NGAIZA SERIES OF ADVANCED COMPUTER SCIENCE

ENJOY & MASTER COMPUTER SCIENCE

COMMANDING THE MACHINE

Java • Python • Operating Systems • Databases

Ngaiza Books App
Scan to download on Google Play

Scan for interactive solutions, extra questions, and all Ngaiza titles on your phone.

Enjoy & Master Computer Science

Topical Questions and Answers: Java · Python · Operating Systems · Databases

Ngaiza Series of Advanced Computer Science

© 2026 Ngaiza Education Hub. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means: electronic, mechanical, photocopying, recording, or otherwise without the prior written permission of the publisher, except for brief quotations in reviews or academic commentary.

First Edition, 2026

ISBN 978-9976-5531-0-9

Writers

Eng. Mwamvua Bakari, Eng. Najma Majid, and Eng. Robert Modestus

Editor

Dr. Isakwisa Tende

Series Creator and Director

Ngaiza Lusima

Published by

Ngaiza Education Hub

P.O. Box 60791

Mbezi Beach, Dar es Salaam, Tanzania

www.ngaizabooks.com

contact@ngaizabooks.com

Every effort has been made to ensure the accuracy of the content in this book. The publisher welcomes corrections and suggestions for future editions.

TABLE OF CONTENTS

Preface i
Student Survival Guide iii

1 Object Oriented Programming with Java 1

The Java platform: JDK, JRE, JVM and bytecode 1
The anatomy of a Java program: syntax, data types, input and output 8
Operators, precedence and expression evaluation 19
Control flow and arrays 29
Methods, parameters and the static/instance divide 45
Classes, objects, constructors and object lifetime 56
Encapsulation, access control and packages 70
Inheritance, super and method overriding 81
Abstraction: abstract classes, interfaces and polymorphism 96
Exceptions, testing and building a complete Java application 110

2 Object Oriented Programming with Python 122

The interpreter and the shape of a program 122
Names, objects and operators 131
Collections and control flow 141
Functions and modules 151
Classes, objects and self 161
Encapsulation and abstraction 173
Inheritance, super and polymorphism 185
Errors, exceptions and a working application 197

3 Operating Systems 211

What an operating system is, and what it is not 211
Classifying operating systems, and how they got here 217
Kernel and shell, the two modes, system calls and interrupts 225
Operating system architectures 234
Processes, threads and the process control block 242
CPU scheduling I: criteria, FCFS and shortest job first 250
CPU scheduling II: preemption, quanta and priority 259
Process synchronisation: race conditions, semaphores and the bounded buffer 269
Deadlock: the four conditions, resource graphs and the Banker's algorithm 276
Memory management: allocation and fragmentation 285
Paging, segmentation and virtual memory 294

File systems, device management and disk scheduling 303

4 Databases and Database Management Systems 313

From files to a database: what a DBMS actually adds 313

Database models, historical and current 319

The relational model: structure, keys and the three integrity rules 326

Conceptual, logical and physical design 333

Entity-relationship modelling: entities, attributes, relationships 339

Cardinality, participation and weak entities 346

Resolving many-to-many, and mapping a diagram to tables 353

Functional dependency, and the anomalies it predicts 361

Unnormalised form to first and second normal form 371

Third normal form, Boyce-Codd, and where fourth normal form begins 380

Testing a decomposition: lossless join, dependency preservation, denormalisation 388

Creating and altering tables: types, and the constraints that hold a schema together 397

Inserting, updating and deleting rows 407

Querying one table: SELECT, filtering, NULL and ordering 415

Aggregation and grouping: the WHERE and HAVING rule 425

Joins: inner, outer, cross and self 434

Subqueries, views and stored routines 444

Transactions and ACID 457

Concurrency, locking and isolation levels 469

Indexing, storage and query optimisation 483

Access control, SQL injection and password storage 498

Connecting from PHP with PDO 509

Parallel and distributed databases 520

NoSQL, data warehousing and big data 529

Index 539

PREFACE

This book was written because you deserve better.

Advanced Level Computer Science rewards precision, yet many students arrive at the paper having learned to recite definitions they cannot apply and to describe programs they have never written. The subject looks approachable, because everybody has used a computer and most students have seen a line of code, and that familiarity is exactly the trap. Candidates memorise lists of features, sit an examination that asks them to define, to compute, to write working code and to judge, and leave believing that computer science is a matter of technical words confidently spelled. It is not. It is one of the few subjects in which almost every claim can be settled by working it out.

This volume is where that becomes literal. A program either compiles or it does not. A query either returns the rows you claimed or it returns different ones. A schedule either averages the waiting time you calculated or it does not. There is no room here for an answer that is nearly right, and there is no need for one, because everything in these pages can be checked by the person writing it. Mastering it under examination conditions takes more than a book to read. It takes practice, attempting real questions, defining, computing, writing and judging, and learning, answer by answer, how to turn understanding into marks. That is what this book sets out to give you.

Why a book of questions and answers?

You do not learn to reason by reading someone else's reasoning, and you do not learn to write a program by admiring a finished one. You learn by attempting a question, getting it wrong, finding out exactly why, and trying again. This book is built entirely around that truth. Every idea in this volume is turned into questions you must answer yourself, each followed at once by a full model answer that shows not only the right conclusion but the definition, the working, the listing and the judgement that earn examination marks. Used honestly, with the model covered until you have written your own, it trains the four things the examination really tests: defining with precision, reasoning from cause to effect, producing code and queries that actually run, and judging how far a claim will hold.

How this book is organised

This volume covers the second half of the Form Five course in four chapters: object oriented programming with Java, the same ideas again in Python, operating systems, and databases and database management systems. The second chapter is deliberately the same subject in a second language, because that contrast is the fastest way to see which parts of object orientation were the idea and which were merely one language's way of writing it. The first half of the course, which carries computer architecture, processor performance, advanced computer networks and C++, is the companion volume, INSIDE THE MACHINE. Neither book is a summary of the other.

Every question is followed at once by its model answer, so that you can attempt it and check yourself immediately. Within each chapter the questions rise through four tiers.

BINDER questions anchor the core. They fix the definition, the mechanism and the arithmetic that must be right before anything else can be, and even here they ask you to reason rather than merely to name. If you can answer these, you own the idea.

REAL questions carry that core into the country itself, from a posho mill at Didia to a grain store at Buhongwa and two clerks at Makuyuni who save the same correction and lose one of them, where the subject actually lives. Many of them hand you another candidate's working and ask you to find the error in it, which is the fastest way to stop making it yourself.

BUILD questions ask you to write the program or the query, not to describe one. There are ninety-nine of them in this volume, because this is where the practical paper is won, and because the practical paper accepts nothing else.

HOT questions push you to examination standard and beyond, drawing several ideas together and demanding a genuine judgement rather than a routine.

Every answer in the book ends with a Master Key, a single sentence built to be carried into the hall. Every listing printed in these pages was compiled and executed before it was set in type, and every output beside a listing is the text that came back from that run, so the outputs are evidence rather than decoration. Read them as evidence, and prove your own code the same way.

Before the questions begin, a short Student Survival Guide shows you how to work this book, and how to beat the three habits that quietly cost good students their marks: the true fact attached to the wrong construct, the program described where it should have been written, and the pair of words that differ by one idea. Read it first. It is the shortest and most important part of the book. At the back, a full index takes you from any term to the pages where it is defined, written or applied.

The team behind this book

Enjoy and Master Computer Science is the work of the Ngaiza Education Hub, built by a team of experienced computer science educators and engineers who share one conviction: that Tanzanian students deserve world-class materials written for their context, their course, and their aspirations.

Writers: Eng. Mwamvua Bakari, Eng. Najma Majid and Eng. Robert Modestus.

Editor: Dr. Isakwisa Tende.

Designer: Mr. Frank Mwita.

I am grateful to every member of this team for their dedication, their expertise, and their refusal to settle for "good enough." Compiling and running every listing, and checking every query against the data printed beside it, is slow work that no reader will ever see and every reader will benefit from. I thank too the students whose hard questions and honest struggles shaped every answer in this book. They reminded us, again and again, that a question a student cannot learn from has failed, no matter how correct its computer science may be.

To every student of Advanced Computer Science opening this book for the first time: the subject is not as hard as its reputation, and it is not as easy as its first pages feel. It is as exact as the definitions you will learn to trust, and as satisfying as a program that runs the first time. Give it your honest effort, with the model answer covered and your pencil moving, and this book will give you its honest best in return.

Ngaiza Lusima
Series Creator and Director

STUDENT SURVIVAL GUIDE

How to turn what you know into marks.

Why a Right Idea Loses Marks in This Subject

Almost everything in this volume can be checked. A program either compiles or it does not. A query either returns the rows you claimed or it returns different ones. That is the great mercy of the subject and also its severity: where a claim can be decided, an examiner is entitled to decide it, and an answer that is nearly right is simply wrong. Three habits of imprecision account for most of the marks that leave a prepared candidate's script.

The first trap is the true fact attached to the wrong construct. Every chapter of this book opens by warning you about it, because most lost marks here are a true fact fastened to the wrong thing. A program is a file, a process is that file running, and a thread is the part of it the processor schedules; calling a thread a process, or calling swapping virtual memory, throws away a sentence that was otherwise correct. The kernel is the core of the operating system, not the whole of it. Overriding is not compile-time polymorphism. A relation is not quite a table. Name the construct, then state the property, in that order.

The second trap is describing the thing you were asked to write. This is the volume where the practical paper lives, and the practical paper accepts nothing else. A question that says write a program wants code that would compile; a question that says write the statement wants SQL a server would accept. An answer explaining what the program would do, however clearly, has answered a question that was not asked. Every listing printed in this book was compiled and executed before it was set in type, and every output beside a listing is the text that came back from that run. Read those outputs as evidence, and get into the habit of proving your own code the same way.

The third trap is the near-identical pair. This subject arrives in sets of words that differ by one idea, and the paper is built on exactly those differences. Preemptive is not non-preemptive. Internal fragmentation is not external. A page is not a frame. There are four conditions for deadlock, not three. A superkey is not a candidate key, and a candidate key is not the primary key. WHERE is not HAVING. Nothing in this book is hard once the words hold still, and nothing in it is safe until they do.

The same question, answered by two candidates.

The question. Write a complete Java program that reports the state of a school computer laboratory: one class called LabRegister, holding the laboratory name, the number of machines installed and the number found faulty, computing the number working rather than storing it, and printing three lines. Then state the two commands that compile and run it.

The answer that fails. The program would need a class with three variables for the name, the installed machines and the faulty ones. It would subtract the faulty from the installed and print the three lines using println. You compile it in the editor and run it. (Every claim is correct and almost none of it is marked: no class was declared, no line was written, and the two commands were the part the question asked for last and most plainly.)

The answer that scores. A twelve line program: public class LabRegister with public static void main inside it; a String for the laboratory and two ints for installed and faulty; then int working = installed - faulty, which computes the total rather than storing it, so it can never fall out of step with the two numbers beneath it; then three System.out.println lines. Compiled with javac LabRegister.java, which produces LabRegister.class, and run with java LabRegister. The public class on line 1 forces the file name, so LabRegister.java is the only name this source may be saved under.

Both candidates knew what the program had to do. Only the second wrote one, and the difference is not knowledge but the habit of putting the thing on the paper instead of talking about it. A candidate who has only ever pressed a button in an editor has never seen either command, and cannot write down what the button did.

How to Use This Book

This is not a book to read. It is a book to work, with a pencil in your hand and, wherever you can reach one, a machine to run your answer on. A model answer you have only read feels learned and is gone by morning. Work every question in four steps.

Cover the model answer. Read the question, then write your full answer on paper, code and all, exactly as you would in the hall, before you look at anything.

Uncover and compare. Read the model twice, once for the computer science and once for the way it is set down: its opening sentence, its working, its listing and the output beneath it. Did you reach the same result, and did you reach it the same way?

Mark yourself honestly. Award a mark only where your answer truly earned it. A missing semicolon, a method that returns nothing where a value was wanted, a WHERE clause left off an UPDATE: each is a lost mark, not a near one.

Run what you wrote, wherever you can. The compiler is a harsher and fairer marker than you are, and it answers immediately. A program that looks correct and a program that is correct are different objects, and only one of them is worth marks.

Climb the ladder, and do not skip the bottom rung. Every chapter rises through four tiers. BINDER fixes the core: the definition, the mechanism and the arithmetic that must be right before anything else can be. REAL carries that core into named Tanzanian settings, a posho mill at Didia, a grain store at Buhongwa, two clerks at Makuyuni who save the same correction and lose one of them. BUILD asks you to write the program or the query, and there are ninety-nine of them in this volume, because this is where the practical paper is won. HOT asks for judgement: to weigh, to compare, and to decide how far a claim holds.

Work the chapters in order. The four chapters run from the languages outward: object oriented programming with Java, the same ideas again in Python, operating systems, and databases. The second chapter is deliberately the same subject in a second language, because that contrast shows which parts of object orientation were the idea and which were merely Java's way of writing it.

Read the Master Key last, never first. Every answer in this book closes with one sentence built to be carried into the hall. It is a summary, not a shortcut, and it is worth nothing to a student who has not first done the work it sums up.

Know that this volume is one of two. The Form Five course runs across INSIDE THE MACHINE, which carries computer architecture, the performance of the processor, advanced computer networks and C++, and COMMANDING THE MACHINE, which you are holding. Neither volume is a summary of the other, and the syllabus asks for both.

How to Write Computer Science for Marks

Marks in this subject are won in sentences and in listings, and a small number of habits win most of them.

Name the construct before you state the property. The class, the object, the process, the thread, the relation, the attribute. Say which thing you mean, then say what is true of it. A true fact attached to the wrong construct scores zero.

Define before you argue. Open by defining every technical term the question names, in one clean sentence each. The definition earns its own mark and it fixes the meaning you will use for the rest of the answer.

Distinguish by giving the difference, not two descriptions. Deadlock and starvation are not distinguished by describing each in turn. In deadlock the resource is held by a peer and is unavailable; in starvation it is free and is handed elsewhere. One sentence, both halves, and the mark is earned.

Compute rather than describe. A question that gives you burst times wants a chart and an average. An answer explaining what shortest job first is for will score almost nothing beside one that works it out.

Write the statement out in full. Not the shape of the query but the query: SELECT, the columns, FROM, the table, WHERE, the condition, and the semicolon. The practical paper accepts nothing else, and a described query is an unwritten one.

Say when, not only what. Overloading is settled by the compiler from the number and types of the arguments; overriding is settled while the program runs, from the class of the object rather than the type of the variable. Name both kinds, say who chooses, and say at which moment.

Check your claim against the data in front of you. Almost every claim in this subject is decidable. If a query is supposed to return three markets, count them.

Give both sides wherever judgement is asked for. Argue the case, then the case against, then decide and say why. State the mechanism and name the costs that are real: an answer that reaches a condition scores where one that reaches a slogan does not.

The Words That Cost Marks

This subject arrives in near-identical sets, and more marks are lost here than anywhere else. Learn the right-hand column to the letter.

Word	In everyday speech	In this subject
Program	Software; an application	A passive file of instructions on disk. That file running is a process; the part of it the processor schedules is a thread
Class	A group or category	A description of state and behaviour. One thing made to that description is an object, and new is what makes it
Parameter	A setting or limit	A variable named in the declaration. The expression supplied at the call is an argument
Overloading	Putting too much on	Two methods, one name, different parameter lists, chosen by the compiler. Same name and same list in a subclass is overriding, chosen at run time
Encapsulation	Wrapping something up	The mechanism: data bundled with the methods that act on it, and access controlled. The result, showing only what is needed, is abstraction
Starvation	Having nothing at all	One runnable process repeatedly passed over, though the resource is free. Where the resource is held by a peer and none can proceed, that is deadlock
Relation	A connection between things	A named set of tuples over a fixed list of attributes. Say relation when designing and table when writing SQL
Key	The thing that unlocks it	A superkey identifies; a candidate key identifies minimally; the primary key is the candidate key chosen; the rest are alternate keys
Transaction	A sale or a payment	A group of statements carried out as one indivisible piece of work, bounded by START TRANSACTION and by COMMIT or ROLLBACK
Index	A list at the back of a book	A separate sorted structure holding a value and a row address, which costs disk and is maintained on every write

How to Write Code and Queries for Marks

The listing is a second answer, marked on its own. Give every one the same furniture, in the same order, and the marks follow.

Answer the specification line by line. A BUILD question lists what the program must hold, compute and print. Take the lines in order, then read them again against your code and point to where each is satisfied.

Compute what follows, store only what you are told. A total that is stored can fall out of step with the numbers beneath it the first time somebody forgets to update it. A total that is computed cannot.

Write the whole program, including the parts that feel like ceremony. The class header, the main line, the braces, the semicolons. They are what makes it a program rather than a description of one.

Say what the two commands are. The practical paper marks the commands as well as the code. javac names the file and produces the class file; java names the class and runs it.

Trace before you claim an output. Follow your own code by hand, writing down what each variable holds after each statement, then print the output you actually expect rather than the one you hope for.

In SQL, know at what stage each clause runs. WHERE filters rows before the groups are formed and so changes the totals themselves; HAVING filters formed groups and leaves their totals as they are. An aggregate written into a WHERE clause is refused outright.

Never write an UPDATE or a DELETE without deciding about the WHERE clause. Leaving it off is not a typing error. It is a different statement, and it acts on every row in the table.

The same query, written by two candidates.

The question. For each market, count the stalls and total the rent, keeping only markets whose total rent is at least 30000.00.

The query that fails. SELECT market, COUNT(*), SUM(rent) FROM stall WHERE SUM(rent) >= 30000.00 GROUP BY market; (The intention is right and the server refuses it. WHERE is evaluated before any group exists, so there is no total there to compare, and it comes back as an invalid use of a group function.)

The query that scores. SELECT market, COUNT(*) AS stalls, SUM(rent) AS collected FROM stall GROUP BY market HAVING SUM(rent) >= 30000.00 ORDER BY market; The condition names an aggregate, so it belongs in HAVING, applied after the groups are formed. Every stall is still counted, the totals printed are the whole ones, and what is dropped is an entire market rather than some of its rows.

How to Handle the Numbers

The arithmetic in this volume is small, it is marked strictly, and the marks live in the working rather than in the answer. Six habits protect them.

Draw the chart before you compute anything. Draw the Gantt chart with every boundary ticked and labelled, read the table off the chart, and take the averages off the table. Chart, table and averages must all come from one schedule, or one of the three is wrong.

Know which time is which. Turnaround is completion minus arrival. Waiting is turnaround minus the burst. Write both definitions at the top of the working and neither can be confused under pressure.

Average by adding and dividing, visibly. Four waiting times of 0, 5, 7 and 7 total 19, and 19 divided by 4 is 4.75. Put the total and the division on the page; they are where the marks sit.

Carry the unit. Units of processor time, bytes, pages, frames. Most wrong answers are right numbers wearing the wrong unit.

In normalisation, work one form at a time and say what you removed. First normal form removes the repeating group, second removes the partial dependency, third removes the transitive one. Name the dependency you are eliminating at each step, and the decomposition explains itself.

Interpret the number in a sentence. An average waiting time of 4.75 is not an answer until you have said what it cost: a process needing a single unit of processor time waited seven, seven times its own burst, and nothing in first come first served can see that.

How to Fail Productively

You will get questions wrong. Every strong candidate does. What decides your grade is the five minutes afterwards.

The unproductive response is to rub out your attempt, read the model, nod, and move on. Your page looks tidy and your hand has learned nothing, because you have only watched someone else think.

The productive response is to stop, and not to erase. Set your answer beside the model and find the exact line where the two part company. Was it the definition, because you used the everyday word? The construct, because the fact was true of something else? The code, because you described it instead of writing it? The clause, because a condition stood at a stage of the query where nothing it named existed yet? The arithmetic, because the chart and the table disagreed? Or the command word, because you explained where you were asked to judge? Five faults need five different cures, and naming yours is half the cure. A wrong answer you have diagnosed is worth more than a right one you copied.

How to Revise on Your Own

Most revision feels productive and is not. Re-reading model answers and copying them out neatly let your eyes move while your memory rests. These methods work instead.

Retype every program from the specification, never from the listing. Cover the code, read the spec lines, write the program, then compile it. The compiler will tell you in one second what an hour of reading will not.

Write queries against the tables in the chapter. Take a table printed in the book and set yourself three questions about it, then write the SQL and work out by hand what it must return before you accept an answer.

Work the scheduling and normalisation families until they are automatic. Five Gantt charts, five decompositions, from figures you have made up yourself, marked against your own working.

Say the difference aloud. Process and thread. Overloading and overriding. Deadlock and starvation. Page and frame. Where you can name both but cannot say what separates them, you have not yet understood either.

Learn the opening sentence of each model answer. It is written to be learned: one sentence, the whole definition, nothing wasted. Learn ten a week and the definition marks stop escaping.

Revise your weakest chapter first, not your favourite. The topic you enjoy is the one you already own; the one you avoid is the one the paper will find.

How to Face the Examination

The paper does not test how clever you are. It tests whether you can define, explain, compute, write working code and judge, by the rules and against the clock. Those are skills, and skills can be practised.

Read every question twice before you write. The second reading catches the command word and the specification line the first reading skimmed.

Obey the command word. Each asks for a particular kind of answer, and giving the wrong kind loses marks even when the computer science is sound. The table below sets out what each demands.

On the practical paper, write the code first. The explanation can be added in the margin; a program that is not on the page cannot be marked at all.

Plan every long answer before you write it. Three minutes with a pencil, listing the points in order, buys more marks than three minutes of extra writing at the end.

Spend your time by the marks. A question worth twenty deserves roughly twice the minutes of one worth ten, whatever your instinct says while you are enjoying a topic.

Leave nothing blank. A definition, one correct line of code, even one correct sentence earns marks that an empty space cannot. Reach every question you are required to answer.

Command word	What it asks for
Define; State	Give the exact meaning in one sentence. No explanation is wanted and none is paid for.
Explain; Account for	Give the reason and the mechanism behind it. Every such answer needs a because.
Describe; Outline	Give an orderly account of what happens, without having to justify it.
Distinguish; Compare	Set the two side by side and give the differences directly. Two separate descriptions do not count.
Write a program	Produce code that would compile and run, complete with its class and its main, not a description of it.
Write the statement; Query	Produce SQL a server would accept, ending in a semicolon, and say what it returns.
Trace; Show	Follow the code or the schedule step by step and record the state after each step.
Compute; Determine	Work the figure in full, every line shown, and state the answer with its unit.
Draw	Produce the labelled diagram or Gantt chart itself. It carries its own marks.
Evaluate; Assess; Argue; Judge	Weigh both sides and reach a judgement, saying how far the claim holds and on what condition.

You now carry what the strongest candidates bring into the hall and most never receive: the discipline to name the construct before you state the property, the habit of writing the thing rather than describing it, an eye for the pair of words that differ by one idea, an honest way of being wrong, and a clear plan for the day itself. The machine described in the companion volume is here put to work, and the first question is waiting a page away. Sharpen your pencil and begin.

Chapter One

OBJECT ORIENTED PROGRAMMING WITH JAVA

Java is the second object oriented language on this course, and C++ was the first. The two share almost every idea and disagree about how it is spelled, and the marks in this topic sit almost entirely in the disagreements. A class is still a class. What differs is that every object is reached through a reference, that the machine your program runs on is itself a program, that memory is reclaimed for you, and that several things C++ allowed are simply not permitted here. Knowing why they are not permitted is worth more than knowing that they are not. Where this chapter sets the two side by side it states what each language does, so nothing here depends on having the C++ material to hand.

The chapter runs from the platform outward. It opens with the toolchain, because a student who cannot say what the compiler produced cannot debug anything. It then takes the anatomy of a program, its types and its input, the operators, and the control structures and arrays that everything else is built from. Only then does it reach objects: methods and the divide between class and instance, constructors and object lifetime, access control and packages, inheritance and overriding, and abstraction through abstract classes and interfaces. It closes by building, testing and debugging a whole application, which is what the practical paper actually asks for.

Two habits earn marks in this topic and almost nothing else does. The first is to name the construct before you state the property, because a true fact attached to the wrong construct scores zero. The second is to run the program. Every listing printed in this chapter was compiled and executed before it was set in type, and every output beside a listing is the text that came back from that run. Read the outputs as evidence, not as decoration, and get into the habit of proving your own code the same way.

The Java platform: JDK, JRE, JVM and bytecode

This section settles what a Java installation actually contains and what each part does with your program. It separates the JDK from the JRE and both from the JVM, follows a source file through javac into bytecode and through the launcher into a running process, and shows why that extra step is what lets one compiled file run on machines that share no instruction set. It closes on the question a candidate is most often asked to argue: whether the extra step costs speed.

BINDER Question 1

The laboratory at Isaka received twenty refurbished machines and a technician who installed Java on all of them. A week later the Form Five class could run the programs their teacher handed out and not compile their own. He had installed the wrong thing.

Define the JDK, the JRE and the JVM. State the relationship between the three, and explain what a machine can and cannot do if it carries only the JRE.

Model Answer

The JDK is the developer's kit and contains the JRE, which is the runtime environment and contains the JVM, which is the program that actually executes your compiled code.

The JVM: the Java Virtual Machine is a program that loads a class file and executes the bytecode inside it. It is the only part of the three that runs anything. It also manages memory for the program, reclaiming objects that can no longer be reached.

The JRE: the Java Runtime Environment is the JVM together with the standard class libraries your program calls, such as String and Scanner. It is everything needed to run a finished Java program and nothing that is needed to produce one.

The JDK: the Java Development Kit is the JRE plus the tools that make a program in the first place: javac, which compiles source into bytecode, javadoc, which generates documentation from comments, and jdb, the debugger. Installing the JDK gives you all three layers at once.

What the laboratory at Isaka was missing: with only the runtime, a machine can execute a class file somebody else compiled and can do nothing with a .java file, because it has no compiler. Since Java 11 there is no separate runtime download to install by mistake; the correct answer on any modern machine is to install the JDK.

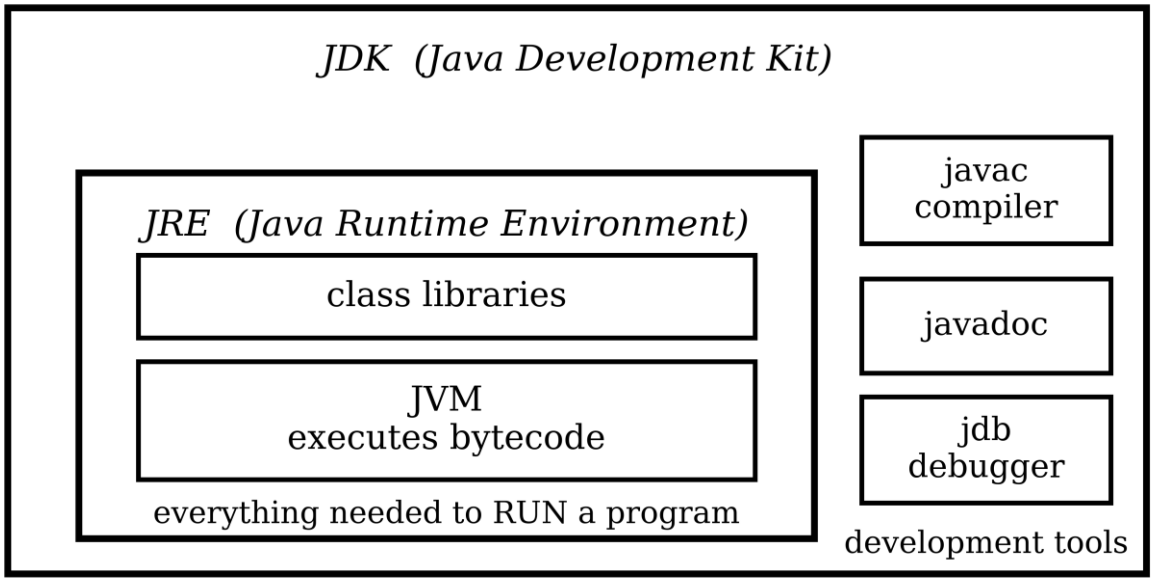


Figure 1.1: *The three layers, drawn as containment. The development tools sit inside the kit and outside the runtime, which is the whole of the distinction.*

Name the layer before you name its job, and the three stop blurring into one another.

The Master Key: *JDK contains JRE contains JVM; only the JVM runs anything, and only the JDK can compile.*

BINDER Question 2

A teacher at Tinde compiled one program on the school's Linux server and copied it onto a memory stick. It ran on Windows machines, the head teacher's laptop, a borrowed Mac. The same trick with C++ failed the term before, at the first machine.

Explain what bytecode is, and how compiling to bytecode rather than to machine code makes a Java program portable. Contrast this with the way a C++ program is built.

Model Answer

Bytecode is the instruction set of the Java Virtual Machine rather than of any real processor, so one compiled file runs on every machine that has a JVM.

What javac produces: javac does not translate your source into instructions for the processor in front of it. It translates it into bytecode, a compact instruction set defined by the language itself, and writes it into a file ending .class. Nothing about that file is specific to Windows, to Linux, or to any chip.

Where the machine dependence went: it moved into the JVM. A JVM for Windows and a JVM for Linux are different programs, written and shipped separately, and each one turns the same bytecode into the instructions its own processor understands. The portability is real, but it is bought by requiring a JVM to be present.

How C++ differs: a C++ compiler produces machine code for one processor and one operating system directly. The source is portable, so the same .cpp file can be compiled anywhere; the executable is not, so a binary built on Linux will not run on Windows. C++ moves the source between machines and recompiles; Java moves the compiled file.

The phrase to be careful with: students are often taught that Java is portable because it is interpreted. That is the wrong reason. Portability comes from targeting a virtual instruction set that every platform implements, and modern JVMs compile hot bytecode to native code as the program runs.

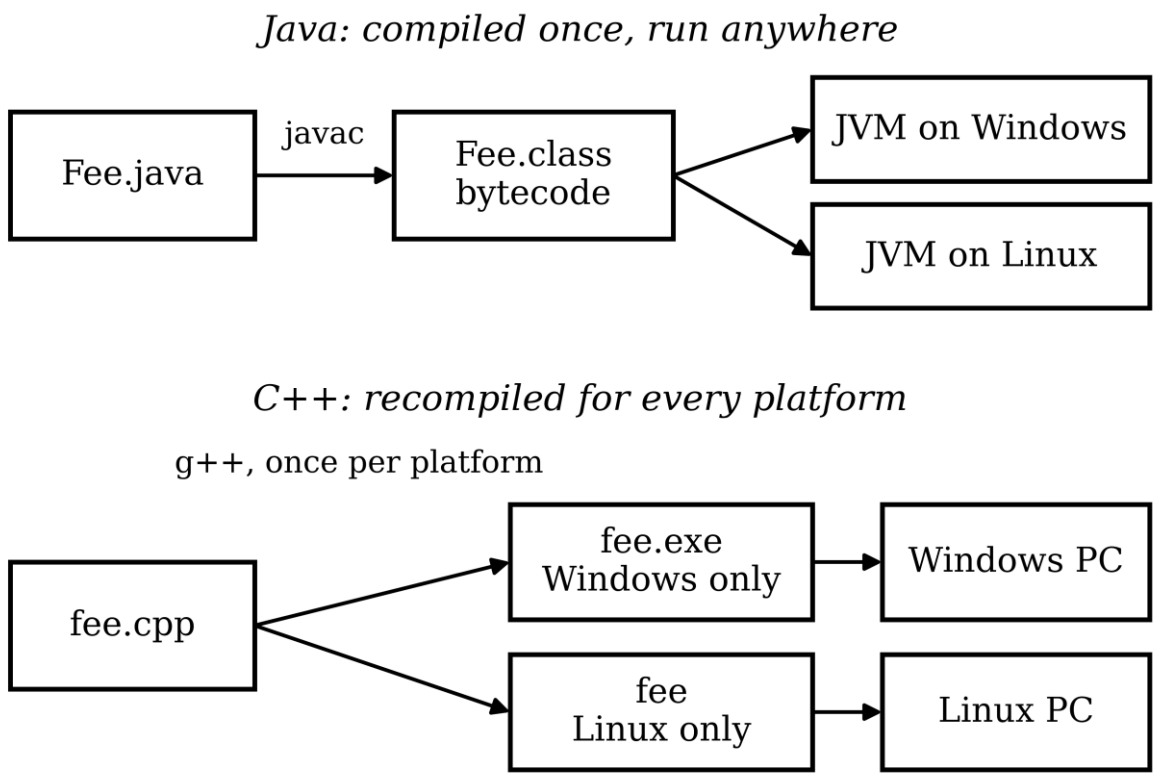


Figure 1.2: *One Java compilation reaching two runtimes, against two C++ compilations producing two binaries that cannot be exchanged.*

Java compiles once and adapts at run time; C++ adapts at compile time and runs directly.

The Master Key: *Bytecode targets the JVM, not the processor, so one class file runs wherever a JVM does.*

BINDER Question 3

Two commands, six seconds apart, and a file that appeared between them. A student at Kisesa could type both from memory and could not say what either one had made. An error message later in the term told her nothing at all.

Describe the steps that take a Java source file to a running program, naming the command and the artefact at each step. Explain why the launcher is given a class name and not a file name.

Model Answer

A source file is compiled by javac into a class file of bytecode, and that class is then loaded and executed by the launcher, java, which is given the name of the class rather than the name of any file.

Step one, compilation: javac Fee.java reads the source and writes Fee.class. The source file must be named after the public class it contains, so a public class Fee must live in Fee.java. A file with no public class in it may be named anything, which is why a two class listing sometimes appears under an unexpected file name.

Step two, launching: java Fee starts a JVM, asks it to find a class called Fee, loads the bytecode, and calls its main method. Writing java Fee.class fails, because the launcher treats what follows as a class name and there is no class called Fee.class.

Why main is declared as it is: the JVM must call main before any object of your class exists, so main is static; it must call it from outside the class, so main is public; and it has no use for a returned value, so main returns void. Every word in the header is there for a reason the launcher imposes.

What survives between the steps: only the class file. Comments, the names of your local variables and the layout of your source are compilation-time matters, and the machine that runs your program never sees them.

Compile a file, run a class, and the two commands stop looking interchangeable.

The Master Key: *javac takes a file name and writes bytecode; java takes a class name and executes it.*

BINDER Question 4

A recruitment notice pinned in a Nansio internet cafe asked for Java and listed four reasons the firm had chosen it. Three of them were properties of the language. The fourth was a property of the machine that runs it.

State four characteristics of the Java language and explain what each one gives the programmer.

Model Answer

Java is object oriented, platform independent through bytecode, automatically memory managed, and supplied with a large standard library.

Object oriented throughout: every method in Java belongs to a class; there are no free functions of the kind C++ allows. This forces the program to be organised around the things it models, and it is why even a program that does nothing but print a line still declares a class to hold main.

Platform independent: compilation targets the JVM, so a compiled class runs unchanged anywhere a JVM exists. The cost is that the JVM must be installed, and that the program starts a little more slowly because the machine it runs on has itself to be started.

Automatic memory management: objects are allocated on the heap and reclaimed by the garbage collector once nothing can reach them. There is no delete to forget and no pointer left dangling, which removes an entire family of C++ faults and replaces it with a smaller one, namely holding a reference longer than you need it.

A large standard library: String, Scanner, the collections and the input and output classes arrive with the runtime rather than having to be found. This is why a Java answer to a small problem is often shorter than the equivalent in a language that ships less.

Characteristic	What it gives you	What it costs
Object oriented	Structure that follows the problem	Even trivial programs need a class
Platform independent	One compiled file runs everywhere	A JVM must be installed
Automatic memory management	No manual freeing, no dangling pointer	Less control over when memory is released
Large standard library	Common work already written	More to learn before you are fluent

Table 1.1: *Each characteristic with the price attached, because a candidate who can name only the benefit cannot answer a question that asks for a judgement.*

Four characteristics, and each of them buys something at a stated price.

The Master Key: *Object orientation, portable bytecode, garbage collection and a wide library, each with a cost.*

REAL Question 5

The program was correct. The student at Kibara had checked every line of it twice against her notes, and the machine still refused to run it, with a message naming a method she had written. She had not written it the way the launcher requires.

The listing below compiles without a single warning and still fails the moment it is run. Name the fault, state exactly what the JVM does about it, and correct it.

Model Answer

The fault is that main is not declared static, so no such method exists for the launcher to call before an object has been made, and the JVM refuses to start the program.

```
1  public class Register {
2      private int machines = 24;
3
4      public void main(String[] args) {
5          System.out.println("machines: " + machines);
6      }
7  }
```

Program 1.1: *A method called main that the launcher cannot use.*

Why the compiler is silent: an instance method called main is perfectly legal Java. Nothing in the language reserves the name, so javac has no reason to object, and the class file is produced exactly as written. The requirement is imposed by the launcher, not by the compiler, and it is enforced only when the class is loaded.

What the launcher is looking for: it searches the named class for a method that is public, static, returns void and takes an array of String. All four must hold. Declare it as returning int and the JVM complains that main must return void; leave out static, as here, and it complains that main is not static.

Why static is the one that matters here: the JVM has no object of your class when it starts and no way to make one, because making one is the sort of thing a program does after it has started. A static method belongs to the class itself and can therefore be called when no instance exists.

The correction: insert static into the header so it reads public static void main. Note that the field on line 2 then becomes unreachable from main, because an instance field needs an instance; either make the field static as well or create a Register object inside main and read the field through it.

The habit worth forming: a message that begins Error and names a class is coming from the launcher, and concerns how the program starts. A message that names a file and a line number is coming from javac, and concerns what the program says. Reading which of the two you have narrows the search before you have looked at a single line.

The compiler judges the language; the launcher judges the entry point, and they fail at different moments for different reasons.

The Master Key: *main must be public, static, void and take String[]; the JVM enforces this at load time.*

REAL Question 6

A Form Five student at Mugumu wrote out the two commands in her exercise book and got the second one wrong. She had added three characters that are on the file, are correct about the file, and are wrong in the command.

Explain what each of the two build commands takes as its argument and what each produces. State what happens if a class file extension is given to the launcher, and why the two commands are not symmetrical.

Model Answer

javac takes the name of a file and writes a file; java takes the name of a class and runs it, which is why one command carries an extension and the other must not.

Command	Argument	What it means	Result
javac Fee.java	A file name	Compile the source in this file	Fee.class is written
java Fee	A class name	Find class Fee and call its main	The program runs
java Fee.class	A class name	Find a class called Fee.class	Error: could not find or load main class
java Fee.java	A source file	Compile in memory and run, single file only	Runs, but writes no class file

Table 1.2: *The two commands are not mirror images of one another, and the third row is the mistake that costs a mark.*

Why the compiler wants a file: javac works on text on disk, and text on disk is found by its file name. It is entirely reasonable for it to insist on the extension, because there may well be a Fee.txt and a Fee.class beside the source.

Why the launcher wants a class: the JVM does not load classes by walking the file system on your behalf. It is given a class name, and it searches the class path for a class of that name. Once a class comes from inside a package or an archive, no file name would identify it usefully anyway.

What the fourth row is doing there: since Java 11 the launcher will accept a single source file, compile it in memory and run it in one step. It is a convenience for one file programs and it leaves no class file behind, so it is the wrong habit to form for a practical paper that asks you to compile and then run.

Reading the failure: the message names the class it could not load, and it names it as you typed it. Seeing Fee.class inside the message is the clue: the launcher was not confused by the file, it went looking for a class with a full stop in its name and did not find one.

Where the launcher looks: it searches the class path, which defaults to the directory you are standing in. Compile into one directory and launch from another and the class will not be found, however correct the name is, because the launcher has not been told where to look. The remedy is to pass the directory with the -cp option rather than to rename anything.

Why this is examined at all: the practical paper asks a candidate to compile and run, and marks the commands as well as the program. A candidate who has only ever pressed a button in an editor has never seen either command, and cannot write down what the button did.

One command names a file and the other names a class, and the extension belongs to only one of them.

The Master Key: *javac takes a file name with .java; java takes a class name with no extension at all.*

BUILD Question 7

The technician at Shirati wanted one number on the board every morning: how many of the laboratory machines were fit to use. Counting by hand took ten minutes and was wrong about twice a week.

Write a complete Java program that reports the state of a school computer laboratory to the specification below, then state the two commands that compile and run it.

The class is called LabRegister and is the only class in the file.

It holds the laboratory name, the number of machines installed and the number found faulty.

It computes the number of working machines rather than storing it.

It prints three lines: the laboratory name, the number installed, and the number working.

Use the values Isaka Secondary School, 24 installed and 5 faulty.

Model Answer

The specification asks for one class, three stored values, one computed value and three printed lines, and the program below satisfies it in twelve lines.

```
 1  public class LabRegister {
 2      public static void main(String[] args) {
 3          String laboratory = "Isaka Secondary School";
 4          int installed = 24;
 5          int faulty = 5;
 6          int working = installed - faulty;
 7
 8          System.out.println("Laboratory: " + laboratory);
 9          System.out.println("Machines installed: " + installed);
10          System.out.println("Machines working: " + working);
11      }
12  }
```

Output

```
Laboratory: Isaka Secondary School
Machines installed: 24
Machines working: 19
```

Program 1.2: *The laboratory register, compiled with javac LabRegister.java and run with java LabRegister.*

Where the specification is satisfied: the three stored values are the local variables on lines 3 to 5, the computed value is line 6, and the three printed lines are lines 8 to 10. Nothing stores the working total, so it cannot fall out of step with the two numbers it is derived from.

The one design decision worth naming: working is computed rather than stored. A stored total would have to be corrected every time a machine failed or was repaired, and the first time somebody forgot, the board would carry a number that no longer followed from anything.

The two commands: javac LabRegister.java produces LabRegister.class in the same directory, and java LabRegister then runs it. The file name is forced by the public class on line 1, so LabRegister.java is the only name this source may be saved under.

Twelve lines, one class, and a total that cannot disagree with the numbers beneath it.

The Master Key: *Store what you are told and compute what follows from it.*

HOT Question 8

The staff room argument lasted a whole break. One teacher held that Java is slow because it is interpreted, and that serious work belongs in C++; the other asked when he last measured anything. Only one was talking about a machine built this century.

Evaluate the claim that a Java program must be slower than the equivalent C++ program because Java is interpreted. Reach a judgement.

Model Answer

The claim rests on a description of the JVM that stopped being accurate long ago, and the honest answer is that Java is slower to start and usually competitive thereafter, so the verdict depends entirely on how long the program runs.

The case for the claim: there is a real extra layer. A C++ binary is handed to the processor and executed; a Java class file is handed to another program first. If that program reads one bytecode instruction at a time and acts on it, as the earliest JVMs did, then every instruction costs several, and the penalty is large and unavoidable.

Why the description is out of date: a modern JVM interprets a method only while it is deciding whether the method matters. Once a method has been called enough times the just-in-time compiler translates it to native code, and from then on it runs at native speed. The description that is out of date is not slow versus fast; it is interpreted versus compiled, because a modern JVM does both.

What the JVM knows that a C++ compiler cannot: it has watched the program run, so it can see which branch is actually taken and compile on that evidence. A C++ compiler must decide everything before the program has run once.

Where Java genuinely does pay: start-up is slower, because a JVM must be created and classes loaded before any of your code runs, and the just-in-time compiler has not yet had anything to watch. Memory use is higher. And garbage collection reclaims memory at a moment the program does not choose, which matters where a deadline is fixed in milliseconds.

Where C++ genuinely does pay: the developer, in defects. The faults Java removes by managing memory are the hardest of all to find, and a program that is correct sooner may be optimised sooner.

The verdict: for a short program that starts, does a little work and exits, the claim holds and the difference is mostly start-up. For a program that runs for minutes or hours, it does not hold, because the just-in-time compiler has long since removed the interpreter from the hot path. Choose C++ where the deadline is hard and measured in microseconds; choose Java where the program is long-lived or the cost of a memory fault is high.

How to argue it in an answer: state the mechanism, name the costs that are real, and let the running time decide the verdict. An answer that reaches a condition scores where an answer that reaches a slogan does not.

The claim was true of the first JVMs and is now true only of programs too short for the just-in-time compiler to matter.

The Master Key: *A modern JVM interprets, then compiles; the penalty is at start-up, not throughout.*

The anatomy of a Java program: syntax, data types, input and output

This section takes a Java program apart and puts it back with every part named. It settles the class that holds main and the header the launcher will accept, the difference between print and println, the eight primitive types and the width the language fixes for each of them, and the String, which is not a primitive at all and does not behave like one. It then teaches the Scanner, so that from this point on a program in this chapter can read what somebody types instead of carrying its data in its own source. Two small matters of layout are settled here as well, because they are the first differences a reader coming from C++ actually sees.

BINDER Question 9

The first program in the Utegi laboratory printed its three words down three lines instead of across one. The class was right, the header was right, and every character between the quotes was right. The student used println three times where two wanted print.

Describe the parts of a minimal Java program and state what each part is for. Distinguish print from println, and explain what an escape sequence is and why one is sometimes unavoidable.

Model Answer

A Java program is a class holding a method whose header the launcher fixes as public static void main(String[] args), with statements inside it; anything else in the file is a comment, and no comment survives compilation.

The class that holds main: every Java method lives inside a class, so a program that prints one line still declares one, and the file takes the name of the public class. The opening brace sits at the end of that line, where C++ conventionally puts it on its own.

The header the launcher accepts: public and static so the launcher can call it before any object exists, void because nothing could usefully be returned, and String[] args for the words typed after the class name. Write String[] args, never the C-style String args[]: both compile, and only the first is the house form. Students are often taught, wrongly, that static here conserves memory.

Comments and javadoc: two slashes comment out the rest of a line, and a slash with a star opens a block that runs until a star and a slash close it. A block opening with two stars is a javadoc comment, which the javadoc tool turns into documentation.

print and println differ: println writes what it is given and then a line terminator, so the next thing printed begins on a fresh line; print writes what it is given and stops. Students are often taught that the two behave alike; they differ by exactly one line terminator, which is why lines 8 to 10 come out as one.

Escape sequences: a backslash inside a literal changes the character that follows it. \n gives a new line and \t a tab, a backslash before a quotation mark puts that mark inside the literal, and two backslashes give one. Without the third, a literal would end early.

```
 1  /**
 2   * The anatomy of a Java program: one class, one method,
 3   * two ways of printing, and three escape sequences.
 4   */
 5  public class Anatomy {
 6      public static void main(String[] args) {
 7          // print leaves the cursor where it stopped
 8          System.out.print("Utegi ");
 9          System.out.print("Secondary ");
10          System.out.println("School");
11          System.out.println("Form Five\nForm Six");
12          System.out.println("She said \"present\".");
13          System.out.println("Folder: C:\\forms");
14      }
15  }
```

Output

```
Utegi Secondary School
Form Five
Form Six
She said "present".
Folder: C:\forms
```

Program 1.3: *One class, one method, and three calls that print across a single line because only the last of them is println.*

Name the part, then state its job, and a listing stops being a shape you copy.

The Master Key: *One class, one fixed main header, and println differs from print by exactly one line terminator.*

BINDER Question 10

Two hundred and fifty six. That is how many values a byte holds in Java, on the newest machine in the country and the oldest still running. A C++ programmer cannot say the same about an int without asking which compiler built the program.

Name the eight primitive types of Java and state the width of each. Explain what fixing those widths gives a Java programmer that a C++ programmer does not have, and state the range of char exactly.

Model Answer

Java has eight primitive types, byte, short, int, long, float, double, char and boolean, and the language itself fixes the width of every one of them, so an int is 32 bits on every machine that will ever run the program.

The four whole number types: byte holds 8 bits, short 16, int 32 and long 64, and all four are signed, so each range is split about zero. An integer literal is an int unless L is written after it, which is why a value too large for an int must carry that suffix.

The two approximate types: float holds 32 bits and double 64, and both store a fraction as an approximation rather than exactly. A literal with a decimal point is a double unless f is written after it, and double is the one to reach for unless there is a reason to want the narrower type.

char, and where its range really stops: char holds 16 bits and is not signed, so its values run from 0 to 65535 and there are 65536 of them. A common error is to give the top of the range as 65536, which is one past the last value any char can hold. A char holds a code, not a picture of a letter.

boolean: boolean holds true or false and nothing else, and it is the one primitive whose storage the language declines to fix, because what matters is that it carries one bit of information and not how many bits are spent on it.

What the fixed widths are worth: in C++ the size of an int is decided by the implementation, so a program that assumed four bytes goes wrong the day it meets a compiler that gives it two. Java removes the question, and the same arithmetic gives the same answer everywhere, which is the portability of bytecode carried down into the types.

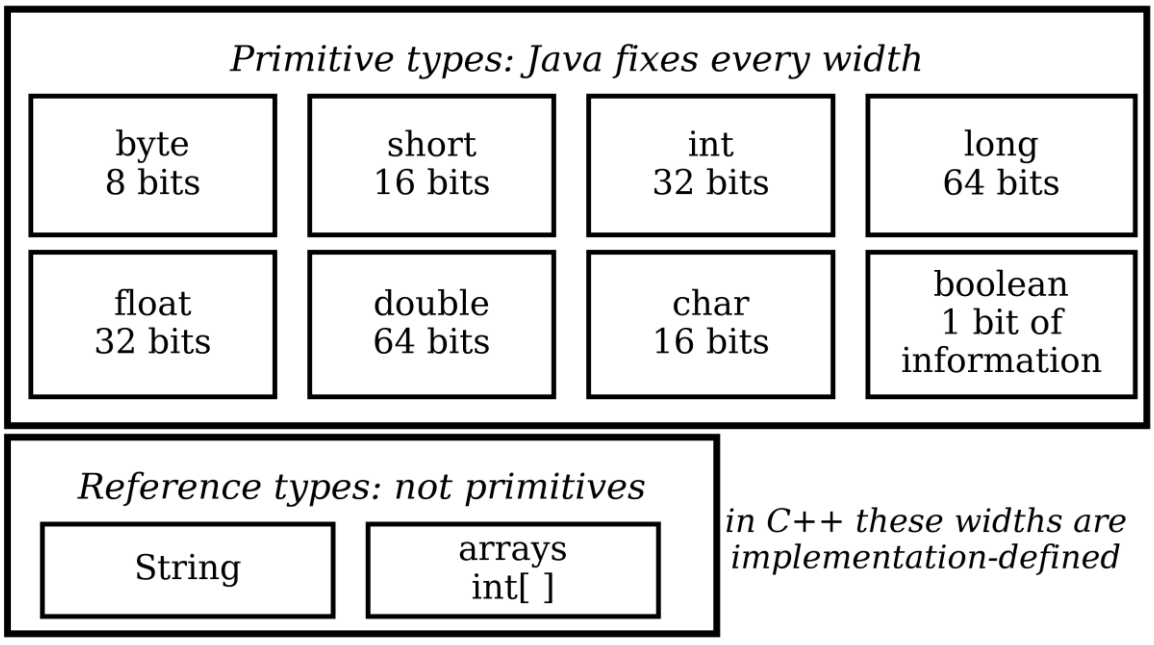


Figure 1.3: *The eight primitives with the width the language fixes for each, set against the reference types, which are a different thing altogether.*

Type	Bits	What it can hold
byte	8	Whole numbers from -128 to 127
short	16	Whole numbers from -32768 to 32767
int	32	Whole numbers from -2147483648 to 2147483647
long	64	Whole numbers from -9223372036854775808 to 9223372036854775807
float	32	Approximate fractions, about seven significant digits
double	64	Approximate fractions, about fifteen significant digits
char	16	One code from 0 to 65535, which is 65536 values in all
boolean	1	true or false; the storage it occupies is not fixed

Table 1.3: *The eight types with the values each can hold. The char row is the one most often mis-stated, and the long row is why a very large count needs its suffix.*

Learn the eight names, the eight widths, and the one range that is nearly always quoted wrongly.

The Master Key: *Eight primitives with widths fixed by the language; char runs 0 to 65535, which is 65536 values.*

BINDER Question 11

The stock book at a Kayanga pharmacy records tablets as whole numbers and prices as fractions, and a program reading it must move a value between the two. One direction Java performs unasked; the other refuses until you write that you accept the loss.

Explain widening and narrowing conversion in Java and state which of the two requires a cast, and why. Explain what a wrapper class is and what autoboxing does.

Model Answer

Java widens a value for you and makes you write a cast to narrow one, because widening cannot put a value out of range and narrowing can, and the cast is how you say you accept what will be lost.

Widening happens by itself: assigning a byte to an int, or an int to a double, needs nothing written, because the wider type has room for every value of the narrower one. The one wart is that an int carrying more significant digits than a float can keep loses the last of them quietly on the way in.

Narrowing needs a cast, and truncates: putting a double into an int requires the type in brackets in front of the value. The fraction is not rounded, it is discarded, so a cast of 1250.75 gives 1250 and not 1251. A common error is to expect a cast to round.

A cast is not a promise that the value fits: casting 300 into a byte compiles, runs and gives 44, because only the low eight bits survive and the rest are thrown away without a word. The compiler stopped objecting the moment the cast was written; it checked nothing on your behalf.

Wrapper classes: Integer, Double, Character and Boolean are classes whose objects each hold one primitive value, for the places where an object is required and a primitive will not serve. They carry useful constants and methods too, such as Integer.MAX_VALUE and Integer.parseInt.

Autoboxing: since Java 5 the compiler inserts the wrapping and the unwrapping itself, so an int may be assigned to an Integer and taken back out with nothing written either way. The convenience is real; the boxed value is still an object reached through a reference.

```
 1  public class Conversions {
 2      public static void main(String[] args) {
 3          int tablets = 250;
 4          double price = 1250.75;
 5
 6          double widened = tablets;
 7          int narrowed = (int) price;
 8          byte overflowed = (byte) 300;
 9
10          Integer boxed = tablets;
11          int unboxed = boxed;
12
13          System.out.println("widened: " + widened);
14          System.out.println("narrowed: " + narrowed);
15          System.out.println("overflowed: " + overflowed);
16          System.out.println("unboxed: " + unboxed);
17          System.out.println("largest int: " + Integer.MAX_VALUE);
18      }
19  }
```

Output

```
widened: 250.0
narrowed: 1250
overflowed: 44
unboxed: 250
largest int: 2147483647
```

Program 1.4: *One value widened without a word, two narrowed by casts, and an int boxed into an Integer and taken out again.*

Widen freely, narrow deliberately, and read a cast as a decision rather than a formality.

The Master Key: *Widening is automatic; narrowing needs a cast, and a cast truncates or drops bits without warning.*

BINDER Question 12

Two students at Nshamba typed the same name into the same program and the program reported that the two names were different. Both had spelled it correctly, letter for letter. The program had not compared the letters at all.

Explain why String is a reference type and what follows from that. Distinguish the == operator from the equals method for Strings, state what immutability means, and explain why the plus sign joins two Strings.

Model Answer

String is a class and not a primitive, so a String variable holds a reference to an object rather than the characters themselves, and == therefore asks whether two variables name one object while equals asks whether two objects hold the same characters.

A variable of class type holds a reference: the characters live in an object and the variable holds the way to reach it. Two variables may name one object, and two separate objects may hold identical characters, and Figure 1.4 shows both arrangements at once.

What each comparison actually answers: == compares the two references, so it is true only when both name the same object. The equals method compares the characters, and it is what you want almost every time. Comparing names, codes or marks with == is the commonest String fault in this topic.

Why the fault survives testing: two identical literals are normally one object, because the compiler keeps a single copy of each literal, so == on them prints true and the program looks correct. Build one of the two at run time and the same comparison prints false, which is lines 7 and 8 of Program 1.5.

Immutable means no method changes it: substring, toUpperCase, trim and concatenation each return a new String and leave the original exactly as it was. A line that calls toUpperCase and does nothing with the result changes nothing at all, which is line 13.

The two jobs of the plus sign: length gives the number of characters and substring gives the characters from the first index up to but not including the second, counting from zero. Plus adds when both sides are numbers and joins when either side is a String, working left to right, which is why one line prints 3 tickets and the next prints 12.

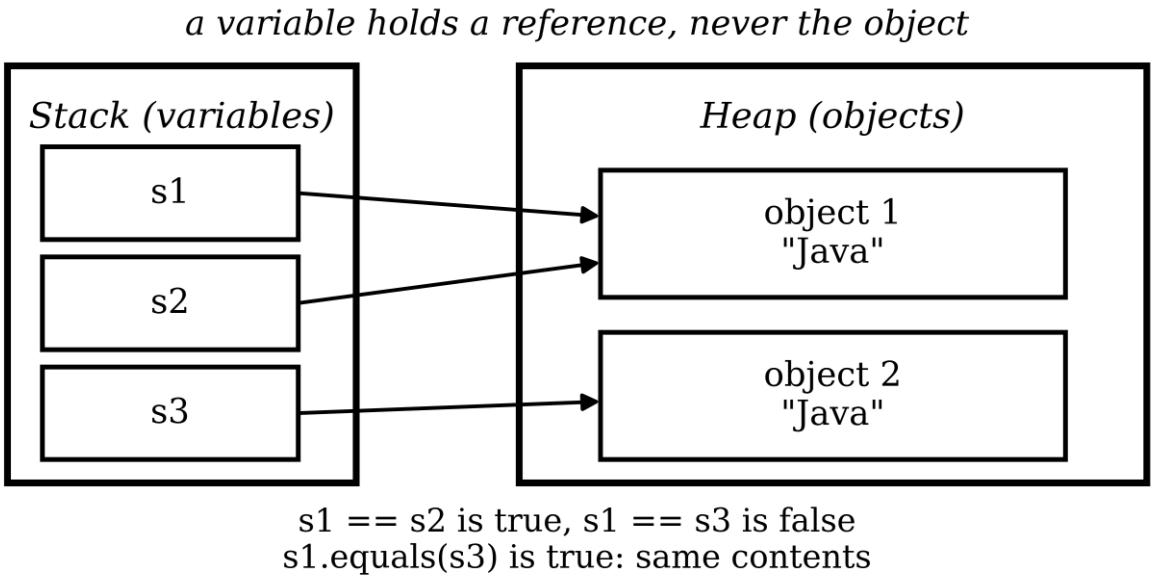


Figure 1.4: *Two variables naming one object and a third naming another object with the same contents, which is the whole of the difference between the two comparisons.*

```
 1  public class Names {
 2      public static void main(String[] args) {
 3          String one = "Nshamba";
 4          String two = "Nshamba";
 5          String three = new String("Nshamba");
 6
 7          System.out.println("one == two: " + (one == two));
 8          System.out.println("one == three: " + (one == three));
 9          System.out.println("equals: " + one.equals(three));
10          System.out.println("length: " + one.length());
11          System.out.println("substring: " + one.substring(0, 3));
12
13          one.toUpperCase();
14          System.out.println("unchanged: " + one);
15          System.out.println(1 + 2 + " tickets");
16          System.out.println("tickets: " + 1 + 2);
17      }
18  }
```

Output

```
one == two: true
one == three: false
equals: true
length: 7
substring: Nsh
unchanged: Nshamba
3 tickets
tickets: 12
```

Program 1.5: *The same seven characters held three ways, and the two answers a comparison can give.*

Ask what a variable holds before you ask what an operator does to it.

The Master Key: *== compares references and equals compares characters, and every String method returns a new String.*

REAL Question 13

The prompt appeared and the cursor did not wait. The program printed its question, printed an empty answer underneath it, and moved on to the next question before anybody had touched a key. One character, invisible, was standing in the queue ahead of the answer.

Explain how a Java program reads what a user types, naming the class and the methods you will use most. Using the listing below, explain why a nextLine that follows a nextInt returns an empty String, and give the cure.

Model Answer

Input comes through a Scanner opened on System.in: nextInt takes the next whole number, nextDouble the next number carrying a fraction, and nextLine the rest of the current line, and it is the join between a numeric read and nextLine that catches almost everybody.

Opening one: import java.util.Scanner at the top of the file, naming the type rather than importing the whole package, then write Scanner in = new Scanner(System.in). System.in is the keyboard. One Scanner serves a whole program and there is no reason to build a second.

The reads worth knowing: nextInt and nextDouble skip any spaces and newlines in front of the number, read as far as the number goes, and stop at the first character that is not part of it. The next method takes one word by the same rule. nextLine is the odd one: it takes everything from the read position to the end of the line.

Where the newline goes: nextInt stops immediately after the last digit, so the newline the user pressed is still sitting in the buffer. The nextLine that follows reads from just after the digits to that newline, finds nothing between the two, and returns an empty String, which line 12 prints as an empty pair of brackets.

The cure: call nextLine once and throw the result away, immediately after the last numeric read, and the nextLine after that collects the line the user meant. Reading the text first and the numbers afterwards avoids the problem rather than curing it, and is worth doing where the order of the questions is yours to choose.

Closing it: call close when the program has finished reading. Closing a Scanner built on System.in closes that input stream with it, so it is done once, at the end, and never in the middle of a program that will read again.

When the input is not what was asked for: nextInt on something that is not a whole number throws InputMismatchException and the program stops there. That is Java refusing loudly rather than reading a wrong value quietly, and what to do about it belongs to a later section.

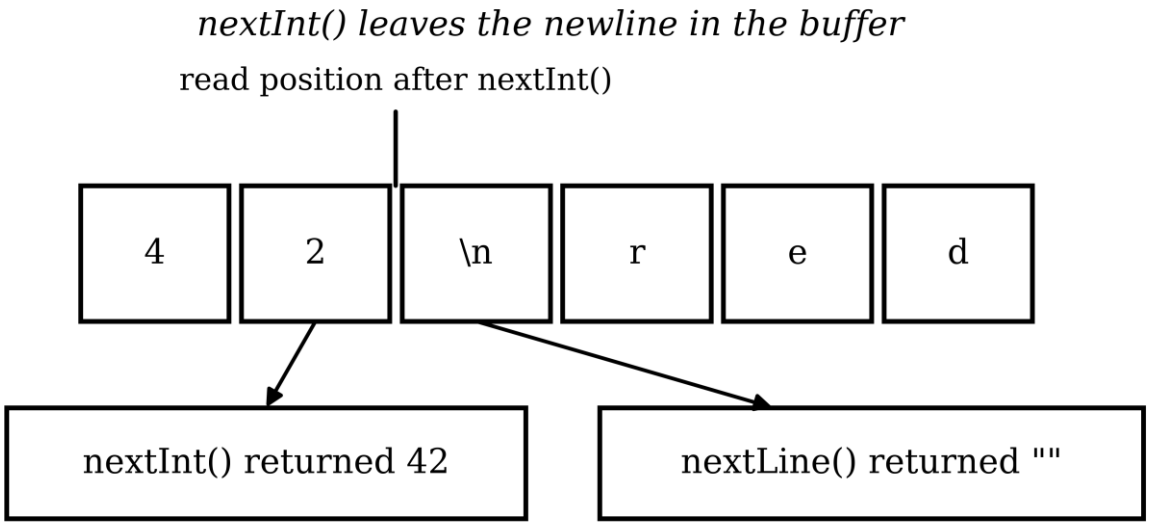


Figure 1.5: *The buffer after nextInt has taken the number. The read position stands in front of the newline, so the nextLine that follows has nothing left to return.*

```
 1  import java.util.Scanner;
 2
 3  public class Buffer {
 4      public static void main(String[] args) {
 5          Scanner in = new Scanner(System.in);
 6
 7          int seats = in.nextInt();
 8          String stub = in.nextLine();
 9          String colour = in.nextLine();
10
11          System.out.println("seats: " + seats);
12          System.out.println("stub: [" + stub + "]");
13          System.out.println("colour: [" + colour + "]");
14
15          in.close();
16      }
17  }
```

Typed at the program

```
42
red
```

Output

```
seats: 42
stub: []
colour: [red]
```

Program 1.6: *Two calls to nextLine after one nextInt, the first collecting nothing and the second collecting the line that was typed.*

Call	What it returns	What it leaves behind	If the input does not match
nextInt()	The next whole number	Everything after the digits, the newline included	Throws InputMismatchException
nextDouble()	The next number with a fraction	Everything after the number	Throws InputMismatchException
next()	The next single word	Everything after that word	Throws NoSuchElementException at the end of the input
nextLine()	The rest of the current line, without the newline	Nothing of that line; the read position passes the newline	Throws NoSuchElementException at the end of the input

Table 1.4: *What each call takes and what it leaves behind. The third column is the one that explains the trap, and it is the column no one reads.*

The Scanner reads exactly what it was asked for and leaves everything else precisely where it was.

The Master Key: *nextInt leaves the newline behind, so a nextLine straight after it returns an empty String.*

REAL Question 14

The message named line seventeen. The program at the Boma Ng'ombe estate office was seventeen lines long, line seventeen closed something the program had opened, and there was nothing wrong with it. The mistake was thirteen lines higher up.

A rainfall tally that will not compile. Name the fault, state exactly what javac reports and why it names a line nowhere near where the mistake was made, and correct it.

Model Answer

The listing does not compile because the closing brace of main and the closing brace of the class are both missing, and javac reports it as reached end of file while parsing at line 17, which is the last line of the file and not the place where anything was left out.

```
 1  import java.util.Scanner;
 2
 3  public class Weighbridge {
 4      public static void main(String[] args) {
 5          Scanner in = new Scanner(System.in);
 6
 7          System.out.println("Bales delivered:");
 8          int bales = in.nextInt();
 9
10          System.out.println("Mass of one bale in kg:");
11          double mass = in.nextDouble();
12
13          double total = bales * mass;
14          System.out.println("Bales: " + bales);
15          System.out.println("Total mass: " + total);
16
17          in.close();
```

Compiler says

```
Weighbridge.java:17: error: reached end of file while parsing
        in.close();
                   ^
1 error
```

Program 1.7: *Seventeen lines that stop in the middle of two open blocks, and what javac says when it runs out of file.*

Naming the fault: two braces are opened and never closed. The brace at the end of line 3 opens the body of the class and the brace at the end of line 4 opens the body of main, and the file stops after line 17 with both of them still open. Nothing else in the listing is wrong.

Exactly what javac does about it: it parses the file, runs out of input, and reports one error, reached end of file while parsing, against line 17. It writes no class file, so there is nothing to run and the launcher cannot even be tried; a practical answer that says the program produces a wrong result has misread what happened.

Why that line number is not the fault: everything down to line 17 is a legal beginning of some longer program. The parser cannot know which block you meant to close or when, so it keeps reading and waiting for the braces to arrive, and the first moment it can be certain that something is missing is the moment there is no more file. The line it names is where the evidence ran out.

Finding the place the message will not give you: the indentation is the evidence, which is the practical reason for four spaces a level and never a tab. Read downwards and check that every line which opened a block has a partner at its own indentation. The brace on line 4 is thirteen lines above the line javac names and its partner is simply absent.

The correction: add two lines at the end, four spaces and a closing brace to end main, then a closing brace at the left margin to end the class. A stray extra brace is the opposite case and gives a different message, class, interface, enum, or record expected, and that one does name the offending line.

An error reported at the last line of a file usually means something was never finished, not that the last line is wrong.

The Master Key: *A missing closing brace is reported at the end of the file, because that is the first place the parser can be sure of it.*

BUILD Question 15

The meter at the Kemondo water kiosk reads in litres, the price list is on the wall, and between them sits a slate where the attendant works out what each customer owes. It is wrong about twice a week, always in the same direction.

Write a complete Java program that prepares a water kiosk bill from what the attendant types, to the specification below.

The class is called WaterKiosk and is the only class in the file.

It reads the litres taken as a whole number and the price of one litre as a number with a fraction.

It then reads the customer name, which may contain spaces, as a whole line.

It computes the amount due rather than reading it.

It prints the customer, the litres, the price and the amount due, one to a line, and closes the Scanner.

Model Answer

The specification asks for two numbers, one line of text that may contain spaces, and one computed total, and the program below satisfies it with a single discarded read on line 13 and no other machinery.

```
 1  import java.util.Scanner;
 2
 3  public class WaterKiosk {
 4      public static void main(String[] args) {
 5          Scanner in = new Scanner(System.in);
 6
 7          System.out.println("Litres taken:");
 8          int litres = in.nextInt();
 9
10          System.out.println("Price of one litre:");
11          double price = in.nextDouble();
12
13          in.nextLine();
14
15          System.out.println("Customer name:");
16          String customer = in.nextLine();
17
18          double due = litres * price;
19
20          System.out.println("Customer: " + customer);
21          System.out.println("Litres: " + litres);
22          System.out.println("Price per litre: " + price);
23          System.out.println("Amount due: " + due);
24
25          in.close();
26      }
27  }
```

Typed at the program

```
23
47.5
Neema Kileo
```

Output

```
Litres taken:
Price of one litre:
Customer name:
Customer: Neema Kileo
Litres: 23
Price per litre: 47.5
Amount due: 1092.5
```

Program 1.8: *The kiosk bill, compiled with javac WaterKiosk.java and run with java WaterKiosk.*

Where the specification is satisfied: the litres are read on line 8 and the price on line 11, the discarded read on line 13 clears the newline the price left behind, the name is taken whole on line 16, and the amount due is computed on line 18 rather than typed by anybody.

The design decision worth naming: the numbers are read before the name, which is exactly what makes line 13 necessary. Asking for the name first would have avoided the discarded read altogether, but a program that reads a number and then a line is the shape you will meet most often, and it is better to hold the cure than to arrange never to need it.

Why the total prints as it does: a value held in a double carries a fractional part even when that part is zero, so 1092.5 is what the machine holds and what it prints. Formatting an amount into two decimal places is a separate matter and the specification did not ask for it.

Read the numbers, clear the line, then read the words, and every value lands where it was meant to.

The Master Key: *A nextLine straight after a numeric read needs a spare call in front of it to swallow the leftover newline.*

BUILD Question 16

The tailor at Katoro writes a slip for every order: the customer initial, the three branch letters at the front of the code, and the number of characters in it. The slip that goes astray is always the one where the customer has two names.

Write a complete Java program that prints an order slip from what the tailor types, to the specification below.

The class is called OrderSlip and is the only class in the file.

It reads the customer name, which contains a space, as a whole line.

It reads the order code, which contains no space, as a single word.

It prints the name, the number of characters in it, the customer initial and the three branch letters at the front of the code.

It ends by printing a slip reference made of the initial, the branch letters and the length of the code, then closes the Scanner.

Model Answer

The specification asks for one line of text, one word, and three pieces cut out of the two, and the program below does it with two reads of different kinds and two calls to substring.

```
 1  import java.util.Scanner;
 2
 3  public class OrderSlip {
 4      public static void main(String[] args) {
 5          Scanner in = new Scanner(System.in);
 6
 7          System.out.println("Customer name:");
 8          String customer = in.nextLine();
 9
10          System.out.println("Order code:");
11          String code = in.next();
12
13          String initial = customer.substring(0, 1);
14          String branch = code.substring(0, 3);
15
16          System.out.println("Name: " + customer);
17          System.out.println("Characters: " + customer.length());
18          System.out.println("Initial: " + initial);
19          System.out.println("Branch: " + branch);
20          System.out.println("Slip: " + initial + branch
21                  + code.length());
22
23          in.close();
24      }
25  }
```

Typed at the program

```
Asha Mwita
KAT-2291
```

Output

```
Customer name:
Order code:
Name: Asha Mwita
Characters: 10
Initial: A
Branch: KAT
Slip: AKAT8
```

Program 1.9: *The order slip, compiled with javac OrderSlip.java and run with java OrderSlip.*

Where the specification is satisfied: the name is read on line 8 and the code on line 11, the initial and the branch letters are cut out on lines 13 and 14, and the slip reference on lines 20 and 21 is built by joining the three pieces the specification names.

Why the two reads are different: the name contains a space, so it has to be taken as a whole line; next would have returned Asha and left Mwita behind for the following read to trip over. The code contains no space, so next is right for it, and it is the difference in the shape of the data that decides which call to use.

What substring counts from: substring(0, 1) gives the character at index 0 and stops before index 1, and substring(0, 3) gives the first three characters. Counting from zero and stopping before the second index is the convention everywhere in Java, and reading that second number as a length is the usual mistake.

Choose the read that matches the shape of the data and the String methods will do the rest.

The Master Key: *nextLine takes a line and next takes a word; substring counts from zero and stops before its second index.*

HOT Question 17

A sheet taped above a machine in an internet cafe listed the size of every C++ type in bytes, written by hand in a column, with a note at the bottom: on this computer. Whoever wrote that note had learned something the hard way.

Java fixes the width of every primitive type, while C++ leaves the size of most of its types to the implementation. Evaluate the two arrangements and reach a judgement.

Model Answer

Fixing the widths is right for Java and leaving them open is defensible in C++, because a language that promises a compiled file will behave identically on machines it has never seen cannot let the arithmetic change underneath it.

What Java is actually fixing: the language definition states the width of every primitive, so an int is 32 bits and stops at the same value everywhere. A program that computes a wrong answer on one machine computes the same wrong answer on the next, and a fault that reproduces can be found.

The case for leaving the size open: C++ lets the compiler choose the size that suits the processor, the intention being that an int is the machine's natural word. On a small controller a 32 bit addition takes several instructions where a 16 bit one takes a single instruction, and a language that insisted would be slower there for no visible benefit.

What that freedom has cost C++: a great deal of code that assumed a size and was wrong the day it was moved. The standard library answered with fixed width types, so a C++ programmer who needs a guaranteed 32 bit integer now asks for one by name, which is Java's arrangement reached from the other direction.

What Java pays for the guarantee: you cannot ask for the machine's natural word, and no integer type is unsigned. byte runs from -128 to 127, so a byte carrying the value 200 in a file arrives in your program as -56. Java 8 added methods that read a signed value as though it were unsigned, but the type itself is still signed.

Where the cost actually falls: for the programs in this chapter it falls nowhere at all, because a 64 bit processor does 32 bit arithmetic in one instruction. The cost is real at two edges only: very large quantities of small numbers, where the spare bits become memory, and work on data somebody else laid out byte by byte.

The judgement: fix the widths. A language whose promise is that one compiled file runs everywhere cannot leave its arithmetic to the machine, because a file that travels and computes a different answer is not portable in any sense a user would accept. Prefer the C++ arrangement only where you compile for one processor you know and the natural word is what you are tuning.

The comparison that scores: say what each language promises before you compare them, name one real cost on each side, and let the promise decide. An answer that says Java is safer and stops there has compared nothing.

The widths are fixed because the compiled file travels, and anything that travels must carry its arithmetic with it.

The Master Key: *Java fixes every primitive width because a file that runs everywhere must compute the same answer everywhere.*

Operators, precedence and expression evaluation

This section settles what each Java operator does and, just as often, what it refuses to do. It separates integer division from real division, distinguishes the value of an increment expression from its effect on the variable, insists that the logical operators take boolean operands and nothing else, and takes the bitwise operators down to the individual bit. It then draws the precedence ladder and works an expression through it by hand, and closes on constants and on the question of when a bracket earns its place.

BINDER Question 18

Zero. That is what the screen showed when a Form Five class at Himo divided twelve by twenty-three, and the calculator lying beside the keyboard showed 0.52. Neither machine was broken. Only one of them had been asked the question the class thought it was asking.

Name the arithmetic operators Java provides and state what each one gives. Explain what division does when both of its operands are whole numbers, what the remainder operator gives, and how the same division can be made to produce a real answer.

Model Answer

Java has five arithmetic operators, and division returns a whole number whenever both of its operands are whole numbers, so 12 / 23 is 0 and the fractional part is discarded rather than rounded.

The five operators: addition, subtraction, multiplication, division and remainder are written +, -, *, / and %. Each takes two operands. The result takes the wider of the two operand types, so an int with an int gives an int, and an int with a double gives a double.

What integer division does: when both operands are integer types, / performs integer division and keeps only the whole number part. 12 / 23 is 0, because twenty-three goes into twelve no times at all. The truncation is toward zero rather than downward, so -7 / 2 is -3 and not -4.

What the remainder gives: % gives what is left after that division has taken all it can. 12 % 23 is 12, because none of the twelve was used. The sign of the result follows the left operand, so -7 % 2 is -1. Java calls this the remainder operator, and remainder and modulus differ exactly on negative operands.

Forcing a real division: make one operand a floating-point value and the whole expression becomes floating-point. Line 8 divides 12.0 by 23 and gets 0.5217391304347826, and line 9 gets the same answer by casting seedlings before the division. Casting the result instead is too late, because the zero has already been produced.

Operator	Name	Expression	Value
+	addition	a + b	35
-	subtraction	a - b	-11
*	multiplication	a * b	276
/	division	a / b	0
%	remainder	a % b	12

Table 1.5: *All five operators on one pair of operands, a holding 12 and b holding 23. Both are int, so every result is an int and the division truncates.*

```
 1  public class SeedlingShare {
 2      public static void main(String[] args) {
 3          int seedlings = 12;
 4          int rows = 23;
 5
 6          System.out.println("12 / 23   = " + seedlings / rows);
 7          System.out.println("12 % 23   = " + seedlings % rows);
 8          System.out.println("12.0 / 23 = " + 12.0 / rows);
 9          System.out.println("cast      = " + (double) seedlings / rows);
10          System.out.println("-7 / 2    = " + -7 / 2);
11          System.out.println("-7 % 2    = " + -7 % 2);
12      }
13  }
```

Output

```
12 / 23   = 0
12 % 23   = 12
12.0 / 23 = 0.5217391304347826
cast      = 0.5217391304347826
-7 / 2    = -3
-7 % 2    = -1
```

Program 1.10: *The same division asked twice of whole numbers and twice of real ones.*

Ask a division what types its operands have, and its answer stops being a surprise.

The Master Key: *Two whole operands give a whole answer; make one of them real and the fraction survives.*

BINDER Question 19

Two statements differed by two characters and by nothing else, and afterwards the same variable held the same number. What differed was what each statement handed back, and the student at Marangu who traced them had never noticed.

Distinguish the prefix and the postfix form of the increment and decrement operators, stating both what each does to the variable and what value the expression itself produces. Explain what a compound assignment operator does that a plain assignment does not.

Model Answer

Both forms add one to the variable and they differ only in what the expression evaluates to: the prefix form gives the new value and the postfix form gives the old one.

The effect on the variable is the same: n++ and ++n both leave n one larger, and n-- and --n both leave it one smaller. Where the expression stands alone as a statement, so that nothing uses its value, the two forms are interchangeable and no reader can tell which was written.

The value of the expression differs: line 4 copies the old value of tickets into taken and then increases tickets, so taken holds 5 while tickets holds 6. Line 8 increases seats first, so issued and seats both hold 6. Read the position of the operator as the order of the work: before the variable means before the value is taken.

Compound assignment: total += 8 means total = total + 8, and -=, *=, /= and %= follow the same pattern. Lines 12 to 15 take total from 40 to 18. The variable is named once rather than twice, so a long name cannot be mistyped on the second attempt, and every one of these operators is right-associative.

The cast the shorthand hides: a compound assignment quietly casts the result back to the type of the left operand, which a plain assignment does not. Line 19 adds 100 to a byte already holding 100 and leaves tally at -56, while tally = tally + 100 would not compile at all. The shorthand is not a pure abbreviation.

```
 1  public class Counters {
 2      public static void main(String[] args) {
 3          int tickets = 5;
 4          int taken = tickets++;
 5          System.out.println("taken " + taken + ", tickets " + tickets);
 6
 7          int seats = 5;
 8          int issued = ++seats;
 9          System.out.println("issued " + issued + ", seats " + seats);
10
11          int total = 40;
12          total += 8;
13          total -= 3;
14          total *= 2;
15          total /= 5;
16          System.out.println("total " + total);
17
18          byte tally = 100;
19          tally += 100;
20          System.out.println("tally " + tally);
21      }
22  }
```

Output

```
taken 5, tickets 6
issued 6, seats 6
total 18
tally -56
```

Program 1.11: *Postfix against prefix, then the compound operators, including one that wraps.*

Decide what the expression is worth before you decide which form to write.

The Master Key: *Prefix gives the new value, postfix the old, and both leave the variable one different.*

BINDER Question 20

The Machame mobile money kiosk divides takings by payouts for its daily average, and on the morning nobody came the ledger showed a dash. A program does not print a dash. It stops, unless the check before the division is in the right order.

State what a relational operator produces and name the six of them. Explain what operand types Java's logical operators accept, describe short-circuit evaluation, and say why it makes the expression count != 0 && total / count > 5 safe.

Model Answer

Every relational operator produces a boolean, and the logical operators accept boolean operands and nothing else, so a Java condition is built out of true and false rather than out of numbers.

The six relational operators: the operators <, >, <=, >=, == and != compare two values and give true or false. The result is a boolean, which in Java is a type in its own right and not an integer wearing a disguise. You cannot add it, and you cannot put a number where one is expected.

The logical operators take boolean operands only: && is logical AND, || is logical OR, and ! is logical NOT. Students are often taught, wrongly, that any non-zero value counts as true. That is C. In Java a condition written as if (a && b) over two int variables is rejected with bad operand types for binary operator, and the logical NOT is !, never ~.

Short-circuit evaluation: && evaluates its left operand first and, if that is false, never evaluates the right one, because the answer is already settled. || does the same once its left operand is true. The right operand is not merely ignored; it is never run, so whatever it would have done does not happen.

Why the guard on line 6 is safe: the division sits inside the right operand. When count is zero the left operand is false, the right is never reached, and the division never takes place. Replace && with the single & and both sides are evaluated whatever happens, the division runs, and the program stops with an ArithmeticException.

The conditional operator: ?: is one operator with one name and three operands, written as a condition, then a value for true, then a value for false. It is an expression and therefore has a value, which is why line 13 can put it on the right of an assignment. The two results must have compatible types.

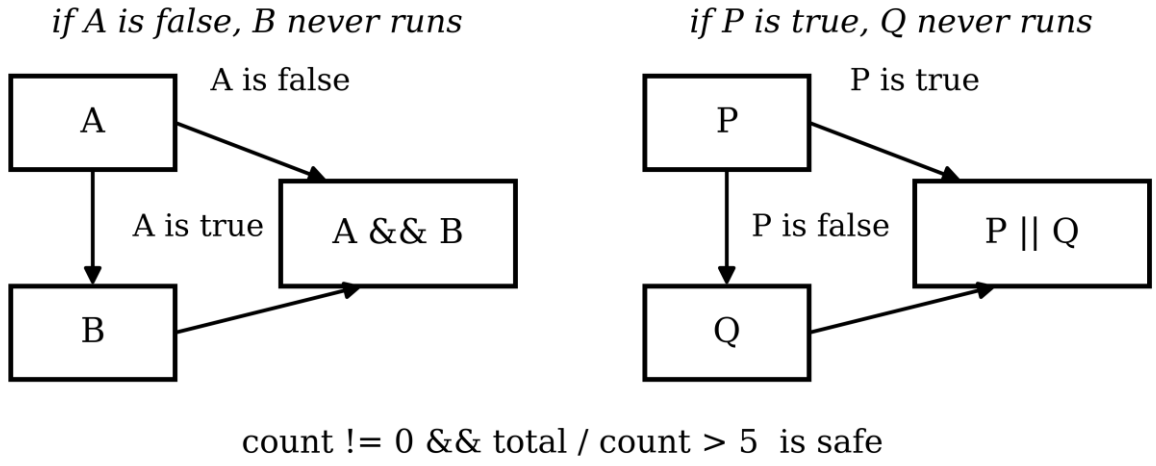


Figure 1.6: *Two routes to each answer, and in each pair one route misses the right operand entirely. That missed route is the short circuit.*

Operator	Meaning	Expression	Value
<	less than	a < b	true
>	greater than	a > b	false
<=	less than or equal to	a <= b	true
>=	greater than or equal to	a >= b	false
==	equal to	a == b	false
!=	not equal to	a != b	true

Table 1.6: *The six relational operators on the same operands as Table 1.5, a holding 12 and b holding 23. The operands are int and every result is a boolean.*

```
 1  public class Guard {
 2      public static void main(String[] args) {
 3          int total = 90;
 4          int count = 0;
 5
 6          boolean safe = count != 0 && total / count > 5;
 7          System.out.println("safe = " + safe);
 8
 9          boolean open = count == 0 || total > 100;
10          System.out.println("open = " + open);
11          System.out.println("not open = " + !open);
12
13          String state = count == 0 ? "no readings" : "ready";
14          System.out.println("state = " + state);
15      }
16  }
```

Output

```
safe = false
open = true
not open = false
state = no readings
```

Program 1.12: *A guard that divides by zero only if it is written the wrong way round.*

Ask what type each operand has before you ask what the condition means.

The Master Key: *Relational operators yield boolean, and && and || take boolean and skip the right side when they can.*

BINDER Question 21

Behind every reading at the Usa River water kiosk stand thirty-two switches, and a program can flip one without disturbing its neighbours. Two of the three shift operators look the same on the page. One turns a negative number into a large positive one.

Explain what the bitwise operators &, |, ^ and ~ do, and how a negative int is held in memory. Distinguish the three shift operators, stating exactly what fills the positions each one empties.

Model Answer

The bitwise operators act on each bit of an integer operand independently rather than on its value, and the three shifts differ in what fills the vacated positions: << brings in zeros at the right, >> copies the sign bit in at the left, and >>> brings in zeros at the left whatever the sign.

The four bit operators: & gives a 1 only where both operands carry a 1, | gives one wherever either operand does, and ^ gives one wherever exactly one of them does. ~ takes a single operand and inverts every bit of it. With 12 as 1100 and 10 as 1010, 12 & 10 is 8, 12 | 10 is 14 and 12 ^ 10 is 6.

Two's complement, and what the tilde really gives: a negative int is held as two's complement: invert every bit of the positive value and add one. The leftmost bit is then 1 for every negative number and 0 for every other one. This is why ~12 is -13 and not something larger: inverting the bits lands one below the negation.

The three shifts: x << n moves every bit n places to the left and fills the right with zeros, which multiplies by 2 raised to n, so 3 << 4 is 48. x >> n moves them right and copies the sign bit into the vacated places, so -16 >> 2 is -4. x >>> n moves them right and brings in zeros regardless of sign, so -16 >>> 2 is 1073741820.

What & and | do to booleans: both accept two booleans as well as two integers, and on booleans they give the same answers as && and ||, with one difference that matters: neither of them short-circuits, so the right operand is always evaluated. Keep && and || for conditions and & and | for bits.

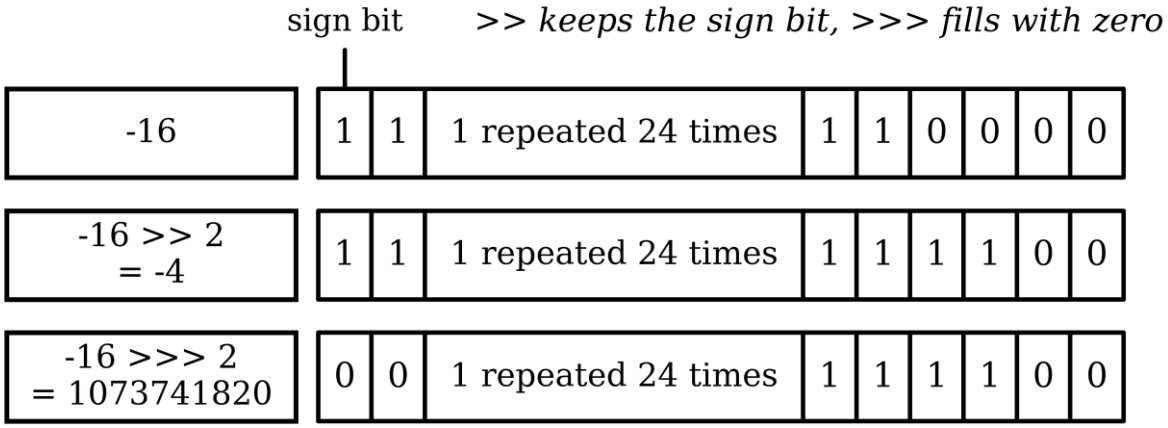


Figure 1.7: *The same thirty-two bits shifted two places by each operator. The leading bit is what separates the two answers.*

```
 1  public class Bits {
 2      public static void main(String[] args) {
 3          int a = 12;
 4          int b = 10;
 5
 6          System.out.println("12 & 10   = " + (a & b));
 7          System.out.println("12 | 10   = " + (a | b));
 8          System.out.println("12 ^ 10   = " + (a ^ b));
 9          System.out.println("~12       = " + ~a);
10          System.out.println("3 << 4    = " + (3 << 4));
11          System.out.println("-16 >> 2  = " + (-16 >> 2));
12          System.out.println("-16 >>> 2 = " + (-16 >>> 2));
13      }
14  }
```

Output

```
12 & 10   = 8
12 | 10   = 14
12 ^ 10   = 6
~12       = -13
3 << 4    = 48
-16 >> 2  = -4
-16 >>> 2 = 1073741820
```

Program 1.13: *Every bitwise operator and every shift, on operands small enough to check by hand.*

Name the fill before you name the direction, and >> stops looking like >>>.

The Master Key: *The signed shift keeps the sign bit, the unsigned shift fills with zero, and ~ lands one below the negation.*

REAL Question 22

*Which happens first? A student at Tengeru wrote 4 + 18 / 4 * 2 - 10 % 4 across the board and the class did not agree on what it came to. One of the answers was Java's, and the rule that decides which one is not a matter of taste.*

Evaluate the expression on line 3 of the first listing by hand, naming the rank that decides each step and saying where associativity rather than precedence does the deciding. Then name the fault in the second listing, state what the compiler reports, and correct it.

Model Answer

Precedence decides which operator takes an operand when two of them compete for it, associativity decides between two operators of equal rank, and the expression works out to 10.

Reading the ladder: the ranks run from 1, which binds tightest, to 14, which binds loosest. Postfix ++ and -- are rank 1, the unary operators are rank 2, * / % are rank 3 and + - are rank 4. Everything is left-associative except the unary operators, the conditional operator and assignment, which are right-associative.

Working the expression: rank 3 goes first, left to right: 18 / 4 is 4 by integer division, then 4 * 2 is 8, then 10 % 4 is 2. That leaves 4 + 8 - 2 at rank 4, worked left to right to give 12 and then 10. Line 4 prints exactly that.

Where associativity does the deciding: in 18 / 4 * 2 two rank 3 operators compete for the 4 in the middle, and precedence cannot separate them. Left associativity gives the 4 to the division, so the integer division happens first and the answer is 8. Grouping the other way gives 18 / 8, and line 9 prints that value as 2 for comparison.

A condition needs no brackets: on line 6 the ranks run % at 3, > at 6, == at 7 and && at 11, each looser than the last, so r > 8 && r % 2 == 0 groups exactly as it reads and comes out true. The ladder was arranged so that a condition written plainly means what it looks like.

Rank 2 holds two operators that are not interchangeable: ! and ~ sit together on rank 2 and there the resemblance ends. ! takes a boolean and ~ takes an integer. Line 4 of the second listing applies ~ to a boolean and javac rejects it with bad operand type boolean for unary operator, with the caret under the tilde. The correction is to write !ready, which prints false.

Where the ladder actually bites: the shifts are rank 5 and the bitwise operators are ranks 8 to 10, all of them looser than + at rank 4. Concatenate a number onto a string and then shift or mask the result without brackets, and Java groups the concatenation first and refuses what follows. That is why lines 6 to 8 and 10 to 12 of Program 1.13 carry brackets and line 9, which uses only rank 2, does not.

rank 1 = highest	operators	associativity	rank	operators	associativity
1	postfix ++ --	left	7	== !=	left
2	unary + - ! ~ (cast)	right	8	& bitwise AND	left
3	* / %	left	9	^ bitwise XOR	left
4	+ -	left	10	| bitwise OR	left
5	shift << >> >>>	left	11	&& logical AND	left
6	< > <= >= instanceof	left	12	|| logical OR	left
13	?: ternary				right
14	= += -= *= /= %= &= ^= |= <<= >>= >>>=				right

Figure 1.8: *The fourteen ranks with their associativity. Rank 1 binds tightest and assignment, at the foot, binds loosest of all.*

```
 1  public class Evaluate {
 2      public static void main(String[] args) {
 3          int r = 4 + 18 / 4 * 2 - 10 % 4;
 4          System.out.println("r = " + r);
 5
 6          boolean ok = r > 8 && r % 2 == 0;
 7          System.out.println("ok = " + ok);
 8
 9          System.out.println("18 / (4 * 2) = " + 18 / (4 * 2));
10      }
11  }
```

Output

```
r = 10
ok = true
18 / (4 * 2) = 2
```

Program 1.14: *One expression from three ranks, one condition from four, and the same division grouped the other way.*

```
1  public class Ready {
2      public static void main(String[] args) {
3          boolean ready = true;
4          System.out.println(~ready);
5      }
6  }
```

Program 1.15: *The logical NOT written as a tilde, which is a different operator altogether.*

Precedence sorts operators of different ranks and associativity sorts operators of the same rank, and an expression needs both before it means anything.

The Master Key: *Rank first, then left to right within a rank; and rank alone never tells you what types an operator will accept.*

REAL Question 23

One capital letter. It sits at the front of a word the compiler knows in lower case, and the message names something else, on a line that looks correct. Move the line from a class body into a method body and the message changes.

The listing below will not compile. Name the fault, state exactly what javac reports and why it reports that rather than what is actually wrong, and correct the line. Then explain how a constant is declared in Java, and what final does and does not fix.

Model Answer

The fault is the capital F in Final, which is not a keyword, so javac reads it as a type name and then finds int where it was expecting the name of the variable.

What javac actually says: the message is <identifier> expected, reported against line 2, with the caret in the gap immediately after the word Final. Nothing in it mentions a capital letter. The compiler was never confused about final, because it never saw final; it saw an unknown type called Final, after which the grammar demands a name.

Why the position changes the message: put that same line inside a method as a local variable and javac reports not a statement instead, followed by a second complaint that a semicolon was expected, because a method body is a list of statements and this is not one. Same character, same cause, two different diagnostics. Never predict a message from the fault; read the one you were handed.

The correction, and how a constant is written: the line should read private static final int MAX_VALUE = 359. final means the variable may be assigned once and never reassigned, static makes it belong to the class rather than to an object so that main can reach it, and UPPER_SNAKE_CASE is the convention that tells a reader at a glance that the value will not move.

A different fault on the same idea: final static int PI = 3.14159 will not compile either, and for an unrelated reason: 3.14159 is a double literal and the variable is an int, so javac reports possible lossy conversion from double to int. The declaration that works is public static final double PI = 3.14159.

What final fixes and what it does not: on a primitive, final freezes the value. On a reference it freezes only which object the variable names; the object itself may still change afterwards. A final reference cannot be pointed at something else and does not make the thing it points at unchangeable.

Reassignment is refused outright: final int limit = 10 followed by limit = 20 is rejected with cannot assign a value to final variable limit, at compile time rather than at run time. The check costs nothing at all and it catches an accidental write to something the program was told to hold still.

```
1  public class Stock {
2      Final int MAX_VALUE = 359;
3
4      public static void main(String[] args) {
5          System.out.println("stock ceiling loaded");
6      }
7  }
```

Program 1.16: *A constant declared with one capital letter too many.*

Read a diagnostic as a fact about where the parser gave up, not as a description of your mistake.

The Master Key: *final means assigned once, and the capital letter in Final makes it a type name the message will never mention.*

BUILD Question 24

The crates went onto the lorry at Oldeani in twelves, and the last of them went on part filled. The tally clerk wanted two numbers on the docket and kept writing three, because he could not settle what to call the part that was left.

Write a complete Java program that reports a landing tally to the specification below, then say which operator supplies each printed number.

The class is called CrateTally and is the only class in the file.

The number of items in a full crate is a constant, declared final and named in upper case.

The program stores the total landed and computes both the number of full crates and the number left loose.

It prints three lines: the total landed, the number of full crates followed by the word crate or crates, and the number left loose.

Use twelve to a crate and 227 landed.

Model Answer

The specification needs one constant, one stored total, two computed values and a choice between two words, and integer division with its remainder supplies both computed values from the same pair of operands.

Where the specification is satisfied: the constant is on line 3 and the stored total on line 4. Line 6 divides to get the full crates and line 7 takes the remainder to get what is left, so 227 / 12 gives 18 and 227 % 12 gives 11. Lines 10 to 12 print the three required lines.

The one design decision worth naming: the loose count is taken with % rather than worked out as landed minus full times PER_CRATE. Both give 11 today, but the remainder says in one symbol what is meant, and it cannot fall out of step with the division above it when somebody changes the crate size.

Why line 8 uses the conditional operator: the word depends on a value, and ?: is an expression, so it produces something that can be assigned to word directly. It is the only Java operator that takes three operands, and it has one name.

```
 1  public class CrateTally {
 2      public static void main(String[] args) {
 3          final int PER_CRATE = 12;
 4          int landed = 227;
 5
 6          int full = landed / PER_CRATE;
 7          int loose = landed % PER_CRATE;
 8          String word = full == 1 ? "crate" : "crates";
 9
10          System.out.println("Landed: " + landed);
11          System.out.println("Full: " + full + " " + word);
12          System.out.println("Loose: " + loose);
13      }
14  }
```

Output

```
Landed: 227
Full: 18 crates
Loose: 11
```

Program 1.17: *The landing tally, compiled with javac CrateTally.java and run with java CrateTally.*

Two operators, one pair of operands, and a docket that cannot contradict itself.

The Master Key: *Division gives the full crates and the remainder gives the rest; ask both of the same two numbers.*

BUILD Question 25

Fold two numbers into one and neither of them is lost. The ticketing desk writes a single integer on every stub, and the route and the seat can both be read back out of it afterwards with one operator each.

Write a complete Java program that packs two small numbers into one integer and recovers them again, to the specification below, and name the operator that does each half of the work.

The class is called TicketCode and is the only class in the file.

Two constants hold the width of the seat field in bits and the mask that selects it, both declared final and named in upper case.

The route number is 37 and the seat number is 14, and each fits inside eight bits.

The program prints the packed code, then the route and the seat recovered from that code alone.

Model Answer

Shifting the route left by the width of the seat field empties eight bits at the bottom, the bitwise OR drops the seat into them, and the unsigned shift and the mask take each field back out again.

Where the specification is satisfied: the two constants are on lines 3 and 4 and the packing is on line 9: 37 shifted left eight places is 9472, the OR adds 14, and the code is 9486. Lines 12 and 13 recover 37 and 14 from that one number, and lines 10, 14 and 15 print the three values.

The one design decision worth naming: the recovery on line 12 uses >>> rather than >>. For a code that is never negative the two agree, but >>> cannot drag a sign bit down into the route, so the field comes back as the unsigned quantity it was packed as whatever the top bit happens to hold.

Why line 9 carries brackets: the compiler does not need them, because << at rank 5 already binds tighter than | at rank 10. They are there so that a reader can see which of the two operators acts first without going to a table to find out.

```
 1  public class TicketCode {
 2      public static void main(String[] args) {
 3          final int SEAT_BITS = 8;
 4          final int SEAT_MASK = 255;
 5
 6          int route = 37;
 7          int seat = 14;
 8
 9          int code = (route << SEAT_BITS) | seat;
10          System.out.println("Code: " + code);
11
12          int backRoute = code >>> SEAT_BITS;
13          int backSeat = code & SEAT_MASK;
14          System.out.println("Route: " + backRoute);
15          System.out.println("Seat: " + backSeat);
16      }
17  }
```

Output

```
Code: 9486
Route: 37
Seat: 14
```

Program 1.18: *Two fields in one integer, packed and then recovered.*

Shift to make the room, OR to fill it, and shift and mask to take it back.

The Master Key: *A shift moves a field and a mask selects one, so one integer can carry two facts safely.*

HOT Question 26

Brackets are free. That is one side of the argument, and it sounds unanswerable until you meet an expression wearing so many that nobody can find the arithmetic underneath. The other side says the ladder is published and a programmer ought to know it.

Evaluate the claim that a programmer who knows the precedence ladder never needs to write a bracket. Reach a judgement.

Model Answer

The claim is true of what the compiler requires and false of what a reader requires, and the judgement turns on whether the operators in the expression come from one family or from several.

The case for the claim: precedence is fixed by the language, not by whoever wrote the compiler, so an expression means one thing everywhere and a bracket adds nothing to its meaning. An expression that reads plainly is easier to check than the same one buried in nested brackets.

Where the claim holds without argument: inside the arithmetic family nobody brackets. Putting * / % above + - is the rule of ordinary algebra, learnt years before any programming, and a * b + c is read as intended by everybody. A condition made of one comparison joined to another reads just as naturally.

Where the claim breaks: the ranks that catch people are the ones ordinary algebra never had to settle. Shifts are looser than addition and the bitwise operators looser still, so both sit below the comparison a reader expects them to bind tighter than. Fourteen ranks is more than anyone holds in mind at once.

What strict typing rescues and what it does not: flags & MASK == MASK groups as flags & (MASK == MASK), and Java refuses it because an int cannot meet a boolean. The same line in C++ compiles and quietly gives the wrong answer, which is the stronger of the two arguments for brackets. Java cannot rescue the arithmetic cases: a - b - c and a - (b - c) both compile, both give an int, and they differ.

The cost a bracket actually carries: it is not free. Every pair is one more thing a reader must match, and a bracket in the wrong place changes the answer in silence, where a missing one at least sometimes fails to compile. An expression defended at every rank is harder to read than the ladder it defends against.

The verdict: trust the ladder inside one family and bracket across families. Where every operator is arithmetic, or where the expression is one comparison joined to another by && or ||, write it plain. Where a shift, a bitwise operator, a conditional operator or an assignment meets anything else, bracket it, because those are the ranks readers do not carry and the ones the ladder settles least obviously.

How to argue it in an answer: name the ranks you are relying on and say what a reader would need to know to check the line. An answer that reaches a condition earns the marks; one that says always use brackets is a rule with no reason attached.

The ladder decides what an expression means; a bracket decides whether a reader can see it without consulting the ladder.

The Master Key: *Rely on precedence within a family and bracket between families, where the ladder surprises people.*

Control flow and arrays

This section supplies the whole of Java's control flow and its arrays, which the common teaching of this topic leaves out altogether. It settles the two selection constructs and the three loops, states what Java demands of a condition and why that demand removes a fault C++ allows to pass, and then takes arrays from declaration through the enhanced for to the index that is one too far. It draws the three loops side by side so that the one which tests after its body can be seen to test after its body, it builds three programs that read their data from the keyboard, and it closes on the choice between a switch and a chain of else if.

BINDER Question 27

One line moved from a C++ practical to a Java one, not a character changed. It had run all term at Katesh and nobody questioned it; in Java it never compiled, and the message spoke about types in a line with no types.

Explain the if, if else and else if forms of selection in Java, and state exactly what the language requires of the condition. Show what Java does with an assignment written where a comparison was meant.

Model Answer

The condition in a Java selection is a boolean expression and nothing else, so an int will not serve as a condition and an assignment written where a comparison was meant is rejected by the compiler instead of being quietly obeyed.

The three forms: if on its own runs its block when the condition is true and does nothing when it is false. if with else runs exactly one of two blocks. A chain of else if tests each condition in turn and runs the block belonging to the first that is true, and an optional else at the end catches everything that matched nothing.

What Java demands between the brackets: the expression must have type boolean. C++ accepts any int and treats every value other than zero as true, so if (count) is legal there. In Java it is a compile error, incompatible types: int cannot be converted to boolean. A comparison such as count > 0 yields a boolean and is what the language wants.

The fault the rule catches: one equals sign written where two were meant. In C++, if (marks = 40) assigns 40 to marks, yields 40, and tests that as true, so the branch is taken every time and no diagnostic is produced. Java evaluates the same assignment to an int, finds an int where a boolean is required, and refuses to compile.

Where the rule still lets you through: if the variable is itself boolean the assignment yields a boolean, so the condition is legal. Written as if (found = true) it compiles, sets found to true and takes the branch every time, exactly as it does in C++. The protection is real and it is not total.

Written in the condition	C++	Java 17
if (count)	Legal. True whenever count is not zero.	Compile error: int cannot be converted to boolean
if (count = 3)	Legal. Assigns 3, then tests 3, so always true.	Compile error: int cannot be converted to boolean
if (count == 3)	Legal. Tests for equality.	Legal. Tests for equality.
if (found = true)	Legal. Assigns true, then tests it.	Legal. Assigns true, then tests it.

Table 1.7: *The same four conditions in both languages. Only the fourth row is a fault that survives the crossing, and it survives because its type is already right.*

```
 1  public class Fare {
 2      public static void main(String[] args) {
 3          int passengers = 3;
 4
 5          if (passengers = 1) {
 6              System.out.println("single fare");
 7          } else {
 8              System.out.println("group fare");
 9          }
10      }
11  }
```

The compiler says

```
Fare.java:5: error: incompatible types: int cannot be converted to boolean
        if (passengers = 1) {
                       ^
1 error
```

Program 1.19: *An assignment standing where a comparison was meant, which C++ would have accepted and run.*

Read the condition first and ask what type it has, because in Java it may only have one.

The Master Key: *A Java condition is a boolean expression, so one equals sign where two were meant is caught by the compiler.*

BINDER Question 28

At Dareda bus stand the fare board names four routes, a price beside each. A program can read it as a chain of comparisons, or as one construct naming the route once and jumping to its line. Only one reads as fast as the board.

Explain the switch statement in Java, naming the switch expression, the case labels, break, default and fall-through. State which types may be switched on and what changed at Java 7.

Model Answer

A switch evaluates one expression, jumps to the case label that matches it, and then runs on through every case after that one until it meets a break or reaches the end of the switch.

The parts of a switch: the switch expression is evaluated once. Each case label is a constant that must be known at compile time and must have the type of the switch expression, so a label may not be a range, a comparison or a variable. The labels are entry points into one block, and they are not separate blocks.

Why break is there at all: control enters at the matching label and does not stop when it reaches the next one. A break is what leaves the switch. Falling through is a deliberate feature of the language and is almost always a mistake in a program, which is why javac reports it as a warning when it is asked for warnings.

What default does: default runs when no label matched. It may be written anywhere among the cases, and if it is not written last it needs a break of its own like any other case. A switch with no default and no matching label simply does nothing at all, which is a silence worth deciding on rather than inheriting.

What may be switched on: byte, short, char and int, the wrapper classes for those, any enum type, and String since Java 7. Not long, not float, not double, not boolean. C++ allows integral types only, so switching on a String is something Java gained and C++ has never had.

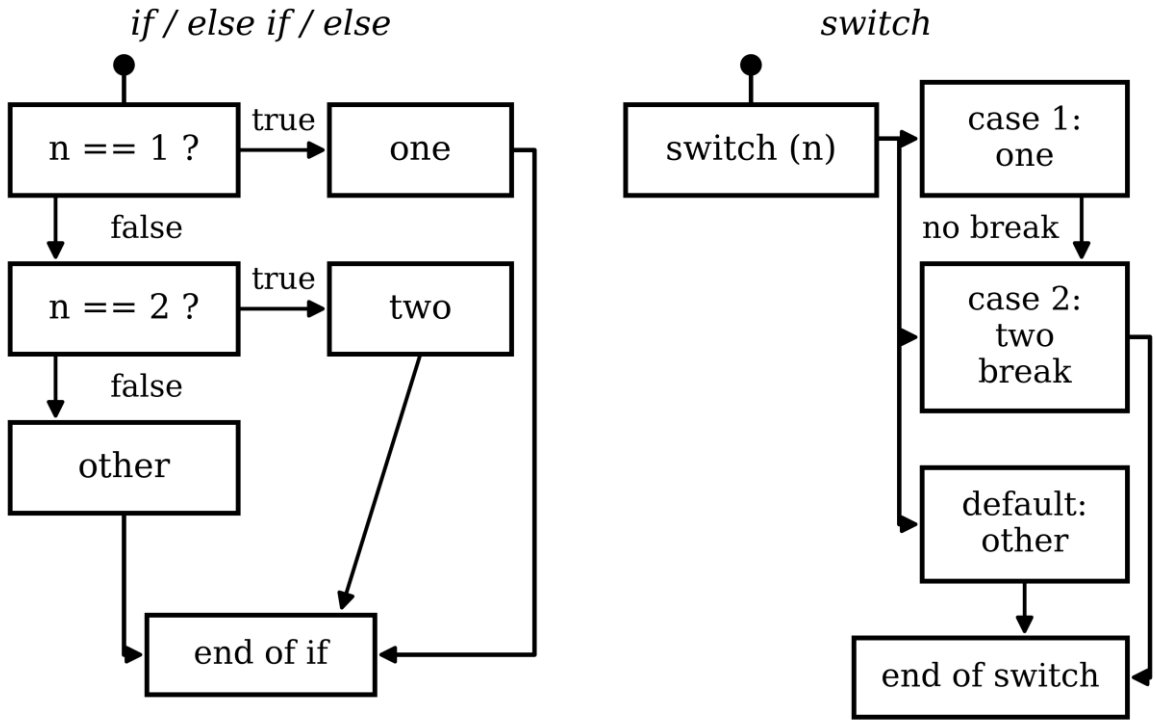


Figure 1.9: *The two selection constructs side by side. Every branch of the chain leads to the end of the if, while case 1 of the switch carries no break and runs on into case 2.*

```
 1  public class FareBoard {
 2      public static void main(String[] args) {
 3          String route = "Magugu";
 4          int fare;
 5
 6          switch (route) {
 7              case "Katesh":
 8                  fare = 2500;
 9                  break;
10              case "Magugu":
11                  fare = 4000;
12                  break;
13              case "Dareda":
14                  fare = 3000;
15                  break;
16              default:
17                  fare = 0;
18                  break;
19          }
20
21          System.out.println("Route: " + route);
22          System.out.println("Fare: " + fare);
23      }
24  }
```

Output

```
Route: Magugu
Fare: 4000
```

Program 1.20: *A switch on a String, which no version of Java before 7 would have compiled.*

One expression, constant labels, and a break at the end of every case you did not mean to continue.

The Master Key: *A switch jumps in and runs on; break is what gets you out, and String labels are legal from Java 7.*

BINDER Question 29

A menu program written at Magugu printed nothing at all. It reached its loop, found the condition false and stepped over the body, so the user never saw the menu and had no idea what to type.

Distinguish the while, do while and for loops in Java and state where each of them tests. Explain what break and continue do inside a loop, and what a break does when it sits inside a nested loop.

Model Answer

A while tests before its body and may run it no times at all, a do while tests after its body and therefore always runs it at least once, and a for gathers initialisation, test and update into one header and tests before its body as a while does.

The while loop: the test comes first, so a condition that is already false costs the program nothing but the test. Something inside the body must eventually make the condition false, because nothing in the header will.

The do while loop: the body runs, then the condition is tested, so the body always runs at least once whatever the condition says. It is the right loop for a menu or a prompt, which must appear before there is anything to judge. Note the semicolon that closes it, which no other loop has.

The for loop: the header carries initialisation, condition and update, separated by semicolons and evaluated in that order, with the update running after each pass. Any of the three may be left empty. A counter declared in the header exists only inside the loop, which is the reason to declare it there.

break and continue: break ends the loop at once and control resumes after it. continue abandons the rest of the current pass and goes to the next test, and in a for loop it runs the update on the way. Both take effect on the loop they stand in and on no other statement.

A break inside a nested loop: it leaves only the innermost loop that encloses it. The outer loop is untouched and starts its next pass, which is what the second half of Program 1.22 shows: the inner loop stops at seat 2 every time and the outer loop still runs all three rows.

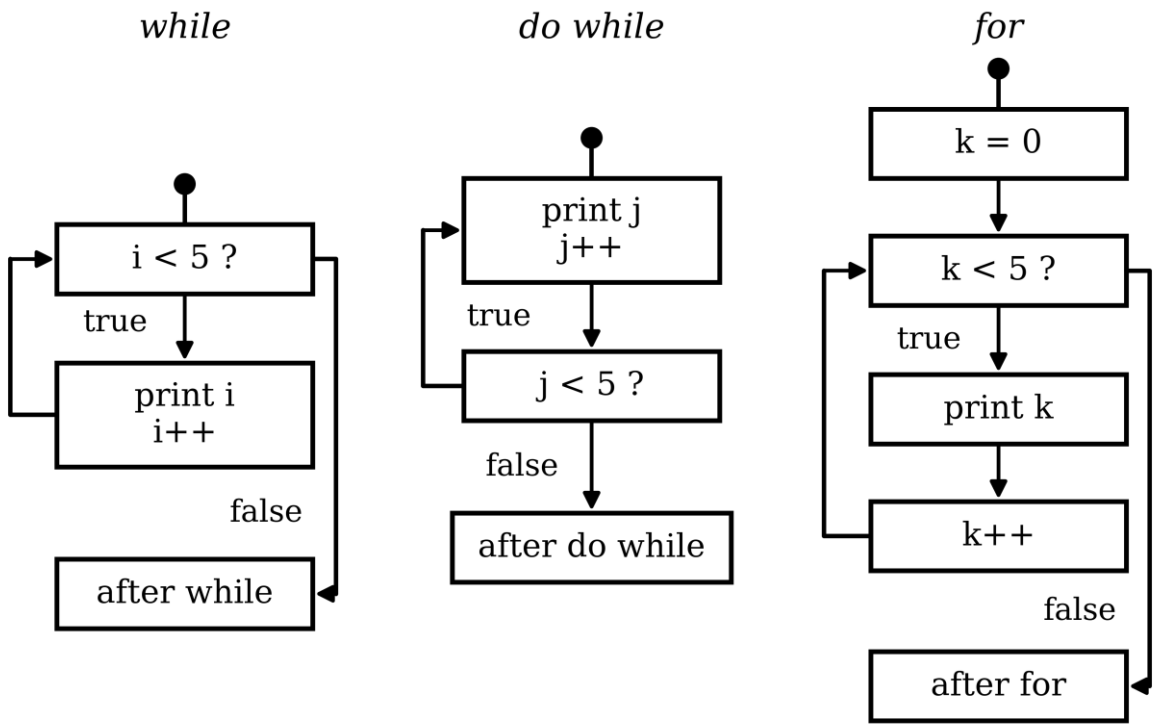


Figure 1.10: *The three loops as flows. Control enters the do while at its body and meets the test afterwards, which is the whole of the difference between it and the other two.*

```
 1  public class Passes {
 2      public static void main(String[] args) {
 3          int stock = 0;
 4          int whilePasses = 0;
 5          int doPasses = 0;
 6          int forPasses = 0;
 7
 8          while (stock > 0) {
 9              whilePasses++;
10              stock--;
11          }
12
13          do {
14              doPasses++;
15              stock--;
16          } while (stock > 0);
17
18          for (int k = 0; k < 3; k++) {
19              forPasses++;
20          }
21
22          System.out.println("while body: " + whilePasses);
23          System.out.println("do while body: " + doPasses);
24          System.out.println("for body: " + forPasses);
25      }
26  }
```

Output

```
while body: 0
do while body: 1
for body: 3
```

Program 1.21: *The same false condition given to all three loops, with the passes counted.*

```
 1  public class Seats {
 2      public static void main(String[] args) {
 3          for (int n = 1; n <= 8; n++) {
 4              if (n == 4) {
 5                  continue;
 6              }
 7              if (n == 6) {
 8                  break;
 9              }
10              System.out.println("seat " + n);
11          }
12
13          for (int row = 1; row <= 3; row++) {
14              for (int seat = 1; seat <= 3; seat++) {
15                  if (seat == 2) {
16                      break;
17                  }
18                  System.out.println("row " + row + " seat " + seat);
19              }
20          }
21      }
22  }
```

Output

```
seat 1
seat 2
seat 3
seat 5
row 1 seat 1
row 2 seat 1
row 3 seat 1
```

Program 1.22: *continue and break in one loop, and a break that leaves only the inner loop of two.*

Choose the loop by where you need the test, and reserve break and continue for the loop they stand in.

The Master Key: *while tests first, do while tests last and so always runs once, and break leaves one loop only.*

BINDER Question 30

The librarian at Haydom keeps forty loan cards in a tray, finding the eleventh by counting from the front. The tray does not object when her hand passes the last card; she finds nothing. A Java array objects the moment the hand goes past.

Explain how an array is declared and created in Java, how its elements are reached, and what length gives you. State the range of valid indices and what Java does with an index outside it. Include the enhanced for and a two dimensional array.

Model Answer

An array is a fixed length block of elements of one type, created with new, reached by an index running from 0 to length minus 1, and carrying its own size in a field called length.

Declaring and creating: int[] marks declares a reference and creates no array at all. new int[6] creates the block and fills every cell with the default of the type: 0 for a numeric type, false for boolean, null for a reference type. The length is fixed when the array is made.

length is a field and not a method: an array carries its size in a field, written marks.length with no brackets. A String carries its size in a method, written name.length() with brackets. Writing marks.length() or name.length is a compile error either way, and the two are examined together because they differ.

The valid indices: they run from 0 to length minus 1, so an array of length 6 has 5 as its last index and marks[6] names nothing. That expression still compiles, because the index is computed while the program runs, and it throws ArrayIndexOutOfBoundsException when the program reaches it.

The enhanced for: for (int mark : marks) visits every element in order and hands you the value, not the position. It cannot go out of bounds, because it never forms an index. Use it to read an array through, and the counted for when you need the position or must assign to the element.

Two dimensions: int[][] terms is an array whose elements are themselves arrays, so terms.length is the number of rows and terms[1].length the length of row 1. The rows need not be the same length. An element takes two indices, each checked separately.

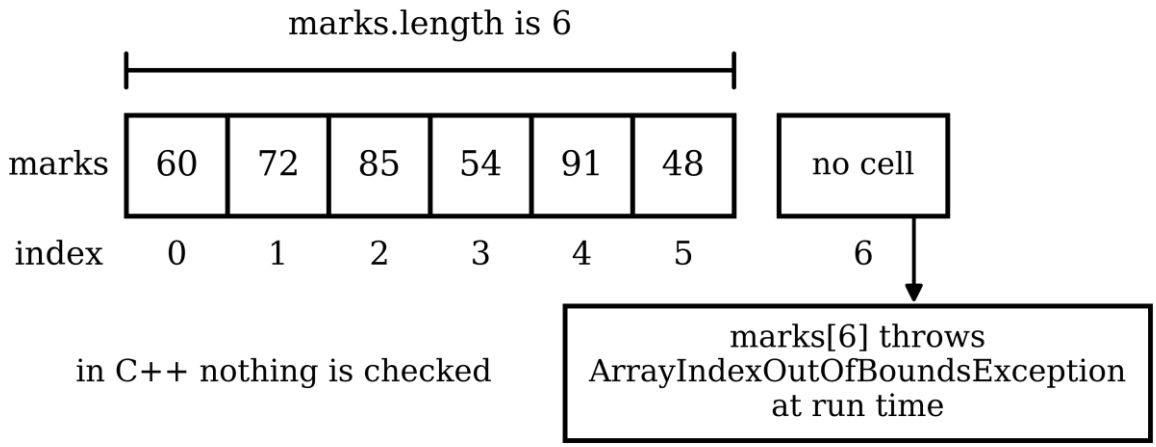


Figure 1.11: *Six cells, indices 0 to 5, and the index that is one too far. Java checks the index before every access; C++ checks nothing and reads whatever lies beyond the block.*

```
 1  public class MarkBook {
 2      public static void main(String[] args) {
 3          int[] marks = new int[6];
 4          marks[0] = 60;
 5          marks[1] = 72;
 6          marks[2] = 85;
 7          marks[3] = 54;
 8          marks[4] = 91;
 9          marks[5] = 48;
10
11          System.out.println("marks.length: " + marks.length);
12          System.out.println("marks[0]: " + marks[0]);
13          System.out.println("last: " + marks[marks.length - 1]);
14
15          int total = 0;
16          for (int mark : marks) {
17              total = total + mark;
18          }
19          System.out.println("total: " + total);
20
21          int[][] terms = new int[2][3];
22          terms[1][2] = 77;
23          System.out.println("terms.length: " + terms.length);
24          System.out.println("terms[1].length: " + terms[1].length);
25
26          String centre = "Haydom";
27          System.out.println("centre.length(): " + centre.length());
28      }
29  }
```

Output

```
marks.length: 6
marks[0]: 60
last: 48
total: 410
terms.length: 2
terms[1].length: 3
centre.length(): 6
```

Program 1.23: *An array of six marks, its length, an enhanced for, and a two dimensional array beside a String.*

Count the cells from zero, take the size from length, and let the enhanced for do the counting wherever you only need to read.

The Master Key: *Indices run 0 to length minus 1; length is a field on an array and a method on a String.*

REAL Question 31

Last year, in C++, the weighbridge program at Basotu added six loads and printed a total larger than the six loads together. Nobody could see where the extra kilogrammes came from, and the program exited happily. The same fault in Java prints no total.

The listing below compiles without a single warning and stops part way through its run. Name the fault, state exactly what the running program does about it, and correct it. Say what a C++ program written the same way would have done instead.

Model Answer

The fault is the loop test on line 12, which is written i <= loads.length and so lets i reach 6, and the seventh pass evaluates loads[6] on an array whose last index is 5, so the JVM throws ArrayIndexOutOfBoundsException and the program ends without printing anything.

```
 1  public class Weighbridge {
 2      public static void main(String[] args) {
 3          int[] loads = new int[6];
 4          loads[0] = 340;
 5          loads[1] = 415;
 6          loads[2] = 280;
 7          loads[3] = 505;
 8          loads[4] = 390;
 9          loads[5] = 460;
10
11          int total = 0;
12          for (int i = 0; i <= loads.length; i++) {
13              total = total + loads[i];
14          }
15          System.out.println("total: " + total);
16      }
17  }
```

The running program says

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
        Index 6 out of bounds for length 6
    at Weighbridge.main(Weighbridge.java:13)
```

Program 1.24: *A loop that runs one pass too many, and an array that says so.*

Why the compiler is silent: the index is a value the program computes while it runs, and javac cannot know that i will reach 6. Everything it can check is correct: loads is an array of int, i is an int, and an int is what an index must be. A bound is a run-time property, so it is checked at run time.

What the running program actually does: six passes succeed and add every load, so the total is already complete and correct when the seventh pass begins. The JVM compares the index 6 against the length 6 before the read, finds it is not less, creates an ArrayIndexOutOfBoundsException and throws it. Nothing catches it, so main ends abnormally, line 15 never runs and the exit status is not zero.

Reading the message: printed as one line, wrapped here to fit the page, it names both numbers, index 6 and length 6, and the frame beneath names the method and source line 13. That is the line holding the read, not the line holding the fault, so read the loop header above it.

The correction: change the test to i < loads.length, which stops i at 5. Better still, drop the index altogether and write the enhanced for, as Program 1.25 does, because a loop that never forms an index cannot form a wrong one. Write the test against length, never against a number you typed.

What C++ would have done instead: nothing. C++ does not check an index, so the read would have fetched whatever bytes happened to lie past the end of the block, added them to the total, printed a number that is simply wrong and exited normally. A write past the end would have damaged something else entirely.

What the check costs and buys: one comparison for every array access, which is not measurable in a program of this size. It buys a program that stops at the fault, names it and points at a line, instead of carrying on with a value nobody can account for.

```
 1  public class Weighbridge {
 2      public static void main(String[] args) {
 3          int[] loads = new int[6];
 4          loads[0] = 340;
 5          loads[1] = 415;
 6          loads[2] = 280;
 7          loads[3] = 505;
 8          loads[4] = 390;
 9          loads[5] = 460;
10
11          int total = 0;
12          for (int load : loads) {
13              total = total + load;
14          }
15          System.out.println("total: " + total);
16      }
17  }
```

Output

```
total: 2390
```

Program 1.25: *The same six loads read with an enhanced for, which has no index to get wrong.*

Java stops at the bad index and says which one it was; C++ reads on and says nothing at all.

The Master Key: *An index outside 0 to length minus 1 throws ArrayIndexOutOfBoundsException at run time, and C++ would have carried on.*

REAL Question 32

Three receipts came out of the printer at Kibaya when the clerk had asked for two, every time, so it was not the printer. The program held one switch, four labels and two breaks. The trouble was not any line that was there.

Trace the listing below and state exactly what it prints. Explain why it prints that, say what javac reports about it and whether the program still runs, and correct it.

Model Answer

It prints three lines, because case 2 carries no break: control enters at case 2, runs line 10, then runs straight on into case 3 and runs line 12 as well, and only the break on line 13 leaves the switch.

```
 1  public class Receipt {
 2      public static void main(String[] args) {
 3          int copies = 2;
 4
 5          switch (copies) {
 6              case 1:
 7                  System.out.println("one copy");
 8                  break;
 9              case 2:
10                  System.out.println("two copies");
11              case 3:
12                  System.out.println("three copies");
13                  break;
14              default:
15                  System.out.println("more than three copies");
16          }
17
18          System.out.println("done");
19      }
20  }
```

Output

```
two copies
three copies
done
```

Program 1.26: *A switch that prints one line more than the value asked for.*

How control enters a switch: the switch expression is evaluated once and matched against the labels. The matching label is an entry point into a single block, not the opening of a block of its own, so nothing about the next label stops the flow of control. Students are often taught, wrongly, that each case is a separate branch like an else if.

Step	What runs	What is printed
The expression is evaluated	copies holds 2	nothing
Control enters at case 2	line 10	two copies
Case 2 has no break	line 12, the body of case 3	three copies
The break on line 13	control leaves the switch	nothing
After the switch	line 18	done

Table 1.8: *The trace. The second and third rows are one pass through one block, and nothing between them tested anything.*

What javac reports: with warnings asked for it reports one warning at line 11, possible fall-through into case, naming the label that can be reached by falling into it. It is a warning and not an error, so the class file is written and the program runs, which is why a warning nobody reads is a fault that survives to the printer.

Where fall-through is genuinely wanted: several labels stacked one above another with a single body underneath, so that case 1: case 2: case 3: share one statement. That is written on purpose and is easy to read. A label with a body and no break is a different thing entirely, and is almost never what was meant.

The correction: put a break at the end of case 2, on a line of its own after line 10. The habit that prevents it is to write the case label and its break together, then fill in the body between them, so the break can never be the thing you forgot.

What the trace teaches about default: default here is written last and so needs no break, because the end of the switch is directly beneath it. Move it above case 1 and it would need one, since it would then fall into case 1. Position changes the requirement, which is a good reason to keep default at the bottom.

A case label says where to start and never says where to stop, and the break is what says where to stop.

The Master Key: *A case without break falls into the next case; javac warns, the compiler still accepts it, and the program runs.*

BUILD Question 33

Two numbers were wanted from a set of eight: how many reached the pass line, and what the class averaged. The head of department had been working them out on the back of the register, and getting one answer on Monday and another on Tuesday.

Write a complete Java program that reports on a set of marks typed at the keyboard, to the specification below.

The class is called MarkReport and is the only class in the file.

It reads how many marks are to be entered, then reads that many marks into an array of exactly that size.

It reports the number of marks entered, the number that are 40 or above, and the average to one decimal place.

The Scanner is closed as soon as the last value has been read.

Use the eight marks 55, 38, 71, 64, 29, 80, 47 and 61.

Model Answer

The specification asks for a count, a filtered count and an average taken from marks that are typed in, and the program below reads them into an array sized by the first number entered and then makes one pass to produce all three.

```
 1  import java.util.Scanner;
 2
 3  public class MarkReport {
 4      public static void main(String[] args) {
 5          Scanner in = new Scanner(System.in);
 6
 7          System.out.println("How many marks?");
 8          int count = in.nextInt();
 9          int[] marks = new int[count];
10
11          System.out.println("Enter " + count + " marks:");
12          for (int i = 0; i < marks.length; i++) {
13              marks[i] = in.nextInt();
14          }
15          in.close();
16
17          int total = 0;
18          int passes = 0;
19          for (int mark : marks) {
20              total = total + mark;
21              if (mark >= 40) {
22                  passes++;
23              }
24          }
25          double average = (double) total / marks.length;
26
27          System.out.println("Marks entered: " + marks.length);
28          System.out.println("At 40 or above: " + passes);
29          System.out.printf("Average: %.1f%n", average);
30      }
31  }
```

Typed at the program

```
8
55
38
71
64
29
80
47
61
```

Output

```
How many marks?
Enter 8 marks:
Marks entered: 8
At 40 or above: 6
Average: 55.6
```

Program 1.27: *The mark report, reading its data from the keyboard.*

Where the specification is satisfied: the array is created at line 9 with the size read at line 8, the filling loop is lines 12 to 14, the Scanner is closed at line 15, and the single pass at lines 19 to 24 produces the total and the count together. The three reported lines are 27 to 29.

The one design decision worth naming: the average is computed as (double) total / marks.length. Without the cast both operands are int, integer division would discard the fraction and the program would report 55 where the true figure is 55.6. The cast is on the total rather than on the result, because casting after the division is already too late.

Reading with the counted for and summing with the enhanced for: the filling loop needs the position, because it assigns to marks[i], so it counts. The second loop only reads, so it takes the enhanced for and gives up the index it does not want. Choosing the weaker loop where the weaker loop suffices is what makes an out-of-bounds fault impossible there.

Three figures from one pass, and an array whose size the user decides at the moment the program runs.

The Master Key: *Size the array from what was read, and cast before you divide or the average loses its fraction.*

BUILD Question 34

A zero is ruled at the foot of the column on a fish landing tally sheet, and it is not a weight. Nobody landed nothing. It says the column ended, and a program treating it as a reading reports a catch nobody brought ashore.

Write a complete Java program that reads weights until a sentinel value ends them, to the specification below.

The class is called LandingTally and is the only class in the file.

It reads weights in kilogrammes, one at a time, until a zero is entered; the zero is not a weight and is not counted.

It reports how many weights were read and the largest of them.

If the very first value entered is the zero, it says that no weights were read and reports nothing else.

Use the weights 18, 24, 31, 12 and 27, then the zero.

Model Answer

The specification asks for a loop whose length nobody knows in advance and whose end is marked by a value that is not data, and the program below reads in the middle of the loop and leaves it with a break the moment the sentinel arrives.

```
 1  import java.util.Scanner;
 2
 3  public class LandingTally {
 4      public static void main(String[] args) {
 5          Scanner in = new Scanner(System.in);
 6          int readings = 0;
 7          int largest = 0;
 8
 9          System.out.println("Enter each weight, then 0 to finish:");
10          while (true) {
11              int weight = in.nextInt();
12              if (weight == 0) {
13                  break;
14              }
15              if (readings == 0 || weight > largest) {
16                  largest = weight;
17              }
18              readings++;
19          }
20          in.close();
21
22          if (readings == 0) {
23              System.out.println("No weights were read.");
24          } else {
25              System.out.println("Weights read: " + readings);
26              System.out.println("Largest: " + largest + " kg");
27          }
28      }
29  }
```

Typed at the program

```
18
24
31
12
27
0
```

Output

```
Enter each weight, then 0 to finish:
Weights read: 5
Largest: 31 kg
```

Program 1.28: *The landing tally, ended by a sentinel rather than by a count.*

Where the specification is satisfied: the reading is line 11, the sentinel test and the break are lines 12 to 14, the largest is kept current at lines 15 to 17, and the count rises at line 18 only once the sentinel has been ruled out. Lines 22 to 27 report, and the empty case has a branch of its own.

The one design decision worth naming: line 15 asks whether this is the first reading before it compares. A program that started largest at zero and never asked would be right for weights, which cannot be negative, and would report zero for any column of readings that all fell below it. The test costs one comparison and removes the assumption.

Why the test sits in the middle of the loop: the value must be read before it can be judged, and must not be counted once it has been judged to be the sentinel. Writing while (true) with a break says that plainly. The alternative reads once before the loop and again at its foot, and writes the same read twice.

The loop ends on a value rather than a count, and the value that ends it never counts as data.

The Master Key: *A sentinel loop reads, tests, then breaks, so the mark that ends the data never becomes part of it.*

BUILD Question 35

There are only two answers to a search of a pharmacy stock book, and one is that the code is not in the book. A program that reports the position and says nothing when it finds none has answered half the question.

Write a complete Java program that fills an array from the keyboard and then searches it, to the specification below.

The class is called StockSearch and is the only class in the file.

It reads five stock codes into an array of five, in the order they are typed.

It then reads one more code and searches the array for it.

It reports the index at which the code was found, or reports that the code is not in the book.

Use the codes 412, 307, 588, 129 and 460, then search for 588.

Model Answer

The specification asks for an array filled by index and then searched by index, and the program below fills it with a counted for, searches with a second counted for that breaks at the first match, and reports the two possible answers through a variable that starts at minus one.

```
 1  import java.util.Scanner;
 2
 3  public class StockSearch {
 4      public static void main(String[] args) {
 5          Scanner in = new Scanner(System.in);
 6          int[] codes = new int[5];
 7
 8          System.out.println("Enter 5 stock codes:");
 9          for (int i = 0; i < codes.length; i++) {
10              codes[i] = in.nextInt();
11          }
12
13          System.out.println("Code to find?");
14          int wanted = in.nextInt();
15          in.close();
16
17          int found = -1;
18          for (int i = 0; i < codes.length; i++) {
19              if (codes[i] == wanted) {
20                  found = i;
21                  break;
22              }
23          }
24
25          if (found == -1) {
26              System.out.println(wanted + " is not in the book.");
27          } else {
28              System.out.println(wanted + " is at index " + found);
29          }
30      }
31  }
```

Typed at the program

```
412
307
588
129
460
588
```

Output

```
Enter 5 stock codes:
Code to find?
588 is at index 2
```

Program 1.29: *Filling an array of five codes and searching it for one of them.*

Where the specification is satisfied: the array of five is created at line 6 and filled by the loop at lines 9 to 11, the code to look for is read at line 14, the search runs at lines 18 to 23, and lines 25 to 29 report whichever of the two answers holds. The code 588 sits third in the book, which is index 2.

The one design decision worth naming: found starts at minus one, which is not a valid index and therefore cannot be confused with one. A program that used 0 to mean not found could never report a match in the first cell. The break at line 21 stops the search at the first match rather than scanning cells that can no longer change the answer.

Why both loops count rather than enhance: the first assigns to codes[i], and the enhanced for cannot assign to an array, only read a copy of each element. The second needs the position, which the enhanced for never offers. Where the position is the answer, the counted for is the loop that has it.

Two answers, both reported, and an index that could never be mistaken for a real one.

The Master Key: *Fill and search by index, start the result at minus one, and break at the first match.*

HOT Question 36

The argument in the laboratory outlasted the lesson. One side held that a switch is tidier than any chain of comparisons and belongs wherever it fits; the other had a problem it would not fit. Both were describing programs the other had not seen.

Evaluate the claim that a switch should always be preferred to a chain of else if statements. Reach a judgement.

Model Answer

The claim holds only where the decision is made on one expression compared for equality against a fixed set of constants known at compile time, and outside that case the chain of else if is not merely acceptable but is the only one of the two that can express the condition at all.

The case for the switch: the expression is named once and tested once, the labels line up beneath one another and read like the board they came from, and no later branch can quietly test a different variable. Since Java 7 a String may be switched on, which removed the commonest reason for reaching past it.

What a switch simply cannot do: the labels must be constants known at compile time, and the match is equality alone. A band such as mark >= 75 cannot be a label, and neither can a test involving a second variable. Every grade-boundary problem is a chain and not a switch.

What the chain costs the reader: the same variable must be checked by eye in every branch, and the order of the branches is part of the meaning: put mark >= 40 above mark >= 75 and the higher band becomes unreachable, with nothing to warn you. The chain is more expressive and more easily made subtly wrong.

The risk the switch carries in return: fall-through. A missing break is not an error, it produces a program that runs, and it gives an answer that is wrong in a way that looks deliberate. That is the strongest argument any chain has, because one branch of a chain cannot run into the branch below it.

Speed is not the argument: a switch over a dense set of int labels may compile to a jump table and reach its label in one step, while a chain tests in order. Over the handful of branches a question of this kind has, the difference cannot be measured.

The verdict: prefer a switch when one expression is compared for equality against a fixed list of constants, because it states that structure plainly. Prefer a chain of else if wherever the test is a range, involves more than one variable, or is not constant. Where the switch fits, write the break before you write the body.

How to argue it in an answer: name the condition that decides it rather than declaring a favourite. An answer that says a switch is tidier scores nothing; an answer that says a switch matches constants for equality, so a range cannot be a label, has given the reason and the limit together.

Question asked of the construct	**switch**	**chain of else if**
What is tested	One expression, evaluated once	A fresh condition in every branch
How a branch is chosen	Equality against a constant label	Any boolean expression at all
A range such as mark >= 75	Cannot be written	Written directly
A test on two variables	Cannot be written	Written directly
Types it accepts	byte, short, char, int, enum, String	Whatever a boolean expression can compare
The fault it invites	A missing break falls into the next case	Branches ordered so an earlier one hides a later one

Table 1.9: *The two constructs answering the same six questions. The third and fourth rows are where the argument is actually settled.*

The switch is the better statement of a narrow thing, and the chain is the only statement of everything else.

The Master Key: *Switch on one expression against constant labels; use else if for ranges, for several variables, and for anything not constant.*

Methods, parameters and the static/instance divide

This section takes a method apart and names every part of it, separates the parameters written in a declaration from the arguments supplied at a call, and shows the compiler choosing between two methods of the same name by the types in front of it. It then draws the line that runs through the whole of Java, between what belongs to the class and exists once and what belongs to an object and exists once for each object, and it settles the two jobs the keyword this does. It closes on the property that misleads a student arriving from C++ more than any other, which is that Java copies every argument, including the reference to an object.

BINDER Question 37

The fare board at the Loliondo stand was repainted twice and still gave the wrong total for a full row of passengers. The clerk beside it used the same numbers and was never wrong. She had one rule for whatever party turned up.

Name the parts of a Java method declaration and state what each part does. Distinguish a method that returns a value from one declared void, and distinguish a formal parameter from an actual argument.

Model Answer

A method declaration has six parts in a fixed order: an access modifier, static or nothing in its place, a return type, the name, a parameter list in brackets, and the body in braces.

```
 1  public class Fare {
 2      public static void main(String[] args) {
 3          int seats = 14;
 4          int total = fareFor(seats, 2500);
 5          report("Loliondo", total);
 6      }
 7
 8      static int fareFor(int seats, int rate) {
 9          return seats * rate;
10      }
11
12      static void report(String route, int total) {
13          System.out.println("Route: " + route);
14          System.out.println("Takings: " + total);
15      }
16  }
```

Output

```
Route: Loliondo
Takings: 35000
```

Program 1.30: *One method that returns a value and one that returns nothing.*

The header, read from the left: public says who may call the method. static, or the gap where it would stand, says whether the method belongs to the class or to an object, and that divide is settled later in this section. int is the type of what comes back, fareFor is the name, and the brackets hold what the method must be given.

void against a returned value: the body is the block in braces and it runs only when the method is called. A method with a return type must hand back a value of that type on every path out, and return ends the method at once. void declares that nothing comes back, so such a call can only be a statement, as line 5 is, while line 4 uses what fareFor returns inside an expression.

Formal parameter against actual argument: the formal parameters are the variables named in the declaration on line 8, and they exist only while the method runs. The actual arguments are the expressions supplied at the call on line 4. They are matched by position and by type and never by name: line 4 and line 8 both say seats and mean two different variables.

Scope, lifetime, and one difference from a field: a parameter, and any variable declared inside the body, lives from that point to the closing brace of its block and is gone once the method returns. A local also has no default

value, so javac refuses one that is read before it is assigned, where a field would quietly have held zero, false or null.

Name the part before you state its job, and a header stops reading as a row of keywords.

The Master Key: *Six parts in a fixed order; the parameters belong to the declaration and the arguments belong to the call.*

BINDER Question 38

Three prices came from one question at the photocopying counter in Kiomboi: one for plain pages, one for colour, one for a customer with her own rate. What decided was not the question but what was put on the counter with it.

Explain method overloading in Java and state exactly what the compiler uses to tell one overloaded method from another. Explain why two methods that differ only in their return type cannot overload one another.

Model Answer

Overloading is declaring two or more methods in one class with the same name and different parameter lists, and the compiler chooses between them from the number and the types of the arguments at the call, never from the return type.

```
 1  public class Shop {
 2      public static void main(String[] args) {
 3          PrintDesk desk = new PrintDesk();
 4          System.out.println("plain: " + desk.cost(20));
 5          System.out.println("colour: " + desk.cost(20, true));
 6          System.out.println("agreed: " + desk.cost(20, 150));
 7      }
 8  }
 9
10  class PrintDesk {
11      int cost(int pages) {
12          return pages * 100;
13      }
14
15      int cost(int pages, boolean colour) {
16          if (colour) {
17              return pages * 400;
18          }
19          return pages * 100;
20      }
21
22      int cost(int pages, int rate) {
23          return pages * rate;
24      }
25  }
```

Output

```
plain: 2000
colour: 8000
agreed: 3000
```

Program 1.31: *Three methods called cost, told apart by what they are given.*

What has to differ: the parameter lists must differ in the number of parameters, in their types, or in the order of those types. The names of the parameters take no part in the choice, so renaming a parameter does not give you a second method; it gives you the same method twice, and javac says so.

Who chooses, and when: javac chooses, while it is compiling, by taking every method of that name and keeping the one whose parameters fit the argument types most closely. The choice is fixed in the class file before the program starts, which is why an overload can never be settled by anything discovered while the program runs.

Why the return type is no help: a call may throw away whatever comes back, and the statement cost(20); on its own is perfectly legal. Two methods that differed only in what they returned would leave that statement with

nothing to choose by, so the language forbids the pair outright and javac reports that the method is already defined.

What overloading is for: one idea earns one name. A class that would otherwise carry costPlain, costColour and costAtRate carries cost three times over, and the caller supplies what he happens to know instead of remembering which of three names matches it.

```
 1  public class PrintDesk {
 2      int cost(int pages) {
 3          return pages * 100;
 4      }
 5
 6      double cost(int pages) {
 7          return pages * 100.0;
 8      }
 9
10      public static void main(String[] args) {
11          PrintDesk desk = new PrintDesk();
12          System.out.println(desk.cost(20));
13      }
14  }
```

The compiler says

```
PrintDesk.java:6: error: method cost(int) is already defined in class PrintDesk
    double cost(int pages) {
           ^
1 error
```

Program 1.32: *Two methods that differ only in what they return.*

The pair of declarations	Overload?	Why
cost(int) and cost(int, int)	Yes	The number of parameters differs
cost(int, boolean) and cost(int, int)	Yes	The second parameter differs in type
cost(int, double) and cost(double, int)	Yes	The order of the types differs, although a call with two ints then matches both and is rejected as ambiguous
int cost(int) and double cost(int)	No	The parameter lists are identical, and the return type takes no part in the choice
cost(int pages) and cost(int sheets)	No	Only the parameter name differs, and names take no part in the choice

Table 1.10: *Five pairs of declarations, and what the compiler makes of each. Only the parameter list decides, and the last two pairs are the same method declared twice.*

One name, several parameter lists, and a compiler that has decided before the program runs.

The Master Key: *Overloads differ by parameter list; the return type is never part of the choice.*

BINDER Question 39

The library at Shelui keeps two numbers on one card: how many books this borrower has out, and how many borrowers there are. The second is the same whichever card you hold, and writing it on every card guarantees they will disagree.

Distinguish a static member from an instance member, stating where each one lives and how many copies of it exist. Explain why a static method can reach neither an instance field nor the keyword this, and state the two jobs this does.

Model Answer

A static member belongs to the class and exists once however many objects are made, an instance member belongs to an object and exists once for each object, and a static method, having no object, can reach neither an instance field nor this.

```
 1  public class Roll {
 2      public static void main(String[] args) {
 3          Student s1 = new Student();
 4          Student s2 = new Student();
 5          Student s3 = new Student();
 6          s1.enrol(60);
 7          s2.enrol(72);
 8          s3.enrol(85);
 9          System.out.println("total is " + Student.total());
10      }
11  }
12
13  class Student {
14      private static int count = 0;
15      private int mark;
16
17      void enrol(int mark) {
18          this.mark = mark;
19          count = count + 1;
20          print(this);
21      }
22
23      static void print(Student s) {
24          System.out.println("mark " + s.mark + ", count " + count);
25      }
26
27      static int total() {
28          return count;
29      }
30  }
```

Output

```
mark 60, count 1
mark 72, count 2
mark 85, count 3
total is 3
```

Program 1.33: *One count held by the class, three marks held by three objects.*

One copy, held by the class: a field declared static is created once, when the class is loaded, and every object reads and writes that one location. Reach it through the class name, as line 9 does, which says at the call that no object was needed to answer.

One copy for every object: a field declared without static is made afresh inside each object. The three Students on lines 3 to 5 hold three separate marks, and changing one leaves the other two as they were. That is why mark is not static and count is.

What a static method may touch: its own locals, its parameters, and the static fields of its class. It cannot name an instance field, because there is no object for that field to sit in; hand it an object, as line 20 does, and every field of that object is reachable through the parameter.

this, where a parameter hides a field: inside an instance method, this names the object the method was called on. The parameter mark on line 17 has the same name as the field, and inside enrol the parameter wins; this.mark on line 18 names the field, or the line would copy the parameter onto itself.

this, handed on as an argument: this is an ordinary reference and may be passed like any other. Line 20 gives the current object to a static method that could not otherwise reach a single field of it. A static method has no this to give away, because it was called on no object.

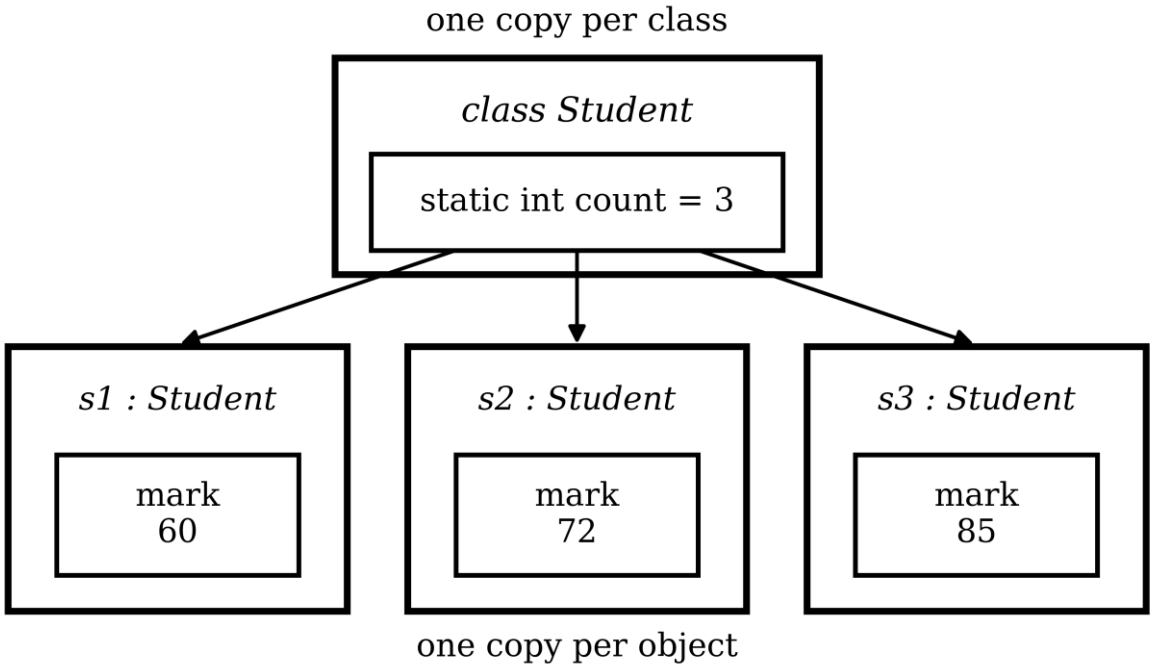


Figure 1.12: *One count held by the class against one mark held by each object. The static field is a single location whichever object reaches it; the three marks are three separate locations that happen to share a name.*

Ask where the value lives and how many copies there are, and static stops being a word added to quieten the compiler.

The Master Key: *A static member exists once for the class; an instance member exists once for every object.*

REAL Question 40

A student at Sepuka wrote a method to swap two numbers, ran it, and found them where he left them. He had written it in C++ the term before, and it worked. Nothing in his Java was wrong except one assumption carried across.

Java is pass-by-value throughout. Trace the listing below, state what each of its four printed lines proves, and explain why the object reached through the parameter changed while the caller's variable still names the object it named before.

Model Answer

Every argument in Java is copied: a primitive parameter is given a copy of the value and a reference parameter is given a copy of the reference, which is why a method can change the object it was handed and can never change which object the caller's variable names.

```
 1  public class Passing {
 2      public static void main(String[] args) {
 3          int x = 5;
 4          Student s = new Student();
 5          s.setMark(60);
 6          change(x, s);
 7          System.out.println("after: x is " + x);
 8          System.out.println("after: s mark is " + s.getMark());
 9      }
10
11      static void change(int p, Student t) {
12          p = 99;
13          t.setMark(99);
14          System.out.println("inside: p is " + p);
15          t = new Student();
16          t.setMark(75);
17          System.out.println("inside: t mark is " + t.getMark());
18      }
19  }
20
21  class Student {
22      private int mark;
23
24      void setMark(int mark) {
25          this.mark = mark;
26      }
27
28      int getMark() {
29          return mark;
30      }
31  }
```

Output

```
inside: p is 99
inside: t mark is 75
after: x is 5
after: s mark is 99
```

Program 1.34: *What a method can and cannot change about its caller.*

What the call on line 6 actually passes: two copies. p receives a copy of the value 5, and t receives a copy of the reference held in s, so for the moment t and s name one object between them. Nothing at all is copied of the object itself, which stays where it was made.

Reading the two lines printed inside: line 12 writes 99 into the copy p and line 14 prints it, so 99 is what the method sees. Line 13 sets the mark of the object t names to 99. Line 15 then aims t at a second object, line 16 gives that one 75, and 75 is what line 17 prints.

Reading the two lines printed after the call: line 7 prints 5, because the assignment on line 12 wrote into the copy and x was never told. Line 8 prints 99, because line 13 reached the caller's own object through the copied reference, and s still names that object, since line 15 moved t alone.

Where the C++ habit misleads: C++ offers a reference parameter, written int followed by an ampersand, which makes the parameter a second name for the caller's variable, so an assignment inside the method reaches back out. Java has no such form. A parameter is always a fresh variable, and the swap the candidate wrote could not have worked.

The two ways to hand a result back: return it, or change an object the caller can already reach. There is no third way, and a method that assigns to its own parameter is writing into a variable that vanishes the moment it returns.

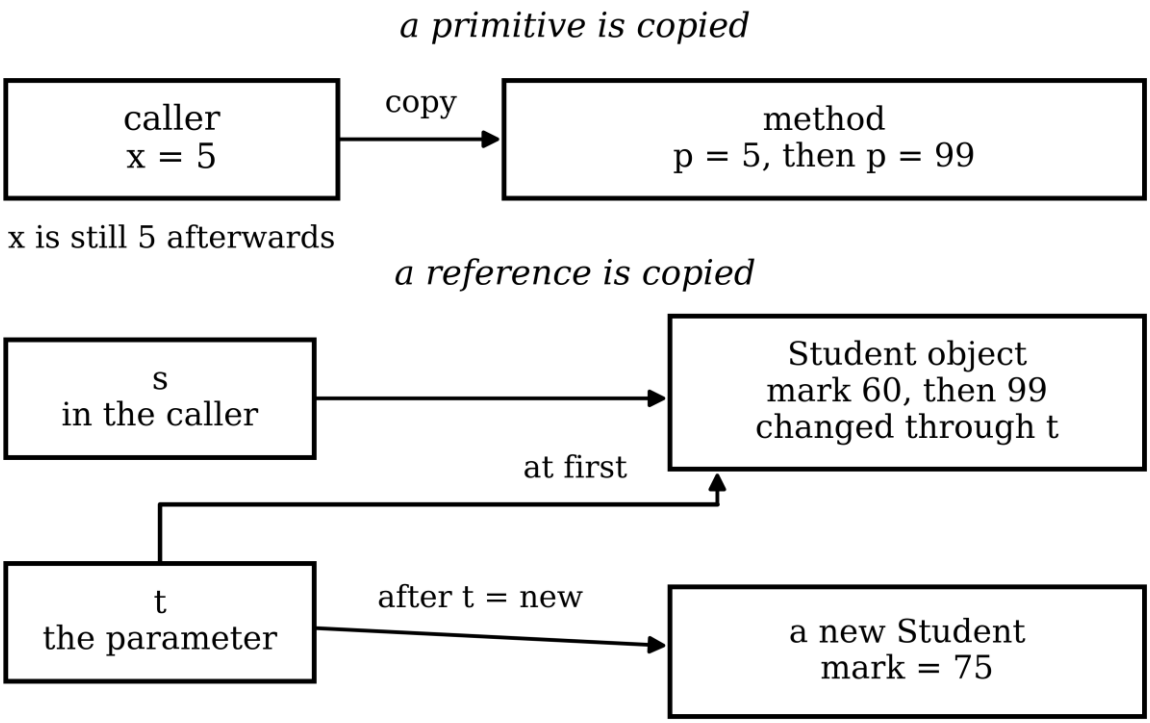


Figure 1.13: *A copied value above, a copied reference below. The method writes 99 into its own copy and x is still 5 afterwards; it changes the mark of the object both names, and the later assignment moves the parameter alone onto a new object.*

Java copies the argument every time, and when the argument is a reference the copy still names the same object.

The Master Key: *A method may change the object; it can never change which object the caller's variable names.*

REAL Question 41

The message named a variable declared four lines above it and said it could not be referenced. The student at Mtinko had typed the name correctly. What the compiler meant was that there was no store for the bags to be in.

A shop counter that will not compile. Name the fault, state exactly what javac reports and what it leaves behind, then give two different correct repairs and say which condition decides between them.

Model Answer

The fault is that main is static while bags is an instance field, so when main runs there is no Store object for bags to belong to, and javac rejects line 5 with the message that a non-static variable cannot be referenced from a static context.

```
1  public class Store {
2      private int bags = 40;
3
4      public static void main(String[] args) {
5          System.out.println("bags in store: " + bags);
6      }
7  }
```

The compiler says

```
Store.java:5: error: non-static variable bags cannot be
        referenced from a static context
        System.out.println("bags in store: " + bags);
                                              ^
1 error
```

Program 1.35: *A static method reaching for a field that belongs to an object.*

Naming the fault precisely: it is not that bags is private, and not that it is uninitialised, because line 2 initialises it. It is that an instance field exists only inside an object, that main runs before any Store has been made, and that line 5 names the field with no object for it to be a field of.

Exactly what javac does about it: it reports an error and not a warning. It names the file and the line, prints the offending line with a caret under bags, and counts one error at the end. It writes no class file whatever, so there is nothing to run: this program does not fail late, it never starts.

The first repair, and when it is right: write private static int bags. There is then one bags for the whole class and a static method may reach it directly. This is the right repair only where the value genuinely belongs to the class, as a price that is the same at every store does.

The second repair, and when it is right: make an object and go through it, which is what lines 5 and 6 of the corrected listing do. Every Store then carries its own bags, which is what a store is. This is the right repair whenever two objects of the class would hold different values.

The question that decides between them: ask whether two Stores would ever hold different numbers of bags. If they would, the field is per-object and the repair is to make an object; if they could not, the field belongs to the class and static states that. Adding static to silence the compiler is the common repair, and it quietly turns data that belongs to each object into one value shared by all of them.

```
1  public class Store {
2      private int bags = 40;
3
4      public static void main(String[] args) {
5          Store depot = new Store();
6          System.out.println("bags in store: " + depot.bags);
7      }
8  }
```

Output

```
bags in store: 40
```

Program 1.36: *The second repair, which gives the field an object to belong to.*

The compiler is not objecting to the name; it is objecting that there is nothing for the name to belong to.

The Master Key: *An instance field needs an instance; make a field static only when the value truly belongs to the class.*

BUILD Question 42

The Ndala cooperative writes one price at the head of the page, a row for every grower. When the price moves, one figure is crossed out and another written in. A ledger copying it onto every row needs fifty corrections and gets one wrong.

Write a complete Java program that records one lot of maize for the cooperative to the specification below, then say why the price is a static field and the number of bags is not.

The class is called Lot and is the only class in the file.

The price of one bag is held once for the whole class, and is 62000.

The grower and the number of bags are held once for each lot, and are private.

One method records a grower and a bag count into a lot, using this to separate each parameter from the field it is named after.

Two further methods return rather than print: one gives a line describing the lot, the other gives its value.

main reads the grower and the bag count from the keyboard, makes one Lot, and prints what the two methods return.

Model Answer

The specification asks for one value held by the class, two values held by each lot, a recording method that uses this, and two methods that return rather than print, and the program below satisfies it in one class.

```
 1  import java.util.Scanner;
 2
 3  public class Lot {
 4      private static int pricePerBag = 62000;
 5      private String grower;
 6      private int bags;
 7
 8      void record(String grower, int bags) {
 9          this.grower = grower;
10          this.bags = bags;
11      }
12
13      String line() {
14          return grower + " brought " + bags + " bags";
15      }
16
17      int value() {
18          return bags * pricePerBag;
19      }
20
21      public static void main(String[] args) {
22          Scanner in = new Scanner(System.in);
23          System.out.println("Grower?");
24          String name = in.next();
25          System.out.println("Bags?");
26          int bags = in.nextInt();
27          in.close();
28
29          Lot lot = new Lot();
30          lot.record(name, bags);
31          System.out.println(lot.line());
32          System.out.println("Value: " + lot.value());
33      }
34  }
```

Typed at the program

```
Juma
18
```

Output

```
Grower?
Bags?
Juma brought 18 bags
Value: 1116000
```

Program 1.37: *The maize lot, with one price for the class and its own bags.*

Where the specification is satisfied: line 4 holds the price once for the class, lines 5 and 6 hold the two values that differ from lot to lot, record on lines 8 to 11 uses this twice to separate a parameter from the field of the same name, and the methods line and value return their answers instead of printing them.

Why the price is static and the bags are not: two lots weighed on the same morning are bought at one price and hold different numbers of bags. A price stored inside every Lot would have to be corrected inside every Lot, and the first one missed would pay a grower the wrong money.

Why the two methods return: a method that returns its answer can be printed, added to a running total, or set beside another lot. A method that prints can only be printed. Returning costs nothing at lines 31 and 32 and leaves every other use of the answer open.

One price for the class, one row for each grower, and no figure written down twice.

The Master Key: *Static for what the class holds once, instance for what each object holds for itself.*

BUILD Question 43

The order book has one column for the garment and one for the price, reached three ways: the standard charge, the charge when the customer brings her own cloth, the charge for work wanted by Friday. The tailor calls all three the price.

Write a complete Java program that prices a tailoring order three ways to the specification below, then say what keeps the three methods from colliding.

The class is called Order and is the only class in the file.

It provides three methods, all named price, told apart only by their parameter lists.

price with one int charges the standard rate of 9000 for each garment.

price with an int and a boolean takes 3500 off each garment when the customer brought the cloth.

price with two ints adds the second as a fixed fee for rush work.

main reads the number of garments from the keyboard and prints all three prices.

Model Answer

The specification asks for one name used three times and told apart by its parameters, and the program below provides exactly that: price with one int, price with an int and a boolean, and price with two ints.

```
 1  import java.util.Scanner;
 2
 3  public class Order {
 4      private static final int RATE = 9000;
 5      private static final int CLOTH = 3500;
 6
 7      static int price(int items) {
 8          return items * RATE;
 9      }
10
11      static int price(int items, boolean ownCloth) {
12          if (ownCloth) {
13              return items * (RATE - CLOTH);
14          }
15          return items * RATE;
16      }
17
18      static int price(int items, int rushFee) {
19          return items * RATE + rushFee;
20      }
21
22      public static void main(String[] args) {
23          Scanner in = new Scanner(System.in);
24          System.out.println("How many garments?");
25          int items = in.nextInt();
26          in.close();
27
28          System.out.println("Standard: " + price(items));
29          System.out.println("Own cloth: " + price(items, true));
30          System.out.println("Rush: " + price(items, 12000));
31      }
32  }
```

Typed at the program

```
4
```

Output

```
How many garments?
Standard: 36000
Own cloth: 22000
Rush: 48000
```

Program 1.38: *Three prices from one name, chosen by the arguments.*

Where the specification is satisfied: the three declarations on lines 7, 11 and 18 share the name price and differ in their parameter lists, and the three calls on lines 28 to 30 are matched to them by the arguments alone. The second parameter is boolean in one and int in the other, and that single difference is enough.

What keeps the three apart: the compiler compares parameter lists, so price with one int is distinct from price with an int and a boolean and from price with two ints. Had the rush fee and the cloth flag both been declared int, the last two would have been one method written twice and javac would have refused the class.

Why these three methods are static: each answer follows from the arguments and the two constants on lines 4 and 5 and from nothing else. No Order object holds anything they would consult, so making one first would be ceremony. A method that would ignore the object it was called on has no business demanding one.

Three prices, one name, and a compiler that tells them apart by what it is handed.

The Master Key: *Overload where the work is the same and only the inputs differ.*

HOT Question 44

A class was told to write static in front of every method, on the ground that the program would not otherwise run. A program whose methods are all static will run. So will one written as a single enormous main, and nobody recommends that.

Evaluate the claim that a method should be declared static wherever the compiler will allow it. Reach a judgement.

Model Answer

Declare a method static when its answer depends only on its arguments and on what the class holds once, and leave it an instance method when its answer depends on the state of one object; the question that decides is whether the method would ever need to know which object it was called on.

Where the habit comes from: main must be static, because the launcher has no object to call it on. A static method cannot then call an instance method without an object, so the first helper is made static to quieten the compiler, and the second, and by the end the whole class is static and nothing has been decided.

The case for static: a method that computes from its arguments alone is honest as a static method. No object need be made to call it, the class name stands in front of the call and says that no state was consulted, and no field that another method changed can alter what it gives back.

The case against, which is the stronger one: an object exists to hold state and an instance method exists to act on it. Once every method is static, every value the program keeps must be static too, and a class meant to describe one lot of maize describes every lot at once. Two lots can then never differ, and the apparatus of an object oriented program, many things each remembering their own state, is out of reach.

The test to apply to one method: write it as though it were an instance method and ask whether the body ever mentions a field. If it does not, and never would, static says so and saves the reader a question. If it does, static is unavailable unless the field is made static too, and that is a decision about the data rather than about the method.

The verdict: reject the claim. Static is right for a computation over its arguments and for a value the class holds once, such as a rate or a count of the objects made; it is wrong wherever the answer belongs to one object among many. Prefer an instance method the moment a method would read or write per-object state, and keep static for where making an object first would be ceremony.

How to argue it in an answer: state the criterion, apply it to one method of each kind, and name what a wholly static program costs. An answer that reaches a criterion scores where one that repeats a rule does not.

Static is a statement about where the data lives, and it deserves to be made on that ground and no other.

The Master Key: *Static when the answer needs no object, instance when it needs this one.*

HOT Question 45

One name, used four times, in a class of eleven methods. Reading the calls afterwards told you nothing about which of the four had run: you had to know the declared type of every argument to know that, and the declarations were on another page.

Evaluate the claim that overloading one name is always better than giving the methods different names. Reach a judgement.

Model Answer

Overload where the methods do the same work on different inputs, and give them different names as soon as what they do differs, because the name is the only thing a reader sees at the call and an overloaded name says nothing about which method was chosen.

The case for overloading: it puts one idea under one name. println is overloaded for every printable type, and nobody has had to remember printlnInt beside printlnDouble. The caller supplies whatever he has and the compiler finds the method that fits, which removes a whole family of clumsy names before it can form.

The case against: a call shows a name and some arguments and nothing else. Where the name is used three times, a reader must know the declared type of every argument before he knows which body will run, and those declarations are usually elsewhere. A distinct name carries that information in the call itself.

Where overloading turns into a defect: cost with pages and cost with pages and a rate do the same arithmetic on different information, and overloading is right. cost with a page count and cost with a customer, where one charges and the other refunds, is a trap: the name promises one thing twice over and delivers two.

The quiet mis-selection: Java widens an argument to make it fit, so a short offered to a class that declares both a method taking int and a method taking long goes to the int, whatever the writer had in mind. The call compiles, runs, and prints a plausible number. Two different names cannot fail in this way.

How many is too many: two or three overloads that differ by one obvious parameter can be held in the head. Past that the reader is choosing from a menu he cannot see, and the compiler begins rejecting calls as ambiguous, which is at least honest but is not what anyone wanted.

The verdict: reject the claim as stated. Overload only where the methods agree in what they do and differ in what they are given, and where a reader meeting the call would not have to ask which one ran. As soon as two methods behave differently, name them differently: the compiler will accept the overload, and the next reader will not thank you for it.

How to argue it in an answer: state the condition, give one example on each side of it, and say plainly what the reader loses. A judgement with a condition attached is worth more than a preference stated twice.

The compiler is content either way, which is exactly why the choice has to be made for the reader.

The Master Key: *One name where the work is one idea; two names where the behaviour is two.*

Classes, objects, constructors and object lifetime

This section takes a class apart and makes an object out of it. It separates the class, which describes, from the object, which is one thing described, follows what new actually does with the heap and with the constructor, and then settles the constructor rules that the common teaching gets wrong more often than it gets right: the return type that turns a constructor into an ordinary method, the compiler-supplied constructor that exists only while you declare none of your own, and the chaining that keeps the work of setting the fields in one place. It closes on the life of an object after the last reference to it is dropped, and on why a language that collects its own memory has no destructor at all.

BINDER Question 46

The enrolment form at Ulyankulu Secondary School is one printed sheet and a filled copy for every student. The blank sheet carries no names, only the headings, the spaces, and the rule that every copy holds the same columns in the same order.

Distinguish a class from an object. Explain what new does, what the variable on the left of it receives, and what the dot operator does. Show that several objects of one class each carry their own field values.

Model Answer

A class is a description of state and behaviour, an object is one thing made to that description, and new is what makes it: it takes memory from the heap, runs the constructor, and returns a reference into the variable on the left.

What the class declares: the fields every object of it will carry, which is its state, and the methods every object will answer to, which is its behaviour. Declaring a class creates nothing: it introduces a type, and until new is called no memory anywhere holds a name or a mark.

```
 1  public class Student {
 2      private String name;
 3      private int mark;
 4
 5      public Student(String name, int mark) {
 6          this.name = name;
 7          this.mark = mark;
 8      }
 9
10      public void report() {
11          System.out.println(name + " scored " + mark);
12      }
13
14      public static void main(String[] args) {
15          Student first = new Student("Neema", 60);
16          Student second = new Student("Juma", 48);
17          Student third = new Student("Asha", 71);
18
19          first.report();
20          second.report();
21          third.report();
22      }
23  }
```

Output

```
Neema scored 60
Juma scored 48
Asha scored 71
```

Program 1.39: *One class, three objects, and three sets of values that do not touch one another.*

What new does: it takes memory from the heap large enough for one Student, runs the constructor over that memory, and hands back a reference to what it made. The three calls on lines 15 to 17 take three separate blocks, so there are three objects rather than one shared between three names.

The variable holds a reference: first does not hold a Student. It holds a reference to a Student on the heap, and that is true of every variable of class type in Java. C++ let a variable be the object itself; Java does not, which is why an object here is always reached through a name and never copied by an assignment.

What the dot does: it follows the reference to the object and names one member of it. On line 19 the dotted expression runs report on the object first refers to, and the values printed are the ones stored in that object. Point the variable elsewhere and the same expression reaches elsewhere.

One class, three sets of values: each object carries its own name and its own mark, so no call can print another candidate. That is what makes a class worth declaring: the fields and the methods are written once, and held separately as often as you call new.

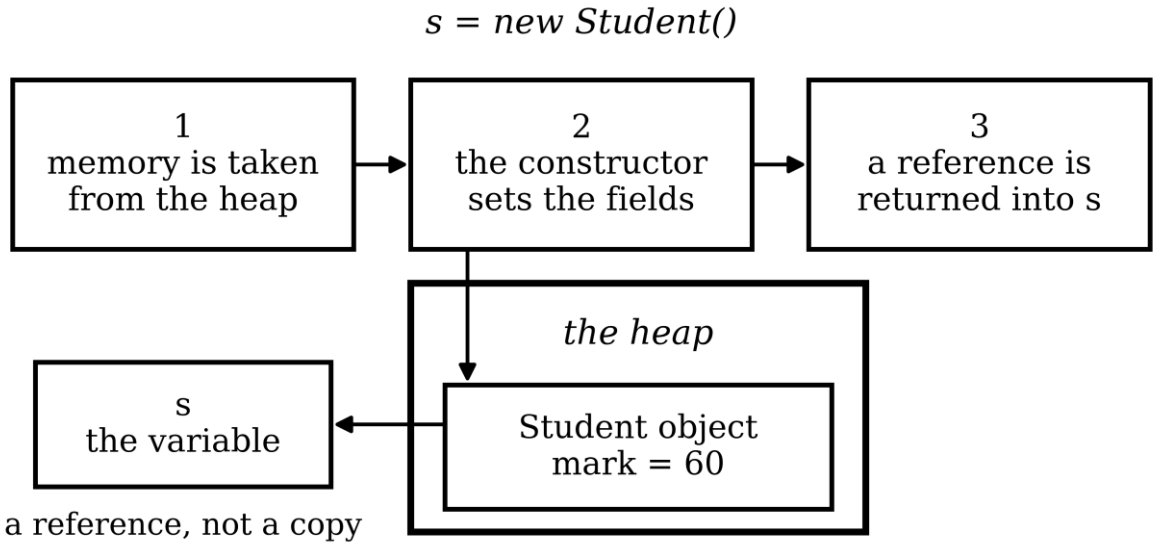


Figure 1.14: *What new does, in the order it does it: memory is taken from the heap, the constructor sets the fields, and a reference is returned into the variable. The arrow into the variable starts at the object, because a reference is what the variable receives and the object itself is what it never receives.*

Keep the description and the thing described apart, and the rest of this section follows.

The Master Key: *A class describes, new makes, the variable holds a reference, and the dot follows it.*

BINDER Question 47

Four lines of output, and nobody had written any of the values in them: null, 0, 0.0 and false. The stock card program at the Namanyere dispensary declared four fields, declared no constructor, and printed those four the moment it ran. Somebody had put them there.

State the rules a constructor must obey. Explain what the compiler supplies when a class declares no constructor, and what that supplied constructor does to the fields. Say what happens to a constructor that is given a return type.

Model Answer

A constructor carries the same name as the class and has no return type at all; the compiler supplies one only while you declare none, and the one it supplies sets every field to the default for its type.

The two rules that identify a constructor: its name is the name of the class, character for character, and it declares no return type. Not int, and not void either. It may take parameters and it may hold as many statements as you like; what it may not do is announce a type it returns, because it is not called to give a value back.

```
 1  public class StockCard {
 2      private String drug;
 3      private int packs;
 4      private double price;
 5      private boolean expired;
 6
 7      public void show() {
 8          System.out.println("drug: " + drug);
 9          System.out.println("packs: " + packs);
10          System.out.println("price: " + price);
11          System.out.println("expired: " + expired);
12      }
13
14      public static void main(String[] args) {
15          StockCard card = new StockCard();
16          card.show();
17      }
18  }
```

Output

```
drug: null
packs: 0
price: 0.0
expired: false
```

Program 1.40: *A class with no constructor of its own, printing what its four fields hold before anything has assigned them.*

What the compiler supplies, and what it does: declare no constructor and the compiler adds a no-argument one. Every field is set to the default for its type before its body runs, so the four fields hold null, 0, 0.0 and false, and the run prints them. Students are often taught, wrongly, that such a field is not initialised; it is, and Table 1.11 gives the value.

Field type	**Value it holds before any constructor body runs**	**As the run above prints it**
byte, short, int, long	0	packs: 0
float, double	0.0	price: 0.0
boolean	false	expired: false
char	the character whose code is zero	not printed here
any reference type, String among them	null	drug: null

Table 1.11: *The default value for every kind of field. These are values that are there, not the absence of a value, and a field of class type holding null holds a reference to nothing rather than no reference at all.*

```
 1  public class StockCard {
 2      private int packs;
 3
 4      public void StockCard() {
 5          packs = 40;
 6      }
 7
 8      public void show() {
 9          System.out.println("packs: " + packs);
10      }
11
12      public static void main(String[] args) {
13          StockCard card = new StockCard();
14          card.show();
15      }
16  }
```

Output

```
packs: 0
```

Program 1.41: *A return type turns a constructor into an ordinary method, and the compiler raises nothing at all.*

The return type that costs you the constructor: write void in front of the name and you have declared a method that happens to share the name of its class. It is legal Java, it compiles without a warning, and new never calls it. The run prints 0 rather than 40, which is the only evidence you will get that the line is not a constructor.

Declaring one takes the supplied one away: the compiler adds its constructor only to a class that declares none. Declare any constructor, of any shape, and the one you never wrote is gone, and code that was calling it stops compiling. The common teaching states that rule and then contradicts it on the next page; Question 51 puts it in front of a compiler.

The name and the missing return type make a constructor, and the compiler writes one only until you write your own.

The Master Key: *No return type, or it is a method; and the supplied constructor sets every field to its type default.*

BINDER Question 48

Every student enrolled through one constructor at Matai came out wrong; the other two were faultless. All three held the same pair of assignments, copied three times, and one had been edited in a hurry. The repair was to stop keeping three copies.

Explain constructor overloading and explain what this followed by arguments does inside a constructor. State where in the constructor it must appear, and say why the work of assigning the fields belongs in the most specific constructor of a set.

Model Answer

Overloading a constructor means declaring several that differ in their parameter lists, and this followed by arguments calls one constructor of the class from another, so that only one of them ever assigns a field.

```
 1  public class Student {
 2      private int mark;
 3      private int form;
 4
 5      public Student() {
 6          this(0);
 7      }
 8
 9      public Student(int mark) {
10          this(mark, 1);
11      }
12
13      public Student(int mark, int form) {
14          this.mark = mark;
15          this.form = form;
16      }
17
18      public void report() {
19          System.out.println("form " + form + ", mark " + mark);
20      }
21
22      public static void main(String[] args) {
23          new Student().report();
24          new Student(60).report();
25          new Student(72, 5).report();
26      }
27  }
```

Output

```
form 1, mark 0
form 1, mark 60
form 5, mark 72
```

Program 1.42: *Three constructors, one chain, and one place where the fields are assigned.*

How the compiler tells them apart: by the number and the types of the parameters, which is the rule that already governs overloaded methods. Two constructors that differ only in the names of their parameters are the same constructor twice, and the class does not compile. Here the three differ by taking no arguments, one, and two.

What the chaining call does: this followed by arguments calls another constructor of the same class. Line 5 declares the no-argument form and immediately forwards to the one that takes a mark, which forwards to the one on line 13 that takes both values. Control comes back down the chain afterwards, so anything written after the call still runs.

It must be the first statement: put any statement above it and the class does not compile, and the message says that the call must be the first statement in the constructor. The reason is that the fields are to be set once, in one known order, before any code can read them. A chain that leads back to where it started is refused as well, as a recursive constructor invocation.

Where the work belongs: in the constructor with the most parameters, because it is the only one that has been given everything. The shorter ones assign nothing; each supplies one default and forwards. A default then appears exactly once in the class, and a change to how a field is set is a change to one line rather than to three.

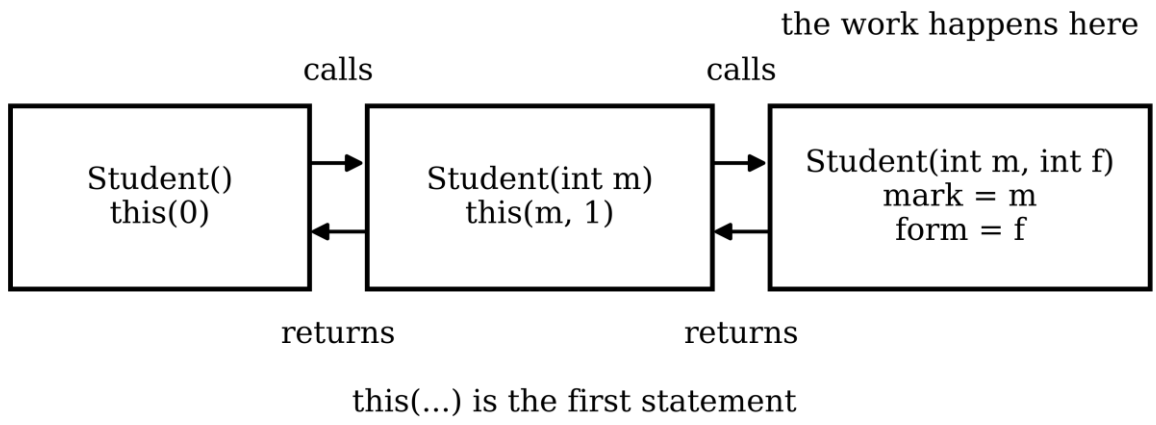


Figure 1.15: *The chain drawn twice: forwards, as each constructor supplies a default and calls the next, and back again, as control returns along it. Only the constructor at the end of the chain assigns the fields.*

Several ways in, one place where the work is done, and the defaults written down once.

The Master Key: *Chain to the most specific constructor, and let it be the only one that assigns a field.*

REAL Question 49

The compiler printed three words and then a name: cannot find symbol, myObj. The object had been made correctly on the line above, and the class was spelled the same on both sides. Nothing in the program was called myObj.

An object is created here and then read again under a different name. Name the fault, state which symbol javac says it cannot find and what that tells you, and correct it.

Model Answer

The object is created under the name myApp on line 5 and then read under the name myObj on line 6, so the compiler reports cannot find symbol against myObj and writes no class file at all.

```
1  public class Application {
2      private int count = 12;
3
4      public static void main(String[] args) {
5          Application myApp = new Application();
6          System.out.println(myObj.count);
7      }
8  }
```

The compiler says

```
Application.java:6: error: cannot find symbol
        System.out.println(myObj.count);
                           ^
  symbol:   variable myObj
  location: class Application
1 error
```

Program 1.43: *An object made under one name and read under another.*

Reading what the compiler printed: three lines of it do the work. The first names the file and the line. The caret sits under the offending name. And the symbol line names the thing it could not find, a variable called myObj, while the location line says where it looked, which was the class Application. No guessing is needed anywhere.

Why the object is not the fault: line 5 is correct in every respect. It creates an Application and puts a reference to it in myApp. The fault is entirely on line 6, where a name that was never declared is used, and the two lines together are the ordinary shape of this mistake: a variable renamed in one place and not the other.

Nothing runs at all: this is a compile-time failure, so there is no class file, and a program that produced no class file does not run. The object on line 5 is therefore never made either. Say that the program fails to compile, not that it prints something wrong; the difference is worth a mark on its own.

Why the compiler cannot repair it for you: Java has no implicit declaration. Every variable must be declared before it is used, and myObj is declared nowhere, so there is no rule by which a compiler could decide that a name it has never met was meant to be a different name that it has. It reports what it knows and stops.

The correction, and the habit: read the field through the name that holds the reference: myApp.count. More useful than the fix is the habit behind it. The message cannot find symbol always prints the symbol it could not find, so searching your source for that exact spelling lands you on the line before you have read anything else.

The message named the missing name, and the missing name was the whole fault.

The Master Key: *Cannot find symbol names the symbol; the object exists under the name you gave it, not the one you typed.*

REAL Question 50

The fleet office at Kasanga keeps one card per bus, the make on one line, the model on the next. The program replacing them declared both fields, assigned neither, and was refused twice. The two names it printed appear nowhere in the class.

A parameterised constructor that assigns fields the class never declares. Name the fault, state exactly what javac reports and how many times it reports it, and correct it.

Model Answer

The class declares fields called brand and model, while the constructor assigns to brandName and to CarModel, neither of which is a field of the class, so the compiler reports cannot find symbol twice, once for each line.

```
 1  public class Car {
 2      private String brand;
 3      private String model;
 4
 5      public Car(String brand, String model) {
 6          this.brandName = brand;
 7          this.CarModel = model;
 8      }
 9
10      public void show() {
11          System.out.println(brand + " " + model);
12      }
13
14      public static void main(String[] args) {
15          Car bus = new Car("Toyota", "Coaster");
16          bus.show();
17      }
18  }
```

The compiler says

```
Car.java:6: error: cannot find symbol
        this.brandName = brand;
            ^
  symbol: variable brandName
Car.java:7: error: cannot find symbol
        this.CarModel = model;
            ^
  symbol: variable CarModel
2 errors
```

Program 1.44: *A constructor assigning to two fields the class never declared.*

Reading which name was printed: there are two errors and the count at the foot says so. Each one names the symbol it could not find, brandName on line 6 and CarModel on line 7, and neither of those names appears on lines 2 and 3 where the fields are declared. The message has already done the comparing for you.

What the qualified form does to the diagnosis: writing this in front of the name says that a field of this object was meant and nothing else will do. There is no field of that name, and the compiler has nowhere else to look, so it reports a missing variable rather than quietly resolving the name to a local or a parameter. The mistake is caught at the line that made it.

Capital letters are part of the name: CarModel differs from model by a capital letter and three characters in front of it, and either difference alone would be enough. Java compares identifiers character for character. A field called model and a name written Model are two different names, and the compiler will not bring them together.

The correction: assign the two fields the class actually declares, so that the constructor sets brand from its first parameter and model from its second. The name on the right of the dot must be one of the names on lines 2 and 3, not the longer name a reader might have expected the field to carry.

Why nothing warns you about an unassigned field: no rule requires a constructor to assign anything. Had this constructor been left empty, the class would have compiled cleanly and every Car would have carried null in both fields, printing null null when shown. The compiler objects to a name it cannot find, never to work you did not do.

Two errors, two names, and both names given to you before you looked at a single line.

The Master Key: *A field is assigned by the name it was declared under, and the compiler prints the name it could not find.*

REAL Question 51

One line was added to a class that had compiled all morning: a spare Car, made with no arguments, to be filled later. Nothing else was touched. It did not compile that afternoon, and the constructor that line called had never been written.

The listing below does not compile. Name the fault, state exactly what the compiler does about it, and give two corrections, saying which of the two you would prefer and on what condition.

Model Answer

Car declares a constructor of its own, so the compiler supplies none, and line 15 calls a no-argument constructor that does not exist.

```
 1  public class Car {
 2      private String brand;
 3      private String model;
 4
 5      public Car(String brand, String model) {
 6          this.brand = brand;
 7          this.model = model;
 8      }
 9
10      public void show() {
11          System.out.println(brand + " " + model);
12      }
13
14      public static void main(String[] args) {
15          Car spare = new Car();
16          spare.show();
17      }
18  }
```

The compiler says

```
Car.java:15: error: constructor Car in class Car cannot be
        applied to given types;
        Car spare = new Car();
                    ^
  required: String,String
  found:    no arguments
  reason: actual and formal argument lists differ in length
1 error
```

Program 1.45: *A class with one parameterised constructor, instantiated as though it still had the supplied one.*

Reading what the compiler printed: the first line names the class and says the constructor cannot be applied to the arguments given. The three lines beneath it are the comparison: required, two Strings; found, no arguments; and the reason, that the two lists differ in length. The message is about arity, not about a missing class.

The rule that produced it: the compiler supplies a no-argument constructor only to a class that declares no constructor at all. Line 5 declares one, so the supplied one is gone. There is no warning when it goes, no option to keep both, and nothing in the class looks different afterwards.

Why the language works this way: a class whose constructor demands a brand and a model is saying that it will not make a Car without them. If the supplied constructor stayed behind, that requirement could be stepped around by leaving the arguments out, and the object would come into existence holding null twice.

The first correction: pass the arguments the constructor asks for, a brand and a model, so that line 15 makes the Car the class says it will make. This is the correction to prefer whenever the caller genuinely knows the two values, because it keeps one way into the class and one place where the fields are set.

The second correction: declare the no-argument constructor yourself and chain it to the one that does the work, giving the two fields whatever a blank card should hold. Prefer this only where an object made that way is genuinely useful; a spare Car carrying the word unknown in both fields is a record of nothing at all.

The contradiction to watch for: students are often taught the suppression rule correctly and then shown, a page later, a listing that creates an object with no arguments from a class whose only constructor takes some. Both cannot be right, and it is not the rule that is wrong. Put the listing in front of a compiler and it says so in one line.

Declare a constructor and you own every way into the class, including the way you had been getting for nothing.

The Master Key: *The compiler supplies a no-argument constructor only while you declare none.*

BUILD Question 52

Fifty shillings a page is the rate at the Bukene printing shop, except when it is not. Long jobs are argued down across the counter, and the price settled before a page is counted. The till must price both kinds of job.

Write a complete Java program that prices two printing jobs, one charged at the standard rate and one at a rate agreed across the counter, to the specification below.

The class is called PrintJob and is the only class in the file.

Each job holds the number of pages and the price of one page in shillings.

The class declares two constructors: one taking pages and a price, and one taking pages alone, which supplies the standard rate of 50.

The shorter constructor assigns no field itself.

The program reads three numbers: the pages of the standard job, then the pages and the agreed rate of the second job.

Each job prints one line giving its pages, its rate and the amount due.

Model Answer

The specification asks for two constructors of which only one assigns anything, and the program below satisfies it in thirty-three lines.

```
 1  import java.util.Scanner;
 2
 3  public class PrintJob {
 4      private int pages;
 5      private int shillingsPerPage;
 6
 7      public PrintJob(int pages) {
 8          this(pages, 50);
 9      }
10
11      public PrintJob(int pages, int shillingsPerPage) {
12          this.pages = pages;
13          this.shillingsPerPage = shillingsPerPage;
14      }
15
16      public void receipt() {
17          System.out.println(pages + " pages at " + shillingsPerPage
18                  + " comes to " + pages * shillingsPerPage);
19      }
20
21      public static void main(String[] args) {
22          Scanner in = new Scanner(System.in);
23          System.out.println("Pages at the standard rate:");
24          int standard = in.nextInt();
25          System.out.println("Pages at an agreed rate, then the rate:");
26          int agreed = in.nextInt();
27          int rate = in.nextInt();
28          in.close();
29
30          new PrintJob(standard).receipt();
31          new PrintJob(agreed, rate).receipt();
32      }
33  }
```

Typed at the program

```
120
400
35
```

Output

```
Pages at the standard rate:
Pages at an agreed rate, then the rate:
120 pages at 50 comes to 6000
400 pages at 35 comes to 14000
```

Program 1.46: *The printing shop till, compiled with javac PrintJob.java and run with java PrintJob.*

Where the specification is satisfied: the two fields are on lines 4 and 5, the shorter constructor is on lines 7 to 9 and assigns nothing, and the constructor on lines 11 to 14 is the only place a field is set. The two jobs on lines 30 and 31 are made one through each constructor, and each prints its own line.

The one design decision worth naming: the standard rate appears once, on line 8, inside the constructor that supplies it. When the shop changes its rate one line changes, no job priced at an agreed rate is disturbed, and no caller has to remember what the standard rate is in order to charge it.

Reading and closing the input: the three numbers are read in the order the specification gives them and the Scanner is closed on line 28, before any object is made. Nothing is read after that, so closing it early costs nothing and cannot be forgotten later.

Two ways to make a job, one place where a job is actually built, and the standard rate written down once.

The Master Key: *Put the default in the shorter constructor and the assignment in the longer one.*

BUILD Question 53

Each stub at Inyonga carries a fare and the passenger's name. The takings are written at the foot of the sheet when the last bus has gone. A stub that carried the running total would be wrong the moment the next passenger paid.

Write a complete Java program that records three ticket sales as three objects of one class and reports the takings for the three, to the specification below.

The class is called Ticket and is the only class in the file.

Each ticket holds the name of the passenger and the fare that passenger paid.

The program reads three passenger names, each followed by the fare paid.

It makes one Ticket object for each sale and keeps the three together.

It prints one line for each ticket, then the takings for the three.

Model Answer

The specification asks for three objects of one class, each holding its own two values, and one total that belongs to none of them, and the program below satisfies it in thirty-four lines.

```
 1  import java.util.Scanner;
 2
 3  public class Ticket {
 4      private String passenger;
 5      private int fare;
 6
 7      public Ticket(String passenger, int fare) {
 8          this.passenger = passenger;
 9          this.fare = fare;
10      }
11
12      public void print() {
13          System.out.println(passenger + " pays " + fare);
14      }
15
16      public static void main(String[] args) {
17          Scanner in = new Scanner(System.in);
18          Ticket[] sold = new Ticket[3];
19          int takings = 0;
20
21          System.out.println("Enter 3 passengers and fares:");
22          for (int i = 0; i < sold.length; i++) {
23              String passenger = in.next();
24              int fare = in.nextInt();
25              sold[i] = new Ticket(passenger, fare);
26              takings = takings + fare;
27          }
28          in.close();
29          for (Ticket sale : sold) {
30              sale.print();
31          }
32          System.out.println("Takings: " + takings);
33      }
34  }
```

Typed at the program

```
Neema 2500
Juma 2500
Asha 1800
```

Output

```
Enter 3 passengers and fares:
Neema pays 2500
Juma pays 2500
Asha pays 1800
Takings: 6800
```

Program 1.47: *The ticket desk, compiled with javac Ticket.java and run with java Ticket.*

Where the specification is satisfied: line 18 makes room for three tickets, line 25 makes one Ticket for each sale read, and the loop at the end asks each of the three to print itself. Two passengers paid the same fare and still hold it separately, because the fields belong to the object and not to the class.

What line 18 actually creates: an array of three references, all of them null until line 25 puts a reference in each. It does not create three Tickets; new Ticket on line 25 does that, three times, and what is stored in the array is three references to the three objects on the heap.

The one design decision worth naming: the takings are added up on line 26 and kept outside every ticket. A total is a fact about the morning, not about any one sale, and putting it in a field of Ticket would give three objects three answers to a question that has one.

Three objects, three sets of values, and the one number that belongs to none of them kept where it belongs.

The Master Key: *Give each object what is true of it alone, and keep what is true of the whole outside.*

HOT Question 54

The line never appeared. A student had written a finaliser that printed one word, made an object, dropped the only reference to it and waited. The program reached its last statement and stopped, having printed nothing, and there was nothing wrong with the machine.

Java has no destructor. Evaluate whether replacing it with a garbage collector is a gain or a loss, and reach a judgement. State what a program can and cannot know about the moment an object is reclaimed.

Model Answer

It is a clear gain where the resource is memory and a real loss everywhere else, and the loss is repaired not by a finaliser, which may never run at all, but by closing what you opened at a moment your own code chooses.

What a destructor gave C++: a member function that runs at a moment you can point to, either when the object goes out of scope or when delete is called on it. Because that is known, the destructor can be trusted with anything, memory, an open file, a lock, and the release is written once inside the class.

What Java put in its place: an object lives for as long as it is reachable. When the last reference to it is dropped it becomes unreachable, and unreachable objects become eligible for collection. At some later moment the collector reclaims the memory. Reachability decides, not scope, and the two are not the same.

```
 1  public class Ledger {
 2      private String village;
 3
 4      public Ledger(String village) {
 5          this.village = village;
 6      }
 7
 8      public void show() {
 9          System.out.println("ledger for " + village);
10      }
11
12      public static void main(String[] args) {
13          Ledger book = new Ledger("Inyonga");
14          book.show();
15
16          book = null;
17          System.out.println("the last reference is gone");
18
19          System.gc();
20          System.out.println("collection was asked for, not commanded");
21      }
22  }
```

Output

```
ledger for Inyonga
the last reference is gone
collection was asked for, not commanded
```

Program 1.48: *Reachability under the control of the program, and collection under nobody's.*

The moment is not the program's to choose: line 16 makes the object unreachable and the run says so, because that much the program controls. What follows does not: the call to gc is a request that a collection be considered,

not an instruction, and the machine is free to do nothing. Every line printed comes from main, because a program that waits to be told by the collector may wait for ever.

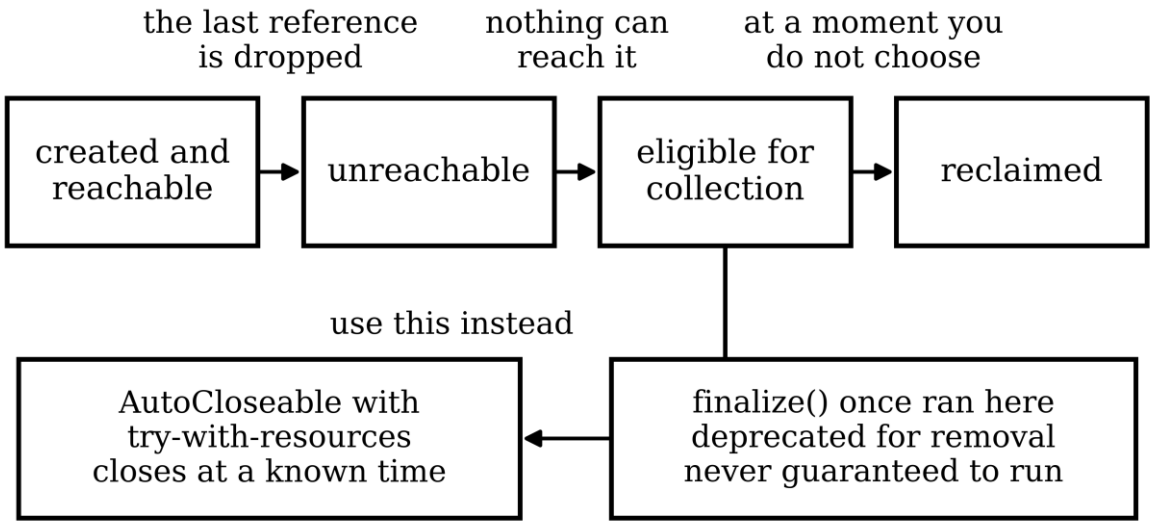


Figure 1.16: *The life of an object after the last reference to it goes. It becomes unreachable, then eligible for collection, and is reclaimed at a moment the program does not choose. The finaliser sits off that path, with what replaced it alongside.*

The finaliser, and why it is gone: finalize was deprecated in Java 9 and deprecated for removal in Java 18, and even before that it was never guaranteed to run at all: a program that ends before a collection happens ends with the finaliser unrun. Something that might never happen can be relied on for nothing, which is why closing a file was never a job for a finaliser. Do not write one.

What replaces it: AutoCloseable, used with try-with-resources, which closes what it opened at the end of the block whatever happens inside. That construct belongs to a later section, on exceptions, and this is the only mention it gets here. What matters now is that the moment of release is chosen by your code and not by the collector.

The question	C++ destructor	Java garbage collector
When does it run?	At a known moment: scope exit, or delete	At a moment the runtime chooses
What if you forget it?	The memory is never given back	There is nothing to forget
What if it happens twice?	Undefined behaviour, and often a crash	It cannot, because the program does not do it
What frees a file or a lock?	The destructor, reliably	An explicit close, written by you

Table 1.12: *The same questions put to both languages. Java answers the memory questions better and the resource question worse, and that is the whole shape of the argument.*

The verdict: a gain, on one condition: stop expecting the collector to release anything except memory. Where the resource is memory, hand it over, and the whole family of double frees, dangling references and forgotten deletes leaves your program at a stroke. Where it is a file, a socket or a lock, close it yourself and never through a finaliser. Where a deadline is measured in microseconds, a pause you did not schedule is a real reason to write C++ instead.

Java took the memory question away and left the resource question exactly where it was.

The Master Key: *No destructor, no guaranteed finaliser, and collection at a moment you cannot name; close resources yourself.*

HOT Question 55

Two versions of one class, two students, one teacher. The first declared one constructor and made every caller supply every value; the second declared four, one of them left behind when a field was renamed. Both compiled; only one could be trusted a year later.

A class may be given one constructor that takes every value, or several overloaded constructors that supply defaults. Evaluate the two designs and reach a judgement.

Model Answer

Declare one constructor that does the work, let every other constructor reach it through a chaining call, and add a shorter constructor only where its default is true of the class rather than convenient for one caller.

The case for a single constructor: every field is assigned in one place, so there is nothing to keep in step and no way for two creations of the same class to leave objects in different states. A reader who wants to know what a new object holds reads one method. Nothing can be added to the class later that quietly bypasses it.

What the single constructor costs: it charges every caller for values it may have no opinion about. A till that always charges the standard rate must still name that rate at every call, and a rate named in twenty places is a rate that will be changed in nineteen. The default has to live somewhere, and the callers are the one place it should not.

The case for several constructors: the common way of making an object becomes short and reads as what it is, and the defaults are stated once inside the class where a maintainer will look for them. The class also documents itself: the parameter lists show which combinations of values the class is prepared to be created from.

The failure that several constructors invite: duplication, which is the fault in the second candidate's class. Four constructors that each assign the fields are four places to edit and nothing to tell you that you edited three. Chaining removes the risk completely, because the shorter constructors then assign nothing and have nothing to fall out of step with.

The test to apply to every default: ask whether it is a fact about the class or a convenience for one caller. A printing job with no agreed rate really is charged the standard rate, so that default is a fact about the shop. A bus whose brand defaults to the word unknown is a fact about nothing; it is a way of avoiding a decision, and it puts an object into the program that no part of the program can use.

The verdict: one constructor that assigns, others that forward to it, and a shorter form only where the default it supplies would be right for every caller who omits the value. On that condition several constructors are better than one, because they put the defaults in the class. Without it, insist on the single constructor and make the caller decide.

Judge a constructor set by how many of them assign a field, and the answer should always be one.

The Master Key: *Many ways in, one way through: chain to the constructor that does the work.*

Encapsulation, access control and packages

This section settles what encapsulation is, what abstraction is, and why the two are not one idea written out twice. It puts packages and the import statement in place first, so that the phrase the same package means something definite by the time the access levels need it, and then gives all four levels with the exact reach of each, including the level you get by writing no modifier at all. It builds two classes whose fields no outsider can damage, one by checking every change and one by permitting none, and it closes on the argument about whether a getter and a setter for every field is encapsulation or only the look of it.

BINDER Question 56

The water kiosk at Mlowo has a meter with one dial and one lever, and nobody drawing water knows how it counts. The attendant's ledger lies open on the table for anybody passing to write in. Only one still agrees with the water.

Define encapsulation and define abstraction, state exactly how the two differ, and point to each of them in the program below.

Model Answer

Encapsulation is bundling data together with the methods that act on it and controlling access to that data, abstraction is exposing only what a user of the class needs and hiding how it is done, and the relation between them is that encapsulation is the mechanism while abstraction is the result.

What encapsulation bundles: the class declaration puts the fields and the methods that work on them in one place, and the modifier on each member decides who may reach it. Both halves are required. A class whose fields are all public has bundled its data and controlled nothing, and it is not encapsulated.

What abstraction exposes: the surface a caller is given, which is the set of public method names together with what each takes and what each returns. In Program 1.49 that surface is serve, getLitresSold and takings. Nothing outside the class learns whether the takings are stored or worked out on demand.

Why one is the mechanism and the other the result: you write private and public, and that act on each member is the encapsulation. What comes back is a class that outsiders can depend on only through its surface, and that is the abstraction.

The wording to avoid: a common error is to define encapsulation as data hiding and abstraction as hiding the implementation, which is one sentence twice over and separates nothing. Say instead what each one does: encapsulation controls reach, abstraction settles what the surface says.

```
 1  public class WaterMeter {
 2      private int litresSold;
 3      private final int pricePerLitre;
 4
 5      public WaterMeter(int pricePerLitre) {
 6          this.pricePerLitre = pricePerLitre;
 7      }
 8
 9      public void serve(int litres) {
10          litresSold = litresSold + litres;
11      }
12
13      public int getLitresSold() {
14          return litresSold;
15      }
16
17      public int takings() {
18          return litresSold * pricePerLitre;
19      }
20  }
21
22  class KioskDay {
23      public static void main(String[] args) {
24          WaterMeter meter = new WaterMeter(25);
25          meter.serve(40);
26          meter.serve(60);
27          System.out.println("Litres sold: " + meter.getLitresSold());
28          System.out.println("Takings: " + meter.takings());
29      }
30  }
```

Output

```
Litres sold: 100
Takings: 2500
```

Program 1.49: *A meter that adds to its own count, and a second class that can reach it only through the three methods it publishes.*

Reading the program: litresSold on line 2 is private, so the only way to change it is serve on line 9, which can only add. KioskDay on line 22 is a separate class and reaches the meter through three public methods, and it could not touch the field if it tried.

Bundle the data with the methods, settle who may reach it, and the surface that is left over is the abstraction.

The Master Key: *Encapsulation is the mechanism, which is access control; abstraction is the result, which is a surface a caller can depend on.*

BINDER Question 57

A Form Five class at Tukuyu saved sixteen programs into one folder over a single term. By the third week there were two files called Student.java, and the second had replaced the first. Java has a construct that stops that happening.

Explain what a package is and where it is declared, state what it does to the name of a class, and explain what an import statement does and what it does not do.

Model Answer

A package is a namespace that corresponds to a directory, the package declaration is the first statement in the file, the package becomes part of the class's real name, and an import copies no code at all; it merely lets a type be written without its prefix.

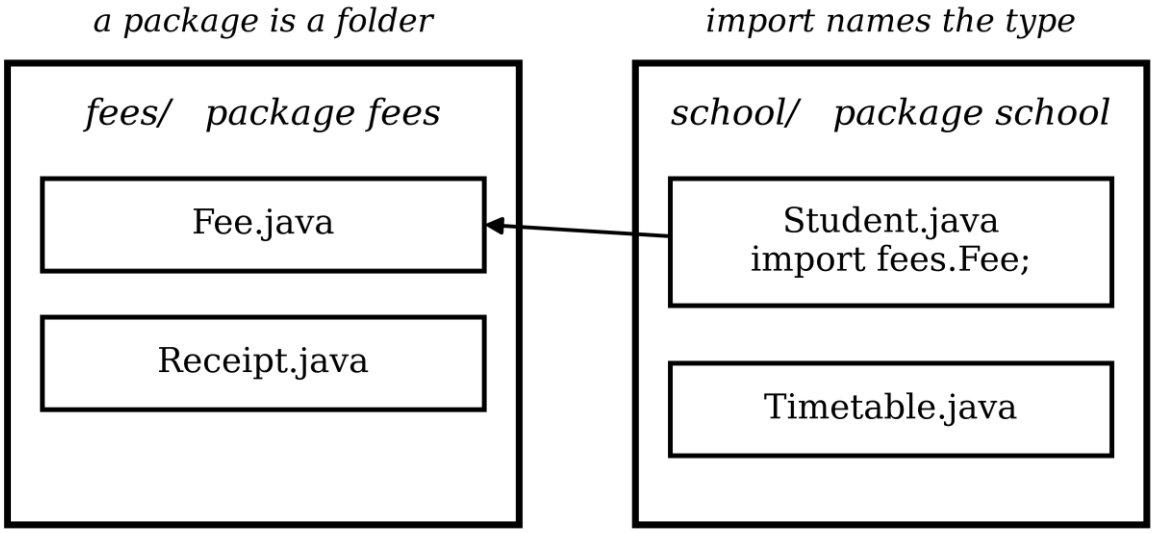


Figure 1.17: *Two packages drawn as the two directories they are, with the import crossing the boundary between them. The arrow carries a name, never any code.*

```
 1  package water;
 2
 3  import java.util.Scanner;
 4
 5  public class Kiosk {
 6      public static void main(String[] args) {
 7          Scanner in = new Scanner(System.in);
 8          System.out.println("Litres drawn today?");
 9          int litres = in.nextInt();
10          in.close();
11
12          int shillings = litres * 25;
13          System.out.println("Litres: " + litres);
14          System.out.println("Shillings: " + shillings);
15      }
16  }
```

Typed at the program

```
180
```

Output

```
Litres drawn today?
Litres: 180
Shillings: 4500
```

Program 1.50: *A class in a named package, launched as water.Kiosk and not as Kiosk.*

The declaration and the directory: the statement on line 1 of Program 1.50 comes before the import and before the class. It puts the class in the namespace water, and javac writes the compiled class into a directory of

that name. A file with no package declaration lands in the unnamed package, where every listing before this section has been.

What the package does to the name: the class on line 5 is not called Kiosk; it is called water.Kiosk, and that is the name the launcher must be given. Two classes may both be called Kiosk so long as they sit in different packages, a protection one folder at Tukuyu did not have.

What an import buys and what it does not: the statement on line 3 names one type, so the rest of the file may write Scanner where it would otherwise have to write java.util.Scanner. Nothing is copied and the compiled class is no larger. The fully qualified name compiles just as well and reads worse.

The one package nobody imports: String, System and Math live in java.lang, which every file imports automatically. That is why a program with no import in it can still declare a String and call System.out.println.

How packages are named: an organisation writes its internet domain backwards and in lower case, so a body whose domain is shule.ac.tz begins its package names tz.ac.shule and adds its own words after that, so that two organisations that have never met cannot choose one name.

A package names a class as much as the class name does; an import only saves typing.

The Master Key: *A package is a namespace that is a directory; import shortens a name, it never moves any code.*

BINDER Question 58

Three, said the notes; four, says the compiler. A student at Mbalizi had learnt private, protected and public, and was then asked which of four levels a member with nothing at all in front of it belongs to.

State the four access levels a member of a Java class may have and give the reach of each one exactly. Say what a member declared with no modifier at all gets, and what happens when a class in another package tries to reach it.

Model Answer

Java has four access levels and not three: private reaches the declaring class alone, the default level that you get by writing no modifier reaches every class in the same package, protected reaches the package and subclasses in other packages as well, and public reaches everything.

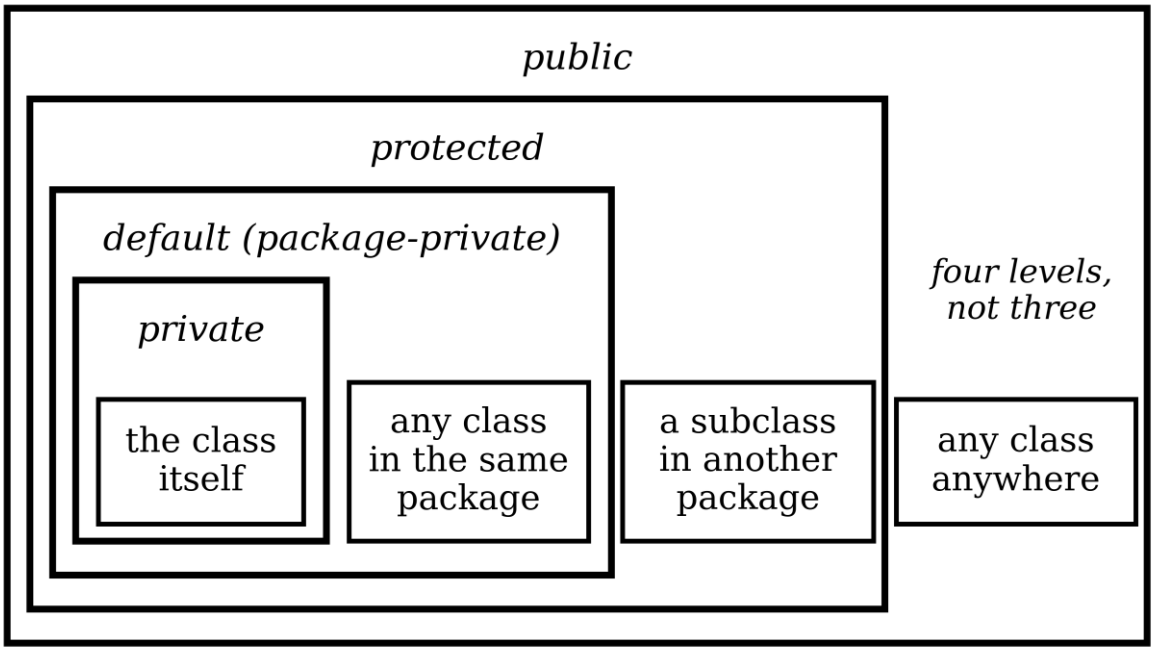


Figure 1.18: *The four levels drawn as nested zones of reach, with the default level, package-private, in its proper place between private and protected.*

private: a private member is reachable only inside the top-level class that declares it, which takes in any class nested within that class and shuts out everything else, including another class in the same file and another class in the same package.

The default level, which has no keyword: writing no modifier gives package-private access, so every class in the same package may reach the member and no class outside it may. There is no word default to type; the level is the absence of a word, which is why it drops off a list of three.

protected: a protected member reaches everything the default level reaches and, in addition, a subclass declared in another package. That extra reach is narrower than it sounds: outside the package the subclass may use the member only through a reference of its own type, never through one typed as the class it inherits from.

public: a public member is reachable by any class anywhere, and it is a promise, not merely a permission. Once a public method is in use by code you do not control, changing it breaks that code, so keep the surface of a class as small as the job allows.

Level	Written as	What may reach it	Where it stops
private	private	The declaring class, nested classes included	Any other class, even one in the same file
default	no modifier at all	Every class in the same package	The package boundary
protected	protected	The package, and subclasses in other packages	Unrelated classes in other packages
public	public	Any class anywhere	Nothing

Table 1.13: *The four levels in order of reach. The second row is the one the common teaching leaves out, and it is the level a field gets when its author decides nothing.*

stock/Bale.java

```
1  package stock;
2
3  public class Bale {
4      int kilogrammes = 60;
5  }
```

shop/Counter.java

```
 1  package shop;
 2
 3  import stock.Bale;
 4
 5  public class Counter {
 6      public static void main(String[] args) {
 7          Bale bale = new Bale();
 8          System.out.println("Kilogrammes: " + bale.kilogrammes);
 9      }
10  }
```

Compiler says

```
shop/Counter.java:8: error: kilogrammes is not public in Bale;
cannot be accessed from outside package
        System.out.println("Kilogrammes: " + bale.kilogrammes);
                                                 ^
1 error
```

Program 1.51: *Two files in two packages. The type is found and the member is out of reach, which is what package-private means.*

Reading the refusal: the field on line 4 of the first file carries no modifier, so it is package-private. Line 8 of the second file is in package shop. The import worked and the class Bale was found; it is the member that is out of reach, and the fixes are to declare the field public or to publish it through a public getter.

Four levels, and the one written by writing nothing is the one to be able to name.

The Master Key: *private, package-private, protected, public: the class, the package, the package plus subclasses, and everything.*

REAL Question 59

The listing at Chimala looked like two classes, and the student counted them: one for the shop, one for the order. The compiler counted one. A closing brace, four lines too low, had turned the second class into part of the first.

A class written inside another class and then instantiated from a static method. Name the fault, state exactly what javac reports, then give two corrections and say what decides which of them is right.

Model Answer

Order is declared inside the body of Tailor, so it is an inner class and every Order needs a Tailor to belong to; main is static and has no Tailor, so javac refuses the instantiation on line 13 and writes no class file at all.

```
 1  public class Tailor {
 2      private String shop = "Chimala Tailoring";
 3
 4      class Order {
 5          private int garments = 6;
 6
 7          public int getGarments() {
 8              return garments;
 9          }
10      }
11
12      public static void main(String[] args) {
13          Order order = new Order();
14          System.out.println("Garments: " + order.getGarments());
15      }
16  }
```

Compiler says

```
Tailor.java:13: error: non-static variable this cannot be
referenced from a static context
        Order order = new Order();
                      ^
1 error
```

Program 1.52: *One class with a class inside it, and a static method trying to make the inner one.*

What the braces actually say: the body of Tailor opens on line 1 and does not close until line 16. Order opens on line 4 and closes on line 10, wholly inside it, so Order is a member of Tailor in the same way that garments is a member of Order. The listing holds one top-level class, not two.

Why an inner class needs an instance: a nested class that is not declared static carries a hidden reference to the object that made it, and that reference is what would let an Order read shop on line 2. Java will therefore not make an Order without being told which Tailor it belongs to.

Exactly what javac does: it stops, reports one error against line 13, and writes no class file, so there is nothing whatever to run. It mentions the word this because the compiler is trying to supply the hidden reference to the enclosing Tailor and finds itself in a static method, where no such object exists.

The first correction: write static class Order on line 4, as Program 1.53 does. A static nested class is a class in its own right that merely lives inside the name of Tailor; it needs no enclosing object and main can make one directly. Notice the price: a static Order can no longer read shop.

The second correction: move Order out of Tailor altogether, either into a file of its own or below Tailor in the same file, since only the public class has to match the file name. It becomes an ordinary top-level class that any other class may use, and the instantiation on line 13 then compiles unchanged.

What decides between them: ask who else uses the class. Nest it and mark it static where it is a detail of Tailor that nothing outside needs; put it at the top level where other classes work with orders too. Leave it inner only where every Order genuinely belongs to one Tailor, and then make it from an instance method rather than from main.

```
 1  public class Tailor {
 2      private String shop = "Chimala Tailoring";
 3
 4      static class Order {
 5          private int garments = 6;
 6
 7          public int getGarments() {
 8              return garments;
 9          }
10      }
11
12      public static void main(String[] args) {
13          Tailor tailor = new Tailor();
14          Order order = new Order();
15          System.out.println("Shop: " + tailor.shop);
16          System.out.println("Garments: " + order.getGarments());
17      }
18  }
```

Output

```
Shop: Chimala Tailoring
Garments: 6
```

Program 1.53: *The first correction, with one word added on line 4 and an instance made on line 13 for the field that does need one.*

Count the braces before you count the classes, because the compiler counts the braces.

The Master Key: *A class inside a class is a member of it; only a static nested class can be made from a static method.*

REAL Question 60

Minus forty litres. The figure went onto the kiosk record without a murmur, and the same figure offered to a second field of the object was refused. Both fields held an int in one class. Only one had anything standing in front of it.

Work through Program 1.54 by hand and write down the five lines it prints. Explain why the value minus forty was taken by one field and refused by the other, and state what a public field can never do about a value it is given.

Model Answer

The public field takes minus forty because nothing at all stands between the assignment on line 21 and the field itself, while the private field can be reached only through setReading, which tests the value first, returns false and leaves the field at zero.

```
 1  public class Tank {
 2      public int litres = 0;
 3      private int reading = 0;
 4
 5      public boolean setReading(int value) {
 6          if (value < 0 || value > 5000) {
 7              return false;
 8          }
 9          reading = value;
10          return true;
11      }
12
13      public int getReading() {
14          return reading;
15      }
16  }
17
18  class Refill {
19      public static void main(String[] args) {
20          Tank tank = new Tank();
21          tank.litres = -40;
22          System.out.println("Public field: " + tank.litres);
23          System.out.println("Accepted: " + tank.setReading(-40));
24          System.out.println("Reading: " + tank.getReading());
25          System.out.println("Accepted: " + tank.setReading(3200));
26          System.out.println("Reading: " + tank.getReading());
27      }
28  }
```

Output

```
Public field: -40
Accepted: false
Reading: 0
Accepted: true
Reading: 3200
```

Program 1.54: *One field anybody may assign to, and one that may be changed only by a method that checks first.*

The trace: line 20 makes a Tank with both fields at zero. Line 21 puts minus forty straight into litres, so line 22 prints it back. Line 23 offers the same value to setReading, the guard on line 6 holds, the method returns false without assigning, and line 24 finds the reading still at zero. Line 25 offers 3200, which passes, so line 26 prints 3200.

What the guard is doing: the test on line 6 turns away anything below zero or above five thousand, which are the readings this tank cannot physically hold. The method reports the refusal by returning false, so the caller can tell a value that was stored from one that was thrown away.

The names these methods take: a getter is named get followed by the field name with a capital letter, takes nothing, and returns the type of the field; a setter is named set, takes one value of that type, and returns either void or, as here, a boolean saying whether it accepted. For a boolean field the getter is named is rather than get.

The alternative to refusing: a setter may clamp instead, storing the nearest legal value and saying nothing. Clamping suits a setting where roughly right beats absent, such as a volume; refusing suits a record, where a wrong figure quietly stored does more harm than no figure at all. What neither of them does is take the value in silence.

What a public field can never do: there is nowhere to put the test. Every assignment reaches the field directly from anywhere the object is visible, and the class gets no moment at which it could look at the value. Nor can the class change what it stores later: publish litres and every caller now depends on there being a field of that name.

A field with a method in front of it can refuse; a field with nothing in front of it takes whatever it is handed.

The Master Key: *The value of a setter is the test inside it, and a setter with no test is a public field written the long way.*

BUILD Question 61

Friday is stock day at the Ilula dispensary, and one shelf came back counted at six hundred and twenty tins of a medicine it has never held five hundred of. A pencil had slipped between shelf and card. The card took the figure anyway.

Write a complete Java program that keeps a stock count which cannot be set to an impossible figure, to the specification below.

The class is called StockCard and the program is one file.

The count of units is private and starts at zero.

setUnits stores a count from zero up to the shelf limit of 500 and returns true; for any other figure it stores nothing and returns false.

getUnits reports the count currently held.

The program reads two counts typed at the keyboard, and after each one prints whether it was accepted and the count now held.

Use the counts 620 and 185.

Model Answer

The specification asks for one field that a single method may change and for that method to refuse two whole ranges of value, and the program below does the checking in one condition and reports the refusal as a boolean the caller can act on.

```
 1  import java.util.Scanner;
 2
 3  public class StockCard {
 4      private static final int SHELF_LIMIT = 500;
 5      private int units = 0;
 6
 7      public int getUnits() {
 8          return units;
 9      }
10
11      public boolean setUnits(int value) {
12          if (value < 0 || value > SHELF_LIMIT) {
13              return false;
14          }
15          units = value;
16          return true;
17      }
18  }
19
20  class Dispensary {
21      public static void main(String[] args) {
22          Scanner in = new Scanner(System.in);
23          StockCard card = new StockCard();
24          System.out.println("Two counts?");
25          int first = in.nextInt();
26          int second = in.nextInt();
27          in.close();
28
29          System.out.println("Accepted: " + card.setUnits(first));
30          System.out.println("Units: " + card.getUnits());
31          System.out.println("Accepted: " + card.setUnits(second));
32          System.out.println("Units: " + card.getUnits());
33      }
34  }
```

Typed at the program

```
620
185
```

Output

```
Two counts?
Accepted: false
Units: 0
Accepted: true
Units: 185
```

Program 1.55: *The stock card, which counts what is possible and refuses what is not.*

Where the specification is satisfied: units is private on line 5, so setUnits on lines 11 to 17 is the only door into it. The condition on line 12 turns away both impossible ranges at once, the storing and the true are lines 15 and 16, and lines 29 to 32 print the pair of lines the specification asks for after each count.

The one design decision worth naming: setUnits returns a boolean rather than ignoring a bad figure in silence. A method that refuses without saying so leaves the caller believing the count went in; here the count of 620 was refused and the first printed line says so, which is how the operator learns that the pencil slipped.

Why the limit is a constant: SHELF_LIMIT on line 4 is private, static and final: one copy for the whole class, named once, and unchangeable. Writing 500 into the condition instead would work and would bury the number in the middle of a test, where nobody revising the shelf would think to look for it.

One field, one way in, and one condition that decides what the card is allowed to say.

The Master Key: *Guard the field with the method, and let the method say out loud when it has refused.*

BUILD Question 62

Nothing in the maize ledger at Kalenga may be altered once the buyer has signed. A wrong figure is struck through and written again beneath, where anybody can still see it. A class can be built to the same rule for one keyword.

Write a complete Java program in which a receipt cannot be altered after it has been made, to the specification below.

The class is called Receipt and the program is one file.

The number of bags and the price of one bag are given to the constructor and may not be changed afterwards.

getBags reports the number of bags.

getTotal reports the total, which is worked out when it is asked for and never stored.

The price of a bag is used inside the class and never published.

The program reads the two figures typed at the keyboard, then prints the bags and the total.

Use 48 bags at 900 shillings each.

Model Answer

The specification asks for an object that is finished the moment it is built, and the program below gets one with two private final fields, a constructor that is the only thing that ever assigns to them, and not a single setter anywhere in the class.

```
 1  import java.util.Scanner;
 2
 3  public class Receipt {
 4      private final int bags;
 5      private final int pricePerBag;
 6
 7      public Receipt(int bags, int pricePerBag) {
 8          this.bags = bags;
 9          this.pricePerBag = pricePerBag;
10      }
11
12      public int getBags() {
13          return bags;
14      }
15
16      public int getTotal() {
17          return bags * pricePerBag;
18      }
19  }
20
21  class Ledger {
22      public static void main(String[] args) {
23          Scanner in = new Scanner(System.in);
24          System.out.println("Bags and price?");
25          int bags = in.nextInt();
26          int price = in.nextInt();
27          in.close();
28
29          Receipt receipt = new Receipt(bags, price);
30          System.out.println("Bags: " + receipt.getBags());
31          System.out.println("Total: " + receipt.getTotal());
32      }
33  }
```

Typed at the program

```
48
900
```

Output

```
Bags and price?
Bags: 48
Total: 43200
```

Program 1.56: *A receipt whose fields are fixed at construction, with a total that no field holds.*

Where the specification is satisfied: lines 4 and 5 declare both fields private and final, so the compiler rejects any assignment to them outside the constructor on lines 7 to 10. getBags on line 12 publishes one of the two, getTotal on line 16 publishes a figure that no field holds, and the price is never published at all.

The one design decision worth naming: the total is worked out at line 17 rather than stored. A stored total is a third number that can fall out of step with the two it came from; a computed one is right every time it is asked for. That is a read-only property, which is a getter with no setter behind it.

What final buys that private alone does not: private stops an outsider assigning to the field. final stops everybody, the methods of the class included, and it says so in the declaration where a reader will see it. An object whose every field is final cannot be put into an illegal state after it is built, because it cannot be altered at all.

Give the values to the constructor, make the fields final, and there is no setter left to get wrong.

The Master Key: *Immutability is private final fields and no setter; what cannot change cannot be corrupted.*

HOT Question 63

The class handed in after the practical had nine private fields, nine getters, nine setters and not one line doing anything else. It had followed the rule it was given. Every field could still be set to any value by anybody holding it.

It is often taught that encapsulation means making every field private and providing a getter and a setter for each one. Evaluate that practice and reach a judgement.

Model Answer

A getter and a setter for every field is not encapsulation, because a setter that only assigns gives an outsider exactly the reach a public field would have given in three times as many lines, and the practice earns its place only where the method does something the field could not.

The case for the practice: it is uniform, it is quick to teach, and it leaves a place where a check can be put later without one caller having to change. Callers depend on setUnits, not on there being a field called units, so the class keeps the freedom to change what it stores behind them.

Where the practice is empty: if the setter does nothing but assign, the object is as open as it would be with a public field: anybody may put any value into it. The word private is in the source and no rule about the object is kept by it, which is the look of encapsulation without the substance.

What actually earns a setter: a test. A setter that turns away a count of minus forty, or clamps it to the nearest legal figure, is doing work that no public field could do, and that is the one argument for it that survives examination. Where a change must also alter something else about the object, the setter is where that happens.

What actually earns a getter: a value the caller genuinely needs. A getter over a computed figure, such as the total in Program 1.56, publishes something no field holds; and a getter with no setter beside it makes a property outsiders may read and may not write, which no field can express.

The option the rule leaves out: having neither. Where nothing about the object changes after it is built, the values go to the constructor, the fields are private and final, and there is no setter to argue about. Such a class cannot be put into an illegal state after construction, and it is shorter than the one with eighteen accessors.

The verdict: make every field private, always; that part of the rule is right and costs nothing. Write a setter only where the value genuinely changes during the life of the object and the change can be checked or must be acted upon, and write a getter only where a caller needs the value. Where nothing changes after construction, make the fields final and write neither.

How to argue it in an answer: name the mechanism, concede the one gain the empty setter still brings, and let the invariant decide the verdict.

The rule is right about private and wrong about the pair of methods it tells you to write next.

The Master Key: *A setter is worth writing when it can refuse; a value that never changes wants final and neither method.*

Inheritance, super and method overriding

This section puts one class beneath another and settles what that buys and what it costs. It gives extends, the test that decides whether extends is the right tool at all, and the single restriction Java places on it, then follows construction base first through a three level chain with super at the head of every constructor. It separates overriding, which the object settles while the program runs, from the overloading already taught with methods, which the compiler settled before the program started, and it makes @Override a rule rather than an ornament, because the fault it catches is silent, legal and expensive. It closes on final, the brake that stops a method being replaced and a class being extended at all.

BINDER Question 64

A print session is a session. Say it aloud, and if it holds, Java lets one class take everything the other has without copying a line. The internet cafe at Idodi had two classes with eleven lines in common, and nobody had noticed.

Explain what extends does in Java, naming the two classes it relates. State what a subclass takes from its superclass and what it does not, give the test that decides whether inheritance is the right tool, and state how many classes a Java class may extend.

Model Answer

extends makes one class a subclass of another, so every member of the superclass that the subclass is allowed to see becomes part of it without being written twice, and a Java class may extend exactly one class and no more.

```
 1  class Session {
 2      private int minutes;
 3
 4      public void use(int extra) {
 5          minutes = minutes + extra;
 6      }
 7
 8      protected int minutes() {
 9          return minutes;
10      }
11  }
12
13  public class PrintSession extends Session {
14      private int sheets;
15
16      public void printed(int extra) {
17          sheets = sheets + extra;
18      }
19
20      public int bill() {
21          return minutes() * 20 + sheets * 100;
22      }
23
24      public static void main(String[] args) {
25          PrintSession session = new PrintSession();
26          session.use(45);
27          session.printed(3);
28          System.out.println("Minutes used: " + session.minutes());
29          System.out.println("Bill: " + session.bill());
30      }
31  }
```

Output

```
Minutes used: 45
Bill: 1200
```

Program 1.57: *A cafe session and a print session, where the second is the first with two fields of its own added.*

What the keyword does: writing extends Session on line 13 names Session the superclass and PrintSession the subclass. Every field and method Session declares is now part of every PrintSession object, so the use method declared on line 4 is called on line 26 although PrintSession never declares it.

What does not come across: constructors are never inherited, so no subclass gets a constructor of its superclass under any name. Neither can the subclass body name a private member: minutes on line 2 is inside every PrintSession object and unreachable from line 21, which is why line 21 goes through the protected method instead.

The reach protected was made for: protected is the level built for this relationship. A subclass declared in another package may reach a protected member, but only through a reference of its own type, never through one typed as the class it inherits from.

The test that decides: put the two names into the sentence a subclass is a superclass and see whether it holds for every object. A print session is a session, so extends is right. A cafe is not a session however many sessions it runs, so a Cafe class must hold Session objects in a field rather than extend one.

One superclass, and where the rest of the answer went: a Java class may name exactly one class after extends; two names will not compile. That is a deliberate restriction, and it is the reason interfaces exist, which the next section takes up.

Member of the superclass	**Available to the subclass?**	**Why**
public and protected methods	Yes, called by name	these are the members inheritance was made to pass on
members with no access modifier	Only inside the same package	the default level stops at the package boundary
private fields and methods	No; the field is in the object but the name is not usable	private reaches the declaring class alone
constructors	No, never	a constructor builds one class; a subclass reaches it with super

Table 1.14: *What crosses from a superclass into a subclass and what does not. Only the last row is absolute, because the first three depend on where the subclass is declared.*

Form	**Shape**	**Java**
Single	one subclass extends one superclass	Allowed
Multilevel	C extends B, and B extends A	Allowed
Hierarchical	two or more subclasses extend one superclass	Allowed
Multiple	one class extends two classes at once	Refused by the compiler

Table 1.15: *The shapes inheritance may take. Java allows the first three and refuses the fourth for classes, which is the gap interfaces are there to fill.*

Say the sentence first: if a subclass is a superclass, extends is right, and if it merely uses one, a field is right.

The Master Key: *A subclass gets every member it may see, never a constructor, and never a second superclass.*

BINDER Question 65

Three lines printed in the order their author least expected. The boarding register at Lupembe was written from Boarder downwards, Boarder was the class asked for, and Person printed first every time. Nothing in the source said to do that, and nothing had to.

Explain what super followed by arguments does and state exactly where it may be written. State the order in which the constructors of a three level chain run, and say what the compiler does when a subclass constructor calls no superclass constructor at all.

Model Answer

super(...) calls a constructor of the immediate superclass, it must be the first statement of the subclass constructor, and construction therefore runs base first: each superclass constructor finishes before the body beneath it begins.

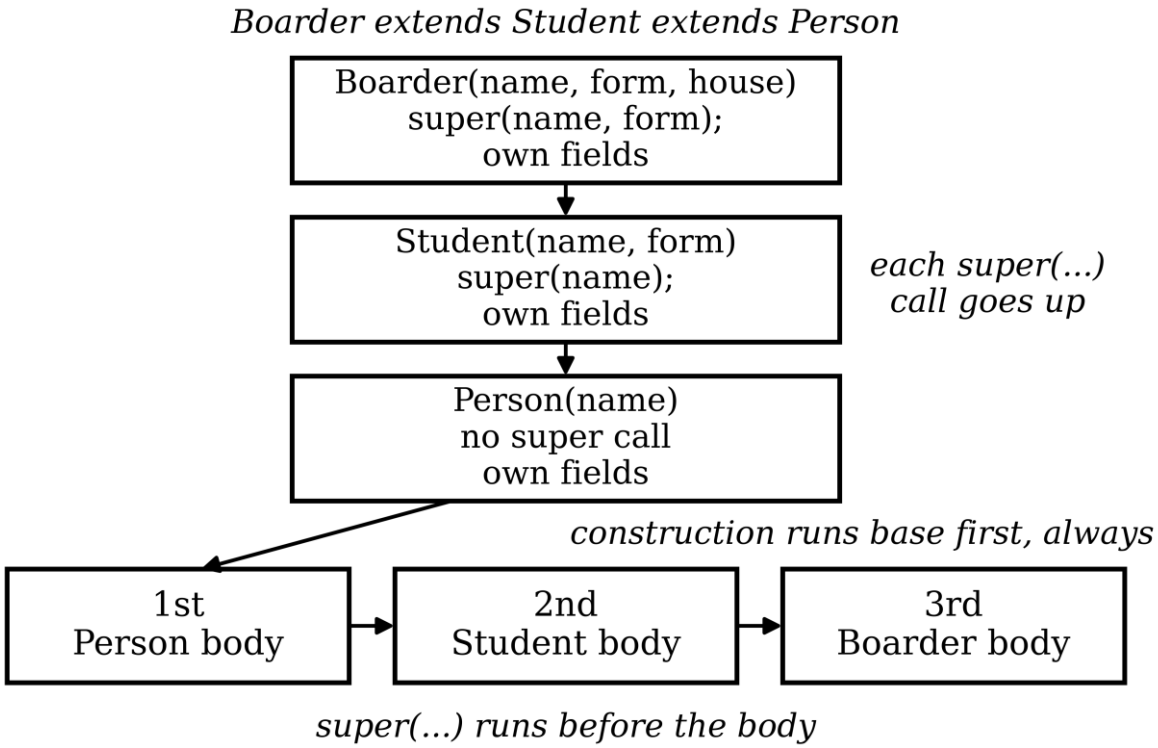


Figure 1.19: *Construction through a three level chain. Every super call runs before the body that contains it, so the calls travel up to Person and the bodies come back down, with Boarder finishing last.*

Why a subclass has to ask: a constructor is not inherited, so a Boarder constructor cannot build the Person part of the object itself. It can only ask the constructor above it to do that work, and super(name, form) on line 24 is that request.

Why it must come first: the superclass part of the object has to be complete before the subclass touches it, or a subclass body could read a field its superclass had not yet set. Java enforces this by position: super with arguments anywhere but the opening line of the constructor is a compile error, and a constructor may open with super or with this, never with both.

What the compiler supplies when you write neither: a constructor that opens with neither gets a call to super with no arguments inserted for it, invisibly. While the superclass has a constructor taking no arguments this is harmless and nobody notices; when it has not, the inserted call is the whole of the trouble, and it is worth meeting on its own.

Reading the order off the run: the request travels up and the work comes down. Line 30 asks for a Boarder, line 24 asks Student, line 14 asks Person, and only then does any body run: Person on line 6, Student on line 16, Boarder on line 26. Rewriting the classes in a different order in the file changes nothing at all.

```
 1  class Person {
 2      private final String name;
 3
 4      Person(String name) {
 5          this.name = name;
 6          System.out.println("Person body runs for " + this.name);
 7      }
 8  }
 9
10  class Student extends Person {
11      private final int form;
12
13      Student(String name, int form) {
14          super(name);
15          this.form = form;
16          System.out.println("Student body runs, form " + this.form);
17      }
18  }
19
20  public class Boarder extends Student {
21      private final String house;
22
23      Boarder(String name, int form, String house) {
24          super(name, form);
25          this.house = house;
26          System.out.println("Boarder body runs, house " + this.house);
27      }
28
29      public static void main(String[] args) {
30          new Boarder("Neema", 5, "Lupembe");
31      }
32  }
```

Output

```
Person body runs for Neema
Student body runs, form 5
Boarder body runs, house Lupembe
```

Program 1.58: *A three level chain that prints as each constructor body reaches its own work.*

The source is written from the subclass down, and the object is built from the superclass up.

The Master Key: *super with arguments must be the first statement, and the superclass constructor always finishes first.*

BINDER Question 66

Two variables at the Peramiho bursary, both declared Fee, both asked for the same total for the same three terms. One answered 180000, the other 185000. The line asking them was the same twice over, and the compiler could not say which answer was coming.

Explain method overriding and state the conditions a method must meet to override another. Explain how the version that runs is chosen, and state what @Override does and why it should be written on every overriding method.

Model Answer

Overriding is declaring in a subclass a method with the same name and the same parameter list as one in its superclass, and the version that runs is chosen while the program is running, from the class of the object rather than the type of the variable.

```
 1  class Fee {
 2      public int total(int terms) {
 3          return terms * 60000;
 4      }
 5  }
 6
 7  class LateFee extends Fee {
 8      @Override
 9      public int total(int terms) {
10          return super.total(terms) + 5000;
11      }
12  }
13
14  public class FeeDesk {
15      public static void main(String[] args) {
16          Fee ordinary = new Fee();
17          Fee late = new LateFee();
18          System.out.println("Ordinary: " + ordinary.total(3));
19          System.out.println("Late: " + late.total(3));
20      }
21  }
```

Output

```
Ordinary: 180000
Late: 185000
```

Program 1.59: *One method name, two classes, and two variables of the same declared type.*

What has to match, exactly: the name and the parameter list, and the returned type must be the same or a subtype of it. Change the parameter list and nothing has been overridden; a second method has been added, which is the overloading already settled with methods. An override may also not be less accessible than the method it replaces, and may not throw a broader checked exception than it declares.

Who chooses the version, and when: the object does, as the call is made. Lines 16 and 17 declare both variables as Fee, so the compiler cannot separate them; line 18 prints 180000 because its object is a Fee and line 19 prints 185000 because its object is a LateFee.

Reaching the version you replaced: super.total(terms) on line 10 calls the superclass version from inside the one that replaced it, which is how a subclass adds to an answer instead of restating it. Drop the word super and the method calls itself for ever.

What @Override is for: it states your intention to the compiler, which then checks it. If the method overrides nothing, javac stops and says so. Without it, a parameter list typed one character wrong is a legal second method, the program compiles, and the superclass version keeps running.

The method every class already has: every class extends Object, whether extends is written or not, so every class inherits toString from it. The inherited version prints the class name and a hash, which tells a reader nothing, and almost every class you write should replace it.

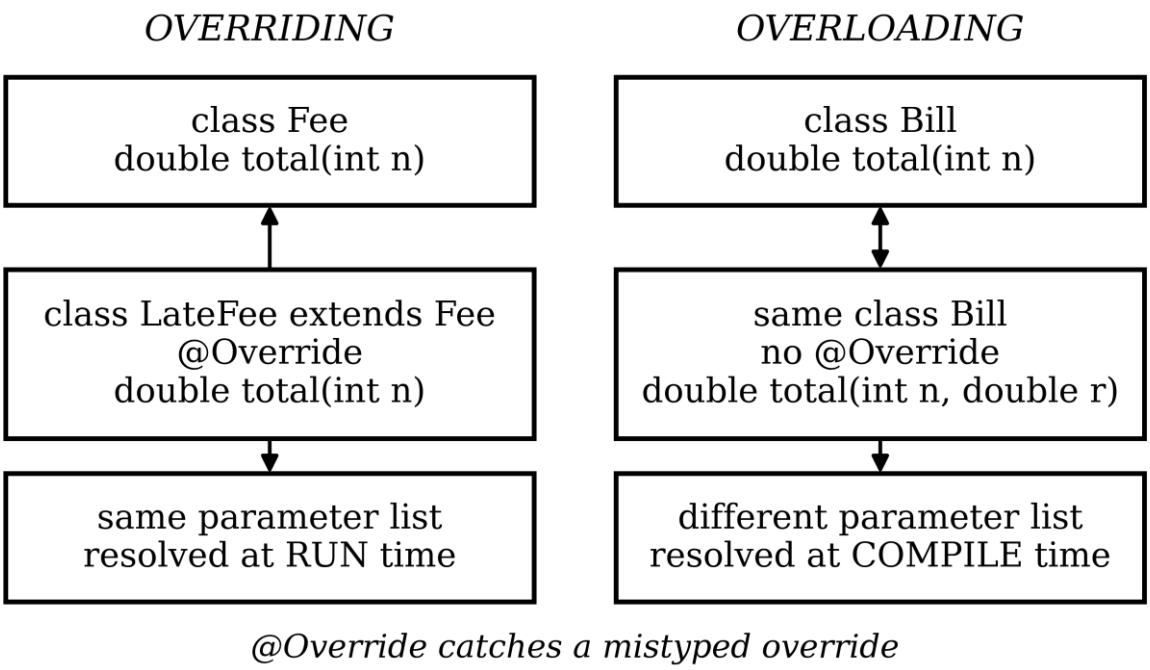


Figure 1.20: *The two mechanisms drawn in outline. The same parameter list in a subclass replaces a method and is settled while the program runs; a different parameter list adds a method and is settled while it is compiled.*

Same parameter list, and the object decides; different parameter list, and the compiler decided long ago.

The Master Key: *An override matches the parameter list exactly and is chosen by the object; write @Override and let javac check you.*

REAL Question 67

The money kiosk at Kigonsera reversed a transfer and handed the customer a slip saying the cash had come in. The clerk printed it again: the same lie. No error, no warning, one word missing from a file nobody had opened in weeks.

Program 1.60 compiles without a single diagnostic and prints the wrong line. Name the fault, state exactly what the compiler did with the subclass method and why it reported nothing, and show what adding @Override does to the same code.

Model Answer

print(String note) in Reversal does not override print() in Slip, because their parameter lists differ; it is a separate method that happens to share a name, so reversal.print() on line 16 finds the inherited version and prints the slip for an ordinary payment.

```
 1  class Slip {
 2      public void print() {
 3          System.out.println("Slip: cash in, ordinary payment");
 4      }
 5  }
 6
 7  class Reversal extends Slip {
 8      public void print(String note) {
 9          System.out.println("Reversal: " + note);
10      }
11  }
12
13  public class Till {
14      public static void main(String[] args) {
15          Reversal reversal = new Reversal();
16          reversal.print();
17      }
18  }
```

Output

```
Slip: cash in, ordinary payment
```

Program 1.60: *A subclass that meant to replace a method and added one instead.*

What the author meant to write: the intention was to replace print so that a reversal announces itself. A parameter was added to the list while the method was being written, and the moment the list changed, the connection to the superclass method was gone.

What the language built instead: Reversal now has two methods called print: the one it inherited, taking nothing, and the one it declared, taking a String. Two methods of one name with different parameter lists overload one another, which is legal and was settled with methods, so there was nothing here for javac to object to.

Why the inherited version ran: line 16 passes no argument, so only print() fits, and the compiler bound the call to it before the program started. The variable and the object are both Reversal and it makes no difference: overriding needs a matching parameter list, and there was none to match.

Why the run was silent as well: the wrong output is a correct execution of a legal program. Nothing throws, the exit status is zero, and the only evidence is a slip in a customer's hand that says the opposite of what happened.

What one annotation changes: Program 1.61 is the same code with @Override written above the method. The annotation asserts that this method overrides one, javac tests the assertion, finds no print(String) anywhere above Reversal, and refuses to compile, naming the line.

```
 1  class Slip {
 2      public void print() {
 3          System.out.println("Slip: cash in, ordinary payment");
 4      }
 5  }
 6
 7  class Reversal extends Slip {
 8      @Override
 9      public void print(String note) {
10          System.out.println("Reversal: " + note);
11      }
12  }
13
14  public class Till {
15      public static void main(String[] args) {
16          Reversal reversal = new Reversal();
17          reversal.print();
18      }
19  }
```

The compiler says

```
Till.java:8: error: method does not override or implement
        a method from a supertype
    @Override
    ^
1 error
```

Program 1.61: *The same classes with the intention stated, and a compiler that now checks it.*

The correction, and the habit behind it: remove the parameter so the list matches, and keep the annotation. Written on every method you intend as an override, it converts the worst failure in this topic, a wrong answer with no message, into the cheapest one, a compile error that names the line.

The compiler will check what you meant, but only if you tell it what you meant.

The Master Key: *Without @Override a mistyped override is a legal overload, and the superclass version quietly keeps running.*

REAL Question 68

A complaint about a constructor the source never calls. The listing below names Delivery once, in the word extends, and javac answers with a complaint about the Delivery constructor pointing at line 16, where no constructor is called. It is pointing at the right place.

Program 1.62 does not compile. Name the fault, state exactly what javac reports, explain why the message lands on line 16 rather than on line 21, and correct it.

Model Answer

The constructor of GradedDelivery opens with neither super nor this, so the compiler inserts a call to super with no arguments, and Delivery declares no constructor that takes no arguments for the inserted call to reach.

```
 1  class Delivery {
 2      private final String crop;
 3
 4      Delivery(String crop) {
 5          this.crop = crop;
 6      }
 7
 8      public String crop() {
 9          return crop;
10      }
11  }
12
13  public class GradedDelivery extends Delivery {
14      private final int bags;
15
16      GradedDelivery(int bags) {
17          this.bags = bags;
18      }
19
20      public static void main(String[] args) {
21          GradedDelivery load = new GradedDelivery(120);
22          System.out.println(load.crop() + ": " + load.bags);
23      }
24  }
```

The compiler says

```
GradedDelivery.java:16: error: constructor Delivery in class
        Delivery cannot be applied to given types;
    GradedDelivery(int bags) {
                             ^
  required: String
  found:    no arguments
  reason: actual and formal argument lists differ in length
1 error
```

Program 1.62: *A subclass constructor that never mentions its superclass.*

What is really on line 16: more than is written there. Every constructor begins by running a constructor of its superclass, and where the author writes no such call the compiler puts one in, with no arguments. Line 16 therefore contains an invisible call to Delivery with nothing passed to it.

Why there is no constructor to receive it: the compiler supplies a constructor taking no arguments only to a class that declares none of its own. Delivery declares one on line 4, so that is the only constructor it has, and it demands a String.

Why the message lands where it does: the error is reported against the subclass constructor header, because that is where the failing call sits. Line 21 is innocent: GradedDelivery(int) exists and 120 fits it perfectly, and a candidate who spends the examination staring at line 21 will find nothing wrong with it.

Reading the three lines below the arrow: required names what Delivery wanted, a String; found names what the inserted call offered, no arguments; and reason says the two lists are not even the same length. The same wording appears when an ordinary call to new misses a constructor, and the only clue that this one is different is where the caret is pointing.

The correction: give GradedDelivery a parameter for the crop and pass it up, as Program 1.63 does on lines 16 and 17. The alternative, adding a constructor with no arguments to Delivery, would compile just as well and would leave every graded delivery carrying a crop nobody set, so it answers the compiler without answering the problem.

```
 1  class Delivery {
 2      private final String crop;
 3
 4      Delivery(String crop) {
 5          this.crop = crop;
 6      }
 7
 8      public String crop() {
 9          return crop;
10      }
11  }
12
13  public class GradedDelivery extends Delivery {
14      private final int bags;
15
16      GradedDelivery(String crop, int bags) {
17          super(crop);
18          this.bags = bags;
19      }
20
21      public static void main(String[] args) {
22          GradedDelivery load = new GradedDelivery("maize", 120);
23          System.out.println(load.crop() + ": " + load.bags);
24      }
25  }
```

Output

```
maize: 120
```

Program 1.63: *The same classes with the request passed upwards, in the only statement that may stand first.*

What the fault proves about inheritance: constructors are not inherited. A subclass that wants the superclass built a particular way must say so itself, every time, in its own first statement.

The compiler wrote a line you did not, and then complained that the line it wrote had nowhere to go.

The Master Key: *Write no super call and one is inserted with no arguments, which fails unless the superclass has such a constructor.*

BUILD Question 69

Item@6d06d69c, said the screen in the Kitangari school library, where a title should have been. Nothing was broken, nothing had failed. The class had never been told how to describe itself, and Java fell back on the only description it had.

Write a complete Java program that prints a library record as one readable line, to the specification below, and state why no formatting method is called anywhere in it.

Item is the superclass and holds the title of the record.

Book extends Item and adds the author.

Each of the two classes replaces toString with a description of itself.

The version in Book builds on the version in Item instead of repeating it.

The author is read from the keyboard, and printing the object prints the line.

Model Answer

The specification asks for two classes, a replacement for toString in each of them, and a subclass version that builds on the superclass version, and the program below satisfies it in thirty four lines.

```
 1  import java.util.Scanner;
 2
 3  class Item {
 4      private final String title;
 5
 6      Item(String title) {
 7          this.title = title;
 8      }
 9
10      @Override
11      public String toString() {
12          return title;
13      }
14  }
15
16  public class Book extends Item {
17      private final String author;
18
19      Book(String title, String author) {
20          super(title);
21          this.author = author;
22      }
23
24      @Override
25      public String toString() {
26          return super.toString() + " by " + author;
27      }
28
29      public static void main(String[] args) {
30          Scanner in = new Scanner(System.in);
31          System.out.println(new Book("Hisabati", in.nextLine()));
32          in.close();
33      }
34  }
```

Typed at the program

```
Mwakyusa
```

Output

```
Hisabati by Mwakyusa
```

Program 1.64: *The library record, compiled with javac Book.java and run with java Book.*

Where the specification is satisfied: Item replaces toString on lines 10 to 13 and Book replaces it on lines 24 to 27, calling the superclass version on line 26. Line 31 hands the object itself to println and reads the author from the keyboard in the same statement.

Why no formatting method is called: println asks whatever it is given for its toString and prints what comes back, and so does every join of an object to a String with a plus. Replacing toString therefore changes how the object appears everywhere at once, which is why a describe method of your own is worth less than an override of this one.

The one design decision worth naming: line 26 calls super.toString rather than naming the title again. Item owns how a title is shown, Book owns only the part it added, and a change to the first is made in one place.

Thirty four lines, two classes, and a record that describes itself wherever it is printed.

The Master Key: *Every class inherits toString from Object; replace it and every printing of the object improves at once.*

BUILD Question 70

Three prices for one transfer. The money kiosk at Nangurukuru charged one shilling in every hundred, half of that inside its own network, and five hundred more outside it. The clerk carried all three rates in her head until the morning she did not.

Write a complete Java program that charges a mobile money transfer three ways, to the specification below, and name the form of inheritance it uses.

Transfer is the superclass and charges one shilling in every hundred sent.

SameNetwork and OtherNetwork both extend Transfer, and nothing extends either of them.

SameNetwork halves the ordinary fee and OtherNetwork adds five hundred to it.

Neither subclass restates the ordinary rate; each reaches it through super.

The amount sent is read from the keyboard, and all three fees are printed.

Model Answer

One superclass with two subclasses beneath it and nothing beneath them is hierarchical inheritance, and the program below writes the rate once, on line 5, with both subclasses reaching it through super.

```
 1  import java.util.Scanner;
 2
 3  class Transfer {
 4      public int fee(int shillings) {
 5          return shillings / 100;
 6      }
 7  }
 8
 9  class SameNetwork extends Transfer {
10      @Override
11      public int fee(int shillings) {
12          return super.fee(shillings) / 2;
13      }
14  }
15
16  class OtherNetwork extends Transfer {
17      @Override
18      public int fee(int shillings) {
19          return super.fee(shillings) + 500;
20      }
21  }
22
23  public class Desk {
24      public static void main(String[] args) {
25          Scanner in = new Scanner(System.in);
26          int amount = in.nextInt();
27          in.close();
28          System.out.println("Plain: " + new Transfer().fee(amount));
29          System.out.println("Same: " + new SameNetwork().fee(amount));
30          System.out.println("Other: " + new OtherNetwork().fee(amount));
31      }
32  }
```

Typed at the program

```
50000
```

Output

```
Plain: 500
Same: 250
Other: 1000
```

Program 1.65: *One rate, two subclasses, and three fees from a single amount typed in.*

Where the specification is satisfied: the ordinary rate is line 5 and appears nowhere else. SameNetwork overrides fee on lines 11 to 13 and OtherNetwork on lines 18 to 20, each calling super.fee for the ordinary charge before adjusting it, and lines 28 to 30 print the three results for the one amount read on line 26.

The one design decision worth naming: neither subclass knows the rate. Change 100 to 80 on line 5 and all three fees move together, whereas a subclass that had written its own division would go on charging the old rate and nothing would report it.

Why the annotation earns its place here: with fee written three times over, a parameter list typed wrongly in either subclass would produce an overload that never runs. @Override on lines 10 and 17 makes that a compile error rather than a wrong fee.

One rate, written once, and two subclasses that adjust it without knowing what it is.

The Master Key: *Hierarchical inheritance puts several subclasses under one superclass; super keeps the shared rule in one place.*

HOT Question 71

Eleven boxes, ten arrows, every arrow labelled extends. The chart was drawn to save typing, no method was written twice. Then one line changed in the top box, four classes at the bottom stopped behaving, and nobody could say why any arrow was there.

Evaluate the claim that a class should extend another whenever doing so saves rewriting the methods of the other. Reach a judgement.

Model Answer

Reuse is a consequence of inheritance and not a reason for it: extends promises what the subclass is as well as what it has, so where only the code is wanted the right answer is to hold an object of that class in a field and call it.

The case for the claim: duplication is a real cost and inheritance really does remove it. Two classes sharing eleven lines will drift apart, one copy will be corrected and the other will not, and the fault that follows is found weeks later. Writing extends makes that impossible in one word.

What extends promises besides the code: it also states that every object of the subclass is an object of the superclass, and every variable declared with the superclass type relies on that. Code holding one may call anything the superclass declares, on an object it has never seen, and expect an answer that makes sense.

Where that promise breaks: the subclass inherits methods that have no meaning for it. A Session has a use method that adds minutes; a Cafe extending Session merely to reuse its counter would acquire that method too, and a cafe with minutes used on it is not an object the rest of the program can trust.

The cost that arrives later: a superclass cannot be changed alone. Every subclass depends on what its methods do and on which of them call which others, so a correction at the top reaches classes the author has never opened. Eleven boxes are eleven classes that must now be understood together.

What to do when only the code is wanted: hold an object of the other class in a private field and call its methods from your own. Nothing is inherited, so nothing is promised, callers see only the methods you chose to write, and the two classes may be changed on different days without either breaking the other.

The verdict: extend where the sentence a subclass is a superclass is true of every object the subclass will ever have, and where you are content for the superclass methods to remain correct for it. Where the sentence is false, or where you would have to override a method in order to stop it doing something, use a field. Shared code alone is never sufficient reason.

How to argue it in an answer: name the test, give one example that passes it and one that fails it, and say what you would write instead in the second case. A verdict resting on that condition earns marks that praise of reuse does not.

Inheritance states what a thing is, and charges for that statement every time the superclass changes.

The Master Key: *Extend for what the subclass is, not for what it saves you typing; otherwise hold a field.*

HOT Question 72

One word, five letters, and two entirely different jobs. In front of a field it says the value will not change again. In front of a method or a class it says nothing about values at all.

State what final does to a method and to a class. Evaluate the claim that a class should be left open to extension unless there is a definite reason to close it, and reach a judgement.

Model Answer

A final method cannot be overridden and a final class cannot be extended at all, and the claim has the burden the wrong way round: a class not designed for extension should be closed, because leaving it open promises a stability its author has never checked he can deliver.

What the keyword forbids here: on a method, final stops any subclass replacing it, and the attempt is a compile error rather than something found later. On a class it stops anything extending it, which closes every method in it at once; String is declared this way. It is the same word used for a constant, doing an unrelated job.

```
 1  class Certificate {
 2      public final String serial() {
 3          return "SN-4471";
 4      }
 5  }
 6
 7  public class Duplicate extends Certificate {
 8      @Override
 9      public String serial() {
10          return "SN-4471-COPY";
11      }
12
13      public static void main(String[] args) {
14          System.out.println(new Duplicate().serial());
15      }
16  }
```

The compiler says

```
Duplicate.java:9: error: serial() in Duplicate cannot
        override serial() in Certificate
    public String serial() {
                  ^
  overridden method is final
1 error
```

Program 1.66: *A subclass that tries to replace a method its superclass has closed.*

The case for leaving a class open: an author cannot foresee every use his class will be put to, and a subclass is often the cheapest way to adapt one without touching its source. A class closed by habit rather than by decision leaves the next reader copying it, and two near identical classes are the worse outcome.

What an open class is actually promising: that every method not marked final may be replaced by code its author will never see, and that the class will go on working when it is. If one method of the class calls another, a subclass replacing the second has changed the first without touching a line of it.

The middle course: a class may be open and still protect the few methods the rest of it relies on. Mark those final and leave the others alone, so a subclass adapts what was meant to be adapted while the calculation the class depends on stays as written.

What closing genuinely costs: a final class forecloses uses nobody thought of, including safe ones, and the reader who needs one must copy the class and maintain a second version. That price is paid by somebody who never made the decision.

The verdict: close what you have not designed for extension, and where extension is part of what the class is for, leave it open and say in writing which methods may be replaced and which of them the class calls itself. A method that other methods of the same class depend on should be final even in an open class, because that is where a silent change does its damage.

How to argue it in an answer: state what final forbids in each position, give the cost on both sides, and let the deciding condition be whether an extension point was designed and written down.

Extension is a service a class offers, and one not built to offer it should say so in the word the compiler enforces.

The Master Key: *final on a method forbids overriding, final on a class forbids extending; close what you did not design to be opened.*

Abstraction: abstract classes, interfaces and polymorphism

This section takes abstraction from the keyword to the running program. It settles what abstract does to a class and to a method, what an interface is and what Java writes into one for you, and what changed at Java 8 when interface methods were allowed bodies. It then names both kinds of polymorphism precisely and shows dynamic dispatch choosing a method body while the program is already running, before putting instanceof and the downcast in place together with the exception a careless cast throws. It closes on the two judgements a candidate is most often asked to reach: when to prefer an interface to an abstract class, and whether Java loses anything by refusing a class two parents.

BINDER Question 73

The record cards at the Mlandizi dispensary carry three printed lines and a fourth left blank, because what belongs there depends on the patient. The form is useful and unfinished on purpose. Java has a keyword for a class built that way.

Explain what the keyword abstract does when it is applied to a class and when it is applied to a method. State what an abstract class may contain, give the rule that binds any class holding an abstract method, and say what a subclass must do about the abstract methods it inherits.

Model Answer

Applied to a class, abstract means the class may not be instantiated but may be extended; applied to a method it means the method is declared without a body, and any class holding such a method must itself be declared abstract.

```
abstract class Card {
    private final String patient;
    protected Card(String patient) {
        this.patient = patient;
    }
    public String heading() {
        return "Card for " + patient + ": " + detail();
    }
    public abstract String detail();
}
class Visit extends Card {
    private final int day;
    Visit(String patient, int day) {
        super(patient);
        this.day = day;
    }
    @Override
    public String detail() {
        return "seen on day " + day;
    }
}
public class Dispensary {
    public static void main(String[] args) {
        Card card = new Visit("Neema", 12);
        System.out.println(card.heading());
    }
}
```

Output

```
Card for Neema: seen on day 12
```

Program 1.67: *An abstract class with a field, a constructor, a concrete method and an abstract method, and the one subclass that completes it.*

What an abstract class may hold: everything an ordinary class may hold. Card has a private field on line 2, a constructor on line 3, a concrete method with a body on line 6 and an abstract method on line 9. The one thing it may not do is be handed to new.

Why an abstract class has a constructor at all: not to make a Card, because no Card can be made. It is there to be called by a subclass constructor through super, as line 14 does, so that the field the abstract class declares is initialised before the subclass sets its own.

The rule most answers leave out: a class containing an abstract method must itself be declared abstract. Take the word abstract off line 1 and leave line 9 as it stands, and javac reports that Card is not abstract and does not override abstract method detail in Card, which is the compiler saying the class cannot keep its own promise.

What the subclass owes: a body for every abstract method it inherits, or the word abstract on its own declaration so that the debt passes to the class below it. Visit supplies detail on lines 17 to 20, so Visit is complete and line 24 may make one.

Why the concrete method works: heading on line 6 calls detail, and detail has no body anywhere in Card. The call compiles because Card declares the method and it runs because the object is a Visit, so the shared work is written once and one hole is left in it.

An abstract class is a partly written class, and the part left out is the part only a subclass can know.

The Master Key: *abstract on a class forbids new; abstract on a method forbids a body and forces the class to be abstract too.*

BINDER Question 74

The agent at the Kidatu mobile money kiosk handles deposits, withdrawals and bill payments, and her ledger asks the same two questions of all three. How each transaction answers them is not the ledger's business. Java has a construct that says only that.

Explain what an interface is and what implements does. State the modifiers Java applies to an interface method and to an interface field when you do not write them, and say what an interface has been allowed to contain since Java 8.

Model Answer

An interface is the list of methods a class promises to supply, declared with interface and taken on with implements; its methods are implicitly public and abstract and its fields implicitly public, static and final.

```
 1  interface Payable {
 2      int CHARGE = 500;
 3      long amount();
 4      default long total() {
 5          return amount() + fee();
 6      }
 7      private long fee() {
 8          return CHARGE;
 9      }
10      static String heading() {
11          return "Kiosk payments";
12      }
13  }
14  class Transfer implements Payable {
15      private final long sent;
16      Transfer(long sent) {
17          this.sent = sent;
18      }
19      @Override
20      public long amount() {
21          return sent;
22      }
23  }
24  public class Kiosk {
25      public static void main(String[] args) {
26          Payable payment = new Transfer(20000);
27          System.out.println(Payable.heading());
28          System.out.println(payment.amount() + " plus " + Payable.CHARGE
29                  + " is " + payment.total());
30      }
31  }
```

Output

```
Kiosk payments
20000 plus 500 is 20500
```

Program 1.68: *An interface holding a constant, an abstract method, a default method, a private method and a static method, with one class implementing it.*

Why you must write public on the implementing method: because the interface has already made the method public, and an implementing method may never narrow access. Line 20 carries public for that reason alone; leave it off and javac says that weaker access privileges are being attempted, and names public as what was there before.

The fields are constants: CHARGE on line 2 is public, static and final whether those three words are written or not. It cannot be assigned to, it belongs to the interface rather than to any object, and line 28 reads it as Payable.CHARGE.

What Java 8 changed: an interface may declare default methods and static methods with bodies, and since Java 9 private methods as well. Lines 4, 7 and 10 are all bodies inside an interface and all compile under Java 17. Students are often taught, without qualification, that an interface method has no body, which was true until Java 8 and is wrong now.

What a default method is for: it hands every implementing class a working version that the class may accept or override. total on line 4 is written once and serves every kind of payment, because it calls amount, which each class supplies for itself.

One class, many interfaces: a class extends at most one class and implements as many interfaces as it needs. That is what interfaces are for: an object may be bound by several contracts at once while still belonging to only one family.

What you write in an interface	What the compiler declares	What you may do with it
int CHARGE = 500;	public static final	Read it as Payable.CHARGE. You may never assign to it
long amount();	public abstract	Every implementing class must give it a body
default long total() { }	public, and it keeps its body	Inherited as it stands, or overridden
private long fee() { }	private, from Java 9 onward	Called only by the other methods of the interface
static String heading() { }	public static	Called as Payable.heading(). It is never inherited

Table 1.16: *The modifiers you do not write and the compiler supplies. Every one of them is examinable, and none of them is visible in the source.*

An interface states what a class must be able to do, and since Java 8 it may also say how, for the part that is the same everywhere.

The Master Key: *Interface methods are public and abstract unless written otherwise; interface fields are public, static and final always.*

BINDER Question 75

One line printed two different things on two passes of the same loop, and nothing in it had changed. What differed was the object at the far end of the reference, and the machine chose which body to run while the program ran.

Name and define the two kinds of polymorphism in Java. Explain what dynamic dispatch is, and state exactly what the type of a reference decides and what the type of the object decides.

Model Answer

Compile-time polymorphism is method overloading, resolved by the compiler from the argument types at the call; run-time polymorphism is an overridden method reached through a superclass-typed or interface-typed reference, resolved during execution by the class of the object.

Compile-time polymorphism: method overloading, which you have already met. Several methods share one name and differ in the number or the types of their parameters, and the compiler picks between them from the arguments written at the call. Nothing about it is left for the running program to decide.

Run-time polymorphism: an overridden method reached through a reference of a supertype, which is where overriding earns its keep. The call is written against the type of the reference, the body that runs comes from the class of the object, and no compiler can know that class in advance, because the same reference may name a different object on the next pass.

Dynamic dispatch is the mechanism: at every call made through a reference, the JVM looks at the class of the object the reference is naming at that moment and runs that class's version of the method. The lookup happens on each call rather than once, which is how one line inside a loop gives a different answer on each pass.

The sentence that carries the whole idea: the reference type decides what will compile and the object type decides which body runs. A Fee reference will not let you call penalty, because Fee does not declare it; a call to total through that same reference runs the LateFee body, because the object is a LateFee.

Where the mark is lost: in calling overriding compile-time polymorphism, and in defining polymorphism as one name taking many forms and stopping there. Name both kinds, say who chooses the body, and say at which moment the choice is made.

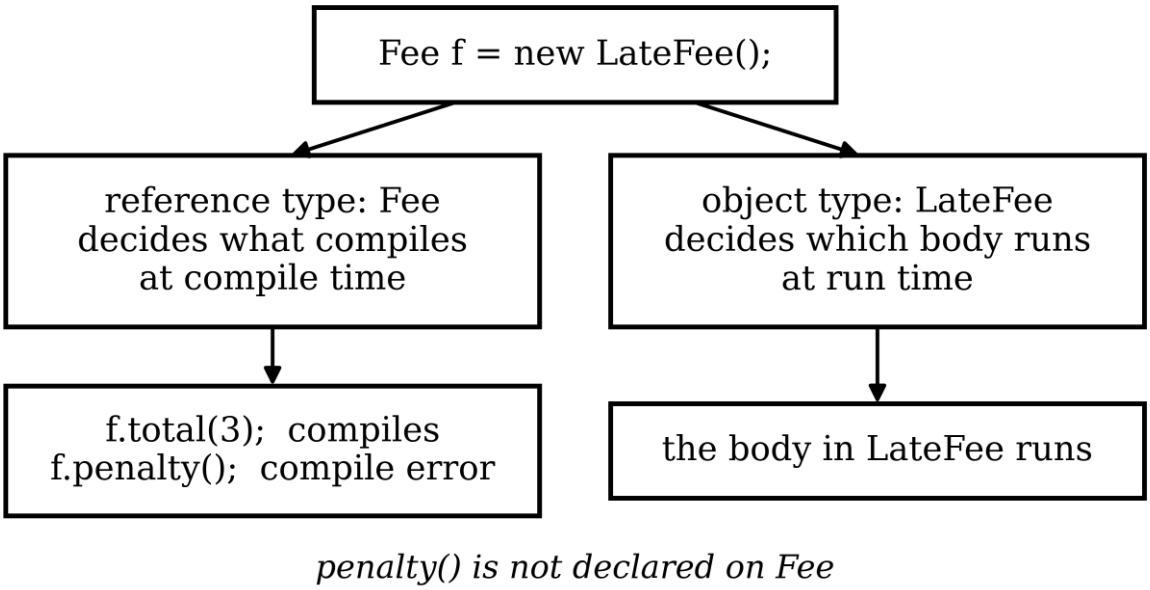


Figure 1.21: *One declaration, two types, and two different questions answered by them. The reference type is consulted by the compiler and the object type by the JVM.*

Question	Compile-time polymorphism	Run-time polymorphism
What it is	Method overloading	Method overriding
Told apart by	The number and the types of the parameters	The class of the object
Chosen by	The compiler	The JVM
Chosen when	While the program is compiled	Each time the call is reached
Reached through	A reference of any type	A supertype reference or an interface

Table 1.17: *The two kinds side by side. The row that decides most answers is the one naming who chooses.*

Two kinds and two moments: one is settled before the program starts and the other cannot be settled until it is running.

The Master Key: *The reference type decides what compiles; the object type decides which body runs.*

REAL Question 76

Three errors came back from a listing with one mistake in it. The tally sheet from the Lugoba fish landing had been typed into an array, the line that picks out the site names would not compile, and the compiler wanted to talk about semicolons.

A type test written as two words. Name the fault, state exactly what javac makes of the line and what it reports, and correct it.

Model Answer

The fault is on line 6, where instanceof has been written as two words; instance of is not an operator, so javac reads it as the declaration of a variable called of and reports three punctuation errors, none of which names instanceof at all.

```
 1  public class Tally {
 2      public static void main(String[] args) {
 3          Object[] entries = {"Lugoba", 48, "Mombo"};
 4          int sites = 0;
 5          for (Object entry : entries) {
 6              boolean isSite = entry instance of String;
 7              if (isSite) {
 8                  sites = sites + 1;
 9              }
10          }
11          System.out.println("sites named: " + sites);
12      }
13  }
```

The compiler says

```
Tally.java:6: error: ';' expected
            boolean isSite = entry instance of String;
                                    ^
Tally.java:6: error: ';' expected
            boolean isSite = entry instance of String;
                                              ^
Tally.java:6: error: not a statement
            boolean isSite = entry instance of String;
                                              ^
3 errors
```

Program 1.69: *A type test written as two words, and the three errors it draws.*

What instanceof is: one reserved word, one operator, and no space anywhere in it. It takes a reference on the left and a type on the right, and it yields true when the object is of that type or of a subtype of it, and false otherwise, including when the reference is null.

What javac made of the line: it read entry as the whole of the initialiser and wanted the declaration to end there, which is the first error and the first caret. It then read instance of as a new declaration, of a variable called of whose type is called instance, and wanted a semicolon after it. String was left standing alone, and a type name by itself is not a statement.

Why no message names the operator: because nothing in the line spells it. Both instance and of are ordinary identifiers, either of which could legally be a type or a variable, so the compiler has no reason to guess at an operator the author did not write. A parse error reports where the parser gave up, not what the author meant.

What it costs: the whole compilation. Three errors are reported, no class file is produced, and not one line of the program runs, however correct the other twelve lines are. A fault the compiler catches is cheap to fix and a fault it cannot catch is not, which is the one consolation here.

The correction: close the gap, so that line 6 reads entry instanceof String. With that single change the listing compiles clean and prints sites named followed by 2, because two of the three entries in the array on line 3 are Strings and the number 48 is not.

One space turned an operator into two identifiers, and the compiler reported the punctuation it wanted rather than the word you meant.

The Master Key: *instanceof is a single word; written apart it becomes two identifiers and the errors never mention it.*

REAL Question 77

One line undid the design of the stock book at the Mombo pharmacy, a class for every kind of medicine. The programmer had asked Java for a medicine that was not any particular medicine. The compiler refused, naming the class as it did.

The listing below will not compile. Name the fault, state exactly what javac reports and where, and give two corrections: one that keeps the abstract class and one that does not.

Model Answer

The fault is on line 13, where new is applied to an abstract class; Medicine declares a method with no body, so no Medicine object can exist, and javac refuses the line with the message that Medicine is abstract and cannot be instantiated.

```
 1  abstract class Medicine {
 2      private final String name;
 3      protected Medicine(String name) {
 4          this.name = name;
 5      }
 6      public String name() {
 7          return name;
 8      }
 9      public abstract int reorderLevel();
10  }
11  public class StockBook {
12      public static void main(String[] args) {
13          Medicine medicine = new Medicine("Paracetamol");
14          System.out.println(medicine.name() + " reorder at "
15                  + medicine.reorderLevel());
16      }
17  }
```

The compiler says

```
StockBook.java:13: error: Medicine is abstract; cannot be instantiated
        Medicine medicine = new Medicine("Paracetamol");
                            ^
1 error
```

Program 1.70: *An abstract class handed to new.*

Why the language forbids it: reorderLevel on line 9 has no body anywhere in Medicine. If an object of Medicine could be made, the call on line 15 would have nothing to run, so the ban is not a matter of style. It is the only way the promise on line 9 can be kept.

What javac reports and where: one error, at line 13, where new stands, and not at line 1 where the class is declared. The message names the class and not the method, so the reader is sent to the line that asked for the impossible rather than to the design that made it impossible.

The correction that keeps the abstract class: write a subclass that supplies a body for reorderLevel and hand new that instead. The variable on line 13 may stay declared as Medicine, because a superclass-typed reference is free to hold a subclass object, and the rest of the program need never learn which kind of medicine it has.

The correction that removes it: if there is in truth no method that only a subclass can write, take abstract off both line 1 and line 9 and give reorderLevel a body. That is the honest answer where the base class has nothing to leave out, and it is the wrong answer where it has.

What an abstract type may still be used for: everything except new. You may declare a variable of it, take it as a parameter, return it from a method and fill an array with references of that type, and every one of those uses is the ordinary way an abstract class earns its keep.

The word behind the keyword: abstraction was settled earlier as exposing only what a user of a class needs and hiding how it is done, with encapsulation as the access control that makes it possible. An abstract class is one way of writing an abstraction down: StockBook can hold a Medicine and use it without ever learning which medicine it holds.

The class is abstract because one method was left unwritten, and new is refused for exactly as long as that stays true.

The Master Key: *An abstract class may be extended, declared and passed about freely, and may never be given to new.*

REAL Question 78

The program printed one line, then stopped with a block of text naming two classes. The cooperative maize ledger at Mang'ula holds two kinds of load, one sorted by quality and one not. The code was written as though it held only the first.

Program 1.71 compiles without a warning and then fails while it is running. Trace what the JVM does at line 27, explain why the compiler allowed a line the JVM refuses, and show the guard that prevents it.

Model Answer

Line 27 casts a Load reference that is naming a Beans object down to Maize; the compiler allows it because some Load could be a Maize, and the JVM refuses it because this one is not, throwing ClassCastException and ending the program where it stands.

```
 1  abstract class Load {
 2      private final int bags;
 3      protected Load(int bags) {
 4          this.bags = bags;
 5      }
 6      public int bags() {
 7          return bags;
 8      }
 9  }
10  class Maize extends Load {
11      Maize(int bags) {
12          super(bags);
13      }
14      public int grade() {
15          return 1;
16      }
17  }
18  class Beans extends Load {
19      Beans(int bags) {
20          super(bags);
21      }
22  }
23  public class Ledger {
24      public static void main(String[] args) {
25          Load load = new Beans(40);
26          System.out.println("bags: " + load.bags());
27          Maize maize = (Maize) load;
28          System.out.println("grade: " + maize.grade());
29      }
30  }
```

The running program says

```
bags: 40
Exception in thread "main" java.lang.ClassCastException:
        class Beans cannot be cast to class Maize (Beans and Maize are
        in unnamed module of loader 'app')
    at Ledger.main(Ledger.java:27)
```

Program 1.71: *A downcast that compiles and then throws.*

Upcast and downcast: assigning a subclass object to a superclass reference is an upcast, is always safe and needs nothing written, which is what line 25 does. Moving back the other way, from the general type to the particular one, is a downcast, must be written out in brackets, and is checked while the program runs.

What the compiler checks before allowing it: only that the cast is possible. Maize is a Load, so a Load reference might be naming a Maize and line 27 is worth compiling. A cast of the same reference to String would be rejected outright, because no object can be both a Load and a String and the compiler can see that from the declarations.

What the JVM does at line 27: it takes the object the reference is actually naming, finds that its class is Beans, asks whether Beans is Maize or a subclass of Maize, and finds that it is not. It then throws ClassCastException,

so no assignment happens, line 28 never runs and main ends abruptly. The line printed above the exception locates the failure exactly.

The guard that prevents it: instanceof, tested before the cast is written. Program 1.72 asks the question on line 27, casts only inside the branch where the answer was true, and prints something sensible in the branch where it was false, so the program now runs to its end and prints two lines.

Why an upcast can never fail: because every Maize is a Load by construction, and no checking is needed for something the compiler has already proved. Only the downcast can be wrong, and only the downcast is checked, which is why a cast that never throws costs nothing and a cast that might throw costs a test.

The newer form, in one sentence: since Java 16 the test and the cast may be written together as if load instanceof Maize maize, which declares the Maize variable and binds it inside the branch; recognise it when you meet it, and write the two steps of Program 1.72 until you do.

```
 1  abstract class Load {
 2      private final int bags;
 3      protected Load(int bags) {
 4          this.bags = bags;
 5      }
 6      public int bags() {
 7          return bags;
 8      }
 9  }
10  class Maize extends Load {
11      Maize(int bags) {
12          super(bags);
13      }
14      public int grade() {
15          return 1;
16      }
17  }
18  class Beans extends Load {
19      Beans(int bags) {
20          super(bags);
21      }
22  }
23  public class Ledger {
24      public static void main(String[] args) {
25          Load load = new Beans(40);
26          System.out.println("bags: " + load.bags());
27          if (load instanceof Maize) {
28              Maize maize = (Maize) load;
29              System.out.println("grade: " + maize.grade());
30          } else {
31              System.out.println("not a maize load");
32          }
33      }
34  }
```

Output

```
bags: 40
not a maize load
```

Program 1.72: *The same classes, with the cast guarded.*

The compiler asks whether the cast could ever succeed and the JVM asks whether this one does, which is why the two disagree.

The Master Key: *An upcast is always safe; a downcast is checked at run time and throws ClassCastException when the object is not of that class.*

BUILD Question 79

Two prices are called at the Mikumi bus stand every morning, one for adults, one for students in uniform. The clerk works both totals for the same seats on the back of a ticket, and one is wrong before it reaches the board.

Write a complete Java program that reports both fares for one day's seats to the specification below, and say where the specification is met.

The class Fare is abstract. It holds a concrete method total, which takes the number of seats, and an abstract method rate.

Two subclasses complete it: Adult at 2500 shillings a seat and Student at 1500.

The number of seats sold is read from the keyboard.

Both fares are held in one array whose element type is Fare.

The program prints, for each element of that array, its rate and its total for the seats read.

Model Answer

The specification asks for one abstract class, two subclasses, one array of superclass references and one loop, and the program below meets it in thirty-two lines, with a single call site on lines 28 and 29 producing two different answers.

```
 1  import java.util.Scanner;
 2  abstract class Fare {
 3      public int total(int seats) {
 4          return seats * rate();
 5      }
 6      public abstract int rate();
 7  }
 8  class Adult extends Fare {
 9      @Override
10      public int rate() {
11          return 2500;
12      }
13  }
14  class Student extends Fare {
15      @Override
16      public int rate() {
17          return 1500;
18      }
19  }
20  public class Desk {
21      public static void main(String[] args) {
22          Scanner in = new Scanner(System.in);
23          System.out.println("Seats sold?");
24          int seats = in.nextInt();
25          in.close();
26          Fare[] fares = {new Adult(), new Student()};
27          for (Fare fare : fares) {
28              System.out.println(fare.rate() + " a seat, total "
29                      + fare.total(seats));
30          }
31      }
32  }
```

Typed at the program

```
40
```

Output

```
Seats sold?
2500 a seat, total 100000
1500 a seat, total 60000
```

Program 1.73: *One array of Fare references, holding two different kinds of fare.*

Where the specification is met: the abstract class is lines 2 to 7, the two subclasses are lines 8 to 19, the array of Fare references is line 26 and the loop is lines 27 to 30. After line 26 neither object is ever named by its own class again, and the loop asks each of them the same two questions.

The one design decision worth naming: total is concrete and rate is abstract, so the arithmetic is written once in the base class and each subclass supplies only the number that differs between them. A third fare costs one class of six lines and changes nothing that already works.

What the output proves: the two printed lines came from one call site. The compiler saw a Fare and allowed the calls on that evidence; the JVM ran the Adult body on the first pass and the Student body on the second, because those are the objects the array is holding.

One loop, one pair of calls, and two answers that the loop itself never chose between.

The Master Key: *An array typed by the superclass holds subclass objects, and each one answers with its own body.*

BUILD Question 80

Can one order be two things at the same time? The question came up over an order book where every garment is priced and traced when the customer comes back. A class may extend only one class: the wrong shape for the job.

Write a complete Java program that treats one order as two things at once, to the specification below, and say where the specification is met.

The interface Priced declares one method, price, taking the number of garments.

The interface Tracked declares one method, reference, and one default method, tag, built from it.

One class, Shirt, implements both interfaces.

The number of garments is read from the keyboard.

The one Shirt object is held through a Priced reference and through a Tracked reference, and one line is printed through each.

Model Answer

The specification asks for two interfaces, one class implementing both and one object held under each of the two interface types, and the program below meets it in thirty-three lines, where lines 28 and 29 name the same object twice.

```
 1  import java.util.Scanner;
 2  interface Priced {
 3      int price(int garments);
 4  }
 5  interface Tracked {
 6      String reference();
 7      default String tag() {
 8          return "order/" + reference();
 9      }
10  }
11  class Shirt implements Priced, Tracked {
12      @Override
13      public int price(int garments) {
14          return garments * 12000;
15      }
16      @Override
17      public String reference() {
18          return "SHIRT";
19      }
20  }
21  public class OrderBook {
22      public static void main(String[] args) {
23          Scanner in = new Scanner(System.in);
24          System.out.println("Garments?");
25          int garments = in.nextInt();
26          in.close();
27          Shirt shirt = new Shirt();
28          Priced priced = shirt;
29          Tracked tracked = shirt;
30          System.out.println(tracked.tag());
31          System.out.println("shillings: " + priced.price(garments));
32      }
33  }
```

Typed at the program

```
3
```

Output

```
Garments?
order/SHIRT
shillings: 36000
```

Program 1.74: *One object, two interfaces, and a default method that neither implementing class had to write.*

Where the specification is met: Priced is lines 2 to 4, Tracked is lines 5 to 10 with its default method on line 7, and line 11 takes both on at once. Lines 28 and 29 make two references of two different interface types, and both of them name the single object made on line 27.

What each reference can see: priced may call price and may not call tag; tracked may call tag and may not call price. The object has both methods and each reference exposes only its own contract, which is the reference type deciding what compiles, exactly as it does through a superclass.

The one design decision worth naming: tag is a default method rather than a method every implementing class must write, because the shape of a tag is the same wherever it is used and only the reference inside it differs. A class needing a different tag is still free to override it.

One object, two contracts, and neither contract able to see the other through its own reference.

The Master Key: *A class may implement many interfaces, and each interface-typed reference sees only what that interface declares.*

HOT Question 81

The argument over one class in a printing shop lasted an afternoon; both students were right. One wanted an abstract class: both kinds of job shared most of their working. The other, an interface: the takings ledger never cared what a job was.

A design needs one type that several classes will share. Evaluate the choice between an abstract class and an interface, and reach a judgement, naming the condition that decides it.

Model Answer

Choose an abstract class when the classes sharing the type also share state and working code, and choose an interface when what they share is a contract only, because a class may implement any number of interfaces and may extend exactly one class.

What only an abstract class can do: hold state. It may declare instance fields, hold a constructor that initialises them, and offer concrete methods that use them, so every subclass inherits the data and the working together. An interface has no instance fields and no constructor, so it can carry no per-object state at all.

What only an interface can do: be one type among many. A class may implement as many interfaces as it needs, and may do so while extending some class of its own, so one class can satisfy the ledger and the printer at the same time without either of them being its parent.

Where the two now overlap: since Java 8 an interface may carry default and static methods with bodies, so shared behaviour is no longer the dividing line it once was. What still divides them is shared state, and that is the test to apply.

The price of choosing the abstract class wrongly: you have spent the single inheritance slot. Every subclass is now committed, and if one of them later needs to extend something else the whole family has to be restructured, which is the most expensive kind of change to make late.

The price of choosing the interface wrongly: every implementing class repeats the fields and the constructor work that one abstract class would have held once. The copies drift apart as the program grows, and the bug that follows appears in one class and not in the others that look identical.

The judgement: prefer the interface unless the classes genuinely share state as well as a contract. Where they share both, use both, an interface to state what the type promises and an abstract class implementing that interface to hold what the subclasses have in common, which is why the standard library ships AbstractList beside List.

How to argue it in an answer: name the thing that decides, which is shared state, then price both mistakes. An answer that reaches a condition earns the marks that an answer listing differences does not.

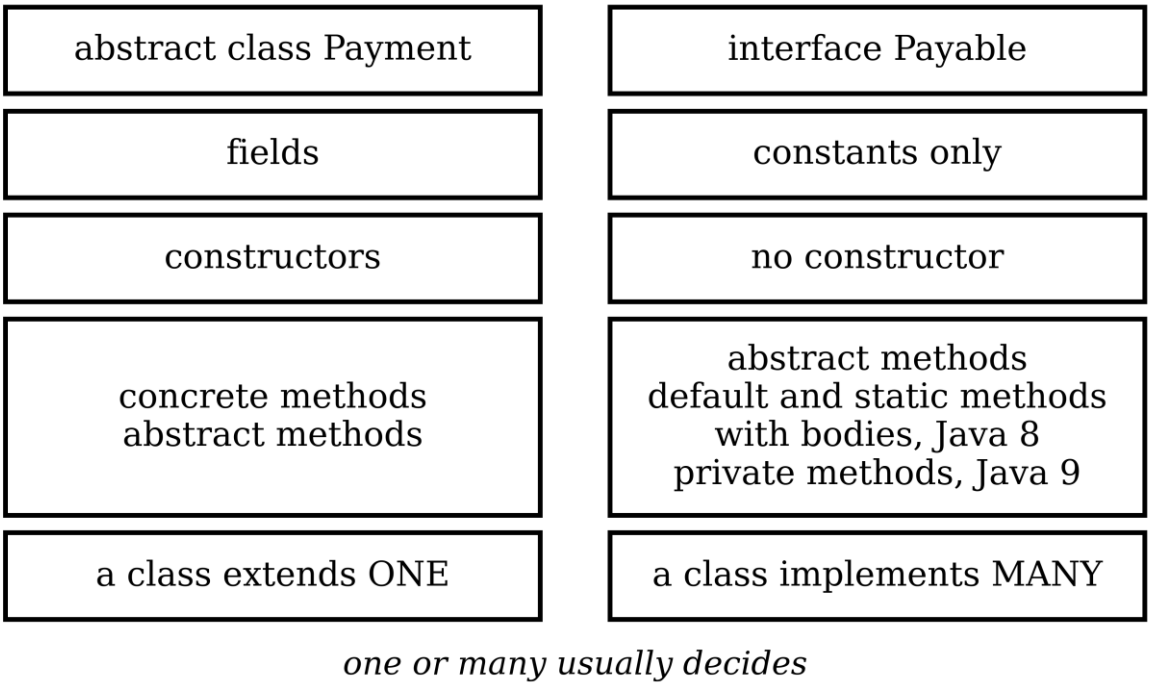


Figure 1.22: *What each of the two constructs may hold. Only the abstract class carries fields and a constructor, and only the interface may be taken on many times over by one class.*

Ask what the classes share; if it is only a promise, an interface says so, and if it is also data, an abstract class says more.

The Master Key: *Interface for a contract, abstract class for shared state, and both together when the design needs each.*

HOT Question 82

It goes up on classroom boards: Java does not support multiple inheritance. The sentence is half a rule. The missing half explains why a class may implement as many interfaces as it likes, and why one may carry a method body.

Java forbids a class to extend two classes and allows it to implement any number of interfaces. Evaluate whether that restriction costs anything real, now that an interface may carry default methods, and reach a judgement about the diamond problem.

Model Answer

The restriction is on inheriting state, not on inheriting types, and Java gives up very little by it, because an interface can supply behaviour through a default method and the one ambiguity that survives is refused by the compiler rather than settled by a rule the programmer must remember.

What the diamond problem is: one class inheriting the same member along two paths. Where the member is a field, the question is how many copies the object holds; where it is a method, the question is which body runs. C++ answers both with rules the programmer must know and apply, and Java answers by not allowing the shape.

How Java removes the harder half: a class extends exactly one class, so there is one chain of fields and one order of construction, and no object can end up holding two copies of anything. Interfaces declare no instance fields, so no number of them can put a second copy back.

What Java 8 brought back: default methods, which are inherited bodies, and therefore behaviour arriving along more than one path. Two interfaces may now offer the same method with two different bodies, so the behaviour half of the diamond is real again even though the state half is not.

What the compiler does about it: it refuses the class. It reports that the two interface types are incompatible and that the class inherits unrelated defaults for the method from both of them, and it reports it at the class declaration, naming both interfaces. Nothing is chosen silently on the programmer's behalf.

How the programmer resolves it: by overriding the method in the class, as lines 12 to 15 of Program 1.75 do. Inside the override either body may still be reached by naming its own interface, as line 14 does twice, so the choice is written down where a reader can see it.

What is genuinely lost: implementation cannot be inherited from two classes, so a class needing the working of two different bases must hold one of them as a field and pass the calls on to it. That is more typing, and it is also a plainer statement of what the object actually is.

The judgement: the restriction earns its price. Ambiguity of state is made impossible rather than merely difficult, ambiguity of behaviour is caught at compile time and must be resolved in writing, and the multiple typing that programmers actually want is still available through interfaces.

```
 1  interface Billed {
 2      default String label() {
 3          return "bill";
 4      }
 5  }
 6  interface Filed {
 7      default String label() {
 8          return "file copy";
 9      }
10  }
11  public class Receipt implements Billed, Filed {
12      @Override
13      public String label() {
14          return Billed.super.label() + " and " + Filed.super.label();
15      }
16      public static void main(String[] args) {
17          Receipt receipt = new Receipt();
18          System.out.println(receipt.label());
19      }
20  }
```

Output

```
bill and file copy
```

Program 1.75: *Two interfaces offering the same default method, and the class that settles between them.*

Java refused the half of multiple inheritance that cannot be made safe, kept the half that can, and left the compiler to insist on an answer where the two collide.

The Master Key: *One class, many interfaces: state is inherited once, and two competing defaults must be resolved by hand or the class will not compile.*

Exceptions, testing and building a complete Java application

This section is where the chapter comes together. It defines an exception and separates it from a compile error, draws the Throwable tree so that checked and unchecked can be read off a position in it rather than guessed from how serious a fault looks, and gives try, catch, finally, throw, throws and try-with-resources each their own job. It then reads a real stack trace to find the line that caused it, tests a method at its boundaries, and assembles two classes into an application that runs from typed input to printed report.

BINDER Question 83

The program compiled without a word of complaint, and the class at Kibaigwa took that as a verdict. It ran for a second and stopped with seven lines of text that nobody had written. None of the seven was a syntax error.

Define an exception and distinguish it from a compile error. Explain how Java decides that an exception is checked or unchecked, and state what that decision requires of a method in which a checked exception may arise.

Model Answer

An exception is an object describing a fault that arises while a program runs, thrown at the point of failure and passed up the chain of calls until something catches it; checked or unchecked is decided by position in the Throwable tree, not by how serious the fault is.

How it differs from a compile error: javac reports a compile error about the text of your program and writes no class file, so nothing runs. An exception belongs to a program that compiled perfectly and has already started: the code was legal, and the data it met was not.

The shape of the statement: The try block holds the code that may fail. Each catch block names one type of exception and says what to do about it. The optional finally block runs on the way out whatever happened, and releases anything left open.

Where the tree makes the decision: Error and its subclasses are unchecked, RuntimeException and its subclasses are unchecked, and everything else below Exception is checked. The mark runs down the branch: IOException is checked because it sits below Exception outside RuntimeException, and ArrayIndexOutOfBoundsException is unchecked because it sits below RuntimeException.

Catch or declare: A method in which a checked exception may arise must either catch it or name it after throws in its header, and javac refuses the method if it does neither. Nothing of the kind is demanded for an unchecked one, because a fault your own code could have prevented is your business.

The two words that are not the same: throw is a statement. It takes one exception object and raises it at that point. throws is part of a method header, takes a list of types, and warns every caller that this method may raise them. One acts, the other declares.

```
 1  public class Meter {
 2      private static int reading(int[] book, int index) {
 3          try {
 4              return book[index];
 5          } catch (ArrayIndexOutOfBoundsException e) {
 6              System.out.println("no reading at " + index);
 7              return -1;
 8          } finally {
 9              System.out.println("finally ran for index " + index);
10          }
11      }
12
13      public static void main(String[] args) {
14          int[] book = {120, 95, 143};
15          System.out.println("got " + reading(book, 1));
16          System.out.println("got " + reading(book, 7));
17      }
18  }
```

Output

```
finally ran for index 1
got 95
no reading at 7
finally ran for index 7
got -1
```

Program 1.76: *The finally block runs on both paths, including the one where the try block has already decided what to return.*

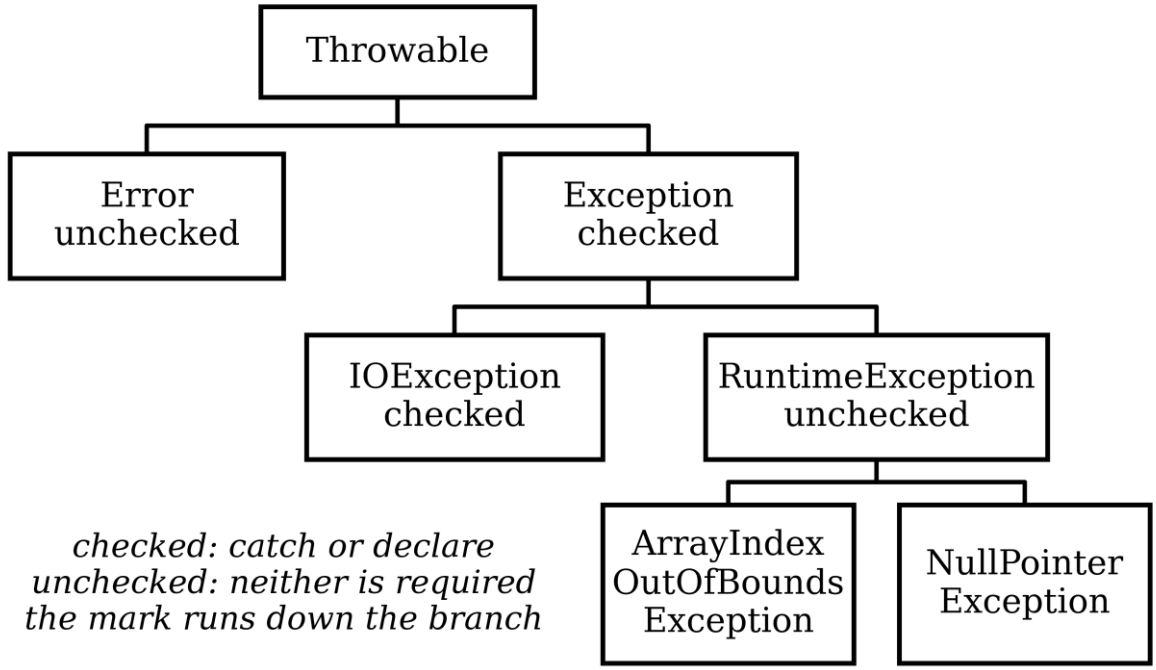


Figure 1.23: *The Throwable tree. Error and RuntimeException carry the unchecked mark down their branches, and everything else below Exception is checked.*

Exception	Checked?	What raises it
ArithmeticException	Unchecked	Integer division by zero
ArrayIndexOutOfBoundsException	Unchecked	An index below zero, or at the length or beyond
NullPointerException	Unchecked	A member reached through a reference holding null
NumberFormatException	Unchecked	Text that is not a number handed to parseInt
InputMismatchException	Unchecked	A Scanner asked for an int and given a word
IOException	Checked	A stream that could not be read or written
FileNotFoundException	Checked	A named file that is not where it was expected

Table 1.18: *The exceptions a Form Five program actually raises. Every unchecked one here descends from RuntimeException, and both checked ones descend from Exception outside it.*

Name the class, find it in the tree, and the rule that governs it follows.

The Master Key: *Position in the Throwable tree decides checked from unchecked; a checked exception must be caught or declared.*

REAL Question 84

A student at Kabuku wrote two catch blocks, both correct in themselves, and put them in the order that felt safest: the general one first, so nothing could slip past. The compiler refused it. The line it named was the second, not the first.

The listing below will not compile. Name the fault, state exactly what javac does about it and why, and correct it. Then say what a finally block added to the same statement would do.

Model Answer

The fault is the order of the two catch blocks: the catch on line 7 names Exception, which matches every exception the try block can throw, so the ArrayIndexOutOfBoundsException block on line 9 can never be entered, and javac rejects a catch block it has proved is unreachable.

```
 1  public class Delivery {
 2      public static void main(String[] args) {
 3          int[] bags = {40, 55, 62};
 4
 5          try {
 6              System.out.println("bag: " + bags[3]);
 7          } catch (Exception e) {
 8              System.out.println("delivery failed");
 9          } catch (ArrayIndexOutOfBoundsException e) {
10              System.out.println("no such bag");
11          }
12      }
13  }
```

Compiler says

```
Delivery.java:9: error: exception ArrayIndexOutOfBoundsException has already been caught
        } catch (ArrayIndexOutOfBoundsException e) {
          ^
1 error
```

Program 1.77: *Two catch blocks in the wrong order, and what javac says about the second one. javac repeats the file name exactly as you gave it on the command line, so a full path appears here only if you typed one.*

What the compiler proved: javac does not merely dislike the order. It compares each catch against the ones written above it, finds that ArrayIndexOutOfBoundsException is a subclass of Exception and so is already handled by line 7, and reports that the exception has already been caught. Nothing about the run is involved; the proof is about reachable code.

The ordering rule: Catch blocks are tried in the order they are written, and the first whose type matches the thrown object wins. A broader type placed above a narrower one therefore swallows it. Write the narrowest first and the broadest last, so that every block you have written can be reached.

The same reasoning by hand: You do not need javac to see this. Ask of each catch block whether any block above it would already have taken the exception, and if one would, the block you are looking at is dead code. That is the whole of the compiler check, and it costs a few seconds to apply while writing.

The correction: Swap the two blocks, so that ArrayIndexOutOfBoundsException comes first and Exception second. Where two unrelated types are handled the same way, a single catch may name both with a vertical bar between them. Best of all, name the exact type you can act on and leave Exception out of the statement.

What finally would do: A finally block added after the last catch runs on all three ways out of the statement: the try block completing normally, the try block throwing where a catch matches, and the try block throwing where none does. In that third case finally runs first and the exception then carries on up to the caller.

Where a resource makes finally unnecessary: A resource opened in the try block used to be closed in finally. Since Java 7 you declare it inside the parentheses of the try instead, and any object whose class implements AutoCloseable is closed automatically however the block ends. That is the deterministic replacement for the finaliser, which was never guaranteed to run at all.

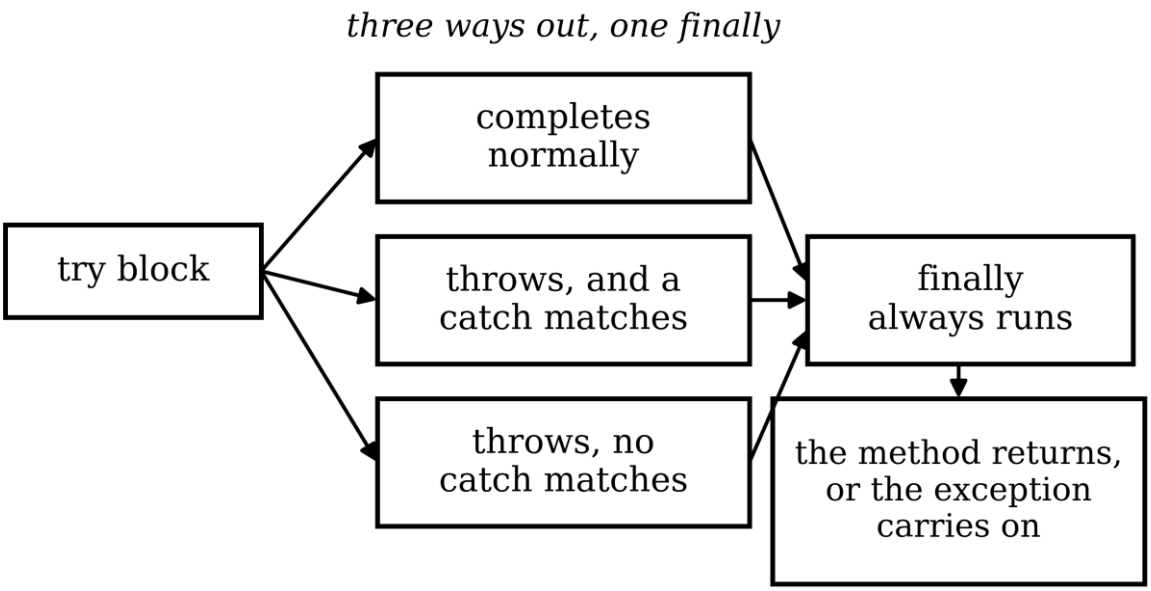


Figure 1.24: *Three ways out of a try block and one finally block, which runs on every one of them before control leaves.*

The compiler is not being fussy about your ordering; it has proved that a block you took the trouble to write can never run.

The Master Key: *Catch the narrowest type first, because a broader catch above a narrower one makes the narrower one unreachable and javac says so.*

REAL Question 85

Seven lines of text came back from the weighbridge program at Amani, and the operator read only the first. Three lines after it name code nobody at the estate had written. The line that mattered was the fifth, carrying a line number.

The listing below compiles cleanly and stops during the run. Read the trace printed beside it. Name the exception and the line of the program that caused it, explain why three lines of the trace name code that is not yours, and correct the program.

Model Answer

The exception is a NumberFormatException, raised because the third entry is not a number, and the line to open first is the topmost line of the trace that names your own class, which is line 3 of Tally.java inside massOf.

```
 1  public class Tally {
 2      private static int massOf(String entry) {
 3          return Integer.parseInt(entry);
 4      }
 5
 6      private static int total(String[] entries) {
 7          int sum = 0;
 8          for (String entry : entries) {
 9              sum += massOf(entry);
10          }
11          return sum;
12      }
13
14      public static void main(String[] args) {
15          String[] entries = {"48", "51", "12kg"};
16          System.out.println("total: " + total(entries));
17      }
18  }
```

The program stops with

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "12kg"
    at
java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)
    at java.base/java.lang.Integer.parseInt(Integer.java:662)
    at java.base/java.lang.Integer.parseInt(Integer.java:778)
    at Tally.massOf(Tally.java:3)
    at Tally.total(Tally.java:9)
    at Tally.main(Tally.java:16)
```

Program 1.78: *A conversion two calls below main, and the trace the run leaves behind. The trace carries no directory names, so this is what you will see.*

The three parts of the first line: Exception in thread main names the thread that died. java.lang.NumberFormatException is the type, and the type alone tells you the family of fault. What follows the colon is the message the exception object carried, and here it reproduces the exact text that could not be converted.

Reading the trace in the right direction: The trace lists calls innermost first. The bottom line is main, where the program began, and each line above it is a call made by the line below. The top line is where the throw actually happened. Reading upward from the bottom retraces the route the program took to reach the fault.

Where the library ends and your program begins: The first three lines beginning at carry java.base and belong to Integer.parseInt, which is not code you can correct and not code that is wrong. The first line naming Tally is the point at which your program handed bad data to a method that was working exactly as documented.

What the numbers are for: Each line beginning at ends with a source file and a line number in brackets. Tally.java line 3 is the return statement inside massOf, and Tally.java line 9 is the call that reached it. The two together give you the failing statement and its caller in one reading, without a debugger.

The correction: Guard the conversion where it happens. Put the parseInt inside a try block in massOf, catch NumberFormatException, and either skip the entry after printing which one it was, or throw a clearer exception naming the entry so that the caller can decide. Silently returning zero would make the total wrong and say nothing.

The habit that shortens the search: Never read only the first line. Find the topmost line naming a class you wrote, open that file at that number, and most of the search is over before you have looked at any code. A trace is a map of the route to the fault, and it is thrown away because it looks long.

The trace already contains the answer, and the work is knowing which of its lines belong to you.

The Master Key: *Read the type, then the message, then the topmost line naming your own class, and open that file at that line.*

BUILD Question 86

A record card at the Mvumi dispensary is stiff and ruled, an item on one line and a count on the next. Once a wrong count reaches it, every total taken afterwards is wrong with it, and nobody can tell which card did the damage.

Write a Java application in two files to the specification below. State which class carries which responsibility, and give the test that shows the refusal works.

Card is a class in a file of its own, holding one item name and one count of units.

A Card cannot be constructed with a count below zero; it throws IllegalArgumentException.

StockBook holds main, reads pairs of item and count from the keyboard, and stores up to five cards.

A card that is refused is reported by name and does not stop the run.

At the end the program prints each stored card, the number kept, and the total units held.

Test it with paracetamol 240, bandage -5, gloves 0 and quinine 90.

Model Answer

The application is in two files because it has two separate duties: Card decides whether a card is valid at all, and StockBook decides what to do about one that is not, so neither file needs to know how the other reaches its decision.

Card.java

```
 1  public class Card {
 2      private final String item;
 3      private final int units;
 4
 5      public Card(String item, int units) {
 6          if (units < 0) {
 7              throw new IllegalArgumentException(
 8                  "units below zero: " + units);
 9          }
10          this.item = item;
11          this.units = units;
12      }
13
14      public int getUnits() {
15          return units;
16      }
17
18      @Override
19      public String toString() {
20          return item + " " + units;
21      }
22  }
```

StockBook.java

```
 1  import java.util.Scanner;
 2
 3  public class StockBook {
 4      public static void main(String[] args) {
 5          Card[] book = new Card[5];
 6          int kept = 0;
 7
 8          try (Scanner in = new Scanner(System.in)) {
 9              while (in.hasNext() && kept < book.length) {
10                  String item = in.next();
11                  int units = in.nextInt();
12                  try {
13                      book[kept] = new Card(item, units);
14                      kept++;
15                  } catch (IllegalArgumentException e) {
16                      System.out.println("rejected " + item
17                          + ": " + e.getMessage());
18                  }
19              }
20          }
21
22          int total = 0;
23          for (int i = 0; i < kept; i++) {
24              System.out.println(book[i]);
25              total += book[i].getUnits();
26          }
27          System.out.println("Cards kept: " + kept);
28          System.out.println("Units held: " + total);
29      }
30  }
```

Typed at the program

```
paracetamol 240
bandage -5
gloves 0
quinine 90
```

Output

```
rejected bandage: units below zero: -5
paracetamol 240
gloves 0
```

```
quinine 90
Cards kept: 3
Units held: 330
```

Program 1.79: *The record card and the book that keeps it, compiled with javac Card.java StockBook.java and run with java StockBook.*

Where the specification is met: The refusal is the throw on line 7 of Card.java, the recovery is the catch on line 15 of StockBook.java, and the report is lines 22 to 28. Because the constructor refuses, no invalid Card object can exist anywhere in the book, and no later total has to be defended.

The one design decision worth naming: The Scanner is declared inside the parentheses of the try on line 8, which is try-with-resources, so it is closed when the block ends however the block ends. The inner try sits inside the loop, which is what lets a refused card cost one card rather than the rest of the input.

Plan, class design, implementation and test: The plan named three duties, validate, store and report. The design gave validation to Card and the other two to StockBook. The implementation is the two files. The test is the table below, which puts one count above the boundary, one below it and one exactly on it.

Typed at the program	Expected	Printed
paracetamol 240	Stored, well above the boundary	paracetamol 240
bandage -5	Refused, below the boundary	rejected bandage: units below zero: -5
gloves 0	Stored, the boundary itself is valid	gloves 0
quinine 90	Stored	quinine 90

Table 1.19: *The test of the refusal, with the expected result written down before the run and the printed result copied from it afterwards.*

Two files, one refusal, and a total that no invalid card can enter.

The Master Key: *Let the class refuse bad data in its constructor, and let the application decide what to do about the refusal.*

BUILD Question 87

The word was nne, and it sat in the column where a number belonged. The clerk at the Hombolo booking desk had written the count in words without thinking. The old program stopped at that line and lost the eleven entries after it.

Write a complete Java program to the specification below, so that one unreadable entry costs one entry rather than the whole run.

The program reads one entry a line from the keyboard until the input ends.

Each entry should be a whole number of seats.

An entry that is not a number is reported and counted, and reading carries on.

At the end the program prints the total seats booked and the number of entries rejected.

Test it with the entries 14, nne and 9.

Model Answer

The specification asks the program to survive bad input rather than stop at it, and that is settled entirely by where the try block is put: inside the loop, so that a failed conversion ends one pass and not the reading.

```
 1  import java.util.Scanner;
 2
 3  public class TicketDesk {
 4      public static void main(String[] args) {
 5          int seats = 0;
 6          int rejected = 0;
 7
 8          try (Scanner in = new Scanner(System.in)) {
 9              while (in.hasNextLine()) {
10                  String typed = in.nextLine().trim();
11                  try {
12                      seats = seats + Integer.parseInt(typed);
13                  } catch (NumberFormatException e) {
14                      rejected++;
15                      System.out.println("not a number: " + typed);
16                  }
17              }
18          }
19
20          System.out.println("Seats booked: " + seats);
21          System.out.println("Entries rejected: " + rejected);
22      }
23  }
```

Typed at the program

```
14
nne
9
```

Output

```
not a number: nne
Seats booked: 23
Entries rejected: 1
```

Program 1.80: *Reading entries that may not be numbers, and carrying on past the one that is not.*

Where the specification is met: Integer.parseInt on line 12 is the only statement here that can throw, and the try on line 11 wraps it and nothing else. The catch on line 13 counts the entry, names it on the screen and falls to the bottom of the loop, so reading resumes with the next line.

The one design decision worth naming: The try sits inside the while, not around it. Put the try around the whole loop and the first bad entry would jump past every entry remaining, which is exactly the failure the specification was written to prevent. The scope of a try block is a decision, not a formality.

Why an unchecked exception is still worth catching: NumberFormatException descends from RuntimeException, so nothing in the language compels you to catch it here. It is caught because there is something useful to do about it, which is the only good reason to catch anything. The Scanner is closed by try-with-resources either way.

One bad line, one printed complaint, and a total that still means something.

The Master Key: *Put the try where the failure is rather than around the whole program, and one bad entry costs one entry.*

HOT Question 88

Two answers to the same practical question sat one above the other on one page at Mlalo. One caught every exception the moment it appeared and printed a polite message. The other caught little and let faults travel. The second was the better program.

A method has called something that has thrown. Argue when that method should catch the exception and when it should let it propagate to its caller, and reach a judgement.

Model Answer

Catch an exception only where you have enough information to do something useful about it and let it propagate everywhere else, because a method that catches without being able to act destroys the one thing its caller needed: the knowledge that the work did not happen.

The case for catching early: An exception caught at the point of failure is caught where the most is known about it: the value that was wrong, the entry that was being read, the index that was outside the array. A handler further away has to guess at all of that, or work without it.

The case for letting it travel: The method that failed usually cannot decide what should happen next. Whether a bad entry is skipped, retried or fatal is a question of policy, and policy belongs to the caller that knows what the program is for. Propagation hands the decision to the level that is able to take it.

What propagation costs: The exception carries its stack trace, so nothing is lost about where the fault arose, but the further it travels the more half-finished work is abandoned on the way. Anything opened between the throw and the catch stays open unless a finally block or a try-with-resources closed it.

The test that decides most cases: Ask what this method would put in the catch block. If the answer is print a message and carry on as though nothing had happened, do not catch. If the answer is skip this entry and read the next, catch, because only this method is placed to take that decision.

The one form that is never defensible: A catch block with an empty body is the worst thing in this section. It takes the exception object, discards the type, the message and the trace, and returns the program to the state the language had just told you was broken. The fault reappears later as a wrong answer with no evidence attached.

Where wrapping beats both: A method may catch a low-level exception and throw a clearer one naming the operation that failed, passing the original in as the cause. The caller gets a message it can act on and the original trace is still attached underneath it.

The verdict: Catch where you can act, propagate where you cannot, and catch the narrowest type you are able to name. Where a method stands between the fault and the decision, wrap rather than swallow. The condition that decides every case is whether the catch block would contain a real action or an apology.

A caught exception should leave the program in a state somebody chose, not the state it happened to be in.

The Master Key: *Catch where you can act, propagate where you cannot, and never catch into an empty block.*

HOT Question 89

Java is unusual among widely used languages in making the compiler enforce exception handling. C++ tried something in the same spirit and then removed it. The languages designed after Java looked at the feature closely and declined to copy it.

Evaluate the claim that checked exceptions were a mistake in the design of Java. Reach a judgement.

Model Answer

Checked exceptions were a good idea applied far too widely, and the honest verdict is that the mechanism earns its keep where the caller can genuinely recover and becomes noise where it cannot, which is why it is still defended for input and output and attacked almost everywhere else.

The case for the feature: A failure the compiler forces you to acknowledge cannot be quietly forgotten. Reading a file may fail for reasons nothing in your program controls, and a language that lets that possibility go unmentioned produces programs that work only on the machine they were written on.

How the argument runs in practice: The declaration is documentation that the compiler checks. A caller reading throws IOException in a header knows before writing a line that this call has a failure path, and knows it without running anything and without reading the body of the method.

The case against the feature: Enforcement is only useful where recovery is possible. Where it is not, the rule produces the same declaration repeated up every level of the program, or, far worse, a catch block written to silence javac rather than to handle anything. The feature then manufactures the very swallowed exception it existed to prevent.

What the later evidence shows: Java itself moved away from the design. The streams and functional interfaces added in Java 8 do not accommodate checked exceptions comfortably, which is why a great deal of code written since wraps a checked exception in an unchecked one purely to get it through.

Where the line actually falls: The distinction that survives the argument is between a fault the caller can do something about and a fault that means the code itself is wrong. A missing file is the first kind; an index outside an array is the second. Java draws that line with the Throwable tree, and the error was to put too much on the checked side of it.

The verdict: Keep checked exceptions where the caller has a real choice to make, which in practice means input and output and not much else, and use unchecked exceptions for faults that mean the program is wrong. The condition is simple: if the declaration tells the caller something it can act on, the enforcement has paid for itself, and if it does not, it is ceremony.

A rule that forces you to write a handler you have nothing to put inside has not made the program any safer.

The Master Key: *Check what the caller can recover from, leave the rest unchecked, and never catch merely to satisfy the compiler.*

HOT Question 90

At the end of the practical the program ran once, printed what was asked, and was handed in. Nobody had tried it with zero, the array's top index, or a word where a number belonged, and two of the three would have stopped it.

A complete program of the size printed in this section has been written. Decide how much testing is enough for it, explain how a single method should be tested, and reach a judgement.

Model Answer

Enough is every boundary of every method that computes or validates, plus one run of the whole program end to end, because in a program of this size the faults live at the boundaries and in the joins between classes and almost nowhere else.

How to test one method: Choose inputs at the boundary of what the method accepts, one just inside it and one just outside. For a count that may not fall below zero, that is zero and minus one. Write down what you expect before you run it, then set expected against actual, because a result decided afterwards is not a test.

What a unit test is: A unit test is a short piece of code that calls one method with a chosen input, compares the value returned against the value expected, and reports pass or fail without a human reading anything. Frameworks such as JUnit run hundreds of them and report only the failures. The idea is what matters, and a main method with an if statement will do.

Testing the joins: Two classes that are each correct can still be wrong together, if one counts in units and the other expects tens, or if one may return null and the other never checks. Only a run of the whole application catches that.

The case for stopping early: Testing costs time. A program of thirty lines that will be read once and never changed does not repay a test for every path through it, and time spent there is time not spent where the marks are.

The case for going further: Every fault found on your own machine is a fault not found in front of somebody else, and the boundary cases are precisely the ones a marker chooses. A test at zero costs a minute and buys the difference between a program that runs and one that stops.

Debugging by halving: When a program fails and the reason is not visible, do not read it again. Print one value from halfway through and see whether it is already wrong, which halves what is left to search; three halvings take thirty lines down to fewer than four. Staring is not a method.

The verdict: Test every boundary of every method that computes or validates, run the whole program once end to end, and stop there. Go further only when the program will be changed by somebody who did not write it, because that is when a test stops being a check and becomes a description of what the code is for.

A program tested only in the middle of its range has been demonstrated rather than tested.

The Master Key: *Boundaries on every method that computes, one whole run end to end, and bisect rather than stare.*

Chapter Two

OBJECT ORIENTED PROGRAMMING WITH PYTHON

Python is the third language in which you meet object orientation, and the second in this book. You already know what a class is, what an object is, and what the four pillars are called, and nothing here asks you to learn any of that again. What this chapter is for is the change. C++ and Java both hand your program to a compiler that checks it against declared types before a line of it runs; Python does neither of those things. Everything the chapter is worth lies in what happens to object orientation when there is no compiler and no declaration.

The chapter runs in the order the ideas depend on one another. It opens on the interpreter, on indentation as syntax and on the naming rules. Names and objects come next, with the operators and program input, then the collection types and control flow, then functions and modules. Only then do classes, objects and self arrive, followed by encapsulation and abstraction, then inheritance and polymorphism, and the chapter closes on errors, exceptions, files and a working application assembled out of everything before it.

Three habits earn marks in this topic and cost almost nothing to acquire. Name the part before you state the property, because most of the marks lost here are true statements attached to the wrong construct. Say when a fault is found and not only what it is, since a question asking what Python does about an error is asking whether you know that nothing is checked until its line runs. And write indentation as carefully in an answer as you would at a keyboard, because in this language the indent is the block, and an answer indented wrongly has written a different program from the one it is describing.

The interpreter and the shape of a program

This section opens the chapter on the difference that causes every other difference in it. It explains what an interpreter does with a source file and what follows from there being no build step, teaches indentation as syntax rather than as layout, gives the rules that make a name legal, and finishes on pass with the two examples it is usually introduced without. Two figures carry the work: the two routes from source to a running program, and the same four statements at two indent levels.

BINDER Question 1

At a printing shop in Buguruni the machine is set up once, then copies come off it all afternoon. Next door a clerk reads each order aloud to the tailor, line by line, and nothing is set up at all.

Explain what an interpreter does with a source file, and state the consequences for a Python program that do not hold for a compiled one.

Model Answer

An interpreter reads a source file one statement at a time and carries out each statement as it reaches it, so the file you write is the file that runs and no second file is produced along the way.

What the interpreter does with the file: It reads a statement, works out what it means, performs it, and moves to the next. A compiler translates the whole source first and writes that translation out as a file, and that file is what the machine later runs. Python produces no such file.

No build step, and no second file: There is no command to run before the program runs. You write kiosk.py, you type python3 kiosk.py, and that same file is read from the top. Each statement is turned into an internal form as the interpreter works, but the form is held in memory and nothing lands beside your source, so there is nothing to rebuild after an edit.

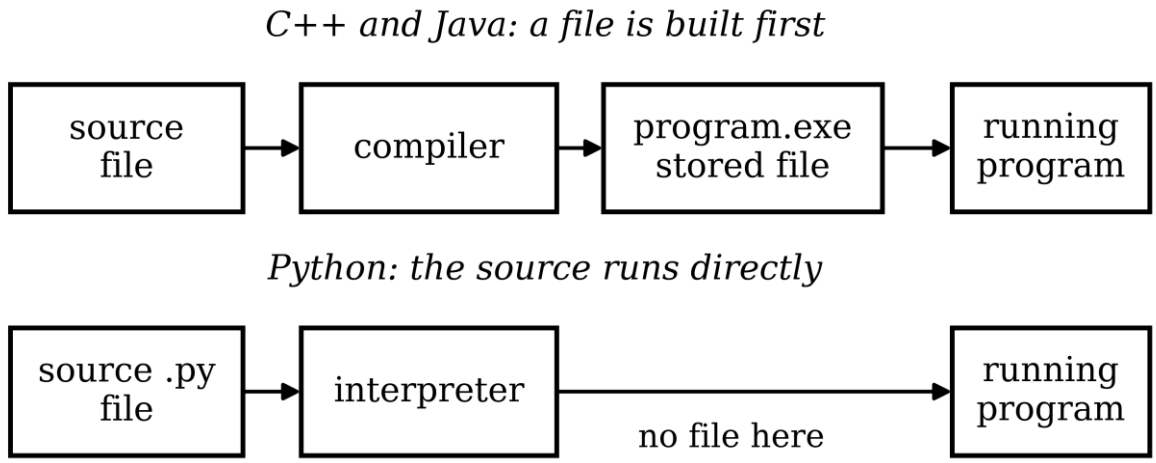


Figure 2.1: *The two routes from source to a running program, and the file that only one of them creates.*

Nothing is checked until its line is reached: A compiler examines every line before the program starts, so a name that was never given a value is reported before anything runs. Python examines a line when it reaches it. A fault far down the file survives everything above it, and the program does real work and prints real output before it stops.

What portability now rests on: A binary built for one processor and one operating system will not run on another, so a compiled program travels as source and is built again on each machine. A Python program travels unchanged and runs wherever an interpreter of a suitable version is installed. The cost has moved rather than gone: that machine must have Python on it.

The interpreter is not a slower compiler but a different arrangement, and every other difference in this chapter follows from it.

The Master Key: *There is no build step, so the file you write is the file that runs, and nothing in it is checked until its line is reached.*

BINDER Question 2

A tally sheet at the Nyamagana fish landing has one column ruled to the right of the others, and the clerk says that indent is what makes a figure a subtotal rather than a total. Move the column and the sheet means something else.

Explain how Python marks the beginning and the end of a block, and contrast that with how C++ and Java mark one.

Model Answer

Python marks a block by indentation alone: a header line ends with a colon, the block is every line below it indented further, and the block ends at the first line that comes back to the level of the header.

The colon and the indent: The colon at the end of a header line announces that a block follows. The block opens at the first line indented past the header and closes at the first line that returns to the header's own column. A block inside a block is indented one further level again.

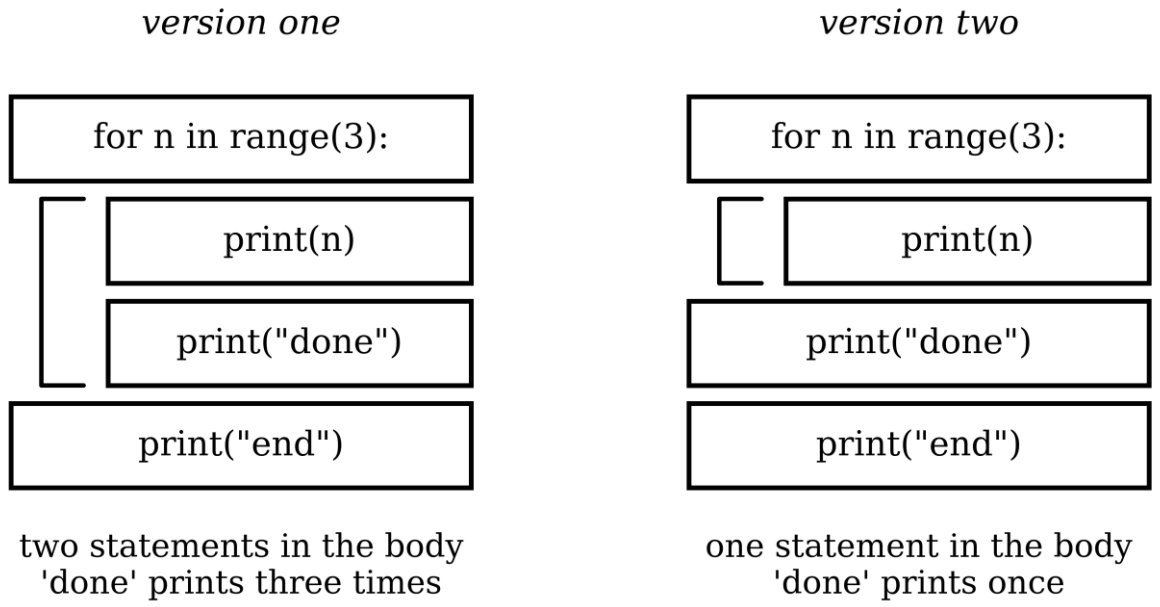


Figure 2.2: *The same four statements at two indent levels, and the two different programs they are.*

How C++ and Java mark the same thing: Both use a pair of braces. The opening brace begins the block and the closing brace ends it, and the indentation between them is a courtesy to the reader that the compiler never reads. A C++ program laid out badly is untidy; a Python program laid out badly is a different program.

```
 1  # Version one: both print statements sit inside the loop body.
 2  print("version one")
 3  for n in range(3):
 4      print(n)
 5      print("done")
 6  print("end")
 7
 8  # Version two: the second print has been moved out of the body.
 9  print("version two")
10  for n in range(3):
11      print(n)
12  print("done")
13  print("end")
```

Output

```
version one
0
done
1
done
2
done
end
version two
0
1
2
done
end
```

Program 2.1: *The same four statements at two indent levels, and the two outputs they produce.*

Reading the two versions: Lines 4 and 5 are both in the loop body, so done appears three times. In the second version only line 11 is in the body, so done is printed once, after the loop has finished. Nothing was added, removed or reordered; one line moved four columns to the left.

Four spaces, and never a tab: The convention every Python author follows is four spaces for one level, with every indent a multiple of four. A tab and spaces must never be mixed inside one block. A tab is drawn as four columns by one editor and eight by another while the interpreter reads it as neither, so a listing containing one does not mean what it looks like it means.

In Python the layout is not a picture of the structure; it is the structure, and moving a line sideways rewrites the program.

The Master Key: *A colon opens a block and the indent decides what is in it, so a line moved four columns is a different program and not a tidier one.*

BINDER Question 3

A stock clerk in Mbagala keeps one peg on the wall. In the morning it holds the pharmacy order; by noon the same peg holds a delivery note, and nobody thinks the peg has turned into a different kind of peg.

State the rules that govern a valid Python name, and explain why an assignment needs no declaration.

Model Answer

A name may be made of letters, digits and the underscore, must not begin with a digit, must not be one of the language's keywords, and is case sensitive; and no declaration is needed because an assignment does not create a variable of a type, it binds a name to an object.

What makes a name legal: The characters allowed are the letters, the digits and the underscore, and the first may not be a digit. A keyword such as if, for, class or pass is reserved. Case matters, so reading, Reading and READING are three separate names, and mistaking one for another shows up only when the line runs.

The conventions the rules do not enforce: Beyond what the language allows, Python authors write ordinary names in lower case with underscores, class names with each word capitalised and no underscores, and fixed values in upper case with underscores. Never use l, I or O alone, because in most printed faces they cannot be told apart from one and nought.

Assignment binds a name: Students are often taught, wrongly, that a name must first be declared with a type before it can be used, as it must be in C++ and Java. There is no such line to write and no keyword that would write it. The assignment below creates an object and binds the name to it, and that is the whole of it.

```
1  # The Mbagala meter was read at dawn; by dusk it had jammed.
2  reading = 480
3  print(reading)
4  reading = "not taken"
5  print(reading)
```

Output

```
480
not taken
```

Program 2.2: *One name bound first to a whole number and then to a piece of text.*

The name has no type, and the object has: On line 2 the name is bound to a whole number and on line 4 to a piece of text. Nothing was converted and nothing was redeclared, because the type never belonged to the name. The object carries the type and the name merely reaches it, so rebinding points it at something else.

A name in Python is a label tied to an object, not a box shaped for one kind of value.

The Master Key: *Assignment binds a name to an object, so the name has no type and the object it reaches has one.*

BINDER Question 4

The choir rota at Kimara has a row for the fifth Sunday, and in most months that row is left blank. Rub it out and the whole table collapses by one row; the blank row is holding a place.

Explain what pass is for, and name the places where a block cannot be left empty.

Model Answer

pass is a statement that does nothing at all, and it exists so that a block the language requires to hold at least one statement can hold one while there is no work for it to do.

Where a block cannot be left empty: Every header ending in a colon opens a block that must contain at least one statement: if, elif and else, for and while, def and class, with, and try, except and finally. A comment will not do, because a comment is not a statement and a block holding only a comment is still empty as far as the parser is concerned.

A body not written yet: The commonest use is to hold a block open while the rest of the program is being written. The branch below is going to refill the tank when the level is low, but the refill rule has not been agreed, so pass stands in its place and the rest of the program runs and can be checked meanwhile.

```
1  # The refill rule has not been agreed yet, so the branch is empty.
2  level = 12
3  if level < 20:
4      pass
5
6  print("pump checked")
7  print(level)
```

Output

```
pump checked
12
```

Program 2.3: *pass holding open a branch whose work has not been written yet.*

A branch meant to do nothing: The second use is permanent. Here a reading of zero is normal on a day the kiosk did not open, so there is deliberately nothing to record, and pass says that plainly to the next person to read the file. A reader who finds an empty branch wonders whether a line was lost; a reader who finds pass knows it was not.

```
1  # A zero reading is normal on a closed day and needs no entry.
2  reading = 0
3  if reading > 0:
4      print("reading recorded")
5  else:
6      pass
7
8  print("day closed")
```

Output

```
day closed
```

Program 2.4: *pass in a branch that is deliberately meant to do nothing.*

pass occupies a block without doing anything in it, which is exactly what an unfinished body and a deliberately idle branch both need.

The Master Key: *A block may never be empty, so pass is the statement you write when the block must exist and the work must not.*

REAL Question 5

The screen at a Kilindoni fuel pump stays blank when the attendant presses the key for the shift slip. Not the header, not the date, not one line of it. The program did not stop halfway; it never started.

Name the fault in the listing below, say exactly what Python does about it and at what moment, and correct it.

Model Answer

The header on line 4 ends with a colon and so promises an indented block, and the line beneath it is not indented, so the block it promised is empty and the file will not parse.

```
1  # Kilindoni fuel pump: a slip is printed when the meter has moved.
2  opening = 1420
3  closing = 1468
4  if closing > opening:
5  print("issue a slip")
6  print("day closed")
```

Python says

```
  File "/home/form5/pump.py", line 5
    print("issue a slip")
    ^
IndentationError: expected an indented block after 'if' statement on line 4
```

Program 2.5: *A header that promises a block, and a file that never runs.*

What the colon promised: A header line ending in a colon is a promise that a block follows, and the block must begin with a line indented past the header. Line 5 begins in the first column, level with the header itself, so the parser arrives at a statement it can only read as the one after the branch. The body is simply missing.

The moment Python finds it: Before a single line executes. Python reads and parses the whole file before it runs any of it, so lines 2 and 3 never run, no name is ever bound and nothing at all is printed. That is why the attendant's screen stayed blank rather than showing a partial slip: this program did not fail during its work, it failed before its work began.

What Python says: The first line names the file, which on your own machine carries your own path, and the line where the missing block was noticed. The message then names the header that wanted the block and the line it is on. The class of the fault is IndentationError, and it is raised by the parser rather than by anything the program does.

The correction: Indent line 5 by four spaces. Line 6 then stays in the first column and runs whatever the meter did, while the slip is issued only when the closing reading is the larger. If the branch were genuinely meant to do nothing, pass would fill it instead, which is the other way of keeping the promise the colon made.

Why the fault is common: It is imported from another language. In C++ or Java a pair of braces enclosing nothing is a legal empty block, so a header written now with its body left for later compiles and runs. Python has no braces to leave empty, and the line that would have been indented is the block, so leaving it out removes the block entirely.

A missing indent is not untidy code; it is a file the interpreter refuses to accept, and refusing it costs nothing because nothing had yet been done.

The Master Key: *A colon with no indented line beneath it is found by the parser, so the program prints nothing at all and stops before its first statement.*

REAL Question 6

A note pinned inside a toolbox at the Buguruni depot says the crate counts go in below the heading. Two clerks copy them in, one pressing the tab key and one pressing the space bar four times, and the columns line up perfectly.

Explain why the listing below never reaches its first statement, and repair it.

Model Answer

Line 4 is indented with four spaces and line 5 with a single tab, and because those two indents cannot be compared reliably the parser refuses the whole file with a TabError before anything runs.

```
1  # Buguruni depot: the running total of crates unloaded.
2  crates = 0
3  for load in range(4):
4      crates = crates + load
5      print(crates)
6  print("unloading finished")
```

Python says

```
  File "/home/form5/crates.py", line 5
    print(crates)
TabError: inconsistent use of tabs and spaces in indentation
```

Program 2.6: *Two lines of one block, one indented with spaces and one with a tab.*

What is on the page and what is in the file: Line 4 begins with four space characters and line 5 begins with one tab character, and both were meant to sit in the loop body. Look at how they have printed: the tab has come out narrower than the four spaces above it, while an editor whose tab stops are four columns wide would have shown the two lines level. One file, two pictures of it.

Why the parser will not settle it: Because a tab has no width of its own, Python measures the indentation of the two lines under more than one assumption about what a tab is worth. When those assumptions disagree about which line is the deeper, the indentation is ambiguous, and Python refuses rather than guess.

The moment Python finds it, and what it says: At parse time, exactly as in the previous question, so nothing is printed and nothing runs. The class differs because the reason differs: there is an indent here, and the complaint is that it cannot be measured against the one above it. TabError names the line where the two ways of indenting first meet.

Why this one hides so well: A missing indent is plain on the page. A tab is one character drawn as a gap of whatever width the display chooses, so no printed listing can be trusted to show which characters are indenting it, and a student may copy one faithfully and still be handed a file that will not parse. This is the commonest way a printed Python listing stops meaning what it appears to mean.

The repair, and the habit that prevents it: Delete the tab and type four spaces. The lasting repair is a habit and a setting: four spaces at every level, every indent a multiple of four, no tab anywhere in a Python file, and an editor told to insert spaces when the tab key is pressed. Most editors will also show whitespace on request, which makes the fault visible.

The look of a listing cannot tell you which characters indent it, and where two indents cannot be compared Python refuses the file rather than choose between them.

The Master Key: *A tab and four spaces in one block are ambiguous rather than untidy, so Python refuses the file with a TabError before any line runs.*

REAL Question 7

Two lines of the weighbridge report at Msasani come up on the screen, the gate name and the morning load, and then the program stops. The lines that appeared are correct. The one that should have followed never comes.

Explain why the listing below prints two lines and then stops, and say what a compiled language would have done with it instead.

Model Answer

The name on line 6 is misspelled, and because a name is looked up only when the line using it runs, the file parses without complaint, lines 2 to 5 do their work, and the program stops at line 6 with a NameError.

```
1  # Msasani gate: two loads weighed in, and a total that never prints.
2  morning = 240
3  evening = 315
4  print("Msasani gate")
5  print(morning)
6  total = mroning + evening
7  print(total)
```

Python says

```
Msasani gate
240
Traceback (most recent call last):
  File "/home/form5/gate.py", line 6, in <module>
    total = mroning + evening
            ^^^^^^^
NameError: name 'mroning' is not defined. Did you mean: 'morning'?
```

Program 2.7: *A listing that parses perfectly, prints two lines, and then stops.*

The file is well formed: Every line obeys the shape of the language. Line 6 is an assignment whose right hand side adds two names, which is a perfectly ordinary statement, and nothing about the spelling of a name is a question the parser asks. So the parser has no complaint, the file is accepted, and the program starts.

What ran before it stopped: Lines 2 and 3 bound two names, line 4 printed the gate and line 5 printed the morning load. Both printed lines stand above the message and both are correct. The program was not refused, it was interrupted, and it had already done part of its job when it was.

When the name is looked up: At the moment line 6 is executed, and not one instant sooner. Python searches for the misspelt name among the names bound so far, fails to find it, and stops there. The suggestion at the end of the message is worked out by comparing the unknown spelling with the names that do exist, which is why it can name the one that was meant.

How this differs from the two before it: Those two failed in the parser and printed nothing, because the file was never accepted. This one failed in the middle of the work, with output already on the screen. That is the distinction to carry: a Python program can fail before it starts or fail while it is running, and only the second kind leaves evidence of what it managed to do.

What a compiled language would have done: Refused it outright. In C++ or Java a name must be declared before it is used, so the misspelling is an undeclared name and the build stops with an error and produces no program at all. Nothing would have printed, because nothing would have been runnable. Python has no such moment, and the fault waits in the file until the line reaches it.

A program that starts is not a program that works, and the only thing that found this fault was the act of running the line that contained it.

The Master Key: *A misspelled name is a run-time fault in Python, so the program parses, prints what it can, and stops only when the line that needs the name is reached.*

BUILD Question 8

At dusk the Tegeta water kiosk closes and the attendant writes one slip: the meter at opening, the meter at closing, and what is owed. The slip is written before the tally book, because the book is copied from the slip.

Write the end of day slip program for the Tegeta water kiosk to the specification below.

The opening and closing meter readings are written into the source, both in litres.

The price of one litre in shillings is a module constant.

Print a heading line, then the litres drawn and the shillings due.

If the meter has not moved, print instead that the kiosk did not open.

Indent every block with four spaces, follow the naming rules of Question 3, and include at least one comment that says something the code does not.

Model Answer

The whole program is arithmetic on two numbers written into the source: the litres drawn are the difference between the two meter readings, and the money due is that difference at the price of one litre.

```
 1  # Tegeta water kiosk, end of day slip.
 2  # Both readings are taken off the meter face itself, never off the
 3  # tally book, which is written up after the slip has been signed.
 4  PRICE_PER_LITRE = 3
 5
 6  opening_reading = 18420
 7  closing_reading = 19260
 8
 9  litres_drawn = closing_reading - opening_reading
10  money_due = litres_drawn * PRICE_PER_LITRE
11
12  print("Tegeta kiosk, end of day")
13  if litres_drawn == 0:
14      print("the kiosk did not open today")
15  else:
16      print("litres drawn:", litres_drawn)
17      print("shillings due:", money_due)
```

Output

```
Tegeta kiosk, end of day
litres drawn: 840
shillings due: 2520
```

Program 2.8: *The end of day slip for the water kiosk at Tegeta.*

How the specification is met: The readings and the price go in as three names on lines 4, 6 and 7, the two quantities the slip needs are worked out on lines 9 and 10, and the heading is printed before either of them is reported. Lines 13 to 17 cover the closed day, and both branches are indented by four spaces, which is what makes them branches.

The one decision worth naming: The price is a module constant in upper case rather than a bare number inside the arithmetic on line 10. That line now says what it is doing instead of multiplying by an unexplained figure, and a price change is made in one place. The four ordinary names are lower case with underscores.

What the comment carries: A comment repeating the code earns nothing, and the one at the top repeats none of it. It records where the readings come from and that the tally book is not their source, which matters when the two disagree and which no line of the program could say.

Every value the slip needs is in the source, so the program can be read from the top and its output predicted before running it.

The Master Key: *A constant named in upper case and a comment that carries what the code cannot are what separate a working program from a readable one.*

HOT Question 9

A boatyard at Kilindoni lets an apprentice start cutting the moment he has a shape in mind. The yard next door checks every drawing before a saw is touched. Both launch boats, and both lose work, in opposite ways.

Argue both sides of the claim that having no compile step helps a beginner and hurts a large program, and reach a verdict.

Model Answer

Both halves are true, and the verdict is that the missing compile step is a gain while one person can still run the whole program in a few seconds and a loss the moment that stops being possible.

The case for the beginner: Edit the file, run the file. There is no build command to learn, no build configuration to get wrong and no class of error belonging to the builder rather than to the language. The gap between writing a line and seeing what it does is a second, and that gap is where a beginner learns.

The second gain, which is larger: A half written program still runs the half that is finished. A compiler refuses the entire file over one undeclared name, so nothing can be tried until everything is legal. Python runs what it can reach, so work may be checked in pieces and a beginner is never blocked by a fault elsewhere.

The cost in a large program: A misspelled name on a branch nobody takes today is not a fault today. It sits in the file, parsing cleanly, until that branch is taken, which may be in front of a user months later. A compiler would have found it without anyone having to think of the case that reaches it.

The cost that grows fastest: Paths through a program multiply faster than anyone can run them, so as it grows the fraction that has ever actually executed falls. A compiler inspects every line whether it is reached or not. In Python, code that has never run is code nobody has checked, and its share only rises.

What replaces the compiler: Three things, and a team with none of them is exposed. Running the program often, so most lines are reached most days. Tests that run it for you and reach branches you would not think to try. And tools that read the source without running it and report an undefined name, which the last section of this chapter puts to work.

The verdict, and the condition that decides it: Prefer Python's arrangement for any program one person can run end to end while working on it, and treat the missing compile step as a real cost once the program holds branches nobody exercises daily. The condition is not size in lines but whether tests and a checker are in place, since those, not the language, make a large Python program safe to change.

The compiler was never the point; being told about a fault before a user finds it was, and Python leaves you to arrange that.

The Master Key: *No compile step is a gain while you can run the whole program yourself, and a cost the day you cannot, so the judgement turns on what you have put in the compiler's place.*

Names, objects and operators

This section separates a name from the object it reaches, which is where most trouble with Python begins. It defines binding, identity and mutability, hand-evaluates the operators by precedence and associativity, and builds two programs that read what the user types and report on it. Three figures carry the ideas that a sentence on its own will not repair: two names on one object, a change made in place against a rebinding, and the precedence ladder with the column the common table leaves out.

BINDER Question 10

At a mobile money kiosk in Goba one cash tin carries two labels, float and till, in the owner's handwriting. A new clerk counts the labels and writes two tins into the record. The tin is still one tin.

Explain what happens when a name is assigned in Python, and distinguish a name from the object it is bound to.

Model Answer

Assignment binds a name to an object: Python works out the expression on the right, obtains the object it produces, and makes the name on the left reach that object, copying nothing and reserving nothing in advance.

What the equals sign does: The right-hand side is evaluated first and yields an object. The name on the left is then made to reach that object, and if the name is new it comes into existence at that moment. There is no declaration to write beforehand, and no type is attached to the name itself.

One object, two names: Writing scores = marks builds no second list. It gives the one list a second name, so both names reach the same object, which is the left half of Figure 2.3. Assigning to scores afterwards moves that one name and leaves marks where it was, which is the right half.

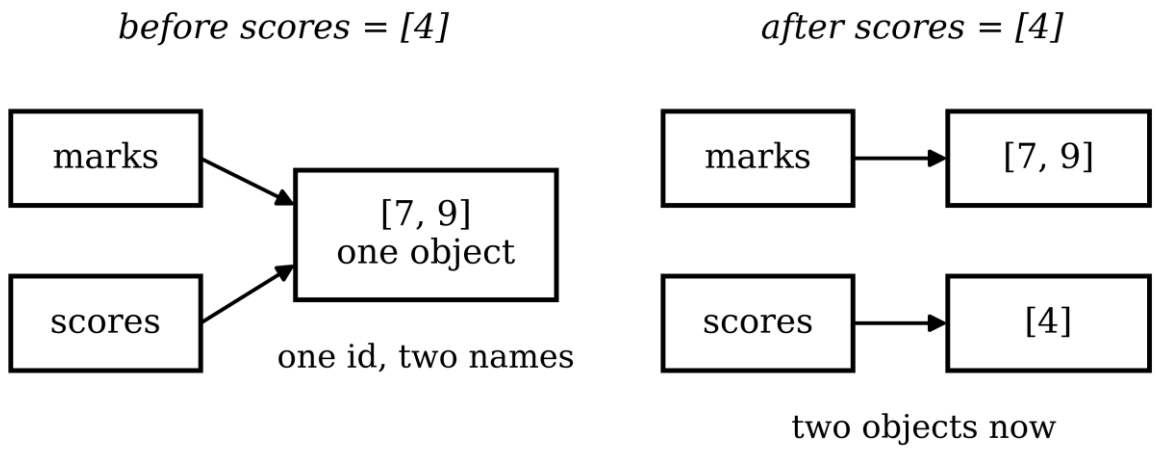


Figure 2.3: *Two names on one object, and the same two names after one of them is rebound.*

Identity, and what id reports: Every object has an identity, meaning its own separate existence for as long as anything reaches it. The built-in id() returns an integer standing for that identity, so two names on one object give one value from id(), and two objects never give the same one while both are alive.

Everything is an object: Everything in Python is an object, whole numbers and pieces of text included, and each object carries its own type. That is why type() in Program 2.9 answers for the value rather than for the name: tally reports int on line 4 and str on line 8, having been rebound on line 7.

```
1  tally = 7
2  depot = "Goba"
3  price = 950.0
4  print(type(tally))
5  print(type(depot))
6  print(type(price))
7  tally = "seven"
8  print(type(tally))
```

Output

```
<class 'int'>
<class 'str'>
<class 'float'>
<class 'str'>
```

Program 2.9: *Three names, three types, and one name rebound to another type.*

A name is a label the interpreter can move at any line; the object is what stays where it is until no name reaches it.

The Master Key: *Assignment never copies: it points one more name at one object.*

BINDER Question 11

Two bicycles lean outside the dispensary at Kibamba, the same green, the same basket, the same bell. The nurse calls them the same bicycle. The watchman, who locked both of them himself, says she is wrong.

Distinguish the is operator from ==, and explain what each one actually compares.

Model Answer

== compares the values two objects hold; is asks whether the names reach the same object, which is a different question with its own answer.

What each one compares: == asks the object's own comparison behaviour and answers on the contents, so two lists built separately from 4 and 8 are equal. is answers on identity alone, the thing id() reports, and no amount of matching contents will ever make it true.

Identity is not a name: A common error is to describe an object's identity as the label it has been given. Identity is the object's own separate existence, and one object may carry several names or none at all. Two names for one list are therefore one object and not two, however the code reads.

```
1  first = [4, 8]
2  second = [4, 8]
3  print(first == second)
4  print(first is second)
5  third = first
6  print(third == first)
7  print(third is first)
```

Output

```
True
False
True
True
```

Program 2.10: *Two equal lists, and two names on one list, compared both ways.*

Reading the four comparisons: Lines 1 and 2 of Program 2.10 build two lists, so line 3 prints True on the contents and line 4 prints False on the identity. Line 5 binds third to the object first already reaches, so lines 6 and 7 both print True, on the contents and on the identity alike.

Never write is against a number or a piece of text: Comparing with is against a value such as 5 asks whether it is one particular object, which Python has never promised. Since version 3.8 the interpreter answers with a SyntaxWarning, and where warnings are treated as failures the program stops there. Use == for values.

Comparing with None: There is exactly one None object in a running program, so testing for it by identity is both correct and idiomatic. That single case aside, keep the identity question for objects you built yourself, and ask every other question with ==.

Two objects holding equal values are still two objects, and only the identity test will tell you so.

The Master Key: *== asks what it holds; is asks which object it is.*

BINDER Question 12

A maize ledger at Msata takes a new line at the end of every delivery. The printed price card beside it cannot be altered at all: a new price means a new card, and the old card goes in the bin.

Explain mutability, classify the everyday Python types by it, and explain why it decides what one name sees when the object is changed through another.

Model Answer

A mutable object can be changed in place, so every name bound to it sees the change at once; an immutable object cannot be changed at all, so what looks like a change is a new object with a name moved onto it.

Changing the object against making another: Figure 2.4 sets the two beside each other. Adding one to a whole number cannot alter the object holding 5, so Python builds an object holding 6 and rebinds the name onto it. Appending to a list alters the one list that is already there, and the name never moves at all.

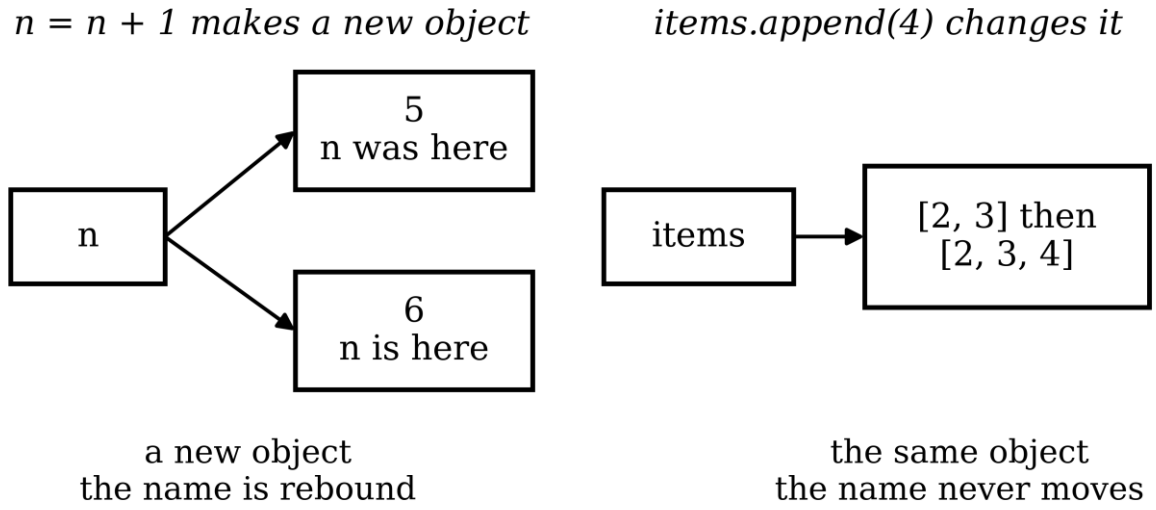


Figure 2.4: *Adding one to a whole number makes a new object; appending to a list changes the object itself.*

Which types are which: int, float, str and tuple are immutable; list, dict and set are mutable, as Table 2.1 records. The division belongs to the type and never to the name, so nothing written on the left of an assignment can make a piece of text changeable or a list fixed.

Type	Mutable	What a change to the value means
int	no	a new object is built and the name is moved onto it
float	no	a new object is built and the name is moved onto it
str	no	every edit yields a new piece of text
tuple	no	the contents are fixed once the object is built
list	yes	appending or assigning changes the one object
dict	yes	adding or removing a key changes the one object
set	yes	adding or removing a member changes the one object

Table 2.1: *The everyday types, classified by whether the object itself can be changed.*

Why one name sees what another did: Two names on one mutable object share everything that happens to it, because there is only one object for anything to happen to. Two names on an immutable object are safe from each other, since neither can alter what they share and an assignment merely moves one of them elsewhere.

One operator, two behaviours: The += operator reads as a single rule and is not one. On a whole number it computes a value and moves the name onto the new object, leaving any other name where it was. On a list it extends the one object in place, and every name bound to that list sees the longer list.

Mutability decides whether a second name is a convenience or a hazard, and the type settles it before you write a line.

The Master Key: *If the object can change, every name on it changes with it.*

REAL Question 13

512, not 64. A carpenter at Miono keys 2 to the power 3 to the power 2 into a borrowed calculator and gets one answer; Python prints the other. Neither machine is broken, and only one of them reads the keys in the order he meant.

Hand-evaluate a group of expressions by precedence and associativity, and state the value of each.

Model Answer

Python binds the tightest operators first, works from left to right between operators of equal tightness, and makes one exception, **, which works from right to left.

How to read the ladder: Figure 2.5 places the operators in binding order, tightest first, and Table 2.2 says the same in words with the column the common table omits. Associativity settles what precedence cannot, which of two equally tight operators goes first. Note where the membership operator in and the identity operator is sit: on the comparison rung, above the three word operators that close the ladder.

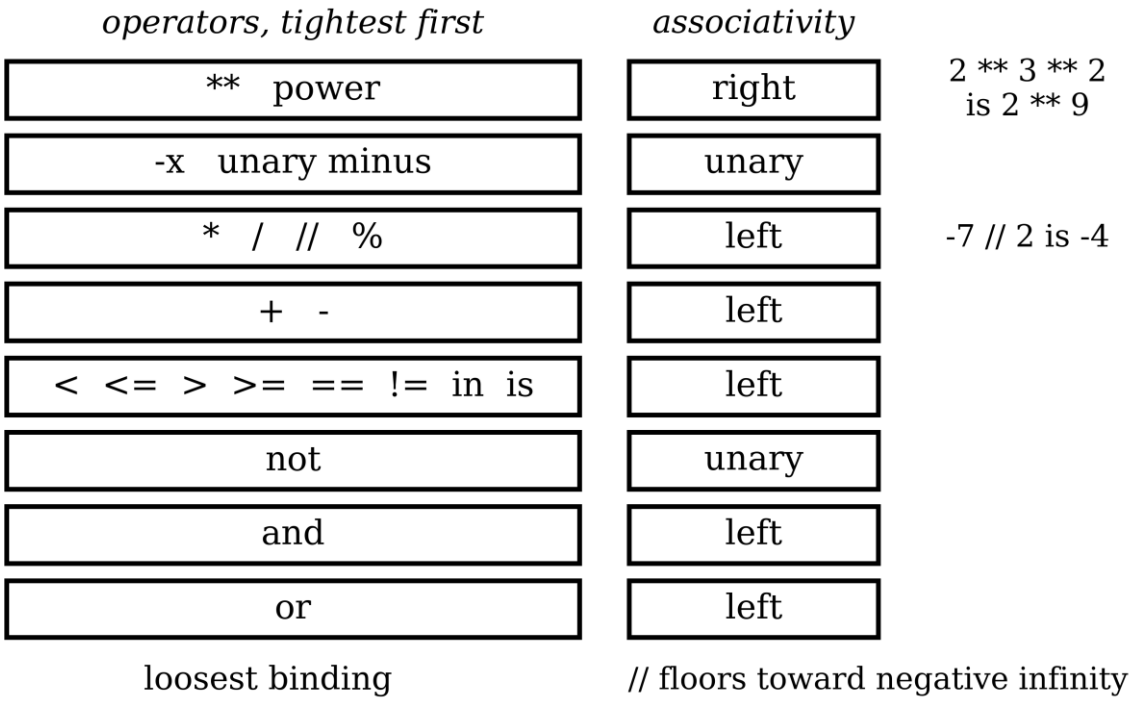


Figure 2.5: *Python operator precedence, highest binding first, with the one operator that associates to the right.*

Rank	Operators	Associativity
1	**	right
2	-x	unary
3	* / // %	left
4	+ -	left
5	< <= > >= == != in is	left
6	not	unary
7	and	left
8	or	left

Table 2.2: *The precedence ladder with its associativity column, tightest binding first.*

The power operator works right to left: 2 ** 3 ** 2 is 2 ** (3 ** 2), which is 2 ** 9, or 512. Reading it from the left instead gives (2 ** 3) ** 2, which is 64, and a table carrying no associativity column cannot tell you which of the two Python means.

Floor division rounds downwards: -7 // 2 is -4. The true quotient is -3.5, and the floor of a value is the whole number below it on the number line, never the one nearer zero. C++ and Java truncate toward zero and give -3 here. The two families agree on positive operands and part company on negative ones.

The remainder follows the quotient: 7 % -2 is -1 and not 1. Python keeps the identity that the quotient times the divisor plus the remainder returns the dividend: 7 // -2 is -4, and -4 times -2 is 8, so the remainder has to be -1. The remainder therefore takes the sign of the divisor.

A mixed expression, one step at a time: In 20 - 3 * 2 ** 2 the power binds first and gives 4, the multiplication gives 12, and the subtraction gives 8. In 100 // 7 % 3 the two operators are equally tight, so the left one goes first: 100 // 7 is 14, and 14 % 3 is 2.

Where brackets earn their place: Brackets override the ladder and cost nothing, so write them wherever a reader would otherwise have to consult it. An expression whose meaning depends on a rung most people cannot recite is a defect waiting for the day somebody edits it, even while it prints the right value today.

```
1  print(2 ** 3 ** 2)
2  print(-7 // 2)
3  print(7 % -2)
4  print(20 - 3 * 2 ** 2)
5  print(100 // 7 % 3)
```

Output

```
512
-4
-1
8
2
```

Program 2.11: *Five expressions evaluated by precedence and associativity.*

Every value above came back from the run printed beside it, not from a rule half remembered.

The Master Key: *Precedence chooses the operator, associativity chooses the end, and floor division always rounds down.*

REAL Question 14

The storekeeper at Bungu adds fifteen bags to the spare tally, and the main tally changes by itself. Nobody has touched the main book, nobody has copied a figure across, and both books are in his own hands.

Trace a program in which two names are bound to one list, and show the one line that gives a separate copy instead.

Model Answer

Program 2.12 prints [40, 25, 15] and then 3, because stock and spare are two names for one list and the append on line 3 changed the object both of them reach.

Line by line: Line 1 builds a list holding 40 and 25 and binds stock to it. Line 2 binds spare to that same object and copies nothing, because an assignment never copies. Line 3 appends 15 through spare, which changes the object itself rather than building a second one.

```
1  stock = [40, 25]
2  spare = stock
3  spare.append(15)
4  print(stock)
5  print(len(stock))
```

Output

```
[40, 25, 15]
3
```

Program 2.12: *A change made through one name and read through another.*

What the two prints show: Line 4 reads the object through stock and prints [40, 25, 15]; line 5 prints 3, the length of the one list. Neither line mentions spare and neither needs to, because the change was made to the object rather than to the name it was made through.

Where the belief goes wrong: Line 2 is commonly read as taking a copy of the list, and nothing in it builds a list. A copy exists only where some call has built a second object, and until one has, every further name is another label on the object that was already standing there. Comparing the two names with is, as Program 2.10 does, settles the question in one line.

The one line that separates them: Replacing line 2 with spare = list(stock) builds a new list holding the same two values, which is what Program 2.13 does. The append then reaches the new object only, so stock still prints as [40, 25] while spare prints as [40, 25, 15].

```
1  stock = [40, 25]
2  spare = list(stock)
3  spare.append(15)
4  print(stock)
5  print(spare)
```

Output

```
[40, 25]
[40, 25, 15]
```

Program 2.13: *The same program with line 2 building a separate list.*

Why aliasing is worth having: A second name copies nothing and costs nothing, which is exactly what is wanted when a long list is handed from one part of a program to another. It becomes a defect only when a change is made through one name and read through another by somebody who believed there were two objects.

Before you trust a second name, find the line that built the second object; where there is none, there is one object wearing two labels.

The Master Key: *A copy is built by a call, never by an assignment.*

REAL Question 15

A stock program at Somanga prints its question, waits while the clerk types 34, and then stops without printing the line it was written to print. The clerk typed what was asked for. Nothing he did was wrong.

Name the fault in a program that does arithmetic on what the user typed, say exactly when Python stops it and what it does, and correct it.

Model Answer

The fault is on line 3: input() returns text, so bags holds the two characters 34 rather than the number 34, and + has no meaning between a piece of text and a whole number.

What input hands back: input() always returns a str, whatever was typed at it. Digits typed at a keyboard arrive as text, and text that looks like a number is still text. Line 1 converts nothing, so by the time line 3 is reached bags is a str while extra, from line 2, is an int.

```
1  bags = input("Bags collected: ")
2  extra = 12
3  total = bags + extra
4  print(f"Total bags: {total}")
```

Typed at the program

```
34
```

Python says

```
Bags collected: Traceback (most recent call last):
  File "/home/form5/bags.py", line 3, in <module>
    total = bags + extra
            ~~~~~^~~~~~~
TypeError: can only concatenate str (not "int") to str
```

Program 2.14: *Arithmetic on what the user typed, with no conversion.*

Exactly when Python stops: Not when the file is read. The program parses without complaint, line 1 prints its prompt and waits, the clerk types 34, line 2 binds extra, and only when line 3 is reached does Python refuse. The prompt standing at the head of the output is the proof: the program had already spoken before the fault was found.

What Python says about it: It raises TypeError and states that a str can only be joined to another str. The lines above the message name the file, give the line number and repeat the expression with the two operands underlined. Line 4 never runs, so the report the program was written to produce never appears at all.

The correction: Wrap the reading in int(), as line 1 of Program 2.15 does. The name then holds a whole number from the moment it is read, the addition on line 3 has two numbers to work with, and the report prints 46. Converting where a value enters the program keeps every line after it honest.

```
1  bags = int(input("Bags collected: "))
2  extra = 12
3  total = bags + extra
4  print(f"Total bags: {total}")
```

Typed at the program

```
34
```

Output

```
Bags collected: Total bags: 46
```

Program 2.15: *The same program with the reading converted by int().*

The other way it could have been read: The + operator is not one operation. Between two numbers it adds, between two pieces of text it joins, and between one of each it refuses rather than guessing which was meant. Had joining been the intention, str() around the number would have been the correction instead.

A program that stops halfway has not failed to run; it ran until it reached a value it could not use.

The Master Key: *Convert what was typed before you compute with it.*

BUILD Question 16

A fisherman at Mchinga counts 47 crates onto the lorry and stacks them five to a load. The buyer asks for four numbers out of those two: the two added, the gap between them, how many full loads, and what will not fill one.

Write a program that reads two whole numbers typed by the user and reports their sum, difference, floor quotient and remainder.

Read two whole numbers typed by the user, the first and then the second.

Report the sum, the difference, the floor quotient and the remainder.

Take the second number as the divisor in both divisions.

Print each result on its own line, labelled, using an f-string.

Model Answer

The program reads each line with input(), converts it with int(), and prints one labelled line for each of the four results with an f-string apiece.

Converting where the value is read: Lines 1 and 2 wrap input() in int(), so everything after them works on whole numbers. Leaving the conversion until the arithmetic would put text on one side of an operator and a number on the other, which is the fault Program 2.14 was built to show.

```
1  first = int(input("First number: "))
2  second = int(input("Second number: "))
3  print(f"Sum: {first + second}")
4  print(f"Difference: {first - second}")
5  print(f"Floor quotient: {first // second}")
6  print(f"Remainder: {first % second}")
```

Typed at the program

```
47
5
```

Output

```
First number: Second number: Sum: 52
Difference: 42
Floor quotient: 9
Remainder: 2
```

Program 2.16: *Sum, difference, floor quotient and remainder of two typed numbers.*

The four results: Given 47 and 5 the program prints 52, 42, 9 and 2. The floor quotient counts the whole fives in 47 and the remainder is what would not fill another, and the two together account for the 47 exactly.

Reading the first line of the output: input() prints its prompt without a newline, so when the two answers are fed to the program from elsewhere the two prompts and the first result share one line. Typed at a keyboard, each answer appears against its own prompt and the output reads as four separate lines.

The specification asked for four labelled lines, and the four operators that produce them are the whole of the program.

The Master Key: *Read as text, convert once, then compute.*

BUILD Question 17

The receipt from a printing shop shows 3 at 1450.50 and a total of 4351.5, one figure short of money. The customer notices before the clerk does, because a price ending in a single decimal looks like a slip.

Write a program that reads a customer name, a quantity and a unit price, and prints one line with the total to two decimal places.

Read a customer name, a whole number quantity, and a unit price that may carry decimals.

Multiply the quantity by the unit price to obtain the total.

Print one line only, naming the customer and the quantity.

Show the unit price and the total to exactly two decimal places.

Model Answer

The program reads three values, converts two of them, multiplies, and prints one line in which the format specification .2f holds both money values at two decimal places.

Three readings, two conversions: The name is wanted as text and is used exactly as it arrives. The quantity is a count, so int() is right for it; the price may carry decimals, so float() is right for it. Casting the price with int() instead would throw away everything after the decimal point.

```
1  name = input("Name: ")
2  quantity = int(input("Quantity: "))
3  price = float(input("Unit price: "))
4  total = quantity * price
5  print(f"{name} buys {quantity} at {price:.2f} each, total {total:.2f}")
```

Typed at the program

```
Neema
3
1450.50
```

Output

```
Name: Quantity: Unit price: Neema buys 3 at 1450.50 each, total 4351.50
```

Program 2.17: *One formatted line, with the total to two decimal places.*

What the format specification does: A colon inside a placeholder separates the value from instructions for printing it. Here .2f rounds the printed figure to two decimal places and keeps the trailing zero, so 4351.5 prints as 4351.50. It changes what is printed and leaves what total holds untouched.

One line, one call: The specification asked for a single line, so there is a single print() on line 5 with one f-string inside it. Each value is placed by name in the order the reader wants to read it, which is what an f-string is for.

Money on a receipt carries two figures after the point whatever the arithmetic came to, and the format specification is what guarantees it.

The Master Key: *An f-string can format as well as fill: the instructions go after the colon.*

HOT Question 18

Two programmers in an internet café argue over a program that ran for a week and then stopped on a line nobody had changed. One blames the language for allowing it. The other says the same freedom is why it existed at all.

Assess the claim that dynamic typing makes Python quicker to write and harder to trust, and reach a judgement.

Model Answer

Both halves of the claim are true, and they are one fact seen from two ends: nothing is checked until the line runs, which takes work out of the writing and puts risk into the running.

What dynamic typing actually is: A name in Python has no type. The object it is bound to has one, and the operation is looked up on that object at the moment the line is reached. There is no earlier pass over the file in which a mismatch between an operator and its operands could have been found.

The case for quicker to write: Nothing is declared, so a value can be read, converted and rebound without a signature to satisfy first. One piece of code serves anything that supports the operation it performs, so the same three lines handle a whole number today and a decimal one tomorrow. A short program reaches a working state in fewer lines and far fewer edits.

The case for harder to trust: The fault in Program 2.14 was invisible until line 3 ran. A branch that is never taken carries its fault untouched to whoever runs the program next, and a name rebound to another type reads on the page exactly like one that was not. What a compiled language refuses before the program starts, Python accepts and discovers later.

Where the two claims meet: A late discovery does not cost the same everywhere. In a program that runs once, in front of the person who wrote it, a TypeError is an irritation put right in seconds. In one that runs unattended at a kiosk, the same fault arrives at a stranger, in the middle of a job, with the work half done and no way to finish it.

What closes the gap: Two habits do most of the work. Convert every value at the boundary where it enters the program, which is why int() belongs around input() rather than three lines later. Then run every branch at least once, because a branch nobody has run is a claim nobody has checked.

The judgement: Prefer Python's dynamic typing wherever every branch will be run before anybody depends on the program, because the freedom then costs nothing that has not already been checked. Where that cannot be promised, the same freedom is a liability and a checked language earns its extra words.

The condition that decides it is not how large the program is but whether its untried paths will ever reach a stranger.

The Master Key: *Dynamic typing does not remove the check; it moves it from the compiler onto you.*

Collections and control flow

This section defines the four collection types Python offers, list, tuple, dictionary and set, and finishes the control flow that decides which of their elements a program touches. It compares the four on the properties that settle the choice, draws the path a loop takes and the three ways it can leave, and traces a key reaching its value. It builds two complete programs from what a user types, one over a list of marks and one over a dictionary of counts, and it gives the dictionary more room than its size suggests, because the section on classes will use one to explain where an object keeps its attributes.

BINDER Question 19

A tailor in Ruponda keeps three books: one for orders in the sequence they arrived, one for prices that never change, and one that finds a customer by name. Python offers those three shapes and a fourth.

Compare the list, the tuple, the dictionary and the set on the properties that decide which one to reach for, and say which one suits a given job.

Model Answer

Python gives you four collection types, and they differ on whether order is kept, whether the collection can be changed after it is built, whether a value may appear twice, and how an element is reached.

	list	tuple	dict	set
ordered	yes	yes	yes since 3.7	no
mutable	yes	no	yes	yes
duplicates	yes	yes	keys unique	no
reached by	position	position	key	membership

Figure 2.6: *The four collection types compared on the properties that decide which one to reach for.*

Type	Ordered	Mutable	Duplicates allowed	Reached by
list	yes	yes	yes	position
tuple	yes	no	yes	position
dict	yes, since 3.7	yes	keys unique	key
set	no	yes	no	membership

Table 2.3: *The four collection types on the four properties that settle the choice.*

The four written down: A list sits in square brackets, a tuple in round brackets, a dictionary in curly brackets with a colon between each key and its value, and a set in curly brackets with no colons. Program 2.18 builds one of each.

Order and change: List, tuple and dictionary keep the order their items went in; a set does not, which is why line 9 sorts before printing. The tuple alone cannot be changed once built, so it is the shape to reach for when a pair of values must never drift apart.

Duplicates and access: A list and a tuple hold a value as often as you like and give it back by position, counting from zero. A set drops a repeat as it arrives, so the three names on line 4 become two. A dictionary allows a repeated value but never a repeated key.

Choosing one for a job: Take a list for a queue of jobs that grows and shrinks. Take a tuple for readings that belong together and must not be edited. Take a dictionary whenever one value labels another. Take a set when the only question is whether something is already there.

Building a list in one line: A list comprehension is a for loop written inside square brackets: the value to store, then the clause that walks the source. Line 12 stores the length of every name in the list above it.

```
 1  places = ["Ruponda", "Lukuledi", "Mahenge"]
 2  prices = (500, 800, 1200)
 3  depths = {"Ruponda": 18, "Lukuledi": 24}
 4  visited = {"Ruponda", "Lukuledi", "Ruponda"}
 5
 6  print(places[0])
 7  print(prices[2])
 8  print(depths["Lukuledi"])
 9  print(sorted(visited))
10  print(len(places), len(prices), len(depths), len(visited))
11
12  lengths = [len(name) for name in places]
13  print(lengths)
```

Output

```
Ruponda
1200
24
['Lukuledi', 'Ruponda']
3 3 2 2
[7, 8, 7]
```

Program 2.18: *One of each of the four collection types.*

The choice is made once, when the collection is created, and every line that follows lives with it.

The Master Key: *Order, changeability, duplicates and how an element is reached are the four questions, and answering them names the type.*

BINDER Question 20

The pharmacy counter at Lukuledi holds a stock book where every medicine name is written once and a quantity sits beside it. Nobody counts through the pages; they turn straight to the name. Python stores things the same way.

Explain how a dictionary holds keys and values, how an entry is added and changed, and what happens when a key that is not there is used.

Model Answer

A dictionary stores pairs, each a key and the value that key stands for, and it reaches a value by naming its key rather than by counting positions.

Keys and values: The key is what you know and the value is what you want. A key must be unchangeable, so a string, a number or a tuple will serve and a list will not. Values are under no such rule and may be anything at all, including another dictionary.

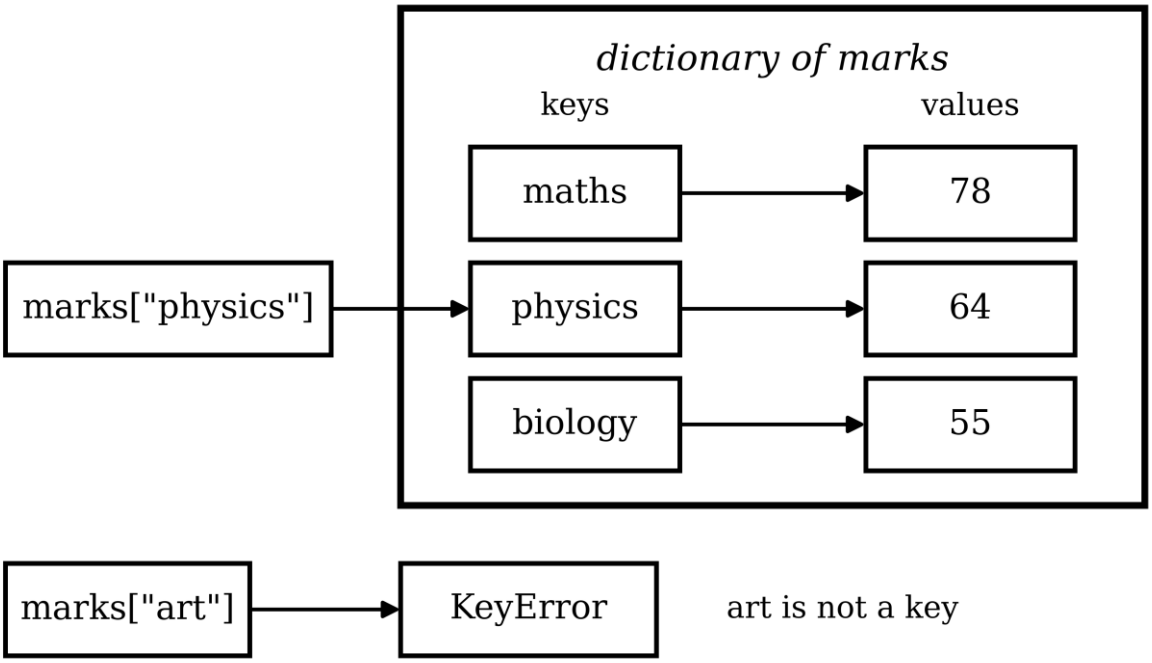


Figure 2.7: *A dictionary reaching a value by its key, and what a key that is not there does.*

Reaching a value: Square brackets carrying a key return the value that key holds, as Figure 2.7 traces for the physics key. Nothing is searched. The dictionary computes where the key belongs from the key itself, which is why a lookup does not slow down as the dictionary grows.

Adding and changing: One line does both. Assigning to a key that is absent creates the pair; assigning to a key that is present replaces its value and leaves the count alone. Line 3 of Program 2.19 adds history, line 4 changes physics, and the dictionary ends with four pairs rather than five.

A key that is not there: Using a key the dictionary does not hold raises KeyError, and the program stops at that line. Figure 2.7 marks that exit. Where a missing key is expected, the in operator answers whether a key is present, and the get method returns a value you choose instead of raising.

Printing a dictionary safely: Walking a dictionary with a for loop gives you its keys, not its values. Sort them first, as line 6 does, and the printed order is the same on every machine and every run; leave them unsorted and the order follows how the pairs were added, which is rarely what a reader expects.

```
 1  marks = {"maths": 78, "physics": 64, "biology": 55}
 2
 3  marks["history"] = 71
 4  marks["physics"] = 68
 5
 6  for subject in sorted(marks):
 7      print(subject, marks[subject])
 8
 9  print(len(marks))
10  print("art" in marks)
11  print(marks.get("art", 0))
```

Output

```
biology 55
history 71
maths 78
physics 68
4
False
0
```

Program 2.19: *Adding an entry, changing an entry, and two safe tests.*

A dictionary is the shape to reach for whenever one piece of data is the name of another.

The Master Key: *A key is looked up, never searched for, and a key that is absent raises rather than returning nothing.*

BINDER Question 21

Sack after sack goes onto the lorry at Mahenge while the loader counts aloud. Beside him a second man fills a water tank and stops when it overflows, counting nothing at all. Two jobs, two different loops.

Explain the for loop with range and the while loop, and say when each one is the right choice.

Model Answer

A for loop runs its body once for every item in something you can walk, and a while loop runs its body for as long as a condition stays true.

What range produces: With one argument it counts from zero up to but not including that argument, so the first loop in Program 2.20 prints zero, one and two. With two it counts from the first up to but not including the second. With three the last is the step, which may be negative.

The stop value is left out: This is where most counting faults begin. The second loop in Program 2.20 is written with a stop of six and its last value is five. To walk a whole list, write range of the length of that list, which gives every valid position and no more.

```
1  for n in range(3):
2      print("count", n)
3
4  for n in range(2, 6):
5      print("bay", n)
6
7  for n in range(10, 0, -3):
8      print("down", n)
```

Output

```
count 0
count 1
count 2
bay 2
bay 3
bay 4
bay 5
down 10
down 7
down 4
down 1
```

Program 2.20: *range with one, two and three arguments.*

What a while loop needs: Three things, and leaving out any one of them hangs the program: something set before the loop, a condition at the top, and a line in the body that moves that condition towards false. Program 2.21 sets both counters on lines 1 and 2 and moves them on lines 4 and 5.

Choosing between them: A for loop is right when you know before you start how many turns there will be, or when you are walking a collection you already hold. A while loop is right when the number of turns depends on something that is only discovered inside the loop, such as filling a lorry until the load passes a limit.

Reading the second trace: Program 2.21 asks how many thirty-five kilogram bags reach one hundred kilograms. It stops after three, at one hundred and five, because the condition is tested before each turn and not after; two bags leave the total at seventy, which is still under the limit.

```
1  total = 0
2  bags = 0
3  while total < 100:
4      bags = bags + 1
5      total = total + 35
6      print(bags, total)
7
8  print("bags needed", bags)
```

Output

```
1 35
2 70
3 105
bags needed 3
```

Program 2.21: *A while loop whose number of turns is not known in advance.*

A loop that never changes what its condition looks at is a program that never finishes.

The Master Key: *Reach for a for loop when the count is known in advance and a while loop when it is not.*

REAL Question 22

Every stop on the Lupiro road gets called out until a passenger says hers, and then the calling stops. On a quiet day the conductor reaches the end and announces that nobody is getting off. Python has a word for that second ending.

Trace a loop that uses break, continue and a loop else, and state exactly what is printed and which exit each loop took.

Model Answer

The program prints five lines, and the three loops leave by three different routes: the first runs out of items, the second breaks out on its third item, and the third runs out and falls into its else.

```
 1  route = ["Lupiro", "Msolwa", "Idete", "Ifunda"]
 2
 3  for stop in route:
 4      if stop == "Msolwa":
 5          continue
 6      print("calling at", stop)
 7
 8  for stop in route:
 9      if stop == "Idete":
10          print("found", stop)
11          break
12  else:
13      print("Idete is not on this route")
14
15  for stop in route:
16      if stop == "Mahenge":
17          print("found", stop)
18          break
19  else:
20      print("Mahenge is not on this route")
```

Output

```
calling at Lupiro
calling at Idete
calling at Ifunda
found Idete
Mahenge is not on this route
```

Program 2.22: *One loop that skips, one that breaks, and one that runs out.*

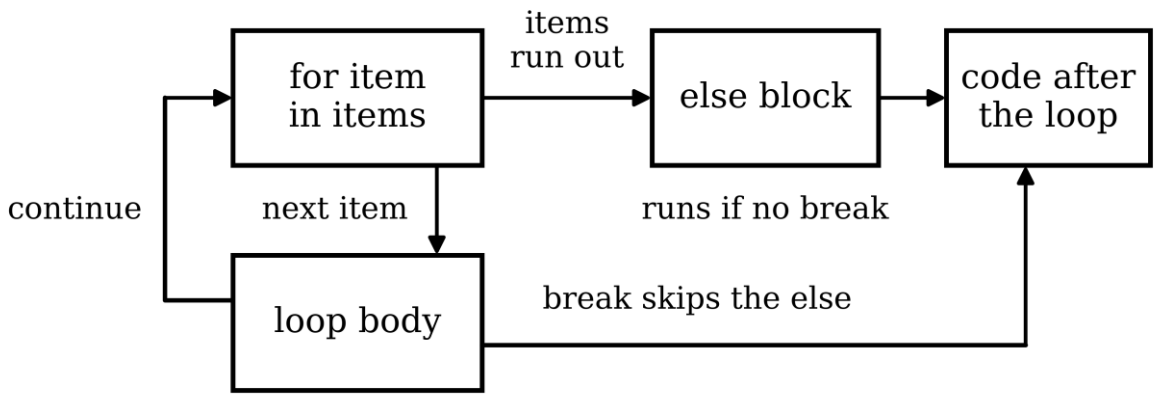


Figure 2.8: *The path a for loop takes, and the three ways it can leave.*

What continue does: It abandons the rest of the body for the current item and goes back for the next one. It does not leave the loop. In the first loop of Program 2.22 the test on line 4 catches Msolwa, so line 6 never runs for that stop and three of the four names are printed.

What break does: It leaves the loop at once, and nothing else in the body runs for that item or any later one. The second loop stops on Idete, so Ifunda is never looked at, and the else on line 12 belonging to that loop is skipped along with it.

The else that belongs to the loop: An else indented level with a for or a while is not an else to any if. It runs when the loop finishes because its items ran out, and it is skipped when a break leaves. Read it as a phrase meaning no break happened, which is why it is the natural place to report a search that failed.

The search that finds: Idete is on the route, so line 10 prints and line 11 breaks. The loop has been left early, the else is not entered, and no line reports a failure. Figure 2.8 draws the route the break takes, running under the else block rather than through it.

The search that fails: Mahenge is not on the route, so the third loop tests all four stops, finds nothing, and reaches the end of its items. The else then runs and prints the only line that loop produces. Without it you would need a flag set before the loop and checked after it.

The exits named: The first loop left by exhaustion with no else attached, the second left by break, and the third left by exhaustion into its else. Those are the only three ways out of a for loop, and knowing which one was taken is the whole of reading such a trace.

The loop else is short to write and easy to read once you hear it as no break happened.

The Master Key: *A loop else runs when the loop ran out of items and never when a break took it out.*

REAL Question 23

A tally program at the Msolwa weighbridge prints two good lines and then dies with red text on the screen. The operator swears nothing changed. Nothing did; the fault was always there, waiting for the last turn of a loop.

Name the fault in the tally program below, say exactly what Python does about it and when, and correct it.

Model Answer

The loop header on line 7 asks for one position more than the list holds, so the program prints its first two lines correctly and then raises IndexError on line 8 when it reaches position three of a list whose last position is two.

```
 1  tally = [12, 9, 15]
 2
 3  print("stations recorded:", len(tally))
 4  print("the first count is", tally[0])
 5
 6  total = 0
 7  for i in range(len(tally) + 1):
 8      total = total + tally[i]
 9
10  print("total sacks:", total)
```

Python says

```
stations recorded: 3
the first count is 12
Traceback (most recent call last):
  File "/home/form5/sacks.py", line 8, in <module>
    total = total + tally[i]
                    ~~~~~^^^
IndexError: list index out of range
```

Program 2.23: *A tally that starts correctly and stops part way.*

Naming the fault: It is an off by one fault in the loop header. The list holds three counts at positions zero, one and two, but adding one to the length makes range produce zero, one, two and three, and there is no position three.

What Python does, and when: Nothing at all until the loop reaches its fourth turn. The file parses cleanly, because the fault is not in the grammar; the two lines above the loop run and print; then line 8 asks for a position that does not exist and Python raises IndexError and stops the program there. The last print never runs.

Why the output is the tell: A program that prints two correct lines and then stops has not failed to parse; it has run into a fault at a particular line. That is the difference between the two ways a Python program can break, and only the second kind lets the reader see how far it got.

Reading the report: Python names the class of the fault and the line that reached it. IndexError means a position outside a sequence, and the line number sends you to line 8; the fault, though, was written on line 7, and the two are rarely the same line.

The correction: Delete the added one, so the header reads range of the length of the list. That produces every valid position and no more, and the program then prints a total of thirty-six. Written that way, the header cannot go wrong when the list grows or shrinks.

The better correction: Walk the list itself rather than a range of positions. A for loop over the counts hands you each value in turn and never mentions a position, so the fault cannot be written at all. Reach for a position only when you genuinely need to know where in the list you are.

A loop that counts positions must be given the positions the list actually has, and no more.

The Master Key: *The last valid position of a list is one less than its length, and asking for the length itself raises IndexError.*

REAL Question 24

Six delivery notes come back to the depot at Idete on Friday evening, and the clerk wants to know how many went to each village. She could sort them into piles. A dictionary does the piling for her.

Hand evaluate the counting loop and the nested loop below, and state exactly what each one prints.

Model Answer

The counting loop prints Idete three, Ifunda two and Msolwa one, in that order because the keys are sorted before printing, then reports three villages and six notes; the nested loop prints three rows of three numbers.

```
 1  deliveries = ["Idete", "Ifunda", "Idete", "Msolwa", "Ifunda", "Idete"]
 2
 3  counts = {}
 4  for stop in deliveries:
 5      if stop in counts:
 6          counts[stop] = counts[stop] + 1
 7      else:
 8          counts[stop] = 1
 9
10  for stop in sorted(counts):
11      print(stop, counts[stop])
12
13  print("villages served:", len(counts))
14  print("notes handled:", len(deliveries))
```

Output

```
Idete 3
Ifunda 2
Msolwa 1
villages served: 3
notes handled: 6
```

Program 2.24: *Counting how often each name appears.*

How the count is built: The dictionary starts empty on line 3. For each name the loop asks whether that key is already there; if it is, line 6 adds one to what is stored, and if it is not, line 8 puts one there. After six turns the dictionary holds three keys.

The count by hand: Idete arrives first and is created with one; Ifunda is created with one; Idete is found and becomes two; Msolwa is created with one; Ifunda is found and becomes two; Idete is found and becomes three. The six notes are accounted for, since three and two and one make six.

Why the printing is sorted: A dictionary walks its keys in the order they were added, and that order depends on the data, so the same program over a different day of notes would print the same counts in a different order. Sorting on line 10 fixes the order and makes the output the same every time.

```
1  for row in range(1, 4):
2      line = ""
3      for col in range(1, 4):
4          line = line + str(row * col) + " "
5      print(line.strip())
```

Output

```
1 2 3
2 4 6
3 6 9
```

Program 2.25: *A loop inside a loop, building one row at a time.*

Counting the turns of a nested loop: The outer loop of Program 2.25 turns three times and the inner loop turns three times for each of those, so the innermost line runs nine times. The inner loop builds one row of text and the outer loop prints it, which is why nine numbers appear on three lines.

The nested result by hand: Row one gives one, two and three. Row two gives two, four and six. Row three gives three, six and nine. The text is emptied again at the top of every outer turn, which is what keeps each row separate; move that line outside the outer loop and all nine numbers land on one line.

Counting with a dictionary needs one pass over the data and no second list to keep in step.

The Master Key: *A count is a dictionary whose keys are the things counted, printed in sorted order so the same data always reads the same.*

BUILD Question 25

The football league table at Ifunda is chalked on a wall, and every Sunday someone rubs out the goal column to work the average again by hand. The same figures are wanted every week: how many, the average, the best and the worst.

Build a marks summary program that reads a set of marks typed by the user and reports the count, the mean, the highest and the lowest, each one labelled.

The first line typed is how many marks follow.

Each mark that follows is typed on a line of its own.

Hold the marks in a list and read them with a loop.

Report the count, the mean to two decimal places, the highest and the lowest.

Every reported figure carries a label.

Model Answer

Program 2.26 reads the count first, fills a list with that many marks, and then reports four labelled figures from the one list it holds.

```
 1  print("Type the number of marks, then each mark.")
 2
 3  count = int(input())
 4  marks = []
 5  for _ in range(count):
 6      marks.append(int(input()))
 7
 8  total = 0
 9  for mark in marks:
10      total = total + mark
11
12  print(f"Marks entered: {len(marks)}")
13  print(f"Mean: {total / len(marks):.2f}")
14  print(f"Highest: {max(marks)}")
15  print(f"Lowest: {min(marks)}")
```

Typed at the program

```
5
78
64
55
71
82
```

Output

```
Type the number of marks, then each mark.
Marks entered: 5
Mean: 70.00
Highest: 82
Lowest: 55
```

Program 2.26: *How many, the mean, the highest and the lowest.*

How the specification is met: Line 3 reads the count and turns it into a whole number. The loop on lines 5 and 6 runs exactly that many times and appends each mark to the list. The count is then the length of the list, the mean is the running total divided by that length, and the highest and lowest come straight from the built in functions.

The one decision worth naming: The loop variable on line 5 is written as a single underscore because the body never uses it. The loop needs a fixed number of turns, not a position, and naming a variable that is never read invites a reader to look for where it is used. The underscore says plainly that there is nowhere.

What the run shows: Five marks were typed and they total three hundred and fifty, so the mean prints as seventy. The highest and lowest are eighty-two and fifty-five. Nothing in the program depends on there being five; a different first line changes the number of turns and nothing else.

A program written round a list scales to any number of marks without a line changing.

The Master Key: *Read into a list first, then ask the list its length, its total, its largest and its smallest.*

BUILD Question 26

Tickets sold at the Lupiro stand go into a tin, and at dusk the clerk tips them out and sorts them by destination. Counting the same name twice by accident is the whole difficulty, and a program never makes it.

Build a ticket tally program that reads destinations typed one to a line and reports how many tickets went to each, in sorted order, with the total.

Each destination is typed on a line of its own.

A blank line ends the typing.

Hold the tally in a dictionary.

Report each destination once with the number of tickets it took, in alphabetical order.

Finish with the total number of tickets.

Model Answer

Program 2.27 keeps one dictionary whose keys are the destinations and whose values are the running counts, and prints those keys in sorted order with the total below them.

```
 1  print("Type each destination, then a blank line.")
 2
 3  counts = {}
 4  total = 0
 5  while True:
 6      place = input().strip()
 7      if place == "":
 8          break
 9      counts[place] = counts.get(place, 0) + 1
10      total = total + 1
11
12  for place in sorted(counts):
13      print(f"{place}: {counts[place]}")
14
15  print(f"Tickets sold: {total}")
```

Typed at the program

```
Lupiro
Mahenge
Lupiro
Ruponda
Mahenge
Lupiro
Mahenge
```

Output

```
Type each destination, then a blank line.
Lupiro: 3
Mahenge: 3
Ruponda: 1
Tickets sold: 7
```

Program 2.27: *A tally kept in a dictionary and reported in order.*

Meeting the specification: The loop on line 5 has no fixed number of turns, so it is a while loop that leaves on a blank line. Line 9 raises the count for that destination, using get so that a destination seen for the first time starts at zero rather than raising. Line 12 sorts the keys before printing.

Why a dictionary and not two lists: Two lists, one of destinations and one of counts, would have to be kept in step by hand at every line that changes either. The dictionary makes that impossible, because the destination and its count are one entry. It also removes the search: a list would be scanned for the name before the count could be raised.

Reading the run: Seven destinations were typed and three of them are distinct, so three lines of counts are printed and the total below them is seven. Lupiro and Mahenge took three tickets each and Ruponda took one, and the counts add back to the total, which is the check worth making.

The blank line is the only signal the program needs, and the tally is complete the moment it arrives.

The Master Key: *Counting into a dictionary needs no second structure, and sorting the keys before printing makes the report the same every run.*

HOT Question 27

Two columns on a stock card at Ruponda, names down one side and quantities down the other, worked for a year. Then someone crossed out a name without crossing out its number, and every reading below it was wrong.

Argue both sides of the claim that a dictionary should always be preferred to two lists held side by side, and reach a stated verdict.

Model Answer

The claim is right in most cases and wrong in a few, and the condition that decides it is whether the two lists must stay in step. Where one list is the label of the other, a dictionary should replace them; where position itself is the meaning, it should not.

What two parallel lists actually are: They are one table pulled apart into two columns, with the position as the only thing holding them together. Nothing in the language records that connection; it lives in the head of whoever wrote the code, and it survives only while every line that changes one column remembers the other.

The case for the dictionary: It makes the pairing part of the data rather than part of the discipline. An entry cannot half exist: deleting a key deletes its value with it, and no arrangement of statements can leave the two out of step. Lookup is direct as well, where a parallel list must be searched for the name first.

The case for keeping two lists: The other side is not empty. A dictionary cannot hold the same key twice, so a record of every ticket sold must stay a list. A key must be unchangeable, so a list of readings cannot become one. And where position is the meaning, as in a queue served in order, a dictionary throws it away.

Where the argument is really decided: On what happens when the data changes. Two lists that are only ever read together are harmless; two lists that are added to, deleted from and sorted are a standing invitation to a fault that shows up far from where it was made. Sorting one column alone destroys every pairing in silence and raises nothing.

What neither side fixes: Both break down as soon as a row needs a third column. Three lists in step are worse than two, and a dictionary of dictionaries reads badly and gives no name to what it holds. At that point the answer is neither: it is a type of your own, with named fields, which is where this chapter goes next.

The verdict: Prefer the dictionary whenever one list is the label of the other and the pair must endure additions, deletions and sorting, which covers nearly every tally, register and price book you will write. Keep two lists only when duplicates must survive, when the keys could not be unchangeable, or when position is itself the information.

Two columns kept in step by hand is a promise, and a dictionary is the same promise kept by the language.

The Master Key: *Reach for a dictionary when one list would only ever be the label of another, and for two lists when position is itself the information.*

Functions and modules

This section defines the function and the module, the two ways a Python program is cut into named parts that can be used more than once. It separates a parameter from an argument and return from print, draws the four places a name is looked up and the two files an import joins, and builds a two-file program whose module holds the one number that must never be written twice. Everything the earlier sections established stands: a name is

bound by assignment, nothing is checked until the line that needs it runs, and a function is itself an object bound to a name.

BINDER Question 28

A tailor in Nyololo measures a customer, writes the figure on a slip, and hands it over. Her apprentice measures, calls the number out loud, and hands over nothing. Both did the work; only one shop can use it.

Define a function in Python, distinguish a parameter from an argument, and distinguish return from print.

Model Answer

A function is a named, reusable block of statements created by def and run each time its name is followed by brackets, and the four words that decide whether you can use one correctly are parameter, argument, return and print.

The shape of a definition: The keyword def is followed by the name, a bracketed list of parameters and a colon, and the indented block below belongs to the function until the indentation returns to the margin. Nothing in that block runs at the moment it is defined. It runs when the name is called.

A parameter against an argument: A parameter appears in the definition and is a name the function will use for whatever it is handed. An argument appears at the call and is the object actually passed. In Program 2.28 buckets is the parameter, while 3 and 2 are the arguments. They are not two words for one thing.

```
1  def charge(buckets):
2      return buckets * 500
3
4
5  owed = charge(3) + charge(2)
6  print(owed)
```

Output

```
2500
```

Program 2.28: *A function that gives its answer back, so the call can sit inside a larger expression.*

Handing a value back against showing it: The statement return hands an object back to whatever called the function, so the call can stand wherever a value can stand. The function print writes characters to the screen and hands nothing back. A function that prints has given nothing back; a function that returns has printed nothing.

```
1  def show_charge(buckets):
2      print(buckets * 500)
3
4
5  result = show_charge(3)
6  print(result)
```

Output

```
1500
None
```

Program 2.29: *A function that prints instead, and the None it hands back for want of a return.*

Reading the two listings: In Program 2.28 the call charge(3) becomes 1500 and can be added to charge(2), because return supplied a number. In Program 2.29 the call puts 1500 on the screen and then evaluates to None, which is what any function gives back when it reaches its end without a return statement.

The check is quick: if you cannot put the call on the right of an equals sign and get something usable, the function printed rather than returned.

The Master Key: *A parameter is written in the definition and an argument is passed at the call, and only return puts a value where the rest of the program can reach it.*

BINDER Question 29

The maize ledger at Mgololo has a running total on its front cover and a fresh total on every page inside. A clerk fills a page, closes the book, and the cover still reads what it read this morning.

Explain scope in Python and the order in which a name is looked up, and say what happens when a function assigns to a name that already exists outside it.

Model Answer

Python looks a name up in four places and stops at the first one holding it: the local scope of the function running now, then any enclosing function, then the module, then the built-in names, which is the order drawn in Figure 2.9.

The four places, innermost first: A name is searched for in the local scope first, then in any enclosing function that contains this one, then in the global scope of the module, and last among the built-in names such as len and print. The first place holding it wins and the search stops there.

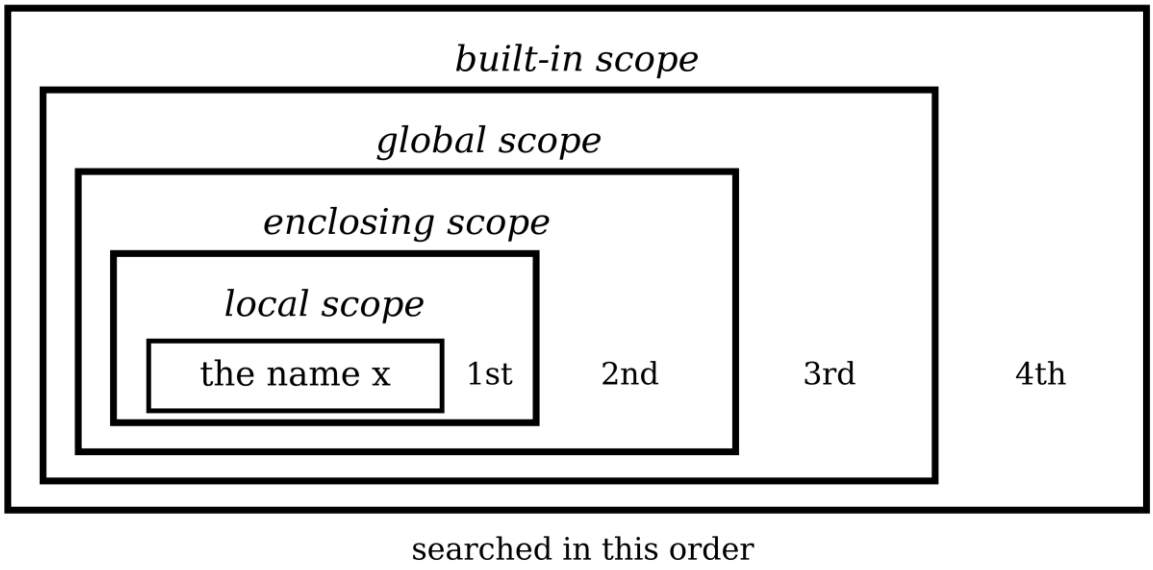


Figure 2.9: *The four places a name is looked up, innermost first.*

Assignment decides where a name lives: If a function assigns to a name anywhere in its body, that name is local to the function throughout it, even where a name spelled identically exists at module level. The function then has a name of its own, and the outer one is untouched by anything the function does.

```
litres_sold = 0


def record(amount):
    litres_sold = amount
    return litres_sold


print(record(40))
print(litres_sold)
litres_sold = record(40)
print(litres_sold)
```

Output

```
40
0
40
```

Program 2.30: *An assignment inside a function makes the name local, and returning the value is the repair.*

Reading the run: The module-level litres_sold starts at 0. The call record(40) gives back 40, yet the next line still prints 0, because the assignment inside record created a local name that vanished when the call ended. The last two lines rebind the module-level name to the returned value.

Why reaching for global is the wrong repair: Declaring a name global inside a function lets it rebind the module-level name, and that hides the effect at the call: the line calling the function shows no sign that anything outside it changed. Returning the value and assigning it where the call is written puts the change in plain sight.

Scope is not about where a name is written but about where Python is willing to look for it.

The Master Key: *A name assigned inside a function is local to that function, so the way to change something outside is to return the value and assign it there.*

BINDER Question 30

Three lorries leave Sao Hill and the loader writes each load on the same clipboard sheet, meaning to start a new one. By the third lorry the sheet carries every load since breakfast, and the total on it belongs to nobody.

Explain default arguments, keyword arguments, *args and **kwargs, and say what a mutable default argument does.

Model Answer

A default argument supplies a value the caller may leave out, a keyword argument names the parameter at the call so position stops mattering, *args gathers any remaining positional arguments into a tuple, and **kwargs gathers the named ones into a dictionary.

A value the caller may leave out: Writing price=500 in the definition gives that parameter something to use when the call supplies nothing. Parameters carrying defaults must come after those that do not, because Python matches positional arguments from the left and cannot step over a gap.

Naming the parameter at the call: A keyword argument is written as name=value at the call, and it reaches the parameter of that name whatever position it sits in. That lets a caller supply the third optional value without repeating the first two, and it makes a long call readable.

Collecting whatever was not matched: A parameter written *args collects the remaining positional arguments into a tuple, and one written **kwargs collects the remaining keyword arguments into a dictionary. One definition then accepts calls of different lengths, which is how a later section makes one name behave as though it had several forms.

Form	How it is written	What it is for
Parameter	def cost(buckets), called as cost(3)	matched by position, left to right; the ordinary case
Default argument	def cost(buckets, price=500)	supplies a value when the call leaves that one out
Keyword argument	cost(3, price=600)	names the parameter at the call, so order stops mattering
*args	def total(*amounts)	gathers the remaining positional arguments into a tuple
**kwargs	def note(**fields)	gathers the remaining keyword arguments into a dictionary

Table 2.4: *How a call supplies values, and what each form is for.*

The default object is built once, not once per call: A default value is created when the def line runs, not each time the function is called, so a list or dictionary written as a default is one object shared by every call that leaves it out. Program 2.31 runs cleanly and is wrong, which is why it catches people.

```
def logged(litres, log=[0]):
    """Add one reading to the log and return the log."""
    log.append(litres)
    return log


print(logged(12))
print(logged(15))
print(logged(9))
```

Output

```
[0, 12]
[0, 12, 15]
[0, 12, 15, 9]
```

Program 2.31: *The same function called three times, and the one list every call has been adding to.*

Reading the growing list: The three calls print two, three and four numbers. Each appended to the same list, because log was bound once to the list built with the definition. The repair is to default the parameter to None and build a fresh list inside the function.

Nothing in the printed source shows the sharing; only calling the function twice does.

The Master Key: *A default argument is a single object created with the definition, so a mutable default is shared by every call that leaves it out.*

BINDER Question 31

A borehole pump log at Igawa lives in one exercise book; the maintenance rules live in another, on a shelf, shared by four pumps. Copy those rules into every log and one correction has to be made four times.

Explain how a Python program is split into modules, show the three import forms, and say what the entry-point guard rests on.

Model Answer

A module is a Python file, and importing one runs it once and binds its top-level names so another file can reach them; the file Python was asked to run holds the name __main__ while an imported file holds its own module name, and the guard rests on that difference.

The three ways of importing: Writing import tools binds the module object to the name tools, and its contents are reached as tools.joined. Writing from tools import SEPARATOR binds that one name directly in the importing file. Writing import tools as helpers binds the module object to a name of your own choosing. Program 2.32 uses all three.

Where Python looks for the file: The search runs down the list in sys.path, which begins with the directory holding the file being run, then any directory named in the PYTHONPATH setting, then the standard library and the installed packages. A file of your own named after a standard module is found first and hides it, so keep module names plain and your own.

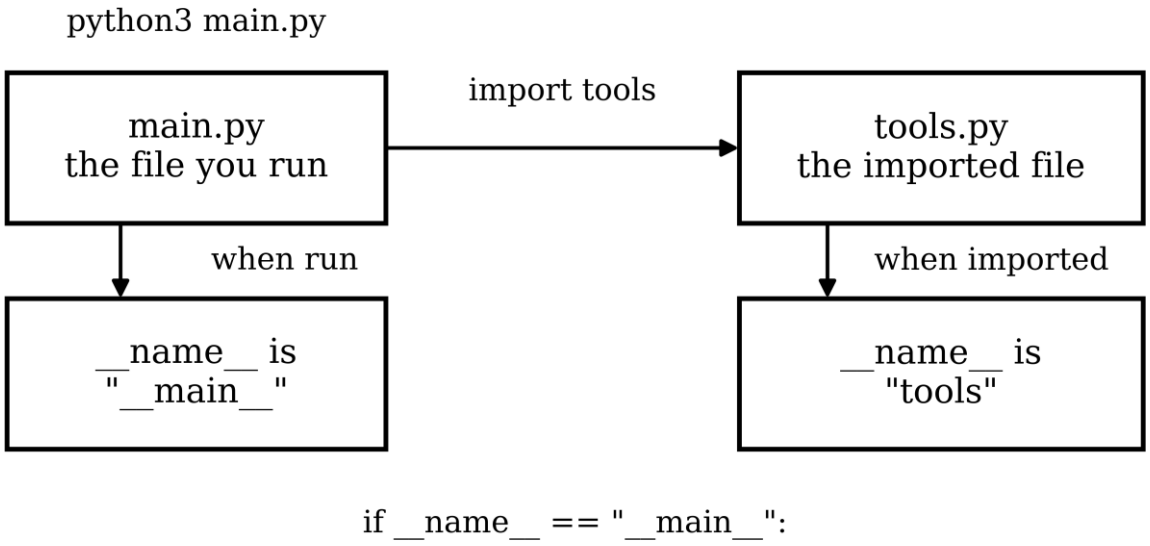


Figure 2.10: *Two files, one import, and what the name of each holds when one of them is run.*

What the name of a file holds: Every module carries a __name__. Python sets it to __main__ in the file it was asked to run and to the module's own file name in every file that arrived by import, which Figure 2.10 shows for both files at once.

tools.py

```
 1  """Small helpers that more than one program needs."""
 2
 3  SEPARATOR = " | "
 4
 5
 6  def joined(parts):
 7      """Return the parts run together with the separator."""
 8      return SEPARATOR.join(parts)
 9
10
11  if __name__ == "__main__":
12      print("tools.py was run on its own")
```

main.py

```
1  import tools
2  from tools import SEPARATOR
3  import tools as helpers
4
5  if __name__ == "__main__":
6      print(tools.joined(["Igawa", "Ubaruku"]))
7      print(SEPARATOR.join(["Sao Hill", "Isyonje"]))
8      print(helpers.joined(["Katumba", "Mgololo"]))
```

Output

```
Igawa | Ubaruku
Sao Hill | Isyonje
Katumba | Mgololo
```

Program 2.32: *One module, three import forms, and a guarded line that never runs because the module was imported.*

What the guard is for: Putting the __main__ guard round the code you want only when the file is run directly lets an import take the functions without the demonstration running too. In Program 2.32 the guarded line inside tools.py prints nothing, because tools.py was imported rather than run, and its name was tools.

A module is a file, and an import is one run of that file followed by a binding.

The Master Key: *The file Python was asked to run holds __main__ and every imported file holds its own name, which is the whole basis of the entry-point guard.*

REAL Question 32

The kiosk program at Ubaruku prints its heading, prints the morning total, and then stops with a message naming line seven. The two lines above it ran perfectly, and nothing about line seven looks wrong.

Find the fault in this kiosk listing, say exactly what Python does about it and when, and correct it.

Name the fault, give the line it sits on, and name what Python raises.

Say exactly when Python discovers it, and what had already been printed by then.

Correct the listing, and say why a compiled language would never have reached this point.

Model Answer

The function is defined on line 1 as daily_total and called on line 7 as daily_totals, so the program parses, runs correctly as far as line 6, and then stops with a NameError, having already printed two correct lines.

```
1  def daily_total(sales):
2      return sum(sales)
3
4
5  print("Kiosk report for Ubaruku")
6  print(daily_total([500, 500, 1000]))
7  print(daily_totals([250, 750]))
```

Python says

```
Kiosk report for Ubaruku
2000
Traceback (most recent call last):
  File "/home/form5/orders.py", line 7, in <module>
    print(daily_totals([250, 750]))
          ^^^^^^^^^^^^
NameError: name 'daily_totals' is not defined. Did you mean: 'daily_total'?
```

Program 2.33: *A listing that starts, works, and then stops, because one call names a function that was never defined.*

The fault: One name was defined and a different name was called. Line 1 creates daily_total; line 7 asks for daily_totals, with a trailing s that no definition anywhere gave. Python never guesses that two similar names were meant to be one, because a name is only ever what assignment or def bound it to.

What Python does about it, and when: It parses the whole file happily, since the listing breaks no rule of grammar, and begins executing. Lines 5 and 6 print their two correct lines. On line 7 the interpreter looks the name up locally, then in the module, then among the built-in names, finds it nowhere, and raises NameError at that instant.

What the message tells you: It names the file, gives line 7, reprints the offending line with the failing name underlined, and, since version 3.10, adds a suggestion of the defined name closest to the one you typed. That suggestion is a hint and not a repair; Python has changed nothing in your file.

The correction: Change line 7 to call daily_total, which is one character. The safer habit is to fix the definition and every call together by choosing one spelling and searching the file for the other, because the same slip in a rarely reached branch will sit undetected until somebody runs that branch.

Why nothing stopped this earlier: A compiled language resolves every call against a declaration before the program is allowed to start, and would have refused this whole file rather than print two lines from it. Python has no build step, so a name is checked at the moment it is reached and not one instant sooner. That is the price of the freedom the first section described.

A program that stops halfway has still told you something true about the lines that ran before it.

The Master Key: *A call that names no existing function is perfectly good grammar, so Python discovers it only when execution reaches that line, and everything above it has already happened.*

REAL Question 33

A water tank at Isyonje is seven metres on every side, and the volume written on the delivery note is forty-nine cubic metres. The note was produced by a program that has never once failed.

Say why this listing raises nothing at all, what its function actually computes, and how a fault of this kind is found.

Say why the listing raises nothing, and what Python is and is not checking.

State what the function computes, against what its name and docstring promise.

Give the correction, and the one hand-worked case that would have caught it.

Model Answer

Nothing is raised because every line is valid Python and every operation can be carried out: cube multiplies its parameter by itself once, which is a legal expression on two integers, so the program prints 49 where the volume of a seven metre cube is 343 and finishes without complaint.

```
1  def cube(n):
2      """Return the cube of the number given."""
3      return n * n
4
5
6  side = 7
7  print("Side in metres:", side)
8  print("Volume in cubic metres:", cube(side))
```

Output

```
Side in metres: 7
Volume in cubic metres: 49
```

Program 2.34: *A function named for a cube, returning a square, printed beside the number it really produced.*

What the function actually computes: It returns the square. A cube of n needs the parameter multiplied three times, as n * n * n, and the body multiplies it twice. The name promises one thing and the docstring on the second line promises the same thing, and the single line that matters delivers something else.

Why nothing is raised: An exception is raised when an operation cannot be carried out. Multiplying one integer by another can always be carried out, so there is no operation here for the interpreter to refuse. Python checks that the arithmetic is possible; it has no way of knowing what the answer was supposed to mean.

The size of what it got wrong: For a side of 7 the listing prints 49 where 343 was wanted, short by 294, and the gap widens with every larger side. A wrong number that is close to the right one is more dangerous still, because the reader of the delivery note has nothing to be suspicious of.

How a fault of this kind is found: By checking the returned value against one worked by hand. Take a side small enough to do in your head: the cube of 2 is 8, and this function gives 4. One case, worked independently and compared with what the call returns, exposes the whole fault in a second, which is why the function must return rather than print.

The correction: Make the body read return n * n * n, then run the hand-worked case again and see 8. Correct the docstring too where it disagrees with the body, because a docstring that describes something the code does not do is a second fault sitting quietly beside the first.

A program that stops tells you where to look, while a program that lies tells you nothing at all.

The Master Key: *An interpreter checks that an operation can be carried out and never that the answer means what the function is called, so only a value you worked yourself can catch a wrong formula.*

BUILD Question 34

Fifty shillings went onto the price of a bucket at Katumba this morning, and the attendant spent the evening hunting the old number through four screens of the program that prints her receipts. She found three of them.

Write a two-file program that prices buckets of water from a module constant and prints a receipt for one sale.

A module named charges.py holds the price of one bucket as a module-level constant.

It holds three functions: the cost of a number of buckets, the change owed from a payment, and one printable receipt line.

A file named main.py imports the module, reads the customer name and two numbers typed at the program, and prints the price, the total and the receipt.

Only main.py runs on its own, and it is guarded so that importing it does nothing.

Model Answer

The program is two files: charges.py holds the price of a bucket and the three functions that use it, and main.py imports that module, reads what is typed, and prints.

charges.py

```
 1  """Charges for water sold at the Ubaruku kiosk."""
 2
 3  PRICE_PER_BUCKET = 500
 4
 5
 6  def cost(buckets):
 7      """Return the charge in shillings for a number of buckets."""
 8      return buckets * PRICE_PER_BUCKET
 9
10
11  def change(paid, buckets):
12      """Return what is handed back when paid covers buckets."""
13      return paid - cost(buckets)
14
15
16  def receipt(name, buckets, paid):
17      """Return one line describing the whole sale."""
18      return f"{name}: {buckets} buckets, change {change(paid, buckets)}"
```

main.py

```
 1  import charges
 2
 3  if __name__ == "__main__":
 4      name = input("Name: ")
 5      buckets = int(input("Buckets: "))
 6      paid = int(input("Paid: "))
 7      print()
 8      print("Price per bucket:", charges.PRICE_PER_BUCKET)
 9      print("Total:", charges.cost(buckets))
10      print(charges.receipt(name, buckets, paid))
```

Typed at the program

```
Mama Neema
4
5000
```

Output

```
Name: Buckets: Paid:
Price per bucket: 500
Total: 2000
Mama Neema: 4 buckets, change 3000
```

Program 2.35: *A tariff module and the guarded file that imports it, priced from one constant.*

What the module holds: PRICE_PER_BUCKET is written once, in capitals because it is a constant, and read three times without being repeated: cost multiplies by it, change calls cost rather than multiplying again, and receipt calls change. Each function returns its answer, so each can be used by the next.

What the file that runs does: It imports charges by name, reads three values typed at it, converts the two numeric ones, and reaches everything through the module name. It prints the constant, the total and the receipt line, and it is guarded, so importing this file later brings in nothing but the import statement.

The decision worth naming: The price lives in the module and not in the file that runs, so a change of tariff is one edit in one place however many programs come to use charges.py. Nothing in main.py knows the number, which is why nothing in main.py has to be found and altered when it changes.

The specification is satisfied by two short files, and the number that changes most often appears in exactly one of them.

The Master Key: *A constant defined in a module and read through the module name is changed in one place, however many files depend on it.*

HOT Question 35

A printing shop in Mgololo has a machine that beeps once for every page it finishes, and the day's tally comes from whoever was in the room counting beeps. Nobody can add up yesterday's.

Argue both sides of the claim that a function should give its answer back rather than print it, and reach a judgement.

Model Answer

The claim is right for nearly every function you will write, and the condition deciding it is whether the function's result is the last thing the program has to say or something another part of the program still has to use.

The case for handing the value back: A returned object can be added, compared, stored in a list, passed to another function or given back again. Printed characters can do none of these. Returning separates working the answer out from showing it, and only the first of those two jobs is hard enough to deserve a function of its own.

A returned value can be checked against a hand-worked one: You can compare what a call gives back with a number you worked yourself, and disagreement is visible at once. A function that prints offers nothing to compare, so the only way to judge it is to read the screen and trust your eye, which is how a wrong formula survives for months.

It survives a change of destination: A function that returns can later have its answer shown on a screen, written into a report or added to a running total, with no change to the function itself. A function that prints has chosen the destination for every caller it will ever have, and chosen it at the wrong moment.

The case for printing: Some functions exist to display. A function whose stated job is to lay out a receipt, a heading or a table has presentation as its whole purpose, and making it return a string that its only caller immediately prints adds a step and buys nothing.

Printing is how you watch a program work: While a function is being written, a print inside it is the quickest way to see what actually happened at that line, and no reasonable programmer refuses that. It is a habit for the half hour you are writing the function, not a property of the finished one.

The judgement, and the condition that decides it: Return, unless the function's stated job is presentation. The condition is a single question: could any other line in this program, now or later, want that value? If it could, return it, and let one function at the edge of the program do the printing. Doing both in the same function is worse than either, because a caller that wants the value gets an unwanted line on the screen with it.

The verdict holds because a value can always be printed later, while printed characters can never be turned back into a value.

The Master Key: *Return the answer unless presenting it is the function's stated job, because printing throws away everything except the appearance of the result.*

HOT Question 36

Two teachers at Nyololo keep the school library's lending program: one wrote it in a single file over a weekend, the other has looked after it for three years and now opens six files to change a date.

Argue both sides of the claim that splitting a small program across modules costs more than it saves, and reach a judgement.

Model Answer

The claim holds while a program has one reader and a life measured in days, and fails the moment either of those changes, because the cost of a file boundary is paid once at the start while its benefit is collected by every later reader.

The case for keeping it in one file: A short program read from top to bottom needs no navigation. Every import is a name the reader must carry into another file, and a program cut up before anybody knows where its joints are ends with its boundaries in the wrong places.

The cost falls due immediately: A boundary costs a file name, an import line, and a decision about which side each function belongs on. All of it is spent on the first day, while nothing yet needs it, which is why the claim feels right to whoever has just paid.

The case for splitting: A module is a unit another program can use whole. The second program needing the same tariff imports it and pays nothing, while a copied block must be found and corrected twice for the rest of its life. A file name also says what is inside.

How many people read it is the first variable: One reader can hold the whole of a short program in mind and needs no map. Four people editing one file collide on the same lines and lose work, while a boundary tells each of them which file is theirs. What matters is the number of people who open the program, not its length.

How long it lives is the second: A program that runs twice next week and is then deleted should be written for whoever is writing it. A program still running years later is read far more often than it was written, and a file name telling a stranger where to look repays its cost at once.

The verdict, and the two things that decide it: Keep a program in one file while it has one reader and a life measured in days, and split it the moment a second person edits it or a second program needs one of its parts. Line count is the wrong measure: a long program with one reader belongs in one file, and a short one that four people share does not.

The claim is true of the program in front of you today and false of the same program a year later, which is why the decision belongs to its readers and its lifetime rather than to its length.

The Master Key: *Split a program when a second person edits it or a second program needs a part of it, and not because it has reached some number of lines.*

Classes, objects and self

This section defines the class and the object, and then supplies the one thing the common teaching leaves out: what self is and where it comes from. It separates an instance attribute from a class attribute by changing the shared value once and watching three objects see the change, separates the instance, class and static method by what each is handed for free, and finishes with three complete programs, every one of which declares a class. Everything the earlier sections established stands: a name is bound by assignment, nothing is checked until the line that needs it runs, and a class is itself an object bound to a name.

BINDER Question 37

Four meters of the same make hang on one wall at Ipinda, each showing a different number. The attendant knows one reading procedure and uses it on all four. Something has to tell that procedure which meter it is standing at.

Define a class, an object, an attribute and a method, and explain what self is and why every instance method carries it.

Model Answer

A class defines a kind of thing, an object is one thing made from that definition, an attribute is a value an object holds, and a method is a definition inside the class reached through an object, which is why it receives that object first, under the name self.

Class and object: The keyword class introduces the definition, and the indented block below says what every object of that kind holds and can do. An object is made by writing the class name with brackets, and every call gives a separate object with its own storage.

An attribute against a method: An attribute is a value the object holds, reached with a dot, as in m.reading. A method is a definition inside the class body, reached the same way and then called, as in m.read(). One the object has, the other it does.

A method is not simply a function: A method is reached through an object and receives that object as its first argument. A plain function receives only what the call passes it. That difference is the whole of what self is, and glossing a method as a function leaves the first parameter unexplained.

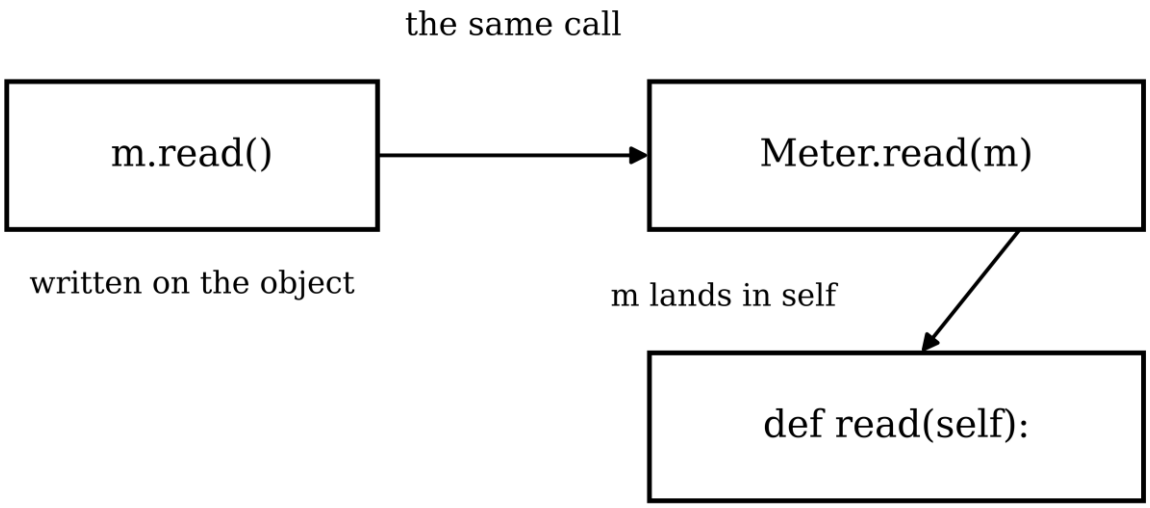


Figure 2.11: *The call written on the object, and the same call written on the class.*

```
 1  class Meter:
 2      def __init__(self, reading):
 3          self.reading = reading
 4
 5      def read(self):
 6          return self.reading
 7
 8
 9  m = Meter(4)
10  print(m.read())
11  print(Meter.read(m))
```

Output

```
4
4
```

Program 2.36: *One class, one object, and the same call written both ways.*

Python hands the object over: Line 10 passes nothing, and yet read declares a parameter. Python supplies the object written before the dot, so m.read() is Meter.read(m), which line 11 writes out in full. Both lines print 4, because both are one call reaching one object.

self is a convention, never renamed: Nothing in the language demands the name. Python binds the instance to whichever parameter stands first, so another word would work. It is called self everywhere and is never renamed, because every reader of Python reads self as the object.

The dot finds the method on the class and hands over the object standing in front of it.

The Master Key: *A method is reached through an object and is handed that object as its first argument, which is what self is and why it stands first.*

BINDER Question 38

One price card hangs above three counters at Itungi, and every attendant works from it. The morning it changes, all three counters charge the new price. One attendant then chalks her own figure on her own counter.

Distinguish an instance attribute from a class attribute, and show what happens when the shared name is assigned on one instance.

Model Answer

An instance attribute belongs to one object and exists once for every object made, while a class attribute is a single value held on the class itself and seen by every instance, so changing it once through the class changes what all of them read.

Where each one is written: An instance attribute is created by assigning to self inside __init__, so it comes into existence once for every object. A class attribute is written in the class body outside every method, so it comes into existence once, when the class definition runs.

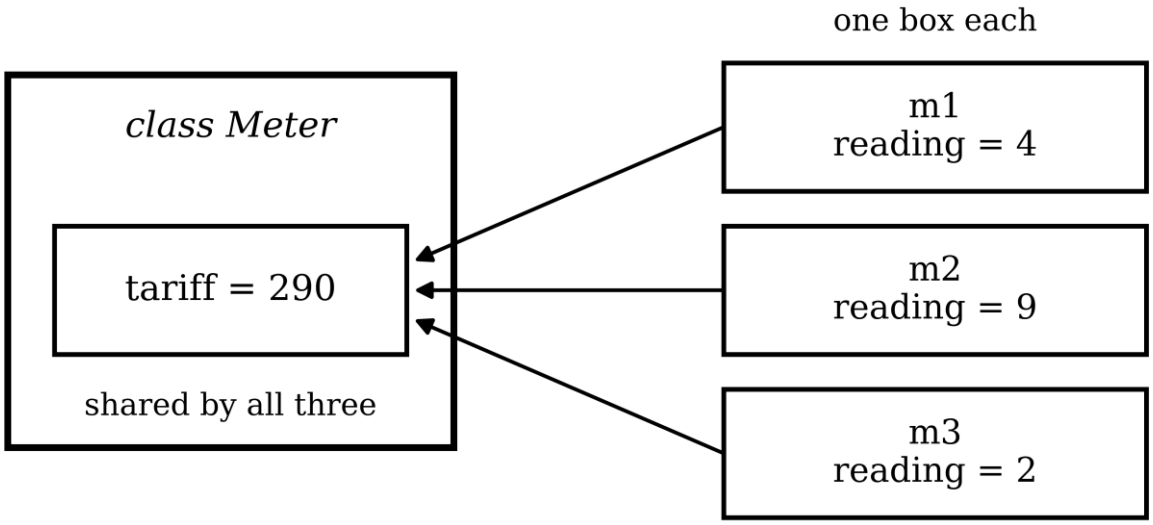


Figure 2.12: *Three objects with their own attributes, and one value they all share.*

One box against three: Figure 2.12 draws the difference as storage. The reading has a box for every meter, so three meters give three boxes holding three numbers. The tariff has exactly one box, and it lives on the class, so no meter has a copy to fall out of step.

```
 1  class Meter:
 2      tariff = 290
 3
 4      def __init__(self, reading):
 5          self.reading = reading
 6
 7
 8  m1 = Meter(4)
 9  m2 = Meter(9)
10  m3 = Meter(2)
11  print(m1.reading, m2.reading, m3.reading)
12  print(m1.tariff, m2.tariff, m3.tariff)
13  Meter.tariff = 310
14  print(m1.tariff, m2.tariff, m3.tariff)
15  m2.tariff = 250
16  print(m1.tariff, m2.tariff, m3.tariff)
17  print(Meter.tariff)
```

Output

```
4 9 2
290 290 290
310 310 310
310 250 310
310
```

Program 2.37: *The shared value changed once through the class, then hidden on one instance.*

Reading the shared value: Line 11 prints three different readings, because each meter holds its own. Line 12 prints 290 three times, since all three read one value through the class. Line 13 changes that value once, and line 14 shows all three now reading 310, though no instance was touched.

What assigning on one instance does: Line 15 looks like a change to the tariff and is not. Assigning to m2.tariff creates an instance attribute of that name on m2 alone, holding 250. Line 16 shows m1 and m3 still at 310, and line 17 shows the class attribute untouched.

The lookup order that explains it: A dotted name is looked for on the object first and the class second, and the search stops at the first place holding it. An instance attribute of the same name therefore hides the shared value for that object, while the others go on reaching the class.

That assignment did not change the class attribute; it made a new one that stands in front of it for a single object.

The Master Key: *A class attribute is one value shared by every instance, and assigning to that name on an instance hides the shared value for that object alone.*

BINDER Question 39

At the Matema landing the tally clerk has three jobs: weigh the catch of one boat, change the price the whole beach works from, and turn kilograms into sacks. Only the first needs a boat in front of her.

Distinguish an instance method, a class method and a static method, and say what each one is handed.

Model Answer

An instance method receives the object as its first argument under the name self, a class method carries the decorator classmethod and receives the class itself under the name cls, and a static method carries the decorator staticmethod and receives nothing automatically at all.

What the decorator decides: A definition inside a class body with no decorator above it is an instance method, and Python hands it the object it was reached through. The decorator classmethod replaces that first argument with the class. The decorator staticmethod removes it, so the definition receives only what the call actually passes, which Figure 2.13 sets out side by side.

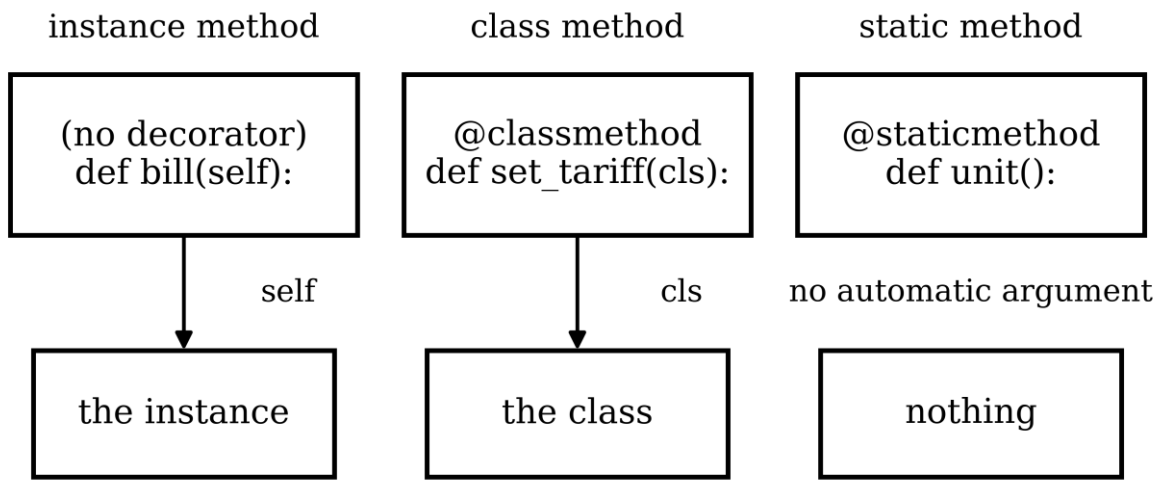


Figure 2.13: *What an instance method, a class method and a static method each receive.*

Kind	First argument	Called as	What it is for
Instance method	self, the object	m.bill()	working with the data of one object
Class method	cls, the class	Meter.set_tariff(310)	reading or changing what the class itself holds
Static method	nothing automatic	Meter.unit()	a helper that needs neither the object nor the class

Table 2.5: *The three kinds of method, what each is handed and what each is for.*

```
 1  class Meter:
 2      tariff = 290
 3
 4      def __init__(self, reading):
 5          self.reading = reading
 6
 7      def bill(self):
 8          return self.reading * self.tariff
 9
10      @classmethod
11      def set_tariff(cls, shillings):
12          cls.tariff = shillings
13
14      @staticmethod
15      def unit():
16          return "litres"
17
18
19  m = Meter(4)
20  print(m.bill(), Meter.unit())
21  Meter.set_tariff(310)
22  print(m.bill(), Meter.tariff)
```

Output

```
1160 litres
1240 310
```

Program 2.38: *All three kinds in one class, with the class method changing the shared value.*

A method that never uses cls is a static method: This is where the common teaching goes wrong. A class method exists to reach the class itself, most often to read or change a class attribute or to build an object by an alternative route. One decorated with classmethod that never mentions cls is a static method under the wrong decorator, and it gives a reader no reason to prefer either.

Reading the run: Line 20 bills four units at 290 and prints 1160, with the unit obtained from a method that was handed nothing. Line 21 calls set_tariff on the class, which assigns to cls.tariff. Line 22 bills the same meter at the new figure, 1240, and shows the class attribute now reading 310.

Why a static method is in the class at all: It could have been a plain function outside, because it receives nothing and reaches nothing. It is written inside because it belongs to the subject the class describes and is found by anyone reading that class. It is called through the class name, and it may be called before any object exists.

Choose by what the method needs to touch: one object, the class, or neither of them.

The Master Key: *An instance method is handed the object, a class method is handed the class and must use cls or it is a static method, and a static method is handed nothing.*

REAL Question 40

Two lorries queue at the Litembo weighbridge. The tally program prints the bale count for the first one and then stops, and the line it stopped on is the line that asks the load how much it weighs.

Name the fault in the weighbridge listing, say exactly what Python does about it and at what moment, and correct it.

Name the fault, give the line it is written on, and name what Python raises.

Say exactly when the fault is discovered and what had already been printed by then.

Correct the listing, and explain why the call itself looks blameless.

Model Answer

The method weight is defined on line 5 without self, so calling it on an object hands the object in as a first argument the definition has no room for, and Python raises TypeError at line 11 after the program has already printed one correct line.

```
 1  class Load:
 2      def __init__(self, bales):
 3          self.bales = bales
 4
 5      def weight():
 6          return self.bales * 50
 7
 8
 9  load = Load(12)
10  print("Bales on the lorry:", load.bales)
11  print("Weight in kilograms:", load.weight())
```

Python says

```
Bales on the lorry: 12
Traceback (most recent call last):
  File "/home/form5/lorry.py", line 11, in <module>
    print("Weight in kilograms:", load.weight())
                                  ^^^^^^^^^^^^^
TypeError: Load.weight() takes 0 positional arguments but 1 was given
```

Program 2.39: *An instance method defined without self, and the call that fails on it.*

The fault: Line 5 defines an instance method with an empty parameter list. Every instance method is handed the object it was reached through, so weight needs a first parameter to receive it, spelled self, and it has none. The body on line 6 already expects that parameter, since it reads self.bales.

What Python does about it, and when: It accepts the definition, builds the class and makes the object without a word, because a definition with no parameters breaks no rule of grammar. Line 10 prints correctly. On line 11 the call load.weight() becomes Load.weight(load), one argument arrives at a definition declaring none, and TypeError is raised at that instant.

Reading the message: It says that weight takes no positional arguments but was given one. The argument it means is the object, and the reason the message reads strangely to a beginner is that the brackets at the call are empty. Nothing visible was passed; Python passed it, exactly as Figure 2.11 draws.

The correction: Write line 5 as def weight(self): and change nothing else. The body was already correct, so twelve bales at fifty kilograms each print 600 and the program runs to its end. One word repairs it, and the word is the one the definition was always going to be handed.

Why nothing caught it sooner: The mismatch is between a definition and a call, and Python compares those only when the call is made. Had the method never been called, this file would have run to the end and printed one line, carrying a method that can never work. That is the price of having no build step.

Empty brackets at the call are the trap here, because the argument that overflows the definition is the one you never wrote.

The Master Key: *An instance method defined without self loads without complaint and raises TypeError the moment it is called on an object, because the object is passed whether or not a parameter waits for it.*

REAL Question 41

A record card at the Gumbiro dispensary opens without complaint, and the program prints a line saying so. Asked next for the name on that card, the same program stops and reports that the card has none.

Explain why this dispensary listing makes its object without complaint and then stops, and correct it.

Name what is misspelled, and say what Python was looking for instead.

Say what happened at the definition and what happened at the line that made the object.

Name what is raised, say which line raises it and why it is not the line at fault.

Model Answer

The constructor is spelled init on line 2 instead of __init__ with two underscores on each side, so Python never calls it, the object is made carrying no attributes at all, and AttributeError is raised on line 12 at the first line that tries to read one.

```
 1  class Card:
 2      def init(self):
 3          self.name = "not yet given"
 4          self.visits = 0
 5
 6      def summary(self):
 7          return f"{self.name}, visits {self.visits}"
 8
 9
10  card = Card()
11  print("Record card opened at the Gumbiro dispensary")
12  print(card.summary())
```

Python says

```
Record card opened at the Gumbiro dispensary
Traceback (most recent call last):
  File "/home/form5/card.py", line 12, in <module>
    print(card.summary())
          ^^^^^^^^^^^^^^
  File "/home/form5/card.py", line 7, in summary
    return f"{self.name}, visits {self.visits}"
             ^^^^^^^^^
AttributeError: 'Card' object has no attribute 'name'
```

Program 2.40: *A constructor whose name is missing its underscores, and the line that finds out.*

The fault: The constructor is __init__, with two underscores before the word and two after. Line 2 gives it four fewer characters. A common error is to copy the name from a heading that prints it as init, and the result is a perfectly ordinary method called init that nothing in Python ever calls.

Two things went wrong before any message appeared: First, line 2 was accepted as the definition of an ordinary method, because init is a legal method name and Python has no idea it was meant to be the constructor. Second, line 10 made an object and succeeded, handing back a Card carrying no attributes whatever. Neither produced a single word of complaint.

What making the object actually did: Calling the class looks for a method named __init__ to run on the new object. This class defines none, so nothing of yours runs and the object arrives empty. Line 11 then prints its line, which is why the reader believes the record card was opened and the program is working.

Where it stops, and why that line is blameless: Line 12 calls summary, which reaches line 7 and asks for self.name. Python looks for that attribute on the object, then on the class, finds it in neither, and raises AttributeError inside summary. The message names the class and the missing attribute, and the failing line is ten lines below the line at fault.

The correction: Rewrite line 2 as def __init__(self): and change nothing else. The two assignments then run at the moment the object is made, and every later line finds the attributes it expects. Check the spelling by counting underscores in pairs rather than by reading the word.

A silent definition and a silent instantiation are worse than a loud failure, because between them they moved the evidence ten lines away from the mistake.

The Master Key: *A constructor spelled anything other than __init__ is never called, so the object is made carrying none of its attributes and the first line that reads one raises AttributeError.*

BUILD Question 42

Eighty kilograms of sunflower seed went into the press before noon and twenty-four litres came out. The afternoon batch was smaller and gave more oil for every kilogram. The sheet on the wall shows only the two totals.

Write a pressing record class for a sunflower oil press, make two objects and print them both.

A class named Pressing holds the batch number, the seed in kilograms and the oil in litres.

One method returns the litres of oil obtained for each kilogram of seed.

The class defines __str__, so that printing an object prints one readable line.

The program makes two pressings and prints each object with its yield to two decimal places.

Model Answer

Program 2.41 meets the specification with one class: three instance attributes assigned in __init__, one method that works the yield out from two of them, and __str__ so that printing an object prints a line the press keeper can read.

```
 1  class Pressing:
 2      def __init__(self, batch, seed_kg, oil_litres):
 3          self.batch = batch
 4          self.seed_kg = seed_kg
 5          self.oil_litres = oil_litres
 6
 7      def litres_per_kg(self):
 8          return self.oil_litres / self.seed_kg
 9
10      def __str__(self):
11          return (f"Batch {self.batch}: {self.seed_kg} kg of seed, "
12                  f"{self.oil_litres} litres of oil")
13
14
15  if __name__ == "__main__":
16      morning = Pressing(1, 80, 24)
17      afternoon = Pressing(2, 50, 16)
18      print(morning)
19      print(f"Yield: {morning.litres_per_kg():.2f} litres a kilogram")
20      print(afternoon)
21      print(f"Yield: {afternoon.litres_per_kg():.2f} litres a kilogram")
```

Output

```
Batch 1: 80 kg of seed, 24 litres of oil
Yield: 0.30 litres a kilogram
Batch 2: 50 kg of seed, 16 litres of oil
Yield: 0.32 litres a kilogram
```

Program 2.41: *A class for one pressing, with a computed yield and a printable form.*

What each part of the class does: The three assignments on lines 3 to 5 give every pressing its own batch number, seed weight and oil volume. The method on line 7 divides one of those by another. The method on line 10 returns the line that describes the object, and print calls it for you on lines 18 and 20.

The decision worth naming: The yield is a method and not a fourth attribute. It is worked out from two values the object already holds, so storing it would be a fourth number able to disagree with the other three the moment either changes. Computed when it is asked for, it can never be stale.

What __str__ is buying: Printing an object of a class that defines none gives the class name and an address in angle brackets, which tells a press keeper nothing. With __str__ defined, the object itself is what the program prints, and the layout of that line is written once and used everywhere the object appears.

Two objects, three attributes each, and one derived figure that never has to be kept in step with them.

The Master Key: *A value computed from attributes the object already holds belongs in a method, and __str__ is what makes printing the object itself worth doing.*

BUILD Question 43

The ticket desk at Nakapanya opens at six, and by nine nobody can say how many tickets have gone out. Each one was written, torn off and handed over. The count exists only if somebody kept it while the tickets were being made.

Write a ticket class that counts how many tickets have been made and reports the count through a class method.

A class named Ticket holds the destination and the seat number of one ticket.

A class attribute counts every ticket made, and __init__ adds one to it.

A class method returns that count and reaches it through cls.

The program reports the count before any ticket exists, makes three tickets, prints each and reports the count again.

Model Answer

Program 2.42 keeps the count on the class rather than inside any object: issued is a class attribute, __init__ adds one to it as each ticket is made, and the class method sold returns it through cls, so the count exists before the first ticket.

```
 1  class Ticket:
 2      issued = 0
 3
 4      def __init__(self, destination, seat):
 5          self.destination = destination
 6          self.seat = seat
 7          Ticket.issued += 1
 8
 9      @classmethod
10      def sold(cls):
11          return cls.issued
12
13      def __str__(self):
14          return f"Seat {self.seat} to {self.destination}"
15
16
17  if __name__ == "__main__":
18      print("Tickets sold before the desk opened:", Ticket.sold())
19      first = Ticket("Nakapanya", 12)
20      second = Ticket("Lumecha", 13)
21      third = Ticket("Nakapanya", 14)
22      print(first)
23      print(second)
24      print(third)
25      print("Tickets sold by the class method:", Ticket.sold())
```

Output

```
Tickets sold before the desk opened: 0
Seat 12 to Nakapanya
Seat 13 to Lumecha
Seat 14 to Nakapanya
Tickets sold by the class method: 3
```

Program 2.42: *A class attribute counting the objects made, reported by a class method.*

Where the count has to live: No single ticket can know how many others were made, so the count cannot be an instance attribute. It is one value on the class, raised by one as each object is built, and it belongs to all the objects at once rather than to any of them.

Why the increment names the class: Line 7 reads Ticket.issued and not self.issued. Written the second way it would read the shared value, add one and store the answer as a new instance attribute on that ticket, leaving the class value at zero. That is the hiding shown in Program 2.37.

What the class method is for: sold takes cls and returns cls.issued, so it reads the class attribute through the class rather than through any object. Line 18 calls it before a single ticket exists, which no instance method could do, and line 25 calls it again after three.

The count is reported at line 18 with no tickets made and at line 25 with three, and no ticket was ever asked.

The Master Key: *A count of how many objects have been made belongs on the class, raised through the class name inside __init__ and read back by a class method that takes cls.*

BUILD Question 44

The meter at Matema read seven hundred and two last month and six hundred and ninety today. Water does not run backwards, so somebody typed a digit wrong, and nothing on the form refused it.

Write a meter reading class that takes what the user types, checks the pair of readings and reports the outcome.

A class named Reading holds the kiosk name and the old and the new meter readings.

One method returns the units used; another checks the pair and reports in words.

A new reading below the old one is refused in words, and nothing at all is raised.

The program builds a Reading from what is typed, twice over, and prints the object and the report each time.

Model Answer

Program 2.43 builds a Reading from three typed values and then asks the object to judge itself: report compares the new reading with the old, refuses in plain words when the new one is lower, and otherwise gives the units used.

```
 1  class Reading:
 2      def __init__(self, kiosk, old, new):
 3          self.kiosk = kiosk
 4          self.old = old
 5          self.new = new
 6
 7      def units(self):
 8          return self.new - self.old
 9
10      def report(self):
11          if self.new < self.old:
12              return f"Refused: {self.new} is below {self.old}"
13          return f"Accepted: {self.units()} units used"
14
15      def __str__(self):
16          return f"{self.kiosk} meter, {self.old} to {self.new}"
17
18
19  if __name__ == "__main__":
20      for _ in range(2):
21          kiosk = input("Kiosk: ")
22          old = int(input("Old reading: "))
23          new = int(input("New reading: "))
24          print()
25          entry = Reading(kiosk, old, new)
26          print(entry)
27          print(entry.report())
```

Typed at the program

```
Ipinda
418
465
Matema
702
690
```

Output

```
Kiosk: Old reading: New reading:
Ipinda meter, 418 to 465
Accepted: 47 units used
Kiosk: Old reading: New reading:
Matema meter, 702 to 690
Refused: 690 is below 702
```

Program 2.43: *A record built from what is typed, with a method that checks it and reports.*

What the object holds and what it works out: The three typed values become three attributes on lines 3 to 5, so each record carries the kiosk it belongs to along with both figures. The method on line 7 subtracts one from the other, and report on line 10 decides which of the two answers the reader is given.

Refusing without raising: The check is an if inside report, returning one string or the other, so nothing is raised and nothing needs to be caught. Both records are dealt with and the program reaches its end normally. A figure that is merely wrong deserves a sentence the clerk can read, not a stopped program.

Why the check belongs inside the class: The object holds both numbers, so it is the only thing in the program able to compare them. A check written outside would have to fetch both values first and would then be repeated at every place a reading is made, which is how two copies of one rule come to disagree.

The second record is refused in a sentence, and the run continues, because refusing is an answer and not a failure.

The Master Key: *A record class that holds its own values can check them itself, and a check that reports in words leaves the program running where a raised error would not.*

HOT Question 45

Every sack leaving the maize store carries two labels: a word on the outside that the buyer reads, and a code on the tag inside that only the storekeeper uses. Neither would do the other one's work.

Assess the claim that every class you print should define __str__, and reach a verdict.

Model Answer

The claim is nearly right and stops one class short of the truth, because the default it is fleeing from is the default of __repr__: define __repr__ on every class you will ever look at while the program is being written, and add __str__ only where the object has a second reader who is not you.

The case for the claim: A class that defines neither prints as its class name and an address in angle brackets. That is useless on a receipt, in a report or on any line a user reads, and the fault appears at the worst moment, when the program is otherwise finished. Defining __str__ removes the problem for good.

What printing an object actually does: The function print asks the object for its string form, which calls __str__. Where a class defines no __str__, that request falls back to __repr__, so a class carrying only __repr__ still prints something worth reading. The fallback runs in one direction only, and that is where the claim starts to leak.

```
 1  class Meter:
 2      def __init__(self, reading):
 3          self.reading = reading
 4
 5      def __repr__(self):
 6          return f"Meter({self.reading})"
 7
 8
 9  m = Meter(4)
10  print(m)
11  print([m, Meter(9)])
```

Output

```
Meter(4)
[Meter(4), Meter(9)]
```

Program 2.44: *A class with no __str__ at all, printed on its own and printed inside a list.*

Reading the two lines: Line 10 prints Meter(4) although the class defines no __str__, because the request fell back. Line 11 prints both objects inside the list, and this is the part the claim cannot cover: a list prints the repr of what it holds, never the str, so a class with __str__ alone shows the angle brackets here.

The two are written for different readers: __str__ is for whoever reads the output, so it may round a figure, drop detail and read as a sentence. __repr__ is for the programmer, so it should be unambiguous, name the class and show the values that tell this object from another of its kind. They are two audiences, not two styles.

The case against writing both on everything: Where a class has one audience, two methods returning nearly the same line are two things to keep in step, and in a year they will not be. The second is written to satisfy a rule rather than a reader, and text nobody reads is not a saving.

The verdict, and the condition that decides it: Define __repr__ on every class you will debug, and add __str__ only when somebody who is not reading the code will read the line. The deciding question is whether the object has two audiences. If it has one, write __repr__ alone, because the fallback lets it serve printing as well, and line 11 shows what a list does.

The claim points at a real defect and names the wrong repair, since the method that covers every case is the one written for the programmer.

The Master Key: *Give every class a __repr__ for the programmer and add a __str__ only where somebody outside the code reads the line, because the fallback runs from str to repr and never the other way.*

Encapsulation and abstraction

This section defines encapsulation and abstraction, the first two of the four pillars, and shows what Python actually gives you to enforce them, which is one convention and one rewriting rule. It draws the three names a single attribute can carry, the two paths a property routes reading and writing along, and an abstract base beside the one instantiation Python refuses to perform. It builds a class that guards a value against every caller and a small family of ticket sales that must all answer the same question. Everything the earlier sections established stands: assignment binds a name, and nothing is checked until the line that needs it runs.

BINDER Question 46

A tin at the Mkumbara water kiosk is labelled DO NOT OPEN, and the lid lifts off in one hand. Every child in the queue knows what is inside, and nobody has ever taken a shilling from it.

Define encapsulation, explain the single and double underscore conventions, and explain name mangling.

Model Answer

Encapsulation is bundling data with the methods that act on it and controlling how the rest of a program reaches that data, and Python has no access modifiers: a single leading underscore is a convention, a double leading underscore triggers name mangling, and neither of them prevents access.

What is bundled, and what is controlled: A class keeps data and the methods that act on it in one place, so values and the operations allowed on them travel together. The half of the definition usually dropped is the second: which of those names the rest of the program may reach.

One underscore is a request: A name carrying a single leading underscore, such as _serial, is marked as belonging to the inside of the class. The interpreter treats it like any other name; it is a message from one programmer to the next, honoured because honouring it costs nothing.

Two underscores rewrite the name: Two leading underscores inside a class body make Python store the attribute under another name: __litres in class Kiosk becomes _Kiosk__litres, which Figure 2.14 draws for an attribute called balance. The renaming is name mangling, and it exists to stop a subclass reusing a name the base class already uses.

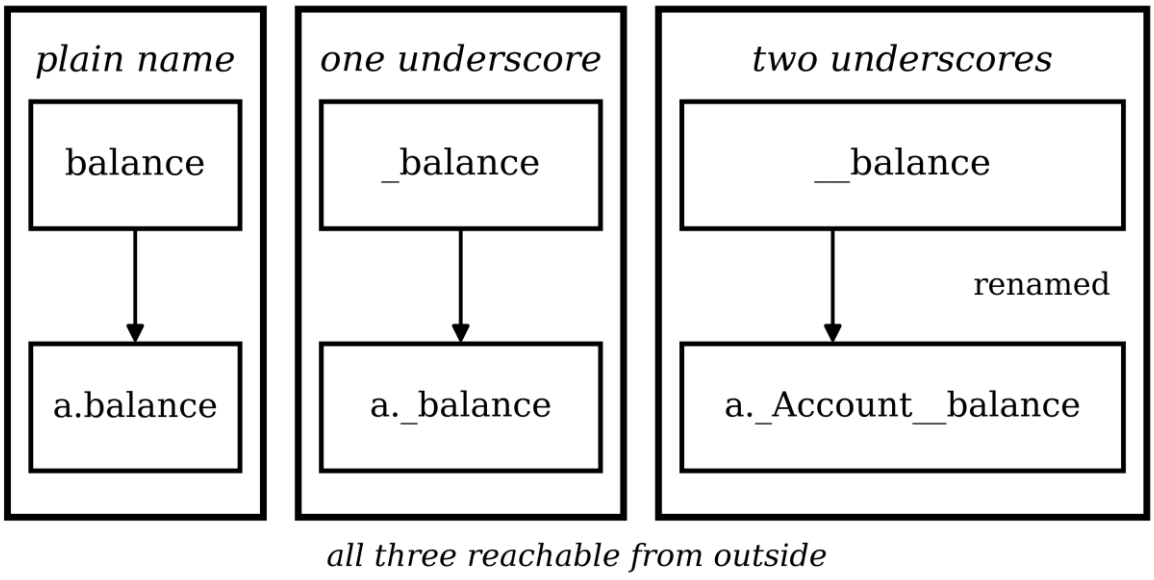


Figure 2.14: *A plain name, a protected name, and the name a double underscore is really stored under.*

```
 1  class Kiosk:
 2      def __init__(self, town, serial, litres):
 3          self.town = town
 4          self._serial = serial
 5          self.__litres = litres
 6
 7      def __str__(self):
 8          return f"{self.town} kiosk, meter {self._serial}"
 9
10
11  kiosk = Kiosk("Mkumbara", 41, 260)
12  print(kiosk)
13  print(sorted(vars(kiosk)))
14  print(hasattr(kiosk, "__litres"))
15  print(kiosk._Kiosk__litres)
```

Output

```
Mkumbara kiosk, meter 41
['_Kiosk__litres', '_serial', 'town']
False
260
```

Program 2.45: *Three attributes written three ways, the names the object really holds, and the value reached through the mangled one.*

Nothing in Python is out of reach: Line 13 lists the names the object really holds, sorted so the mangled one is in plain view. Line 14 asks whether the spelling __litres exists and gets False, because nothing is stored under it, and line 15 reaches 260 through the mangled name. Students are often taught, wrongly, that a double underscore makes an attribute private; that last line is the answer.

Mangling protects a class from an accident and never from a person, and the label on the tin is worth writing even so.

The Master Key: *A single underscore asks the rest of the program to keep out, a double underscore renames the attribute to _ClassName__attribute, and neither of them locks anything.*

BINDER Question 47

The sunflower press at Makuyuni has one lever and a locked steel door. The woman working it can tell you the litres she gets from a sack; she has never seen the screw turning inside, and has never needed to.

Distinguish encapsulation from abstraction, and show both at work in one class.

Model Answer

Encapsulation is the mechanism and abstraction is the result: encapsulation bundles data with the methods that act on it and controls access to that data, while abstraction exposes only what a user of the class needs and hides how the work is done.

The question each one answers: Encapsulation answers who may touch this data. Abstraction answers how much a user has to know before this class is usable. The first is a decision about names and access; the second is a decision about the surface the class shows the world.

Why the mechanism comes first: You encapsulate in order to abstract. Keeping the litres and the capacity inside Tank is encapsulation; offering fill and level and nothing else is abstraction. Take the first away and the second collapses, because a caller free to reach the data directly has no reason to go through the interface at all.

Neither word means hiding: Hiding describes both of them badly. Encapsulation is bundling plus control, and control includes deciding what stays reachable. Abstraction is a choice about what to show, and a class showing nothing would be useless. A common error is to give the two words one definition and lose the distinction between them.

Question	Encapsulation	Abstraction
What it is	bundling data with the methods that act on it, and controlling access to it	exposing only what a user needs and hiding how it is done
What it controls	which names the rest of the program may reach	how much a user has to know
Where you see it	the underscore on an attribute name	the list of methods the class offers
What Python provides	a convention, and name mangling	a small public surface, and the abc module
Which one it is	the mechanism	the result

Table 2.6: *Encapsulation against abstraction, and the question each one answers.*

```
 1  class Tank:
 2      def __init__(self, capacity):
 3          self.__capacity = capacity
 4          self.__litres = 0
 5
 6      def fill(self, litres):
 7          room = self.__capacity - self.__litres
 8          self.__litres = self.__litres + min(litres, room)
 9
10      def level(self):
11          return round(self.__litres * 100 / self.__capacity)
12
13      def __str__(self):
14          return f"Makuyuni tank at {self.level()} per cent"
15
16
17  tank = Tank(5000)
18  tank.fill(2000)
19  print(tank.level())
20  tank.fill(4000)
21  print(tank)
```

Output

```
40
Makuyuni tank at 100 per cent
```

Program 2.46: *One class encapsulating two values and abstracting away the arithmetic that uses them.*

Reading the two ideas off the listing: The two values are inside and named so that nothing outside is meant to touch them, which is the encapsulation. The caller writes fill and level and learns neither that the second delivery of 4000 was clipped to the 3000 the tank had room for, nor that a level is a percentage worked out by division, which is the abstraction.

One of them is what you do to the data, and the other is what the reader of your class is spared.

The Master Key: *Encapsulation is the mechanism and abstraction is the result, so a class is encapsulated in order that it can be abstract, and neither word on its own means hiding.*

BINDER Question 48

Nobody at the Mererani tap noticed the change. The handle turns the same half turn it always did, and behind the wall a meter was fitted in March that now counts every litre drawn from it.

Explain the property and its setter, and say why Python prefers them to a pair of get and set methods.

Model Answer

A property is a method reached as though it were a plain attribute: the method under @property runs when the name is read, the method under the matching setter runs when the name is written, and not one character of the code doing the reading and the writing has to change.

How the pair is written: The getter is a method carrying @property whose name is the attribute a caller will write. The setter carries @litres.setter, takes the same name and one extra parameter for the incoming value, and is the only route a write can travel down.

What the calling code sees: It writes meter.litres and meter.litres = 265, with no brackets anywhere. A pair of get_litres and set_litres methods would put brackets into every line that used them, and the day the pair was introduced, every one of those lines would have to be found and rewritten.

Why Python needs no defensive getters: Where the choice is permanent, programmers write get and set methods on the first day for attributes that will never need a check, because changing their minds later breaks every caller. Python lets a class publish a plain attribute and route it through a property on the day a check is finally wanted.

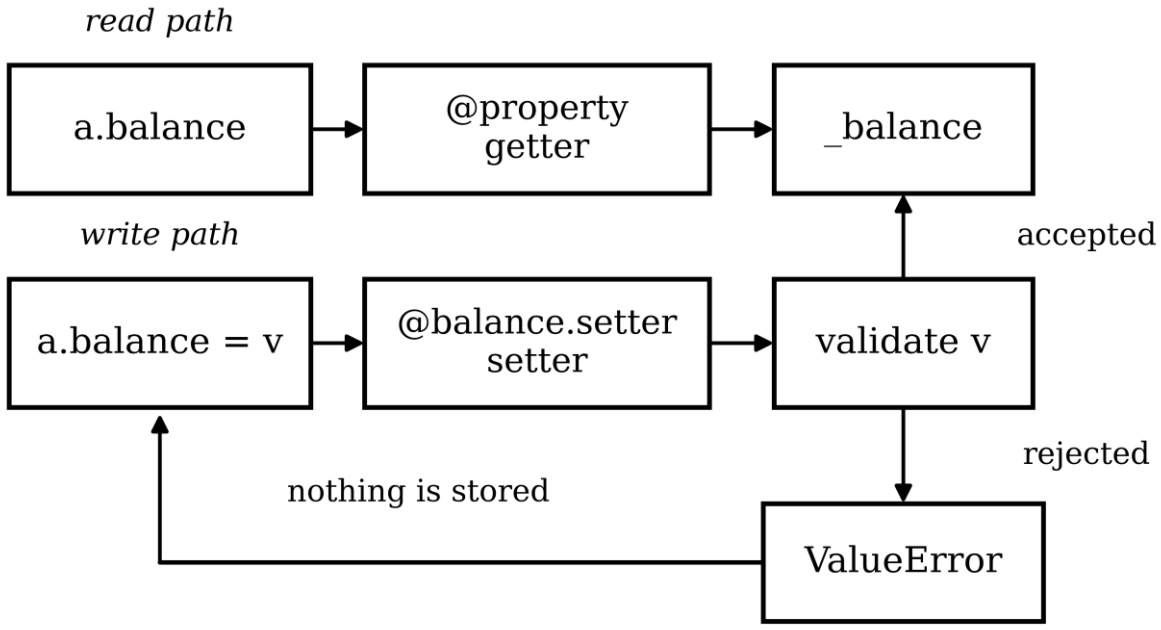


Figure 2.15: *Reading and writing an attribute routed through a property, and where the check sits.*

```
 1  PRICE = 50
 2
 3
 4  class Reading:
 5      def __init__(self, litres):
 6          self._litres = litres
 7
 8      @property
 9      def litres(self):
10          return self._litres
11
12      @litres.setter
13      def litres(self, value):
14          self._litres = value
15
16      @property
17      def cost(self):
18          return self._litres * PRICE
19
20      def __str__(self):
21          return f"Mererani meter: {self._litres} litres"
22
23
24  meter = Reading(240)
25  print(meter.litres, meter.cost)
26  meter.litres = 265
27  print(meter.litres, meter.cost)
28  print(meter)
```

Output

```
240 12000
265 13250
Mererani meter: 265 litres
```

Program 2.47: *A stored value reached through a property and its setter, beside a second property that is not stored at all.*

A property need not be stored anywhere: Reading meter.cost runs a multiplication and hands back a number the object never held. Figure 2.15, drawn for an attribute called balance, marks where a check belongs on the write path, and this listing leaves it empty; the next question fills it.

Reading the run: The first line prints 240 and 12000, the second prints 265 and 13250 after the single assignment on line 26, and nothing outside the class multiplied anything or knew that the litres sit in an attribute called _litres.

The property is why a Python class can begin plain and grow careful without disturbing its callers.

The Master Key: *A property turns a plain attribute into a checked one without changing a single line of the code that reads or writes it.*

REAL Question 49

A clerk at Naberera types a price of minus fifty into the stock book by mistake. The program takes it without a murmur, the shelf label still shows the old figure, and nobody can now say which of the two the shop is using.

Work through validation in a property setter, and say what a setter must do when it refuses a value.

Say where the check belongs and why that is the only place it works.

Trace the two assignments to price in the listing and give the value stored after each.

Say what the setter must do when it refuses, beyond not storing the value.

Model Answer

A setter that validates has three jobs, and the third is the one most often left out: judge the incoming value, leave the stored value untouched when the value fails, and report the refusal, because a write

discarded in silence leaves the program holding one number and the person who typed it believing another.

Where the check belongs: Inside the setter, which is the single route a write can take. Put it in the code doing the assigning and it has to be written again at every place that assigns, and it will be forgotten at one of them, usually the one added last.

```
 1  class Tariff:
 2      def __init__(self, price):
 3          self._price = price
 4
 5      @property
 6      def price(self):
 7          return self._price
 8
 9      @price.setter
10      def price(self, value):
11          if value <= 0:
12              print(f"Refused: {value} is not a price above zero")
13              return
14          self._price = value
15
16      def __str__(self):
17          return f"Naberera bucket: {self._price} shillings"
18
19
20  tariff = Tariff(500)
21  tariff.price = 600
22  print(tariff.price)
23  tariff.price = -50
24  print(tariff.price)
25  print(tariff)
```

Output

```
600
Refused: -50 is not a price above zero
600
Naberera bucket: 600 shillings
```

Program 2.48: *A setter that refuses a value, says so, and leaves the value already stored exactly where it was.*

What a refusal must not do: It must not store the bad value, and it must not stay quiet. The setter printed in the common teaching assigns when the value is good and does nothing whatever when it is not, so the line that failed looks on the screen exactly like the line that worked, and the program carries on.

Reporting is the cheap half of the job: One printed line naming the value and the reason turns an invisible failure into a visible one, and it costs a single statement. The other answer is to stop the program at that point by raising the ValueError drawn in Figure 2.15, which needs machinery the last section of this chapter supplies; refusing and saying so is this section's answer.

Reading the trace: Line 21 assigns 600, which passes the check, so line 22 prints 600. Line 23 assigns -50, which fails on line 11, so the setter prints its refusal and returns on line 13 before reaching the assignment on line 14. Line 24 then prints 600 again.

Why reading it back is part of the answer: Printing the attribute after a refused write is what turns a claim into evidence. Without that line the reader has to take on trust that nothing was stored; with it, the same 600 appears before and after, and the last line shows the object still describing itself with the value it kept.

A refused write should be as loud as an accepted one is quiet.

The Master Key: *A setter that refuses must leave the old value in place and say that it refused, because a silent discard leaves the program and its user believing two different numbers.*

REAL Question 50

The ticket desk program at Melela prints the cash total, prints the mobile money total, then stops dead on the third line. Nothing was edited between them, and the third line looks like the two that had just worked.

Find the fault in this ticket desk listing, say what Python refuses to do and at which line, and correct it.

Name the fault and give the line it sits on.

Say what Python does about it, and how much had already been printed by then.

Correct the listing, and say what the abstract method was there to guarantee.

Model Answer

Line 26 asks Python to build a Sale, and Sale is an abstract base class holding an abstract method that nothing has implemented, so the listing parses, runs, prints two correct totals and then stops with a TypeError raised at that line.

```
 1  from abc import ABC, abstractmethod
 2
 3
 4  class Sale(ABC):
 5      def __init__(self, seats, fare):
 6          self.seats = seats
 7          self.fare = fare
 8
 9      @abstractmethod
10      def total(self):
11          pass
12
13
14  class Cash(Sale):
15      def total(self):
16          return self.seats * self.fare
17
18
19  class Mobile(Sale):
20      def total(self):
21          return self.seats * self.fare + 500
22
23
24  print("Cash:", Cash(3, 12000).total())
25  print("Mobile:", Mobile(2, 12000).total())
26  print("Desk:", Sale(1, 12000).total())
```

Python says

```
Cash: 36000
Mobile: 24500
Traceback (most recent call last):
  File "/home/form5/tickets.py", line 26, in <module>
    print("Desk:", Sale(1, 12000).total())
                   ^^^^^^^^^^^^^^
TypeError: Can't instantiate abstract class Sale with abstract method total
```

Program 2.49: *A listing that works twice and is then refused, because the class it asks for is the abstract one.*

The fault: Line 26 instantiates the base class itself. Sale exists to say that every sale must be able to produce a total; it supplies no way of producing one, and the abc module is what turns that statement into something Python enforces rather than something a reader is asked to notice.

What Python does about it, and when: It parses the file, since nothing here breaks a rule of grammar, and it runs. Lines 24 and 25 print two correct totals. On line 26 the attempt to build the object is refused before any of the object's own code runs, and a TypeError is raised at that instant.

What the message tells you: It names the class Python would not build and the abstract method left unimplemented, which is how you find the missing one in a base declaring several. The wording of that sentence was changed after version 3.11, so read the two names in it rather than memorising the line.

The correction: Instantiate a concrete subclass: make a Cash or a Mobile on line 26 and the program runs to the end. If a plain Sale is genuinely wanted, then Sale is not abstract and the decorator should come off it. What cannot stand is the listing as printed, which asks for both at once.

The one place abstraction is enforced: Every other thing this section describes is a convention that asks politely. This refusal is the one point at which Python stops a program rather than trusting it, and it stops it for a single reason: a promise was declared in the base and no implementation of it exists in the class being built.

The two totals above the message are not consolation; they are the evidence that everything except the direct instantiation worked.

The Master Key: *A class holding an unimplemented abstract method cannot be instantiated, and Python refuses at the line asking for the object, after everything above it has already run.*

BUILD Question 51

4100 at the Kijungu borehole in the morning, 4380 at dusk, and the reader who comes on Monday writes 4200 in the same column. The pump has not stopped turning since Friday, and one of those three numbers cannot be right.

Write a program whose Pump class guards its meter reading with a property setter, and run it against an accepted reading and a refused one.

A class named Pump holds the town and one meter reading, and the reading is reached through a property.

The rule: a reading is accepted only when it is not lower than the reading already stored, because a meter counts upward.

A refused write prints the value refused and the value kept, and leaves the stored reading where it was.

The three readings are typed at the program, and only the file that runs prints anything.

Model Answer

The program is one class, Pump, whose reading is reached through a property, and whose setter carries the single rule that a meter counts upward: a value below the one already stored is refused, reported and dropped, and the stored reading survives it.

```
 1  class Pump:
 2      def __init__(self, town, reading):
 3          self.town = town
 4          self._reading = reading
 5
 6      @property
 7      def reading(self):
 8          return self._reading
 9
10      @reading.setter
11      def reading(self, value):
12          if value < self._reading:
13              print(f"Refused: {value} is below {self._reading}")
14              return
15          self._reading = value
16
17      def __str__(self):
18          return f"{self.town} meter stands at {self._reading}"
19
20
21  if __name__ == "__main__":
22      opening = int(input("Opening reading: "))
23      evening = int(input("Evening reading: "))
24      monday = int(input("Monday reading: "))
25      print()
26      pump = Pump("Kijungu", opening)
27      pump.reading = evening
28      print(pump)
29      pump.reading = monday
30      print(pump)
```

Typed at the program

```
4100
4380
4200
```

Output

```
Opening reading: Evening reading: Monday reading:
Kijungu meter stands at 4380
Refused: 4200 is below 4380
Kijungu meter stands at 4380
```

Program 2.50: *A meter reading guarded by the setter of a property, run against one reading it accepts and one it refuses.*

What the class guards and what it leaves open: The town is a plain attribute, because one town name is as good as another. The reading is not, so it is stored as _reading and published as a property, and the only way to write it runs through the setter holding the rule.

What the run shows: 4380 is accepted and the meter reports itself. 4200 is refused, the refusal names both numbers, and the meter still stands at 4380 afterwards. The three values are typed at the program, so the same listing takes any three readings somebody wants to try on it.

The decision worth naming: The rule lives in the class and not in the code that reads the numbers. A second program pointed at this class cannot get a lower reading into it either, which is the difference between checking one lot of data and guarding an attribute.

Anything that writes a reading obeys the rule, because there is no other way in.

The Master Key: *A rule written into a setter is enforced for every caller a class will ever have, and a program taking numbers from a person is exactly where that matters.*

BUILD Question 52

Every fare at the Ngerengere bus desk is worked out differently depending on how it was paid. Each one still has to produce a single number for the tally sheet the clerk adds up at closing time.

Write a program whose abstract Sale class forces two kinds of ticket sale to produce a total, and use one sale of each kind.

An abstract base named Sale holds the seats and the fare and declares one abstract method, total.

Cash returns the seats times the fare. Mobile adds a fixed service fee to that same product.

Printing a sale shows its seats and its total, and the base is the only place that layout is written.

The program makes one sale of each kind, prints both, and prints the takings of the two together.

Model Answer

The program is an abstract base and two concrete subclasses: Sale declares that every sale can produce a total and cannot be built itself, Cash and Mobile each supply that total in their own way, and the file that runs makes one of each and prints the pair.

```
 1  from abc import ABC, abstractmethod
 2
 3  SERVICE_FEE = 500
 4
 5
 6  class Sale(ABC):
 7      def __init__(self, seats, fare):
 8          self.seats = seats
 9          self.fare = fare
10
11      @abstractmethod
12      def total(self):
13          pass
14
15      def __str__(self):
16          return f"{self.seats} seats: {self.total()} shillings"
17
18
19  class Cash(Sale):
20      def total(self):
21          return self.seats * self.fare
22
23
24  class Mobile(Sale):
25      def total(self):
26          return self.seats * self.fare + SERVICE_FEE
27
28
29  if __name__ == "__main__":
30      cash = Cash(3, 12000)
31      mobile = Mobile(2, 12000)
32      print(cash)
33      print(mobile)
34      print("Ngerengere takings:", cash.total() + mobile.total())
```

Output

```
3 seats: 36000 shillings
2 seats: 24500 shillings
Ngerengere takings: 60500
```

Program 2.51: *An abstract base fixing the question, and two classes that answer it differently.*

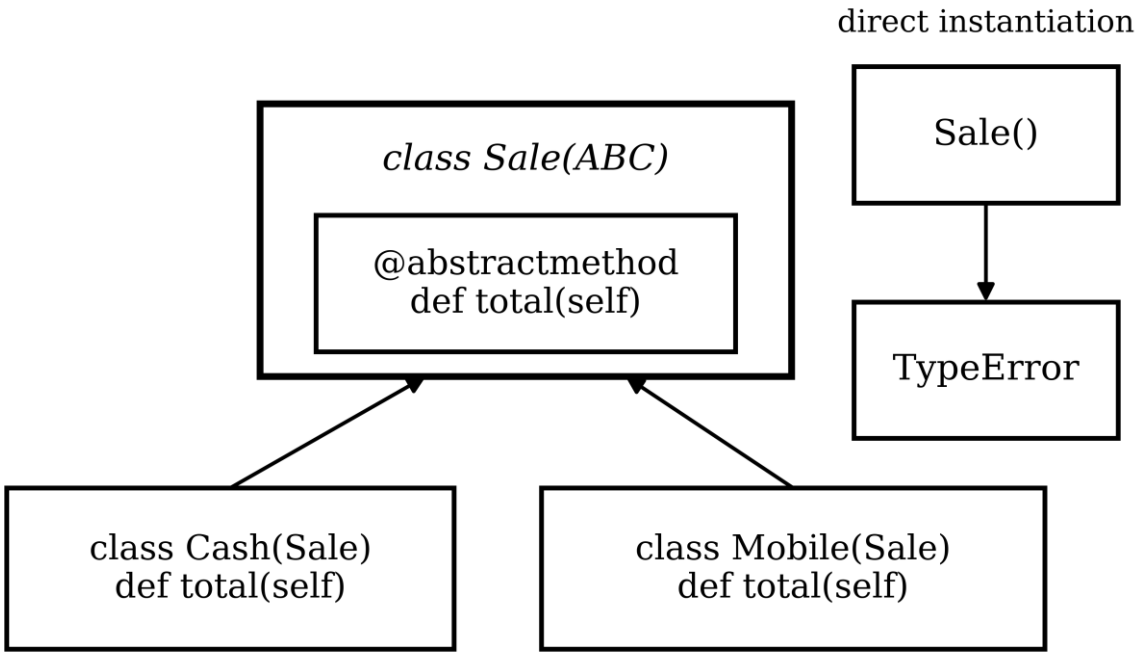


Figure 2.16: *An abstract base, two classes that implement it, and the instantiation Python refuses.*

What the base fixes and what it leaves open: Sale fixes the data every sale carries and the one question every sale must answer. It fixes no part of the answer. Marking total abstract makes that requirement something Python collects rather than something a reader is trusted to honour, which is what Figure 2.16 shows on its right.

What the run prints: 36000 for three seats at 12000 each, 24500 for two of the same seats once the 500 fee is added, and 60500 for the two sales together. Each of those numbers was worked out inside the class that knows how its own kind of sale is priced.

The decision worth naming: The layout of a printed sale is written once, in the base, and the fee is a module constant rather than a number buried in a method. A third way of paying is a class of three lines, and nothing already written has to be opened to add it.

The specification is met by one base that states the question and two classes that answer it.

The Master Key: *An abstract base states what every subclass must be able to do, and Python refuses to build anything that has not done it.*

HOT Question 53

A card taped above a repair counter says: parts in the blue tray are spoken for. There is no lock on the tray, no lid and no list of names, and in eleven years nothing has walked out of it.

Argue both sides of the claim that a leading underscore is worthless because it prevents nothing, and reach a judgement.

Model Answer

The claim is right about enforcement and wrong about engineering: one underscore prevents nothing, and it still does most of the work an access modifier does, because the reader it is written for is a colleague deciding what to depend on, not an intruder deciding what to break.

The case for the claim: Nothing stops the write. A name carrying one underscore is read, changed and deleted like any other, and a class putting its whole state behind underscores has published all of it. A protection defeated by not caring is not a protection.

The language offers no help either: There is no compiler to refuse a program that touches an underscored name, and no refusal when it runs. The underscore is a comment standing where another language would put a keyword, and it can mislead a reader arriving from that language.

The case against the claim: The convention is universal, and that is the property doing the work. Every Python programmer reads _reading as internal, so the underscore separates the part of a class other code may depend on from the part its author intends to change next month. An access modifier draws that same line.

What a convention buys that a keyword does not: It is free to relax. A private keyword chosen wrongly forces a change to the class and to everything compiled against it, while an underscore chosen wrongly is a rename. It also leaves open the two honest reasons for reaching inside: reading a value while hunting a fault, and checking one from a program written to try the class out.

Who the reader is decides the argument: If the reader is a colleague choosing which names to build on, a convention every one of them knows is enough, and nothing stronger would change what they do. If the reader is hostile, no keyword in any language was ever enough, because the source and the data are already on their machine.

The verdict, and the condition that decides it: Keep the underscore: it is worthless as a lock and valuable as a boundary. The condition is whether the people reading the code share the convention. Inside a Python program, where they all do, it does the work of a modifier at none of the cost. Where a class is reached by people or tools that do not share the habit, the boundary has to be written down as a documented interface, and the underscore becomes a note beside it rather than the rule.

A convention everybody knows is not enforcement, and it is a long way from nothing.

The Master Key: *A leading underscore prevents nothing and still marks the line between what a class promises and what it merely happens to contain, and that line is the one a reader needs.*

HOT Question 54

A carpenter cuts the joints for a roof before the walls are up, and the timber is beautiful. The builder then moves one wall by half a metre, and every joint has to be cut a second time.

Argue both sides of the claim that every class hierarchy should begin with an abstract base class, and reach a judgement.

Model Answer

The claim is wrong as a rule and right in one situation, and what decides it is whether you can already name the thing every member of the family must be able to do, because an abstract base is a promise collected from every subclass and a promise written too early is the most expensive line in the file.

The case for starting abstract: It states the shared question once. A base declaring total abstract fixes the one thing every sale must answer, and a subclass forgetting to answer it fails when somebody tries to build it rather than when somebody calls a method nobody wrote.

It gives the family a name and a home: The data every member carries has one location, and a reader learns the whole family from the base. Where several classes genuinely are kinds of one thing, the base was going to be written anyway, and declaring it abstract only stops it being built by accident.

The case against: An interface is a commitment, and the cost of getting it wrong is paid by every class that already implements it. Change one abstract method after four subclasses exist and all four stop being buildable at once; that is the machinery working, and it is still four edits you invited.

Most families do not exist yet: You usually have one class. A base built over one class is a guess about the second and the third, made when you know least about either. The cheap order is to write the second class first and lift the base out of what the two share.

The family works without the base: A method is found on the object in front of you rather than on a declared type, so classes sharing no base can already be used the same way. The base adds a guarantee, not an ability, which is why the decision turns on certainty and not on whether the program runs.

The verdict, and the condition that decides it: Begin with an abstract base only when two things are already true: two or more classes exist that are genuinely kinds of one thing, and you can name the method every one of them must supply. Until both hold, write the concrete class. Taken as a rule the claim turns a sharp tool into a ceremony, and the bill arrives on the day the interface turns out to have been wrong.

An abstract base earns its price once the family and its shared question are known, and is a guess in the clothes of a guarantee before then.

The Master Key: *Declare an abstract base when you can already name the method every member of the family must supply, and not before, because an interface is cheap to write and expensive to change.*

Inheritance, super and polymorphism

This section defines inheritance and polymorphism, the last two of the four pillars, and supplies the three things this topic is examined on and almost never shown: super(), multiple inheritance and the method resolution order. It draws the constructor chain, the diamond with the single order Python searches it in, and one loop reaching three classes that share no ancestor. Its argument is duck typing, and it is where a reader who has written C++ and Java finds out why Python needs no interface keyword, no virtual and no declared base class for a call to reach the right code.

BINDER Question 55

A tailoring order book in Kimamba keeps one page of standing rules at the front and a page for each garment behind it. The apprentice copies the front page onto every garment page, so a change to the rules means hunting the whole book.

Explain inheritance and overriding in Python, and explain super() and constructor chaining.

Model Answer

Inheritance makes one class a specialised version of another, so the subclass has everything the base defines without repeating a line of it; overriding is redefining an inherited method; and super() is how a subclass constructor calls the one above it, without which the inherited attributes never exist.

What a subclass begins with: Writing class Student(Person) names Person as the base, and the new class begins with every method Person defines. Nothing is copied across; the subclass has a second place to look. Neither Student nor Prefect gives itself a printable form, and both print through the one Person defines.

```
 1  class Person:
 2      def __init__(self, name):
 3          self.name = name
 4
 5      def report(self):
 6          return "attends assembly"
 7
 8      def __str__(self):
 9          return f"{self.name} {self.report()}"
10
11
12  class Student(Person):
13      def __init__(self, name, level):
14          super().__init__(name)
15          self.level = level
16
17      def report(self):
18          return f"attends assembly and studies {self.level}"
19
20
21  class Prefect(Student):
22      def __init__(self, name, level, duty):
23          super().__init__(name, level)
24          self.duty = duty
25
26      def report(self):
27          return f"studies {self.level} and must {self.duty}"
28
29
30  head_boy = Prefect("Juma", "Form Five", "lock the Ulaya block")
31  print(head_boy.name, "|", head_boy.level, "|", head_boy.duty)
32  print(head_boy)
33  print(Student("Neema", "Form Six"))
```

Output

```
Juma | Form Five | lock the Ulaya block
Juma studies Form Five and must lock the Ulaya block
Neema attends assembly and studies Form Six
```

Program 2.52: *Three classes, each constructor calling the one above it before doing its own work.*

Redefining what you inherited: A subclass that defines a method the base already has replaces it for its own objects, and that is overriding. Both Student and Prefect redefine report, so one name produces three different sentences, and the version on Person is untouched.

Calling up before your own work: A subclass constructor must call super().__init__(...), or the base never runs and the attributes it would have created do not exist. The call goes up first and the lines assigning the subclass's own attributes run after it, so the base's attribute appears first and the most specialised one last.

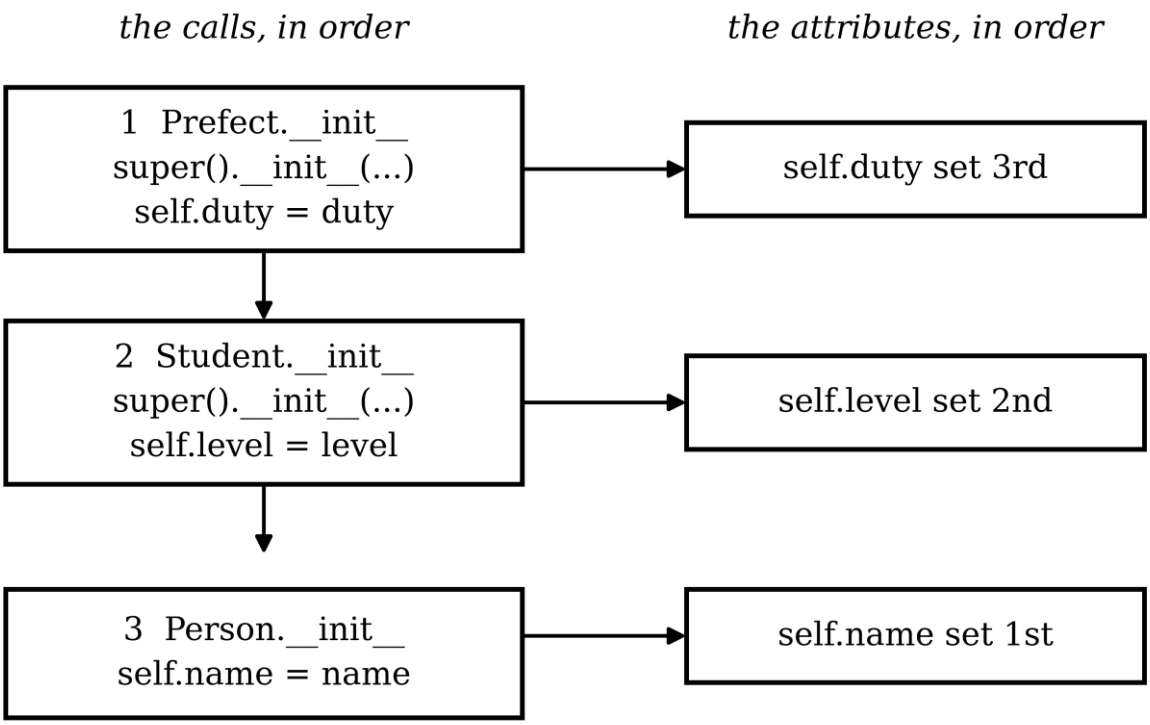


Figure 2.17: *Each constructor calling up before it sets its own, and the order the attributes are really set in.*

Reading the two columns: Figure 2.17 puts the calls down the left and the attributes down the right. The calls run from Prefect to Student to Person, each handing over before doing its own work, while the attributes appear in the opposite order, name first and duty last.

Why super beats naming the base: You could name Person directly and call its constructor yourself, and it works until the hierarchy changes or a second base arrives. super() asks Python for the next class in the resolution order instead, so the line survives a rearrangement above it.

A subclass inherits everything it does not redefine, and inherits nothing whatever from a constructor it never calls.

The Master Key: *A subclass takes every method it does not override, and takes the base's attributes only when its own constructor calls super() first.*

BINDER Question 56

Water reaches the Msowero dispensary down two pipes that both start at one tank. The caretaker keeps a written order for tracing a fault, and it takes him past the tank once, never twice.

Explain multiple inheritance and the method resolution order, and state the two properties of that order a reader can rely on.

Model Answer

Multiple inheritance lets one class name more than one base, and the method resolution order is the single sequence Python works out from those bases and then searches, one class at a time, until it finds the name that was asked for.

Naming more than one base: Writing class A(B, C) gives A whatever B offers and whatever C offers. The diamond appears the moment B and C have a base in common, because D is then reachable from A by two different routes, and a search that walked both routes would arrive at D twice.

Why one order is needed: A call on an A object has to reach exactly one method, so Python cannot leave two routes open. It works the sequence out once, when the class is created, and stores it; every later lookup walks that one list. Where no consistent sequence exists, Python refuses to create the class at all.

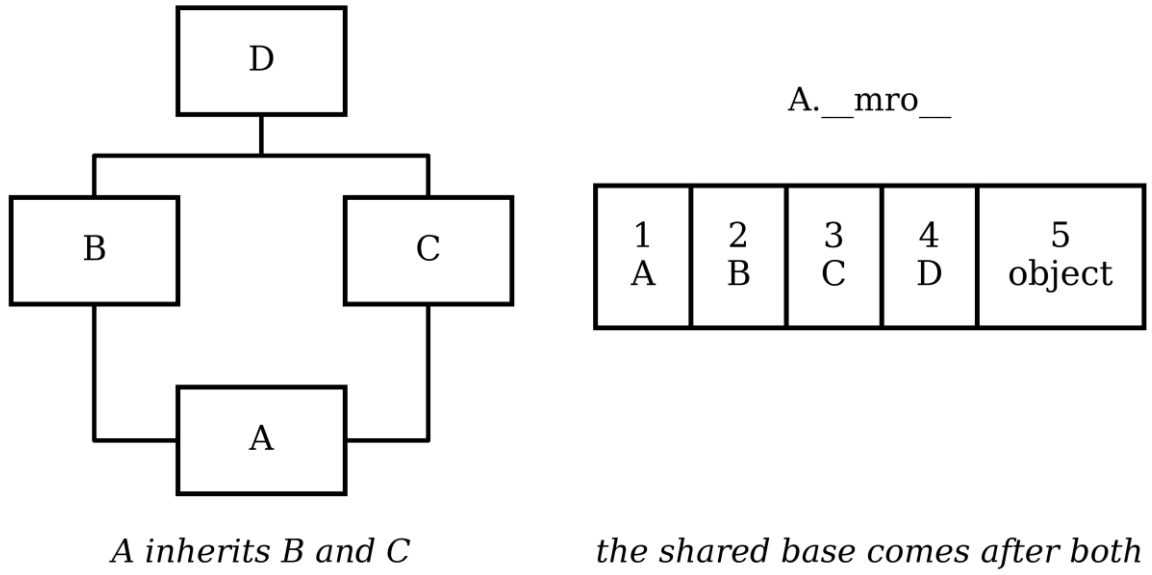


Figure 2.18: *Four classes in a diamond, and the single order Python searches them in.*

The two properties you can rely on: A class always comes before its own bases, and a base shared by several classes comes after every one of them. Those two together fix the order in Figure 2.18 as A, then B, then C, then D, then object. Guessing depth first would give A, B, D and reach the shared base before C, which is the answer to avoid.

```
 1  class D:
 2      pass
 3
 4
 5  class B(D):
 6      pass
 7
 8
 9  class C(D):
10      pass
11
12
13  class A(B, C):
14      pass
15
16
17  for cls in A.__mro__:
18      print(cls)
```

Output

```
<class '__main__.A'>
<class '__main__.B'>
<class '__main__.C'>
<class '__main__.D'>
<class 'object'>
```

Program 2.53: *A diamond of four classes, printing the order Python reports for itself.*

Reading the printed order: Every class carries its own resolution order, and A.__mro__ hands it over as a sequence of five classes. Each printed line is Python's own display of a class, giving the module it was created in and then its name, except the built-in object, which carries none; the five lines read A, B, C, D and object.

One order, worked out once when the class is created, is what makes a diamond safe to walk.

The Master Key: *Python flattens every base into a single order in which a class always precedes its own bases and a shared base comes after everything that inherits it.*

BINDER Question 57

A dog barks at the gate, the assembly bell rings, and the staffroom radio starts the news. Three things at the Ulaya school that share no part between them, and one word covers what all three of them just did.

Explain polymorphism in Python and explain duck typing, and say what happens to a second definition of one method name.

Model Answer

Polymorphism is one name reaching different code depending on the object it is called on, and duck typing is why Python needs no declaration to arrange it: the method is looked up on the object in front of you at the moment of the call, so anything carrying a method of that name will serve.

The two shapes the idea takes: The first is overriding inside one hierarchy, where a subclass redefines a method and its own objects reach the new version. The second is unrelated classes that happen to offer the same method name, and it is the shape a language with declared types cannot manage without extra machinery.

```
 1  class Dog:
 2      def speak(self):
 3          return "Woof"
 4
 5      def __str__(self):
 6          return "the dog at the gate"
 7
 8
 9  class Bell:
10      def speak(self):
11          return "Ding ding ding"
12
13      def __str__(self):
14          return "the assembly bell"
15
16
17  class Radio:
18      def speak(self):
19          return "Habari za asubuhi"
20
21      def __str__(self):
22          return "the staffroom radio"
23
24
25  things = [Dog(), Bell(), Radio()]
26  for thing in things:
27      print(thing, "says", thing.speak())
```

Output

```
the dog at the gate says Woof
the assembly bell says Ding ding ding
the staffroom radio says Habari za asubuhi
```

Program 2.54: *One loop, three classes with nothing in common, and three different methods reached.*

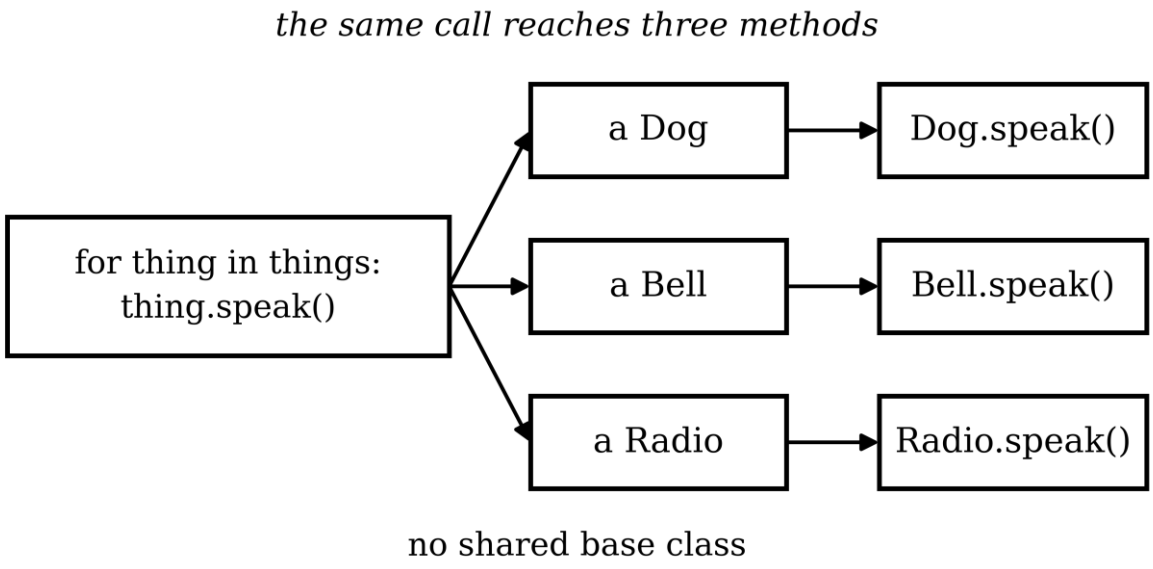


Figure 2.19: *One call site, three unrelated objects, and the method found on each of them.*

Nothing has to be declared: C++ reaches the right code through a common base and a method marked virtual, and Java through a superclass or an interface. Dog, Bell and Radio promise nothing and share nothing above them, yet the loop reaches all three, because Python asks the object itself for speak when the line runs.

Python has no method overloading: A second def of a name inside one class replaces the first outright. Nothing is chosen by how many arguments arrive or what type they are, because a def binds a function to a name and the second binding wins. Where several forms of one call are wanted, a default argument or *args supplies them.

```
 1  class Till:
 2      def charge(self, sheets):
 3          return sheets * 50
 4
 5      def charge(self, sheets, rate):
 6          return sheets * rate
 7
 8
 9  till = Till()
10  print(till.charge(20, 150))
```

Output

```
3000
```

Program 2.55: *Two definitions of one method name, and the only one of them that survives.*

What duck typing costs: Nothing checks that a method exists until the call runs. Put one object without speak into that list and the loop prints the earlier lines and then stops there, whereas a compiler would have refused the whole program. The freedom and the exposure are one property.

The method belongs to the object and not to a type written down beside it, which is the whole of why no interface keyword was needed.

The Master Key: *A method is looked up on the object at the moment of the call, so a shared method name is all that unrelated classes need, and a second definition of one name replaces the first.*

REAL Question 58

The savings group at Berega opened a treasurer's record this morning. The program printed its heading and the cash in hand, then stopped on a line that only asks for the treasurer's name.

Find the fault in this savings listing, say exactly what Python does about it and when, and correct it.

Name the fault and give the line it sits on.

Say exactly when Python discovers it, and what had already been printed by then.

Correct the listing, and say why the object was allowed to be built at all.

Model Answer

The constructor of Treasurer on line 7 never calls super().__init__(...), so the assignment on line 3 never runs and no attribute called name is ever created; the program builds the object, prints two correct lines, and then raises AttributeError inside statement at line 11.

```
 1  class Member:
 2      def __init__(self, name):
 3          self.name = name
 4
 5
 6  class Treasurer(Member):
 7      def __init__(self, name, cash_box):
 8          self.cash_box = cash_box
 9
10      def statement(self):
11          return f"{self.name} holds {self.cash_box} shillings"
12
13
14  asha = Treasurer("Asha", 20000)
15  print("Treasurer opened for the Berega savings group")
16  print(asha.cash_box)
17  print(asha.statement())
```

Python says

```
Treasurer opened for the Berega savings group
20000
Traceback (most recent call last):
  File "/home/form5/savings.py", line 17, in <module>
    print(asha.statement())
          ^^^^^^^^^^^^^^^^
  File "/home/form5/savings.py", line 11, in statement
    return f"{self.name} holds {self.cash_box} shillings"
              ^^^^^^^^^
AttributeError: 'Treasurer' object has no attribute 'name'
```

Program 2.56: *A subclass constructor that does not call up, and the line where the missing attribute is finally asked for.*

The fault: Line 7 gives Treasurer a constructor of its own, and line 8 assigns cash_box and nothing else. Nothing in it reaches the constructor above, so the assignment on line 3 never happens. The parameter name is accepted at line 7, used nowhere, and thrown away when the call ends.

What Python does about it, and when: It parses the whole file, since no rule of grammar is broken, and starts. Line 14 builds the object without complaint. Lines 15 and 16 print correctly, because cash_box really was assigned. Line 17 then calls statement, line 11 asks the object for name, the lookup fails, and AttributeError is raised at that instant.

Why the object was built at all: An object holds whatever its code has assigned and no more. There is no declared list of fields to be checked against, so an object missing an attribute is a perfectly ordinary object until something asks for the attribute. That is why the failure arrives late and away from the line that caused it.

Where the message points: The two frames are worth reading in the right order. The lower frame, line 11, is where the lookup actually failed, inside the method. The upper frame, line 17, is only the call that led there. The bottom line names the class of the object and the attribute it does not have.

The correction: Make the first line of the constructor read super().__init__(name), so that the base runs and assigns name before cash_box is dealt with. The habit worth taking away is that a subclass constructor calls up as its first statement, unless there is a stated reason for it not to.

The attribute printed successfully on line 16 came from an object that was already broken when it was made, and only line 17 was in a position to notice.

The Master Key: *A subclass constructor that never calls up leaves the base's attributes uncreated, the object is built anyway, and the failure waits for the first line that reads one of them.*

REAL Question 59

Ask four people at Ilongero market where the bus stops and you get four answers, all of them true and none of them first. A program has the same trouble, and it settles the order before anybody asks.

Work out the method resolution order of this hierarchy by hand, say which method a call reaches, and check both against what Python reports.

Model Answer

Working it by hand gives Ticket, Timed, Priced, Record, object, five classes in one order, and the call ticket.tag() therefore reaches the version on Priced rather than the one on Record, which is exactly what Program 2.57 prints.

Working the order out by hand: Start with the class itself, Ticket. Then take its bases in the order they were written, Timed and then Priced. Then take what those two have in common, Record, which must wait until both of them have been placed. Then object, which every class ends at. Five names, one order.

Which method the call reaches: Ticket defines no tag and Timed defines none, so the search moves on to Priced, which does, and stops there. The tempting answer is Record.tag, reached by climbing from Timed to its base; the order forbids that route, because Record has to come after every class that inherits from it.

```
 1  class Record:
 2      def tag(self):
 3          return "Record.tag"
 4
 5
 6  class Timed(Record):
 7      pass
 8
 9
10  class Priced(Record):
11      def tag(self):
12          return "Priced.tag"
13
14
15  class Ticket(Timed, Priced):
16      pass
17
18
19  ticket = Ticket()
20  print(ticket.tag())
21  for cls in Ticket.__mro__:
22      print(cls)
23  print(type(ticket) is Record, isinstance(ticket, Record))
24  print(issubclass(Ticket, Priced), issubclass(Priced, Ticket))
```

Output

```
Priced.tag
<class '__main__.Ticket'>
<class '__main__.Timed'>
<class '__main__.Priced'>
<class '__main__.Record'>
<class 'object'>
False True
True False
```

Program 2.57: *A hierarchy where the naive answer and the real one differ, printing both the order and the method reached.*

Checking the hand work against the run: The first printed line is Priced.tag, and the five that follow are the order, class by class, in the sequence worked out above. Three classes stand ahead of Record, which is why a method defined on Record is the last one a Ticket will ever find and the first one a hurried reader will name.

Asking about an object and asking about a class: The call isinstance(ticket, Record) asks whether this object was made from Record or from anything below it, and issubclass(Ticket, Priced) asks the same question about two classes without needing an object. Both answer True downward only: Priced is not a subclass of Ticket, and the last line prints False for it.

Why an exact type check is the wrong test: Writing type(x) == Record, or type(x) is Record, asks whether the object was made from that class and nothing else, so it answers False for a Ticket, which is what the second to last line prints. Code guarded that way silently refuses every subclass written after it, and isinstance is the test that does not.

The order is worth working by hand once, because the answer a hierarchy makes obvious is the one Python was designed not to give.

The Master Key: *Take the class, then its bases from left to right, then anything they share, and a shared base comes last of all; ask isinstance rather than for an exact type, because only isinstance accepts a subclass.*

BUILD Question 60

The weighbridge book at Puma has three sums in it before ten o'clock: grain by plain weight, maize at the mill rate, and sunflower less the pressing fee. One clerk works all three.

Write a complete program that prints a weighbridge line and a day's total for a mixed list of produce deliveries.

A base class holds the farmer and the weight in kilograms, and values a load at the plain store rate.

Two subclasses override the valuing method, one for maize at the mill rate and one for sunflower at the press rate less a pressing fee.

The sunflower class takes one extra piece of information and chains to the base for the rest.

One loop prints a line for every delivery in a mixed list and adds up a total.

Model Answer

The program is one base class and two subclasses that override value, so the loop at the foot of it names no class at all and still reaches three different calculations.

Where the specification is met: Delivery holds the two facts every load has and values it at 500 a kilogram. Maize overrides value and nothing else, since the rest suits it already. Sunflower needs a third fact, so it takes a constructor of its own, chains up for the farmer and the weight, and then values the load at 900 a kilogram less the fee.

What the loop does not have to know: The loop prints load and adds load.value() without asking what any of them is. Each object carries its own version, so the three printed lines come from three different bodies of code reached through one name, and the total is the sum of whichever versions ran.

```
 1  class Delivery:
 2      def __init__(self, farmer, kilos):
 3          self.farmer = farmer
 4          self.kilos = kilos
 5
 6      def value(self):
 7          return self.kilos * 500
 8
 9      def __str__(self):
10          return f"{self.farmer}: {self.kilos} kg worth {self.value()}"
11
12
13  class Maize(Delivery):
14      def value(self):
15          return self.kilos * 620
16
17
18  class Sunflower(Delivery):
19      def __init__(self, farmer, kilos, press_fee):
20          super().__init__(farmer, kilos)
21          self.press_fee = press_fee
22
23      def value(self):
24          return self.kilos * 900 - self.press_fee
25
26
27  loads = [Delivery("Msowero store", 40), Maize("Mama Zawadi", 90),
28      Sunflower("Puma group", 75, 5000)]
29  total = 0
30  for load in loads:
31      print(load)
32      total += load.value()
33  print("Total:", total)
```

Output

```
Msowero store: 40 kg worth 20000
Mama Zawadi: 90 kg worth 55800
Puma group: 75 kg worth 62500
Total: 138300
```

Program 2.58: *A base class, two overriding subclasses, and one loop that names none of them.*

The decision worth naming: The rate lives in the method that overrides value, not in a test inside the loop. A fourth kind of load is one new class and no change anywhere else, whereas a chain of tests on a kind field would have to be found and extended every time the store handled something new.

Three classes, one method name, and a loop with no knowledge of any of them satisfy the whole specification.

The Master Key: *Put the difference between two kinds of object in a method they both define and neither the loop nor the total ever has to ask which kind it is holding.*

BUILD Question 61

One reading at Dumila, three readings at Kimamba, and the tally clerk wants both totals added together at the close of the day. Her book has a column for weights and no rule anywhere for adding two books.

Write a complete program with a tally class that takes any number of readings in one call and joins two tallies with the plus sign.

A class holds a place and a weight, and may be created with a starting weight or without one.

One method accepts no readings, one reading, or several, in a single call.

The plus sign joins two tallies into a third, and the less-than sign compares two by weight.

A printable form shows the place and the weight.

Model Answer

The class supplies both halves of the specification: weigh accepts any number of readings because *args gathers them into a tuple, and the dunder methods __add__ and __lt__ give the plus sign and the less-than sign a meaning for two Load objects.

```
class Load:
    def __init__(self, place, kilos=0):
        self.place = place
        self.kilos = kilos

    def weigh(self, *readings):
        for reading in readings:
            self.kilos += reading

    def __add__(self, other):
        return Load(f"{self.place} and {other.place}",
                    self.kilos + other.kilos)

    def __lt__(self, other):
        return self.kilos < other.kilos

    def __str__(self):
        return f"{self.place}: {self.kilos} kg"


first = Load("Dumila")
first.weigh(120)
second = Load("Kimamba", 60)
second.weigh(40, 35, 25)
print(first)
print(second)
print(first + second)
print(first < second)
```

Output

```
Dumila: 120 kg
Kimamba: 160 kg
Dumila and Kimamba: 280 kg
True
```

Program 2.59: *One method taking several shapes of call, and two operators given a meaning of their own.*

How one definition takes several calls: The constructor carries a default, so a Load may be created with a starting weight or without one. The method weigh is written once and called with one reading and then with three, because the star gathers whatever arrives into a tuple the body walks. That is the simulation of overloading.

What the dunder methods buy: Writing first + second makes Python look on the object for __add__, and first < second makes it look for __lt__, exactly as print looks for __str__. Giving an operator a meaning for a class of your own is operator overloading, and it survives because it is resolved on the object, not on a declared type.

The decision worth naming: __add__ returns a third Load rather than changing either of the two it was given, which is what a plus sign leads a reader to expect. An operator that quietly altered its left operand would satisfy the specification and surprise everybody.

One definition with a star in it does the work several signatures would do in C++ or Java, and three dunder methods make the class behave like something built in.

The Master Key: *Simulate overloading with a default argument or a starred parameter, and give an operator a meaning by defining the dunder method Python looks up on the object.*

HOT Question 62

Two men give the lorry driver at the sisal estate his orders. This morning the owner wants the vehicle at the weighbridge and the loader wants it in the field, and both of them are his boss.

Argue both sides of the claim that multiple inheritance should be avoided, and reach a judgement.

Model Answer

The claim is right about inherited state and wrong about inherited behaviour, and that distinction is the whole verdict: two bases that each hold data of their own are trouble, while two bases that contribute methods and hold nothing are among the tidiest arrangements Python offers.

The case for the rule: A diamond gives one shared base two routes into a class, and a reader looking at a call cannot see which of them the method came from. The answer depends on an order that was worked out at class creation and is written nowhere in the source, so the code no longer says what it does.

Two bases holding data is the real danger: If both bases assign an attribute of the same name, one of them wins and the loss is silent: no message, no warning, and an object whose state is half of what each base believed it had. Constructors compound it, since a chain that does not run cleanly through every base leaves some of them uninitialised.

The order is computed, not obvious: Adding a base, or writing two bases the other way round, recomputes the order for the whole hierarchy. A call that reached one method yesterday can reach a different one today with no change to the line that made it, and a change of that kind is very hard to notice in review.

The case against the rule: A class that adds behaviour and holds no data of its own, a mixin, carries none of that risk. Several small classes that each contribute one method, combined by naming them as bases, is clearer than copying the method into every class that needs it or wrapping each class by hand.

What the other two languages decided: Java forbids inheriting from two classes and permits any number of interfaces, which is precisely the split between state and behaviour drawn in a keyword. C++ allows the full form and pays for it with virtual inheritance. Python allows it and settles it with one computed order, which is more permissive than either.

The verdict, and the condition that decides it: Avoid multiple inheritance where more than one base assigns attributes of its own, and use it freely where the extra bases contribute only methods. The test is one question asked of each base: does it hold data? If two of them do, hold an object of one class inside the other instead, and call through it.

The rule survives as advice about state and fails as advice about behaviour, which is why a blanket ban costs more than it saves.

The Master Key: *Inherit behaviour from as many classes as you like, inherit state from one, and where two bases both hold data, hold an object of one of them instead.*

HOT Question 63

A spare-part counter keeps one order slip that takes a part number, and a quantity only where somebody wants more than one. The shop opposite prints three different slips, one for each way a customer might ask.

Argue both sides of the claim that the absence of method overloading is a weakness in Python, and reach a judgement.

Model Answer

It is not a weakness, and the condition is what the several forms would have differed in: where they differ only in how many values arrive or what type they are, dynamic typing has already removed the reason for them, and where they differ in what they do they deserved separate names in any language.

What overloading gives a language: Several definitions share a name and differ in their parameter lists, and the compiler chooses between them from the declared types at the call. Every legal shape of call is written down, and one matching no definition is refused before the program runs.

The case that losing it is a weakness: A definition ending in a starred parameter tells the reader nothing about how many values are expected or what they should be. The list of legal calls that C++ and Java write into the

source has to live in a comment or in somebody's head, and a call nobody thought of is discovered by running it.

The cost falls at the wrong moment: A compiled language settles which definition a call matches before anything executes, and refuses the file if none does. Python accepts any call that parses, so a rarely used branch can carry a wrong one for months.

The case that nothing was lost: Overloading exists mostly to choose between declared types, and Python declares none. A pair of C++ functions taking an int and a double collapse into one Python function that works for both, so most of what overloading was for disappears rather than being missed.

What Python supplies instead: A default argument covers the shorter call, a starred parameter covers a call of any length, and operator overloading survives untouched, because a dunder method is found on the object and needs no declared type to be chosen by.

Where the honest cost lands: It lands on reading rather than on writing. One definition that quietly handles several shapes of call is easy to write and hard to describe, and the reader has only the name and whatever was written down about it.

The verdict, and the condition that decides it: Not a weakness, on the condition that the forms differ only in how many values arrive. Where two forms do genuinely different work, giving them one name was always a poor idea. Judge a language by what it makes unnecessary as well as by what it lacks.

A feature whose main purpose is to choose by declared type cannot be missed by a language that has no declared types, though the reader still pays for its absence.

The Master Key: *Dynamic typing removes the reason for method overloading and keeps one cost of losing it, which is that the written list of legal calls has to move into the name and the documentation.*

Errors, exceptions and a working application

This section owns everything a program does when it meets something it cannot do. It defines the exception hierarchy and the four blocks of a try statement, teaches the reading of a traceback from its last line upward, exposes a handler that reports success on wrong data and replaces it with a custom exception class, and closes the chapter with a complete application that reads a file, reads what is typed at it, and handles both by name. Everything the earlier sections built is used here unchanged and none of it is taught again.

BINDER Question 64

Held down together, two keys should stop a runaway program at the Nzuguni internet café. The screen keeps scrolling. Whoever wrote it made sure the program would survive anything, and it is surviving the request to stop.

Explain how Python arranges its exception classes, and say why an except clause that names no class is a fault.

Model Answer

Every exception is an object of a class, and the classes form a tree: BaseException is the root, Exception sits beneath it and gathers the faults a program can answer, and KeyboardInterrupt and SystemExit sit beside Exception rather than beneath it, which is why a bare except is a fault.

What is raised is an object: When an operation cannot be carried out, Python builds an instance of a class and raises it. The class is the name of the fault: ValueError for a string that is no number. Naming a class after except says which fault this handler answers.

Catching a class catches everything under it: A handler naming Exception answers every class below it in the tree, which is nearly everything a program raises. A handler naming ValueError answers that class alone. The nearer the named class sits to the fault, the smaller the promise the handler has made.

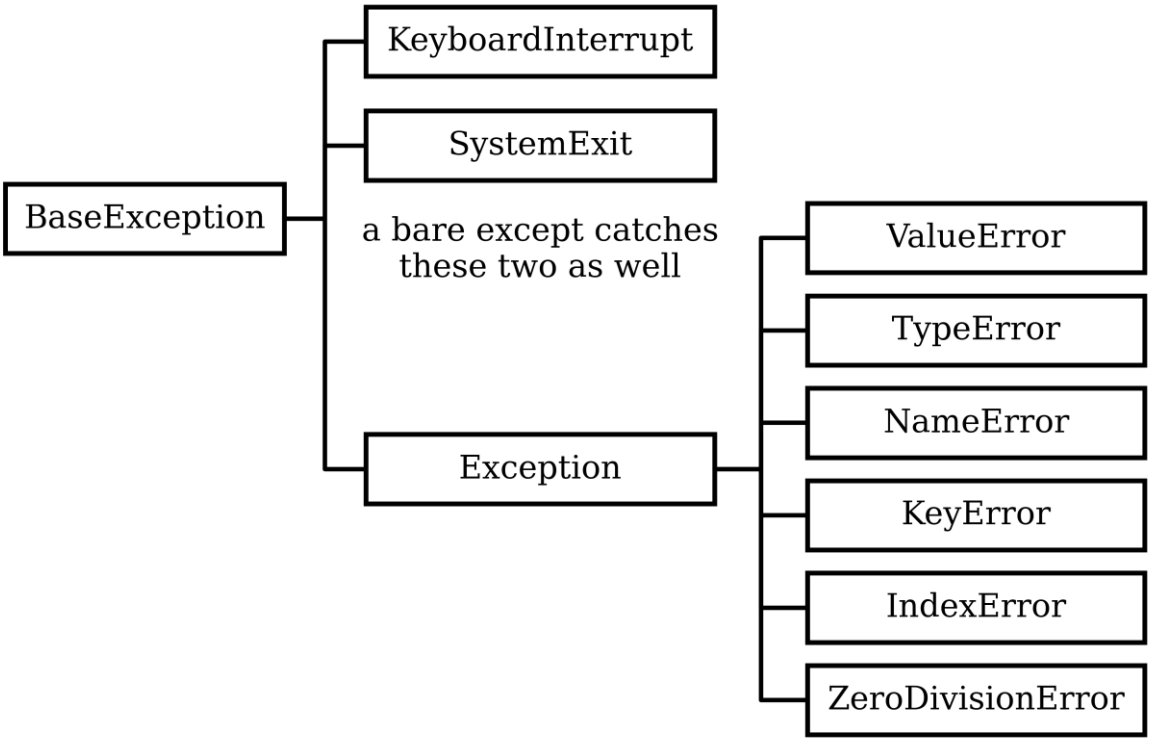


Figure 2.20: *The part of the exception tree this chapter uses, and the two classes that sit outside Exception.*

The two that sit outside Exception: KeyboardInterrupt is raised when the person at the machine holds down Ctrl and C, and SystemExit is raised by the program's own request to stop. Both inherit from BaseException directly, as Figure 2.20 draws them, so a handler naming Exception leaves both alone.

Why a bare except is a fault: An except with no class after it is called a bare except, and it catches everything under BaseException, those two included. The reader who holds down Ctrl and C finds the keypress swallowed and the program still running, and a program that asked to stop is prevented from stopping by its own handler.

```
 1  class MeterRun:
 2      def __init__(self, place, readings):
 3          self.place = place
 4          self.readings = readings
 5
 6      def average(self):
 7          try:
 8              return sum(self.readings) / len(self.readings)
 9          except ZeroDivisionError:
10              return 0.0
11
12      def __str__(self):
13          return f"{self.place}: {self.average():.1f} units a day"
14
15
16  print(MeterRun("Makiungu", [180, 200, 190]))
17  print(MeterRun("Ndago", []))
```

Output

```
Makiungu: 190.0 units a day
Ndago: 0.0 units a day
```

Program 2.60: *A meter average that answers one named class and lets everything else past.*

Reading the two meters: The first has three readings and averages 190.0. The second has none, so len gives 0, the division raises ZeroDivisionError, and the handler names that class and answers with 0.0.

Name the class you mean, and the handler answers that fault and nothing else.

The Master Key: *KeyboardInterrupt and SystemExit inherit from BaseException rather than from Exception, so an except naming no class catches the reader's attempt to stop the program and the program's own attempt to exit.*

BINDER Question 65

Whatever the day has been, the shutter at the Kintinku spare-part counter comes down at six. Good sales, no sales, a leaking roof at four: the same key turns. One part of the evening never depends on how the day went.

Explain try, except, else and finally, and say which of the four blocks runs in each of the three cases.

Model Answer

The try block holds the line that might fail, an except block runs only when the class it names is raised, the else block runs only when nothing was raised at all, and the finally block runs in every case, including the one where nothing caught the exception and the program is on its way out.

As little as possible inside try: The try block should hold the one call that can fail and nothing else. Every line that ought to happen only when that call succeeded belongs in else, where no handler can reach it.

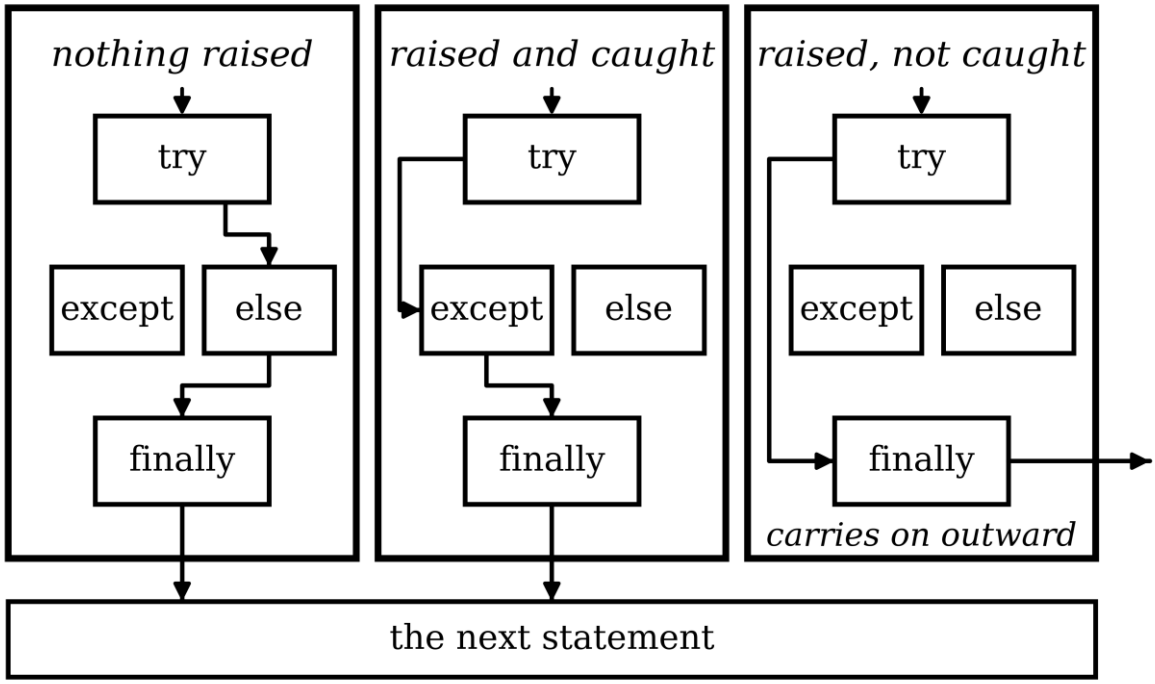


Figure 2.21: *Which blocks run when nothing is raised, when a raise is caught, and when it is not.*

What else protects you from: If the work that follows a successful conversion sits inside try, a ValueError raised by that later work is answered by a handler written for the conversion, and the message on the screen names the wrong thing entirely.

What finally is for: Work that must happen whether or not the line succeeded: closing a file, handing back a key, printing the line that says the drawer was shut. It runs after except, after else, and on the way out when nothing caught the exception at all.

Block	Nothing raised	Raised and caught	Raised, not caught
try	runs to the end	stops at the failing line	stops at the failing line
except	skipped	runs	skipped
else	runs	skipped	skipped
finally	runs	runs	runs, then the exception carries on outward

Table 2.7: *Which of the four blocks runs in each of the three cases.*

```
 1  class Till:
 2      def __init__(self, place):
 3          self.place = place
 4          self.taken = 0
 5
 6      def add(self, typed):
 7          try:
 8              amount = int(typed)
 9          except ValueError:
10              print("refused:", typed)
11          else:
12              self.taken += amount
13              print("accepted:", amount)
14          finally:
15              print("drawer closed at", self.place)
16
17      def __str__(self):
18          return f"{self.place} took {self.taken}"
19
20
21  till = Till("Kintinku")
22  till.add("500")
23  till.add("5o0")
24  print(till)
```

Output

```
accepted: 500
drawer closed at Kintinku
refused: 5o0
drawer closed at Kintinku
Kintinku took 500
```

Program 2.61: *All four blocks, run once on a good amount and once on a bad one.*

Reading the two calls: The first amount converts, so except is skipped, else adds it to the till and finally reports. The second was typed with a letter o where a zero was meant, so int refuses it, except reports it, else is skipped, and finally reports again. The till holds 500.

The four blocks answer four different questions, and a line placed in the wrong one is how a handler ends up catching something nobody meant it to catch.

The Master Key: *The else block runs only when the try block raised nothing and the finally block runs whatever happened, which is why try should hold only the line that can fail.*

BINDER Question 66

Twelve lines of text land on the screen at Sanza, and the reader starts at the top, as one starts anything. The line that names what actually went wrong is the last one.

Explain how to read a traceback, and say which line of it to read first.

Model Answer

Read a traceback from its last line upward: that line names the exception class and the message, the frames above it run from the outermost call at the top down to the line that actually failed at the bottom, and the file and line number in that lowest frame is where to look.

```
 1  class WeighNote:
 2      def __init__(self, place):
 3          self.place = place
 4          self.kilos = []
 5
 6      def as_kilos(self, typed):
 7          return int(typed)
 8
 9      def record(self, typed):
10          self.kilos.append(self.as_kilos(typed))
11
12      def load(self, rows):
13          for row in rows:
14              self.record(row)
15
16      def __str__(self):
17          return f"{self.place}: {sum(self.kilos)} kg"
18
19
20  note = WeighNote("Zuzu")
21  print("Weighing at Zuzu")
22  note.load(["412", "398", "4l2"])
23  print(note)
```

Python says

```
Weighing at Zuzu
Traceback (most recent call last):
  File "/home/form5/weighbridge.py", line 22, in <module>
    note.load(["412", "398", "4l2"])
  File "/home/form5/weighbridge.py", line 14, in load
    self.record(row)
  File "/home/form5/weighbridge.py", line 10, in record
    self.kilos.append(self.as_kilos(typed))
                      ^^^^^^^^^^^^^^^^^^^^
  File "/home/form5/weighbridge.py", line 7, in as_kilos
    return int(typed)
           ^^^^^^^^^^
ValueError: invalid literal for int() with base 10: '4l2'
```

Program 2.62: *A weighbridge run that stops on its third row, and the traceback it printed.*

The last line is the answer: It gives the class, a colon, and the message. Here it names ValueError and then quotes the value handed to it, 4l2, which is a weight typed with a letter l where the digit 1 was meant.

The frames run outermost to innermost: Each frame gives a file, a line number, the name of what was running, and the source line itself. The top frame is where the program started, each frame below it was called by the one above, and the lowest frame is the one that failed.

Where to put your finger: In the lowest frame, which here is line 7 of weighbridge.py, inside as_kilos. That is where the exception was raised. The three frames above it record only the route by which the program arrived.

What the carets add: Since version 3.11 the interpreter marks the part of the line that failed. In the third frame the carets sit under self.as_kilos(typed) rather than under the whole line, so a line carrying two calls no longer leaves you guessing.

Why the frames above still matter: The message names the value and the lowest frame names the line, but neither says where the value came from. The frames above answer that: load was walking a list of rows and record passed the third of them down.

A traceback is the most useful thing Python ever prints, and it is printed only because nobody caught the exception.

The Master Key: *The last line of a traceback names the class and the message and the lowest frame names the file and line where the exception was raised, so a traceback is read from the bottom upward.*

REAL Question 67

The stock report from the Ihumwa pharmacy printed the word complete, and the shelf holds eighty-two boxes fewer than the report says. Nothing was flagged. The program was written to keep going whatever happened, and it did.

Say why this stock listing reports success on wrong data, name what hid the fault, and rewrite it so the fault is reported.

Say what was actually wrong with the data, and what the program printed instead.

Say why nothing was reported, naming the block that hid it.

Rewrite the loading method so the fault is either reported honestly or allowed to stop the program.

Model Answer

Nothing was reported because the handler catches Exception, does nothing with what it caught, and lets the program carry on printing: the third row is not a number, the loop is abandoned where it stood, and the program then announces five rows read and 42 boxes where the five rows really hold 124.

```
 1  class StockCount:
 2      def __init__(self, place):
 3          self.place = place
 4          self.rows = 0
 5          self.boxes = 0
 6
 7      def load(self, rows):
 8          try:
 9              for row in rows:
10                  self.boxes += int(row)
11          except Exception:
12              pass
13          self.rows = len(rows)
14
15      def __str__(self):
16          return f"{self.place}: {self.rows} rows, {self.boxes} boxes"
17
18
19  count = StockCount("Ihumwa")
20  count.load(["24", "18", "thirty", "40", "12"])
21  print(count)
22  print("Stock check complete")
```

Output

```
Ihumwa: 5 rows, 42 boxes
Stock check complete
```

Program 2.63: *A stock count that runs cleanly, reports success, and is 82 boxes wrong.*

What was actually wrong: The third row is the word thirty, written out where every other row carries digits. The call to int cannot make a number of it and raises ValueError at that row, which is a typing fault in the stock book and not a fault in the program.

Why nothing was reported: The except block caught it and its body is pass, so the exception object was discarded and the lines below the try ran as though the loop had finished normally. An exception that is caught and not answered leaves no trace anywhere: no message, no traceback, no exit status.

The try is wrapped round the wrong thing: The whole loop sits inside try, so the raise from the third row abandons the fourth and fifth rows as well. A try round the single conversion would have lost one row. As it stands, two rows were added and the count of rows read was taken from the length of the list.

Why this is worse than stopping: A stopped program prints a traceback naming the line and the value, and the clerk knows to look. This one prints 42 against a true 124, short by 82, and adds the word complete. Nobody who reads that screen has any reason to open the stock book.

The repair, and the class that names the fault: The rewrite gives the fault a name of its own. RowError inherits from Exception, the add method raises it when a row is not all digits, and load catches that one class around a single row and prints which row it was. The raise statement is what starts an exception on its way.

```
class RowError(Exception):
    """A stock row that is not a whole number of boxes."""


class StockCount:
    def __init__(self, place):
        self.place = place
        self.rows = 0
        self.boxes = 0

    def add(self, row):
        if not row.isdigit():
            raise RowError(f"{row} is not a whole number")
        self.boxes += int(row)
        self.rows += 1

    def load(self, rows):
        for row in rows:
            try:
                self.add(row)
            except RowError as trouble:
                print("rejected:", trouble)

    def __str__(self):
        return f"{self.place}: {self.rows} rows, {self.boxes} boxes"


count = StockCount("Ihumwa")
count.load(["24", "18", "thirty", "40", "12"])
print(count)
```

Output

```
rejected: thirty is not a whole number
Ihumwa: 4 rows, 94 boxes
```

Program 2.64: *The same data, a custom exception raised by name, and a count the reader can trust.*

What the repaired run says: It names the row it refused and counts 94 boxes from the four rows it could read. That total is still not 124, and it should not be: a word in a column of figures is not stock, it is a typing fault, and the program now says so.

The honest alternative was always available: report the row and carry on, or let the exception through and stop.

The Master Key: *A handler that catches a class and answers it with nothing converts a fault that would have stopped the program into a wrong answer that nobody is told about.*

BUILD Question 68

Two, typed as a word, is what the clerk at the Ndago booking desk entered for the number of tickets. The program stopped. Nine passengers waited while it was started again and the whole booking retyped.

Write a program whose class reads a number of tickets typed at a booking desk and asks again until a whole number above zero arrives.

A class holds the place, the fare, and the tickets sold so far.

One method reads a number typed at the program and catches ValueError, asking again after each bad attempt.

It also refuses anything below one, and returns only when a good number has arrived.

The program prints what is owed and how many tickets were sold.

Model Answer

The reading method loops until it has something usable: it converts inside a try, catches ValueError and asks again, checks the converted number against the smallest sale allowed, and returns only when both conditions hold.

```
 1  class TicketDesk:
 2      def __init__(self, place, fare):
 3          self.place = place
 4          self.fare = fare
 5          self.sold = 0
 6
 7      def ask(self):
 8          while True:
 9              typed = input("Tickets: ")
10              try:
11                  number = int(typed)
12              except ValueError:
13                  print("Whole numbers only, please.")
14                  continue
15              if number < 1:
16                  print("At least one ticket, please.")
17                  continue
18              return number
19
20      def sell(self):
21          number = self.ask()
22          self.sold += number
23          return number * self.fare
24
25      def __str__(self):
26          return f"{self.place}: {self.sold} tickets sold"
27
28
29  if __name__ == "__main__":
30      desk = TicketDesk("Ndago", 2500)
31      owed = desk.sell()
32      print("To pay:", owed)
33      print(desk)
```

Typed at the program

```
two
0
3
```

Output

```
Tickets: Whole numbers only, please.
Tickets: At least one ticket, please.
Tickets: To pay: 7500
Ndago: 3 tickets sold
```

Program 2.65: *A booking desk that refuses a word and a zero without stopping.*

Two different refusals, one loop: The word two cannot be converted, so ValueError is raised and caught, and the loop goes round. The zero converts perfectly and is refused by a plain comparison instead. A value that is unusable and a value that is impossible are different faults, and only the first of them is an exception.

Why the check sits outside the try: Only the conversion can raise, so only the conversion is inside the try block. Putting the comparison in there as well would let a fault in the comparison be reported as a bad number typed by the clerk.

The decision worth naming: The method returns a good number rather than storing one, so the caller decides what the number is for. Three attempts were needed here and the program never stopped: two refusals and one sale of three tickets at 2500 each.

The specification is met by one loop, one named class of exception, and one comparison that has nothing to do with exceptions at all.

The Master Key: *A validation loop catches the class the conversion can raise, refuses the values that convert but make no sense, and returns only when both tests are passed.*

BUILD Question 69

The tally file for Monday is not on the machine at the Makiungu landing site; somebody saved it to a phone. A program that opens it by name has to decide, this morning, what to do about a file that is not there.

Write a program whose class loads a landing tally from a text file beside it and says so plainly when the file is not there.

A file named landings.txt sits beside the program, one landing to a line, the place and the weight separated by a comma.

A class is created with the name of a file and holds the rows it managed to read.

Its loading method opens the file with with, handles FileNotFoundError by name, and reports whether it succeeded.

The program loads two files, one of which does not exist, and prints what it read from each.

Model Answer

The class is created with a file name and reads it inside a with statement, which closes the file whether the reading ended well or badly, and the whole opening sits inside a try that names FileNotFoundError, so a missing file becomes one printed line instead of a traceback.

landings.txt

```
Makiungu,42
Ndago,17
Kintinku,26
```

tally.py

```
class TallySheet:
    def __init__(self, path):
        self.path = path
        self.rows = []

    def load(self):
        try:
            with open(self.path, encoding="utf8") as sheet:
                for line in sheet:
                    place, kilos = line.strip().split(",")
                    self.rows.append((place, int(kilos)))
        except FileNotFoundError:
            print("No file named", self.path)
            return False
        return True

    def total(self):
        return sum(kilos for place, kilos in self.rows)

    def __str__(self):
        return f"{self.path}: {len(self.rows)} rows, {self.total()} kg"


if __name__ == "__main__":
    for name in ("landings.txt", "monday.txt"):
        sheet = TallySheet(name)
        if sheet.load():
            for place, kilos in sheet.rows:
                print(place, kilos)
            print(sheet)
```

Output

```
Makiungu 42
Ndago 17
Kintinku 26
landings.txt: 3 rows, 85 kg
No file named monday.txt
```

Program 2.66: *A tally sheet read from the file beside it, and the same class asked for a file that does not exist.*

What with is doing: It opens the file, hands it to the name after as, and closes it when the block ends, whether the block ended by falling off the bottom or by raising. A file closed by hand after the last line is a file left open by every fault that happens before it.

Why the handler names FileNotFoundError: A missing file is the one fault this method knows what to do about: say which name it looked for and report that it failed. A row with no comma in it raises something else, which this handler leaves alone, so a damaged file cannot be reported as a missing one.

What the run shows: The first file exists, so three rows are read and the sheet reports 85 kg. The second does not, so one line is printed and the program continues to its end without a traceback, which is the whole difference the handler makes.

The same class handles both files, and the only thing that changes between them is which of the two branches printed.

The Master Key: *A file read inside with is closed whatever happens, and a missing file handled by name turns a traceback into a printed line the reader can act on.*

BUILD Question 70

Yesterday's numbers are in the book, today's are on the dials, and the round of kiosks at Zuzu has to be walked before six. One dial was replaced overnight and now reads lower than the book says.

Assemble the whole kiosk application: a class, a book of yesterday's readings, today's figures typed at the program, and the takings for the day.

A file named meters.txt holds one kiosk and its previous reading to a line.

A class holds one meter and works out the units used from the reading typed today.

A reading below yesterday's is refused by raising ValueError, and a reading that is not a number raises it too.

The program handles FileNotFoundError when the book is opened and ValueError at each meter, and prints the day's takings.

The round starts only when the file itself is run.

Model Answer

The application is one class, one loading function and one guarded block: the class turns a typed reading into units and a charge, the function builds a meter from every line of the book, and the guard handles the two faults that can actually happen, a missing book and a reading it cannot use.

meters.txt

```
Makiungu,1180
Ndago,940
Kintinku,1325
Sanza,806
```

kiosk.py

```
PRICE_PER_UNIT = 35


class Meter:
    def __init__(self, place, previous):
        self.place = place
        self.previous = previous
        self.units = 0

    def take(self, typed):
        reading = int(typed)
        if reading < self.previous:
            raise ValueError("below yesterday's reading")
        self.units = reading - self.previous

    def charge(self):
        return self.units * PRICE_PER_UNIT

    def __str__(self):
        return f"{self.place}: {self.units} units, {self.charge()}/="


def loaded(path):
    meters = []
    with open(path, encoding="utf8") as book:
        for line in book:
            place, previous = line.strip().split(",")
            meters.append(Meter(place, int(previous)))
    return meters


if __name__ == "__main__":
    try:
        meters = loaded("meters.txt")
    except FileNotFoundError:
        print("No meter book at meters.txt")
    else:
        takings = 0
        for meter in meters:
            typed = input(f"{meter.place} reading: ")
            try:
                meter.take(typed)
            except ValueError as trouble:
                print("rejected:", trouble)
            else:
                print(meter)
                takings += meter.charge()
        print("Day's takings:", takings)
```

Typed at the program

```
1204
900
1361
nine
```

Output

```
Makiungu reading: Makiungu: 24 units, 840/=
Ndago reading: rejected: below yesterday's reading
Kintinku reading: Kintinku: 36 units, 1260/=
```

```
Sanza reading: rejected: invalid literal for int() with base 10: 'nine'
Day's takings: 2100
```

Program 2.67: *The whole application, reading its book, taking four readings and refusing two of them.*

One handler, two sources: The reading at Sanza was typed as a word and int raised ValueError; the reading at Ndago was a number below yesterday's and the class raised ValueError itself. Both reach the same handler and print the same way, which is what naming a class rather than a message buys you.

Where each handler sits: The missing book is not a fault of any meter, so it is handled where the book is opened, and the round never begins. A bad reading concerns one meter only, so it is handled inside the loop, and the other three meters are read normally.

The decision worth naming: The class raises and never prints, and everything the operator sees is printed by the guarded block at the edge of the program. That is why the same Meter could be used tomorrow by something that writes to a file instead of a screen.

Twenty-four units at Makiungu and thirty-six at Kintinku make 2100 shillings, and the two refused meters contributed nothing rather than something wrong.

The Master Key: *A complete application handles each fault where something can be done about it, and lets the class raise rather than print so the messages all come from one place.*

HOT Question 71

The boy at the Sanza motorcycle counter is told to sort out whatever goes wrong. He can find a missing bolt. When the roof leaked onto the stock he mopped it, told nobody, and the owner learned about it in March.

Argue both sides of the rule that a program should catch only what it can do something about, and reach a judgement.

Model Answer

The rule is right, and the condition that decides it is whether you can say in one sentence what the program should do instead: if you can, catch the class and do it, and if the only answer is to carry on and hope, let the exception through.

A handler is a decision, not a precaution: Writing except is a claim that you know what the program should do next. Ask again, skip this row and report it, fall back on a value you can name in the output: each of these is an answer. Where there is no answer, the handler adds nothing and removes the traceback.

What happens to an exception nobody catches: It propagates: the frame it was raised in is abandoned, then the frame that called it, and so on outward. If no handler answers on the way, the interpreter prints the traceback and stops the program, leaving a non-zero exit status that whatever started the program can notice.

Why stopping is often the right outcome: A program that stops has damaged nothing further, and the traceback is the most complete description of the fault that anybody will ever get: the class, the value, the file, the line, and the calls that led there. Catching it quietly destroys all of that.

The case for catching broadly: Some programs must not stop. A clerk typing at a desk should not meet a traceback because a finger slipped, an overnight run over nine hundred rows should not lose the good rows to one bad one, and a program holding a file open has work to do before it goes.

Where a broad catch belongs: At the edge, where the program meets a person or a whole batch, and never in the interior. The method that converts a reading knows what went wrong and not what to do about it; the loop that called it knows what to do and would never have heard, had the method swallowed it.

The verdict, and the condition that decides it: Catch a class only where you can name the action that follows. If the honest answer is that the program cannot go on sensibly, let the exception propagate, because stopping with a traceback is a better outcome than continuing on data you have already been told is wrong. An uncaught exception is not a failure of the programmer; a caught and unanswered one is.

The rule survives because a program is judged by what it does with the fault, and doing nothing is the one answer that is never worth writing down.

The Master Key: *Catch an exception where you can say what the program should do instead, and let it propagate everywhere else, because a program that stops with a traceback has told the truth about itself.*

HOT Question 72

The maize mill at Makiungu has not broken down once this season. Its scale has been weighing two kilos light since April, and every sack that has left the yard has been paid for at that figure.

Argue both sides of the claim that a program which never crashes is not necessarily correct, and reach a judgement.

Model Answer

The claim is true, and the condition that decides it is whether anybody has ever worked out the expected answer independently: without that, a clean run proves only that every operation the program attempted could be carried out, which is a statement about arithmetic and never about meaning.

What a clean run actually proves: That nothing raised. Multiplying two numbers can always be carried out, whichever two numbers they are, so an interpreter has no opinion about whether the product was the quantity you wanted. It checks that an operation is possible, never that the answer means what the name promises.

The case for treating robustness as enough: It is not nothing. A program that runs to the end on a full day of real data has survived every shape of input that day contained, and users report crashes while quietly working around wrong totals.

The fault that raises nothing: Question 33 is the case in one line: a function named for a cube multiplied its parameter twice instead of three times, printed 49 where 343 was wanted, and finished cleanly. No exception exists for a formula that is wrong, because every operation in it succeeded.

What catches it, since Python cannot: A case whose answer you worked yourself, compared with what the call gives back. The cube of 2 is 8 and that function returns 4, and the disagreement is visible in a second. This is the whole of testing, and it needs a function that returns rather than prints.

Which cases are worth choosing: The ordinary one, the boundary, and the one that must be refused: a full file, an empty file, a reading equal to yesterday's. A program deserves trust only over the shapes of input it has actually been run on.

What debugging is, once the tool is chosen: Naming a tool is not debugging. Debugging is stating what you expected a line to produce, finding what it really produced, whether by printing the value or by stopping at a breakpoint and stepping, and narrowing the distance between the two until one line disagrees.

The verdict: Never crashing is a necessary property and not a sufficient one, so the verdict is that a program is correct only where its answers have been compared against values worked outside it. A swallowed exception makes this worse, turning a fault that would have stopped the program into a wrong answer with nothing on the screen to show for it.

The chapter ends where it began: the interpreter will tell you that an operation is impossible, and it will never tell you that an answer is untrue.

The Master Key: *A program that never crashes has shown only that its operations were possible, so correctness is established by comparing its answers with values worked independently and never by the absence of a traceback.*

Chapter Three

OPERATING SYSTEMS

Almost everything else you study in computer science is something you can point at. A logic gate is a thing, a cable is a thing, a class in a program is at least a page you can read. The operating system is the one piece of software you are asked to study that works hardest when you cannot see it at all. It decides which program gets the processor next and for how long, where in memory each one sits, which of them is allowed to touch the disk, and what happens when two of them want the same thing at the same moment. When it does all of that well, the machine simply feels quick. When it does any of it badly, the machine feels broken, and nobody can say why.

The chapter builds from the outside in. It begins with what an operating system is and how the different kinds differ, then goes through the boundary between an ordinary program and the privileged code beneath it, and how the inside of that code can be organised. From there it takes the process: what one is, the five states it moves between, and how the operating system chooses which one runs next, worked out properly with charts and averages rather than described in general terms. Then come the two problems that arise the moment processes share anything, which are interference and deadlock, and they are kept apart because they are different problems with different cures. The last three sections handle memory: how it is divided, how a program can run without all of it present, and how the same questions return once the data has to survive on a disk.

Three habits earn the marks in this topic. The first is to compute rather than describe: a question that gives you burst times wants a chart and an average, and an answer that explains what shortest job first is for will score almost nothing beside one that works it out. The second is to name the part before you state the property, because most of the marks lost here are lost by attaching a true fact to the wrong construct, calling a thread a process or calling swapping virtual memory. The third is to be exact about the words that come in near-identical sets. Preemptive is not non-preemptive, internal fragmentation is not external, a page is not a frame, and there are four conditions for deadlock, not three. Nothing in this chapter is hard once the words hold still.

What an operating system is, and what it is not

This section settles what an operating system actually is before anything is asked of it. It gives the two views that every later section leans on, places the operating system correctly among the other software on a machine, and names the six things it does.

BINDER Question 1

A Kariakoo hardware stall keeps one boy behind the counter. Four customers shout for nails, hinges and pipe at once, and none of them ever touches the shelves. Take the counter away and something more than politeness is lost.

Define an operating system, and explain the two views of it: the operating system as a resource manager and the operating system as an extended machine. State what each view makes easy to see.

Model Answer

An operating system is the system software that manages a computer's hardware on behalf of every program running on it, and that presents that hardware to those programs as something simpler than the hardware itself.

The resource manager view: Seen this way, a machine is a pile of scarce resources with more claimants than supply. The operating system hands out processor time, allocates memory, admits one process to a device and makes the rest wait, then takes back what it lent when the work is done.

The extended machine view: Seen this way, bare hardware is awkward and the operating system covers it with something better. A disk understands only cylinders, heads and sectors; the operating system offers a named file that a program can open, read and close. The difficult machine has been extended into a comfortable one.

What each view answers: Ask why the operating system ran this process now, or refused that memory request, and the resource manager view answers. Ask why one program runs on two unlike machines, or why a program asks for a file by name, and the extended machine view answers. One piece of software, two questions.

Where the operating system sits: Hardware is at the base, the operating system above it as one part of the system software on the machine, applications above that, and users at the top. Students are often taught, wrongly, that a separate layer called system software sits above the operating system. The operating system is system software.

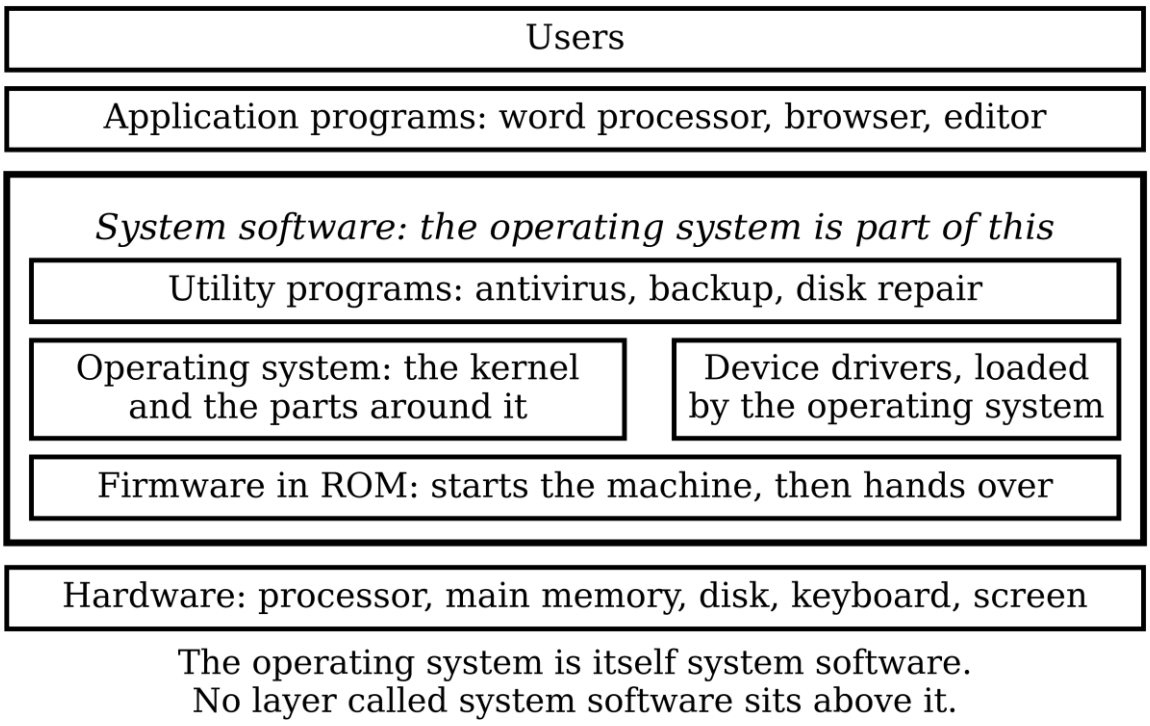


Figure 3.1: *Hardware at the base, users at the top, and the operating system inside the system software rather than underneath it.*

Define it by what it manages and what it presents, and the two views follow from the definition.

The Master Key: *An operating system manages the hardware and presents it as something easier to use.*

BINDER Question 2

The office at a Mwenge secondary school runs on one clerk. She holds the timetable, the store keys, the filing cabinet, the printer and the visitors book, and she takes the counter herself. The week she travels, six separate things stop.

State the six functions of an operating system, explain what each one does, and give one consequence of each that a user could observe.

Model Answer

The six functions are process management, memory management, file management, device management, security and protection, and the user interface, and four of the six manage a hardware resource directly.

Process management: The operating system decides which process holds the processor, for how long, and in what order the others wait. You observe it when a browser and a spreadsheet are open together and neither of them stops moving while the other is working.

Memory management: The operating system decides which bytes of main memory belong to which process, and keeps each process out of the others. You observe it when one program fails and the rest carry on, instead of the whole machine going down with it.

File management: The operating system turns a disk of numbered blocks into named files inside folders, and records which blocks are still free. You observe it when a document saved last term opens by its name, with no block number typed anywhere.

Device management: The operating system owns the drivers and the queues that stand between a program and a keyboard, a screen or a printer. You observe it when two people print within a minute of each other and each job comes out whole rather than interleaved.

The two that manage no hardware: Security and protection decides who may do what, and you observe it when a mistyped password is refused. The user interface, a shell or a desktop, is how a request is made at all, and you observe it whenever you start a program without knowing which file holds it.

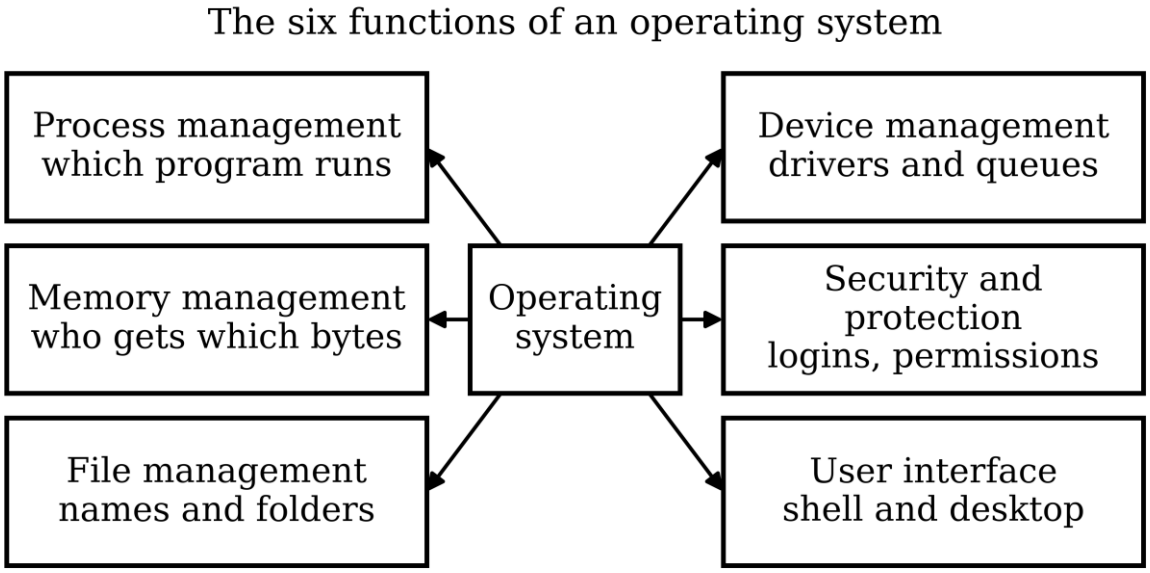


Figure 3.2: *The six functions of an operating system, four of them managing a hardware resource and two of them governing access and presentation.*

Function	**Where this chapter develops it**
Process management	Section 5, processes and threads; Sections 6 and 7, scheduling
Memory management	Section 10, partitioning; Section 11, paging and virtual memory
File management	Section 12, files, directories and allocation
Device management	Section 3, how devices are served; Section 12, disk scheduling
Security and protection	Section 12, permissions on files and directories
User interface	Section 3, the kernel and the shell

Table 3.1: *Each function and the section of this chapter that develops it.*

Name the function, then name the thing you can watch it doing, and the six stop being a list to recite.

The Master Key: *Six functions: the processor, memory, files, devices, who may do what, and how you ask.*

BINDER Question 3

A repairman in Manzese rebuilds machines from a single memory stick: the operating system, a printer driver, an antivirus, a compiler for the school next door. He calls the whole stick system software. Every machine also runs software he has never installed.

Distinguish system software from application software, and place the operating system, a device driver, a compiler, an antivirus program and firmware correctly in that division.

Model Answer

System software serves the machine and the programs on it, application software serves the person using the machine, and all five of the items named here fall on the system side.

The test that separates them: Ask who the program is for. If it exists so that the machine works, or so that other programs can run, it is system software. If it exists to do the user's own job, writing a letter or keeping accounts, it is application software.

The operating system and its drivers: The operating system is system software, and it is the piece that loads the others. A device driver is system software written for one device, which the operating system loads at start up or when the device appears. It is not part of the kernel source and can usually be replaced without rebuilding the operating system.

The compiler and the antivirus: A compiler is system software because it serves programs rather than people, turning source into something the machine can run, and it runs as an ordinary program under the operating system. An antivirus is a utility program, system software that maintains the machine instead of doing the user's work.

Firmware is not the operating system: Firmware is software held in read only memory on the board itself. It starts the machine and then hands control to the operating system, which is stored on the disk and loaded afresh every time. A common error is to treat the two as one thing because both are there before any application.

Item	Which side	What it actually is
Operating system	System	Manages the hardware and loads everything else
Device driver	System	Code for one device, loaded by the operating system
Compiler	System	Turns source into something the machine can run
Antivirus program	System	A utility that maintains the machine
Firmware in ROM	System	Starts the machine, then hands over to the operating system
Word processor	Application	Does the work of the person at the keyboard

Table 3.2: *The five items placed on the system side, with a word processor for contrast.*

Five of the six rows are system software, and the word processor is the only one that is not.

The Master Key: *System software serves the machine; application software serves the person at it.*

BINDER Question 4

At a Tandale grain mill one man holds the key to the hopper. Nobody else may open it, not the loaders, and every sack in the yard waits on him. He is not the mill, and the mill cannot run without him.

Explain what a kernel is and why an operating system needs one. State what distinguishes kernel code from the rest of the operating system.

Model Answer

The kernel is the core of the operating system: the part that is always resident in main memory and that runs with privileges the rest of the operating system does not have.

What is inside it: The kernel holds the code that arbitrates: the part that decides which process gets the processor, the part that hands out memory, and the part that reaches devices through their drivers. Anything that must not be duplicated, and must not be bypassed, belongs there.

Why an operating system needs one: Arbitration needs a single arbiter. If every program could reach the hardware for itself, two programs could write over the same memory or the same disk block, and nothing could stop one program taking the processor and never giving it back. The kernel exists so that there is one place where those decisions are made.

What separates kernel code from the rest: Privilege and residence. Kernel code runs at a level of privilege that ordinary programs are not given, and it stays in memory while the machine is on. The shell, the file utilities and the graphical desktop are operating system software too, but they run as ordinary programs and must go through the kernel to touch hardware.

The error to avoid: Students often say that the kernel is the operating system. It is the core of it, not the whole of it, and a machine loaded with only a kernel would give a user nothing to type into. The kernel is also not a program that anyone starts or stops; it is running before anything else can.

Say kernel when you mean the privileged core, and operating system when you mean the core and everything around it.

The Master Key: *The kernel is the resident, privileged core through which every request for hardware finally passes.*

REAL Question 5

A wall chart above the laboratory door at a Buguruni school gives three statements about where the operating system belongs. It has been taped there for two years and copied into hundreds of notebooks. Only one of the three is right.

Three statements about where the operating system sits among the software on a machine are given below. Identify the one that is correct, explain precisely what is wrong with each of the other two, and correct them.

Statement A: the operating system sits below a separate layer of system software, and the applications sit above that layer.

Statement B: the operating system is itself system software, and the applications run directly on top of it.

Statement C: the operating system is the desktop and the menus that the user sees, and below it there is only hardware.

Model Answer

Statement B is correct: the operating system is itself system software, and nothing called system software sits between it and the applications.

Why B is right: System software is a category, not a layer. The category holds the operating system, the drivers the operating system loads, the utility programs and the firmware. Applications sit above the whole category, and users above them. Figure 3.1 draws the machine in exactly that order.

What A gets wrong: Statement A invents a layer. It makes system software a separate slab of code with the operating system underneath, which would mean the operating system both manages the hardware and is managed by something above it. Nothing on a real machine answers to that description.

Correcting A: Hardware at the base, then system software, of which the operating system is one part alongside the drivers, the utilities and the firmware, then applications, then users. The utilities that Statement A is reaching for are peers of the operating system inside the category, not a layer above it.

What C gets wrong: Statement C confuses the whole with one of its parts. The desktop and the menus are the user interface, which is one of the six functions of an operating system. Underneath the interface is not hardware but the rest of the operating system: the part that schedules, the part that hands out memory, the part that runs the file system.

Correcting C: The user interface is the part of the operating system a user can see, and the operating system continues below it. A machine with no desktop at all, answering only typed commands, still has a complete operating system underneath.

The habit that catches both: Both wrong statements come from reading a diagram as a stack of layers and then naming the layers loosely. Name the part before you state its position, and neither error survives.

Two of the three statements are wrong, and both are wrong about the same thing: what the phrase system software actually names.

The Master Key: *The operating system is one part of the system software, not a floor beneath it.*

REAL Question 6

The single machine in the Magomeni school office was bought second hand and does nothing clever. In one afternoon it saved a document, refused a password, queued three printouts and warned about the disk. How many of its six functions is that?

A school machine is observed doing the five things listed below. Name the operating system function responsible for each, and say what the machine would do instead if that function were absent.

One: a document reopens showing the page it was closed on.

Two: a browser and a spreadsheet are both open, and neither of them freezes.

Three: printouts sent within a minute of each other come out one after another, each one whole.

Four: a mistyped password is refused and the machine returns to the login prompt.

Five: a warning appears that the disk is nearly full.

Model Answer

The five observations are file management, process management, device management, security and protection, and file management again, so they name four of the six functions between them.

The document that reopens where it was left: File management. Without it a disk is a numbered sequence of blocks and nothing records which blocks hold your document, let alone which page you stopped on. You would have to remember a block number and read it back yourself.

Two programs, neither of them frozen: Process management. Without it whichever program started first would hold the processor until it finished, and the second would not move at all until then. Nothing would be deciding when to take the processor from one and give it to the other.

The printouts that come out whole: Device management. Without it whichever program wrote to the printer last would win, and the pages of the two jobs would come out interleaved. The queue exists because the printer can serve one program at a time and the operating system decides which.

The password that was refused: Security and protection. Without it anyone who reached the keyboard would have every other person's files, and there would be no way to say that a student may read a document but not delete it.

The warning about the disk: File management again. Without it nothing would be counting free blocks, so a write would simply fail part way through and leave a half written file behind with no warning of any kind.

What the afternoon did not show: Two of the five are file management, so the five observations reach four functions, not five. Memory management and the user interface are the two that are missing: memory management is the function you notice only when it fails, and the interface is what you are looking at while you watch the other four.

Every one of these is invisible until it is absent, which is why the absence is the best way to name it.

The Master Key: *Ask what the machine would do without the function, and the function names itself.*

HOT Question 7

A pump controller in a Msasani boatyard runs one program forever: read the float switch, drive the motor, stop at the top switch. It has no operating system and has not failed in three years. The workshop now wants it to keep a log.

A small embedded controller runs a single program on bare metal with no operating system at all. Argue whether an operating system is necessary, and reach a judgement about when one earns its cost.

The controller board has 32 kilobytes of program memory and no disk.

It runs one program: read the float switch, drive the motor, stop at the top switch.

A small operating system kernel for this board occupies 20 kilobytes.

The workshop wants to add a second job: logging running time to a memory card.

Model Answer

An operating system is not necessary for this controller as it stands, and installing one would leave only twelve of its thirty two kilobytes for the program itself; it becomes necessary the moment the board must do two things at once or answer to more than one user.

The case for bare metal: With one program there is nothing to arbitrate. The program owns the processor, the memory and the pins outright, so no scheduling decision exists, because no second claimant exists. Timing is exact as well: the pump responds after a known number of instructions, every time, with nothing else able to take the processor away.

What the operating system would cost here: The board holds thirty two kilobytes of program memory. A small kernel for it occupies twenty, which leaves twelve kilobytes for the pump logic, the log and everything else. On a board this size the cost of an operating system is not an abstraction; it is most of the memory.

The case for an operating system: Add the log and the picture changes. Something must decide what happens when the float switch changes while the memory card is being written. Written bare, that decision becomes a hand coded loop of flags inside the program, and it is the same decision a scheduler makes, made worse by somebody who did not know that is what they were writing.

The second user nobody counted: If a technician can plug in a cable and read the log while the pump is running, the board now serves two users, and something must keep one from disturbing the other. That is protection, and it is operating system work whether or not an operating system is present.

The verdict: On balance, leave this controller bare while it runs one program for one user, and give it an operating system as soon as it must run two things at once or answer to a second user. The condition is concurrency, not complexity: a long and difficult single program still needs no operating system, and two short ones do.

An operating system earns its cost at the point where something has to be shared, and not before.

The Master Key: *One program and one user need no operating system; two of either do.*

HOT Question 8

The Kariakoo bus stand has one conductor. To a passenger he is a man who takes the fare and gives a seat. To the depot he is deciding which of two hundred people get sixty. Both descriptions are true; only one explains his temper.

The resource manager view and the extended machine view describe the same software. Argue which of the two is the more useful for explaining why operating systems are designed as they are, and reach a judgement.

Model Answer

The resource manager view is the more useful of the two for explaining why operating systems are designed as they are, because almost every hard decision in an operating system is a decision about who gets a scarce thing.

What the resource manager view explains: Scheduling, allocation and protection all fall out of it. Why one process runs before another, why a memory request is refused, why a device is given to one program and withheld from the rest: each is an allocation decision, and each has a measurable objective behind it.

What the extended machine view explains: Interfaces and portability fall out of it. Why a program asks for a file by name rather than by cylinder and sector, why the same program runs on two unlike machines, and why the set of services an operating system offers looks similar across very different designs.

The case for the extended machine view: It is the stronger of the two for a user, and it explains the shape of what an operating system offers. Take it seriously and you can predict the services a new system will provide before reading a line about it, because every operating system must hide the same awkward hardware behind the same kinds of call.

Why it still loses: The extended machine view describes what the operating system offers but not why any of it is difficult. It would be satisfied by a machine with one program on it, where every request could simply be granted. Nothing in that picture explains why an operating system is hard to write.

The judgement: Prefer the resource manager view when the question is why the operating system behaves as it does, because contention is what forces every design decision, and prefer the extended machine view when the question is what the operating system offers a program. Design pressure comes from scarcity, so the resource manager view wins.

Both views describe the same software at the same time, and the one that explains the design is the one that starts from scarcity.

The Master Key: *The extended machine view says what an operating system offers; the resource manager view says why.*

Classifying operating systems, and how they got here

This section supplies the classification the common teaching leaves out. It separates the way an operating system processes work from the kind of device it runs on, keeps four similar words apart, and puts dates and named systems on the evolution.

BINDER Question 9

At a Mikocheni duplicating shop the clerk stacks forty job cards in one tray at eight in the morning and collects the finished output at noon. Nobody touches the machine between. Computing removed three different kinds of waiting, in a fixed order.

Define batch processing, multiprogramming and time-sharing, and explain what problem each was introduced to solve.

Model Answer

Batch processing, multiprogramming and time-sharing are three ways of moving work through one machine, and each was introduced to remove a different kind of idle time.

Batch processing: Jobs with similar requirements are collected together and run one after another, with no interaction from the user while they run. A resident monitor loads the next job the moment the last one ends. The waste it removed was operator idle time, the minutes a human spent setting up between jobs.

Multiprogramming: Several programs are held in main memory at the same time, and when the running program starts an input or output operation the processor is handed to another program instead of standing still. The waste it removed was processor idle time during input and output.

Time-sharing: The processor is switched between several users so rapidly that each believes the machine is his alone, and one person's keystroke is answered while the others are still typing. The waste it removed was the user's own waiting time, since a small job queued in a batch might not return until morning.

The chain from one to the next: Each stage removed the waste the stage before it left behind, so the order is not arbitrary. Batch removed the idle operator, multiprogramming removed the idle processor, and time-sharing removed the waiting person.

A common error: Batch processing is often treated as merely old. It is not. A payroll run or an overnight stock valuation is batch work today, because the defining feature is that the job runs without interaction, not the decade in which it was written.

Way of working	What happens	Idle time it removed	User contact while running
Batch processing	Jobs are queued and run one after another	Operator idle time between jobs	None
Multiprogramming	Several programs are in memory; the processor moves to one that is ready	Processor idle time during input and output	None required
Time-sharing	The processor is switched between users many times a second	The user's own waiting time	Continuous

Table 3.3: *The three ways of processing work, and the waiting each one abolished.*

Batch and time-sharing describe how work reaches the processor; multiprogramming describes what is in memory while it gets there.

The Master Key: *Batch removed the operator's idle time, multiprogramming the processor's, and time-sharing the user's.*

BINDER Question 10

A Tegeta lorry brakes on a wet slope, and the anti-lock controller must decide within four milliseconds whether to release the wheel. On the same road a driver's music stutters for a moment. Both are called real-time, yet only one of them has failed.

Define real-time, distributed, embedded and network operating systems, and distinguish a hard real-time system from a soft one with an example of each.

Model Answer

A real-time operating system answers within a stated deadline, a distributed system spreads one workload across machines that appear as one, an embedded system is built into a device to run a fixed set of tasks, and a network operating system lets separate machines share resources.

Real-time and its deadline: A real-time operating system must produce its answer inside a fixed time bound, so a correct result delivered late counts as a failure. What is promised is the worst case, not the average, which is why such a system is built to be predictable before it is built to be quick.

Hard against soft: A hard real-time system that misses one deadline has failed outright: anti-lock braking that decides after the wheel has locked has not braked. A soft real-time system that misses one degrades instead: video playback drops a frame, the viewer sees a stutter, and the film goes on.

Distributed: Several independent computers, each with its own memory, are presented to the user as a single system, and the work is placed on one of them without the user choosing. A clerk who books a seat cannot tell which machine served the request.

Embedded: An embedded operating system is built into a device that nobody calls a computer, runs a fixed set of tasks, and occupies a small amount of storage, often firmware. A router and a washing machine controller each carry one.

Network: Each machine keeps its own operating system and its own name, and the network operating system adds shared files, shared printers and shared logins across them. The user knows which machine he is working on, which is exactly what a distributed system hides.

Type	What defines it	Where you meet it
Hard real-time	A missed deadline is a failure	Anti-lock braking, a pacemaker
Soft real-time	A missed deadline degrades the service	Video playback, a voice call
Distributed	Many machines presented to the user as one	Booking shared across four offices
Embedded	Fixed tasks inside a device, small storage	A router, a washing machine
Network	Separate machines sharing resources	A file server on an office network

Table 3.4: *Four types, and the pair of real-time systems that differ in what a missed deadline costs.*

Hard and soft real-time differ in the cost of one missed deadline, and distributed and network systems differ in whether the user can tell the machines apart.

The Master Key: *Hard real-time fails on one missed deadline; soft real-time only degrades.*

BINDER Question 11

Four words beginning with multi are written on the wall of a Kimara repair shop, and the owner uses them interchangeably while three of his laptops lie open on the bench. Only one of the four says anything about the number of processors.

Distinguish multiprogramming, multitasking, multiprocessing and multithreading, stating exactly what is multiple in each.

Model Answer

The four words differ in what there is more than one of: multiprogramming has several programs in memory, multitasking has several turns on one processor, multiprocessing has several processors, and multithreading has several threads inside one process.

Multiprogramming: Several programs are resident in main memory at the same time. The processor still runs one instruction stream, but it never stands idle while one program waits on input or output, because another program in memory is ready to use it.

Multitasking: One processor is shared between several programs in turns short enough that a person sees them all progressing at once. Multitasking is time-sharing applied to the programs of one user, and it requires no second processor of any kind.

Multiprocessing: The machine has two or more processors, so two instruction streams genuinely run in the same instant rather than in alternation. This is the only one of the four that is a statement about the hardware as well as about the software.

Multithreading: One process is divided into several threads which share its memory and its open files. The thread, not the process, is the unit the processor is given, so a single process can keep working while one of its threads waits.

How to tell them apart: Ask what is multiple: programs in memory, turns on one processor, processors, or threads inside one process. A single-processor laptop running a browser and a music player is multitasking, and it becomes multiprocessing only when a second processor is fitted.

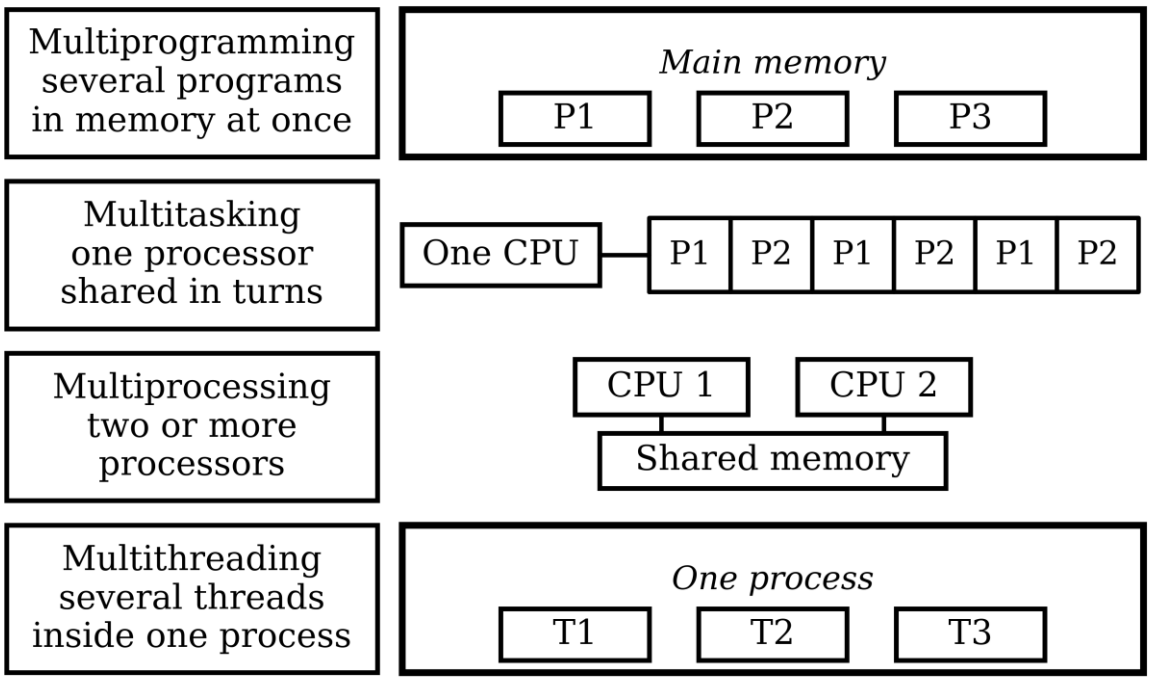


Figure 3.3: *The four words side by side, each drawn beside the thing there is more than one of.*

One machine can be all four at once, so saying which you mean means naming what is multiple.

The Master Key: *Count what is multiple: programs, turns, processors or threads.*

BINDER Question 12

A Kibamba college keeps a machine from 1998 in a store room beside a handset bought last month. Between the two lies half a century of operating systems, and almost nobody can name the systems or the years in order.

Trace the evolution of operating systems from the earliest machines to the present, naming a real system and an approximate date for each stage.

Model Answer

Operating systems developed in eight stages, from the resident batch monitors of the 1950s, through OS/360 in 1964 and UNIX in 1969, to MS-DOS in 1981, the Macintosh in 1984, Windows in 1985, Linux in 1991 and Android in 2008.

The 1950s and the resident monitor: The earliest machines were loaded one job at a time by hand. A resident monitor, a small permanently loaded program on machines such as the IBM 704, read the next job from a card deck and started it without a human present.

OS/360 in 1964: IBM announced one operating system for a whole family of compatible machines instead of a fresh one for every model, and it supported multiprogramming. It came nine years after the monitors of the middle 1950s and is the ancestor of general-purpose systems since.

UNIX in 1969: Written at Bell Laboratories by Ken Thompson and Dennis Ritchie, five years after OS/360, UNIX was multi-user and time-sharing from the start. Rewriting it in C four years later made it portable between machines, which no earlier operating system had been.

The personal machine, 1981 to 1985: MS-DOS reached the IBM Personal Computer in 1981, twelve years after UNIX, single-user and single-tasking with a command line. The Macintosh followed three years later with a graphical interface as standard, and Windows one year after that as a graphical environment running on top of MS-DOS.

Free kernels and the handset: Linus Torvalds released the Linux kernel in 1991, six years after Windows, free and open source. Android reached its first handset in 2008, seventeen years after Linux, carrying that kernel onto a touch screen; the whole timeline covers fifty-three years.

Date	System	What it made possible
1950s	Resident batch monitors	Jobs run one after another with no operator between them
1964	OS/360	One operating system across a family of machines, with multiprogramming
1969	UNIX	Multi-user time-sharing, and portability once rewritten in C
1981	MS-DOS	A single-user, single-tasking command line on the personal computer
1984	Macintosh	A graphical interface as standard equipment
1985	Windows	A graphical environment running on top of MS-DOS
1991	Linux	A free and open source kernel, developed in the open
2008	Android	A Linux kernel on a touch screen handset

Table 3.5: *Eight stages, each with a real system and the year it appeared.*

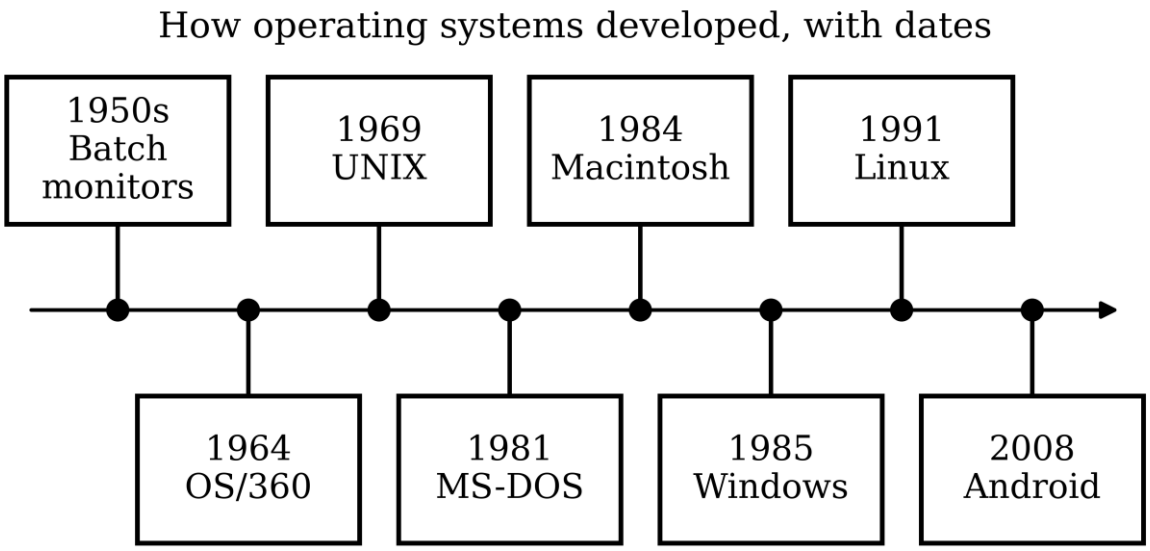


Figure 3.4: *The same eight stages on one time line, earliest at the left.*

Each stage answered the limitation of the one before it, which is why the dates fall in the order they do.

The Master Key: *Batch monitors in the 1950s, OS/360 in 1964, UNIX in 1969, MS-DOS in 1981, the Macintosh in 1984, Windows in 1985, Linux in 1991 and Android in 2008.*

REAL Question 13

Six installations around Mbagala were logged in one afternoon by a technician on a motorcycle, from a sugar depot to a radio station. Each moves work through its processor in a different way, and one of the six refuses to be only one thing.

Classify each of the six installations below by the way it processes work, and justify the classification in one sentence each.

Kimara: a sugar depot queues 3,600 delivery notes as a single overnight run, with no operator present and the report collected next morning.

Tegeta: a school computer room runs 24 terminals from one server, and every student expects a reply while the others are typing.

Goba: a water pump controller, built into the pump housing, must close a valve within 8 milliseconds of a pressure reading or the pipe bursts.

Mbagala: a bus company runs booking on four servers in four offices, which the clerk sees as one seat map.

Chanika: a clinic keeps a separate computer in each consulting room, each with its own operating system, sharing patient files over the local network.

Mikocheni: a radio playout machine holds the newsroom editor, the mixer and the transmission log in memory together, giving the processor to whichever is ready.

Model Answer

Each installation is classified by the way it moves work through the processor rather than by the kind of box it occupies, and one of the six needs both axes named before the classification means anything.

Kimara, batch processing: The 3,600 delivery notes are queued as one job and run with nobody present, and the output is collected afterwards. No interaction is possible while the run proceeds, which is the defining feature of batch work and has nothing to do with the age of the machine.

Tegeta, time-sharing: Twenty-four terminals share one server and each student expects an answer while the others type, so the processor must be switched between users many times a second. At 10 milliseconds a turn, a full round of 24 turns finishes in 240 milliseconds, which a person reads as instant.

Goba, two answers on two axes: This is the installation that is two things at once. On the processing axis it is hard real-time, since an 8 millisecond bound means answering up to 125 times a second and a late valve bursts the pipe. On the device axis it is embedded, since the controller is inside the pump.

Mbagala, distributed: Four servers present one seat map and the clerk cannot tell which of them answered. Many machines are made to behave as one, which is distribution rather than a shared folder.

Chanika, network: Each consulting room keeps its own computer and its own operating system, and the server adds shared files across them. The user knows which machine he is on, so this is a network operating system and not a distributed one.

Mikocheni, multiprogramming: The editor, the mixer and the log are all resident in memory and the processor goes to whichever is not waiting on the disk. Nothing here is said about users or deadlines, so multiprogramming is the honest classification.

Five of the six take one label, and the sixth takes one on each axis.

The Master Key: *Classify by what the installation does with work, and name the axis whenever two answers are both true.*

REAL Question 14

A neatly typed page in a training folder at a Chanika computer shop offers three statements about multitasking and multiprocessing, and the trainee has learned all three. Two of them are true sentences about the wrong word.

Identify which of the three typed statements is correct, explain what is wrong with the other two, and correct them.

Statement A: Multitasking means the computer has more than one processor, so two programs truly run in the same instant.

Statement B: Multiprocessing means the operating system switches between programs so quickly on one processor that the user cannot tell.

Statement C: Multitasking is one processor shared in turns between several programs, and multiprocessing is two or more processors running instructions in the same instant.

Model Answer

Statement C is the correct one, and A and B have exchanged the two definitions, so each of them is a true description of the other term.

Why C is right: Multitasking is one processor shared in turns between several programs, and multiprocessing is two or more processors running instructions in the same instant. C names the processor count in each, which is the only thing that separates them.

What is wrong with A: A claims that multitasking requires more than one processor. It does not. A single-processor machine running a browser and a music player is multitasking, and the turns are short enough that the user sees no gap. A should read that multitasking shares one processor in turns.

What is wrong with B: B describes rapid switching on one processor and calls it multiprocessing. That description belongs to multitasking. B should read that multiprocessing means two or more processors running instructions in the same instant, with real simultaneity rather than the appearance of it.

The two errors are one error: A and B carry each other's definitions, which is what makes the page look plausible, because every sentence on it is true of something. Either statement read alone would leave the reader confident and wrong, so the pair has to be corrected together.

The number that settles it: Count the processors, then count the elapsed time. Two programs each needing 30 seconds of processor time finish in 60 seconds on one processor however the turns are arranged, and in 30 seconds on two. Multitasking rearranges the waiting, and multiprocessing removes half of it.

Correcting the page: The corrected page needs two lines only: multitasking is one processor in turns, multiprocessing is several processors at once, and the machine in front of you may well be doing both.

Two of the three sentences are accurate English about the wrong term, which is the hardest kind of error to notice.

The Master Key: *Multitasking shares one processor in turns; multiprocessing needs a second processor.*

REAL Question 15

A Goba engineer is asked to choose operating systems for three installations in one week: a bottling line, a college payroll and a district office. She is given only their timings, and the timings decide everything.

Choose an operating system type for each of the three deployments below, and justify the choice against the stated requirement.

Tegeta: a bottling line whose capper must stop within 4 milliseconds of a jam signal, on every occasion, or glass breaks.

Kibamba: a college payroll of 7,200 records that must be processed between 19:00 and 05:00 with nobody at a terminal.

Chanika: a district office where 30 clerks share one server and each keystroke must be answered inside 150 milliseconds.

Model Answer

The stated requirement decides the type: a bound that must never be exceeded calls for a hard real-time system, an unattended overnight workload calls for batch processing, and many interactive users on one machine call for time-sharing.

Tegeta takes a hard real-time system: The requirement names a bound that must hold on every occasion, and a 4 millisecond bound means the controller must be able to respond 250 times a second. A general-purpose system with an excellent average would still break glass on the one occasion it was late. The stated figure is a ceiling and not a target, and only a system that can state its own worst case can be held to it.

Kibamba takes batch processing: Nobody is at a terminal, so interactive response is worth nothing here. The requirement is that 7,200 records clear inside the 10 hours between 19:00 and 05:00, which is 720 records an hour, so the work is queued as one job and left to run unattended.

Chanika takes time-sharing: Thirty clerks each need an answer inside 150 milliseconds, so a turn of 150 divided by 30, which is 5 milliseconds, brings the processor round to every clerk inside the stated bound. Serving many interactive users from one machine is what time-sharing was built for, and no deadline guarantee is needed here, since a clerk kept waiting an extra moment loses nothing but the moment.

Justifying against the number, not the reputation: Each choice above cites the stated figure: the bound of 4 milliseconds, the window of 10 hours, the reply inside 150 milliseconds. A justification that says real-time is faster, or that batch is old-fashioned, has answered a question nobody asked.

What the wrong choices would cost: Put the bottling line on the district office server and the capper waits behind thirty clerks. Put the payroll on the real-time controller and its guarantees buy nothing, because no deadline is attached to a run that need only finish by morning.

Three requirements, three types, and in each case the number in the requirement is what chose it.

The Master Key: *Read the timing in the requirement first, because it names the type before any preference does.*

HOT Question 16

A Mikocheni welder times his generator changeover with a stopwatch: usually two seconds, once in a while eleven. The average is excellent and the machine still ruined a weld. A real-time system is sold on a promise the average cannot make.

It is often said that a real-time operating system is simply a fast operating system. Argue for and against this claim, and reach a judgement.

Model Answer

A real-time operating system is not simply a fast one: it trades average speed for a guaranteed worst case, and speed with no bound attached is exactly what a real-time requirement cannot use.

The case for the claim: Real-time systems really are quick. Their kernels are small, they answer a device in microseconds, and on identical hardware the faster of two systems misses fewer deadlines. A system too slow to finish the work inside the bound cannot meet that bound at any price, so speed is a genuine part of the story.

Speed is reported as an average: A general-purpose system is tuned for throughput and for the typical case, and the mechanisms that produce good averages, caching, buffering and pushing work into a quieter moment, all make the worst case both worse and harder to know in advance.

A real-time system is judged at its worst: What such a system sells is a bound. A response that averages 2 milliseconds but reaches 40 milliseconds once an hour has failed a 10 millisecond deadline, overshooting it by 30 milliseconds, however good the average looks on the datasheet.

The engineering differs, it is not merely quicker: Predictability is designed in. The worst case is calculated before the system ships, code paths that could take an unbounded time are removed, and the designer accepts a slower average in order to keep the ceiling where it was promised.

Where the claim is not silly: For soft real-time work a fast general-purpose system is often the right purchase, because a dropped video frame degrades the service instead of ending it, and raw speed genuinely does reduce how often that happens.

The judgement: The claim is rejected wherever a deadline must never be missed, because on that requirement a fast general-purpose system will not do at any clock speed: it guarantees nothing. Where the deadline is soft, the verdict turns over, and the faster general-purpose machine is the sensible buy.

Speed is a means, and the thing a real-time system actually delivers is a bound that holds on the worst day.

The Master Key: *A fast system is quick on average; a real-time system is on time at its worst.*

HOT Question 17

A Kimara shopkeeper sorts four hundred items by shelf size; his supplier sorts the same four hundred by what each one does. Both catalogues are correct, and only one of them answers a customer who asks what a thing is for.

Operating systems can be classified by processing mode or by the device they run on. Argue which classification is the more useful, and reach a judgement about when each is the right one to use.

Model Answer

The processing-mode classification is the more useful when the question is what an operating system does, and the device classification is the more useful when the question is what to install on a particular machine, so neither one replaces the other.

What each axis sorts: The processing-mode axis sorts by how work reaches the processor: batch, multiprogramming, time-sharing, real-time, distributed. The device axis sorts by the machine the software runs on: desktop, server, handset, and the small controller built into an appliance.

The case for the device axis: It is the axis a buyer uses, and it is honest about that. Told that a machine is a handset, you can predict its interface, its power budget and the software available for it, and you can name what to install in one word. When the decision is a purchase or an installation, this axis answers it.

Where the device axis stops working: It classifies the deployment rather than the operating system. One Linux kernel is found on a server, on a desktop, inside a handset as Android and inside a router, so the device name tells you almost nothing about what the system guarantees or how it treats waiting work.

The case for the processing-mode axis: It predicts behaviour. Hard real-time tells you a deadline is guaranteed, time-sharing tells you many users are served in turns fast enough to feel private, and batch tells you no answer arrives until the run ends. Each label commits the system to something a user can check.

Where the processing-mode axis stops working: A modern general-purpose system is several of these at once: multiprogramming, multitasking, multi-user, and soft real-time wherever sound is playing. Asked for a single label it has no honest one, and the axis then has to be applied to a named part of the system instead.

The judgement: Classify by processing mode when explaining or predicting what a system will do, which makes it the more useful of the two for understanding, and by device when choosing what to install on the hardware in front of you. The tie-break is whether the question is about behaviour or about purchase.

The device axis names where an operating system sits, and the processing-mode axis names what it promises.

The Master Key: *Device answers what to install; processing mode answers what the system will actually do.*

Kernel and shell, the two modes, system calls and interrupts

This section explains how a program that is not allowed to touch the hardware gets the hardware to do something. It separates the kernel from the shell, gives the two modes and the instruction that switches between them, and follows one request all the way down and back.

BINDER Question 18

At the Kivukoni ferry office a clerk takes your fare, writes a docket and pushes it through a hatch. Behind the hatch, unseen, someone decides which boat moves. Replace the clerk tomorrow and the crossings run as before.

Distinguish the kernel from the shell. Explain why the shell is an ordinary program and what follows from that.

Model Answer

The kernel is the part of the operating system that stays in memory and runs in kernel mode with complete access to the hardware, while the shell is an ordinary user-mode program that reads what you type and asks the kernel, through system calls, to carry it out.

What the kernel is: at start-up a small bootstrap program held in firmware loads the kernel into memory, and it stays resident there until the machine is switched off. It runs in kernel mode, so it alone may write to a device register, alter the memory map or take the processor away from a running program. Every service the operating system offers is reached through it.

What the shell is: the shell is the command interpreter. It reads a line, works out which program you meant, asks the kernel to load and run that program, and shows you what comes back. A graphical desktop is a shell as well, doing the same job with a pointer instead of typing.

Why the shell is an ordinary program: the shell holds no privilege of its own. It runs in user mode, waits in the ready queue like everything else, and cannot read a disk sector by itself; it obtains one by asking. None of it is built into the kernel, and ending it does not stop the machine.

What follows from that: a shell can be replaced without replacing the operating system, one machine can carry several at once, and two users on the same machine can be given different ones. A shell that fails takes one process down. The common teaching calls the shell the user interface and stops, which hides all of this.

The kernel is where the authority sits, and the shell is one of the programs that queue up to use it.

The Master Key: *The kernel is the part that may act, and the shell is a part that asks.*

BINDER Question 19

The pump attendant at Vikindu can fill a drum all day, but the padlock on the main tank valve is not his. When a driver reached for it, an alarm sounded before his hand arrived. Something was watching the reach, not the result.

Explain user mode and kernel mode, the mode bit, and what a privileged instruction is. State what happens when a user-mode program attempts one.

Model Answer

A processor runs in one of two modes and a single bit in a status register says which: in kernel mode, where the bit reads 0, every instruction is permitted, and in user mode, where the bit reads 1, the privileged instructions are not.

Kernel mode: kernel mode, sometimes also called supervisor mode, is the state in which the processor will execute anything at all: input and output instructions, the instruction that halts the machine, the loading of memory-management registers. Kernel code runs here and nothing else does.

User mode: user mode is where every application runs, the shell and the compiler among them. Arithmetic, branching and access to memory inside the program's own space all work normally. Anything that could damage another program or the machine itself is refused by the hardware.

The mode bit: the mode bit is hardware, not a variable the operating system keeps, and the processor consults it as it decodes each instruction. No user-mode instruction can set it to 0. Only a trap or an interrupt does that, and each of those also moves control into kernel code, so gaining privilege and leaving your own program happen in the same act.

What a privileged instruction is: a privileged instruction is one the hardware will execute only while the bit reads 0. Halting the processor, setting the interval timer, masking interrupts, issuing a command to a disk controller and loading the relocation register are the usual members of the set.

What happens when one is attempted: the hardware raises a trap. It is not a compilation error, because the same instruction is perfectly legal inside the kernel, and it is not silently ignored. The mode bit goes to 0, control passes to the trap handler, and the handler ordinarily ends the offending process.

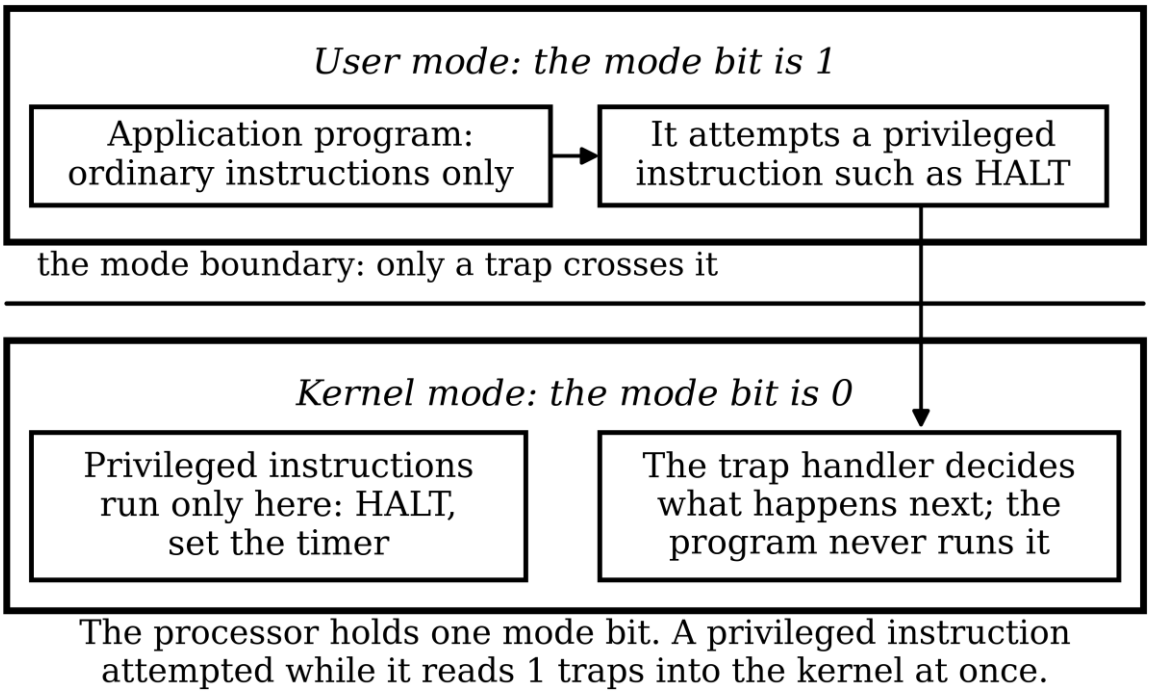


Figure 3.5: *The two modes, the boundary between them, and the trap that is the only way across it.*

The separation is enforced by the processor itself, which is why no program can simply decide to become privileged.

The Master Key: *One bit decides what the processor will obey, and only a trap or an interrupt may change it.*

BINDER Question 20

At a hardware counter in Mbezi you do not walk into the store yourself. You fill a slip, hand it over, and a storeman brings the six-inch nails. Nobody has ever seen a customer behind the shelves.

Define a system call and name the six families of system call, giving one example of each. Explain why a system call is not an ordinary function call.

Model Answer

A system call is a request from a user-mode program for a service that only the kernel can perform, and it differs from an ordinary function call because it changes the privilege of the processor and enters the kernel at an address the kernel fixed, never at one the caller chose.

How the request is made: the program puts a call number in a register and the arguments in registers or in a block of memory whose address it passes. It then executes the trap instruction. The kernel reads the number, checks the arguments, and dispatches to the routine that serves that number.

Why it is not an ordinary function call: an ordinary call jumps to an address inside the same program at the same privilege and comes back with a value. A system call crosses the mode boundary, moves to a kernel stack, and may enter only through the one gate the kernel opens. It therefore costs a great deal more than a call within the program.

The library that sits in between: programs seldom execute the trap themselves. A library routine packages the arguments and does it for them, which is why a system call is so often mistaken for a function call: the source shows the wrapper, not the crossing underneath it.

The six families: the calls a machine offers fall into six families, and one ordinary command uses several of them. Running a program takes process control to create it, file management to read it from the disk, and information maintenance to ask the system what the time is.

Family	What it asks for	Example
Process control	create, run, wait for and end a process	fork, exec, wait, exit
File management	make, open, read, write and delete a file	open, read, write, close
Device management	claim a device, use it and give it back	request device, release device
Information maintenance	read or set a value the system keeps	get the time, get the process identifier
Communication	open a channel and pass data between processes	send message, receive message
Protection	set and read who may use a resource	set permissions, allow user

Table 3.6: *The six families of system call, with one example of each.*

A system call is the only door in the wall, and the kernel decides where it stands and what may come through.

The Master Key: *A system call is a request across a privilege boundary, not a jump within a program.*

BINDER Question 21

The dispensary at Nyanguge has one bell and nine rooms. It rings, and the nurse does not run about asking who needs her; the number showing on the board sends her straight to room four. The board is doing more work than the bell.

Explain the interrupt cycle and the purpose of the interrupt vector. Distinguish a hardware interrupt from a software interrupt.

Model Answer

An interrupt is a signal that makes the processor finish the instruction it is on, save enough state to come back, run a kernel routine chosen by the interrupt number, and then resume the interrupted work as though nothing had happened.

The cycle: after each instruction the processor tests the interrupt line. If a signal is waiting, it saves the program counter and the status register, clears the mode bit to 0, takes the interrupt number from the device, reads the handler address out of the vector, and jumps to it.

How the cycle ends: the handler services the device and acknowledges it so the line is released, then executes a return-from-interrupt instruction. That restores the saved program counter, the status register and the previous mode together. The interrupted program is never told that any of it happened.

The interrupt vector: the interrupt vector is a table of handler addresses held at a fixed place in memory and indexed by interrupt number. The entry for a device holds the address of the routine that serves it, so the hardware reaches the right code in one lookup instead of asking every device in turn who called.

A hardware interrupt: a hardware interrupt is raised by a device on a physical line and arrives with no warning: a disk has finished a transfer, a key has been pressed, the interval timer has run out. It is asynchronous, because nothing the running program did caused it.

A software interrupt: a software interrupt, also called a trap, is raised by the instruction the processor is executing: a system call, a division by zero, an address outside the program's own space. It is synchronous, and running the program again reproduces it at the same instruction.

The timer interrupt deserves a line of its own, because it is what lets the kernel take the processor back from a program that would otherwise keep it.

The Master Key: *An interrupt tells the processor to go elsewhere, and the vector tells it exactly where.*

REAL Question 22

A boat clerk at Kimbiji wants one line from the landing register, which is locked in the safe. He cannot open the safe. He writes the date on a chit, and four minutes later the line comes back to him copied onto a sheet.

Follow a single request from a user program that wants to read a file, from the library call to the return of the data. Name every boundary it crosses and say what changes at each.

A program running in user mode has a file already open and asks for one record of 512 bytes.

The record is not in memory, so the disk has to be read.

The disk transfers whole blocks of 4096 bytes, and the record is one eighth of a block.

Model Answer

The request crosses two boundaries in each direction, the mode boundary between the program and the kernel and the boundary between the kernel and the device, and at every crossing something changes hands: either the privilege the processor is running at, or the ownership of the data.

Step 1, the library call: the program makes an ordinary call to the library routine that reads a file, passing the descriptor, the address of its own buffer and the count of 512 bytes. Nothing has been crossed yet. This is a call inside the program, at user privilege, and the mode bit still reads 1.

Step 2, the trap: the library loads the call number for a read and executes the trap instruction. The mode bit goes to 0, the processor moves to a kernel stack, and control lands at the kernel entry point. This is the first boundary, and what changed at it is privilege, not the process.

Step 3, the kernel checks and looks: the kernel confirms that the descriptor is open and that the buffer address belongs to the calling program, then searches its own buffers for the block. It is not there, so the disk must be read, and the calling process is put into the waiting state until it arrives.

Step 4, the driver starts the disk: the driver writes the block number and the destination address into the controller registers and starts the device. This is the second boundary, between the kernel and the hardware, and what changed is that the work now belongs to the device. The processor is free and runs something else.

Step 5, the interrupt: the controller places all 4096 bytes into a kernel buffer and raises an interrupt when the block is complete. That is the device boundary crossed back. The processor stops, the vector sends it to the disk handler, and the handler makes the waiting process ready again.

Step 6, the copy: the kernel copies the 512 bytes that were asked for out of its own buffer into the buffer inside the program. The copy is needed because kernel memory is not visible to the program and the device was never given the address of user memory.

Step 7, the return: the kernel puts the count of bytes read into the return register and executes the return-from-trap instruction. The mode bit goes back to 1, the user stack is restored, and the library routine returns to the line after the call with the data in place.

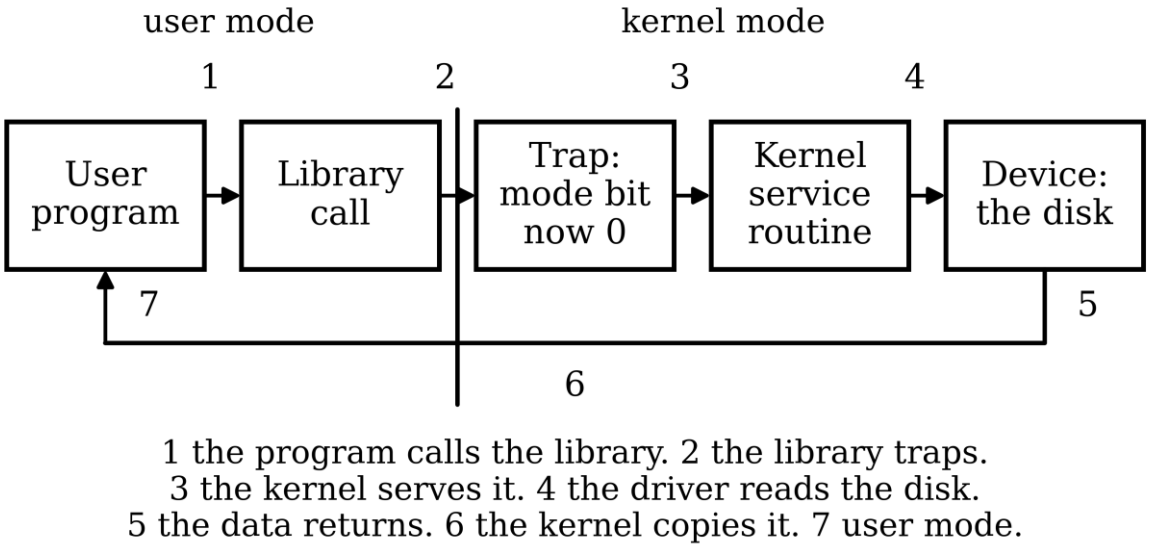


Figure 3.6: *One read followed down into the kernel, out to the disk and back, with the mode boundary marked.*

Nothing inside the program changed but the contents of its buffer, and it never once held the privilege that fetched them.

The Master Key: *A read goes down two boundaries and back up two, and the program's whole part in it is to ask.*

REAL Question 23

A lorry at Lamadi carries one thousand and twenty-four sacks of rice. The turnboy can hand them down one at a time, or start a conveyor and stand aside. The difference is not effort; it is how often he is called back.

A device transfers a block of 4096 bytes. Compare programmed input and output, interrupt-driven input and output, and direct memory access on that transfer, computing how many times the processor is involved under each.

A device is to transfer one block of 4096 bytes into memory.

The processor moves data one word at a time, and a word on this machine is 4 bytes.

One involvement means one moment at which the processor must leave other work to attend to this transfer.

Model Answer

The block is 1024 words, so programmed input and output involves the processor 1024 times, interrupt-driven input and output involves it 1025 times, and direct memory access involves it twice, once to set the controller going and once to take the completion interrupt.

The size of the block in words: divide the block by the word. 4096 bytes at 4 bytes to a word gives 1024 words. All three methods move those same 1024 words; what differs between them is who does the moving and how often the processor has to be there for it.

Programmed input and output: the processor moves every word itself, so it is involved 1024 times. It also discovers that the device is ready by polling, reading a status register over and over until the bit changes, so between words it is not free at all. It is spinning.

Interrupt-driven input and output: the device raises one interrupt when it is ready and the processor still copies every word, so the count is 1025, one interrupt plus 1024 copies. That is one more than programmed transfer, not fewer, and the point is worth stating plainly: what interrupts buy back is the waiting, not the copying.

Direct memory access: a separate controller moves the block. The processor writes the memory address, the count and the direction, starts it, and hears nothing more until a single completion interrupt arrives, so it is involved twice for the whole block whatever the block size happens to be.

The ratio, which is the thing to carry: 1024 divided by 2 is 512, so on this block direct memory access touches the processor 512 times less often than programmed transfer does. The gap widens with every byte added, because one count grows with the block and the other never moves off two.

What a different word size would do: the word size, not the method, sets the first two counts. On a machine with a 2 byte word the same block is 2048 words, so programmed transfer would cost 2048 involvements and interrupt-driven transfer 2049, while direct memory access would still cost two.

Method	Who moves the data	When the processor is involved	Involvements
Programmed input and output	the processor	once for every word, and it polls between words	1024
Interrupt-driven input and output	the processor	once when the interrupt arrives, then once per word	1025
Direct memory access	the controller	at set-up, and again at completion	2

Table 3.7: *The same 4096 byte block moved three ways, and the processor involvements each way costs.*

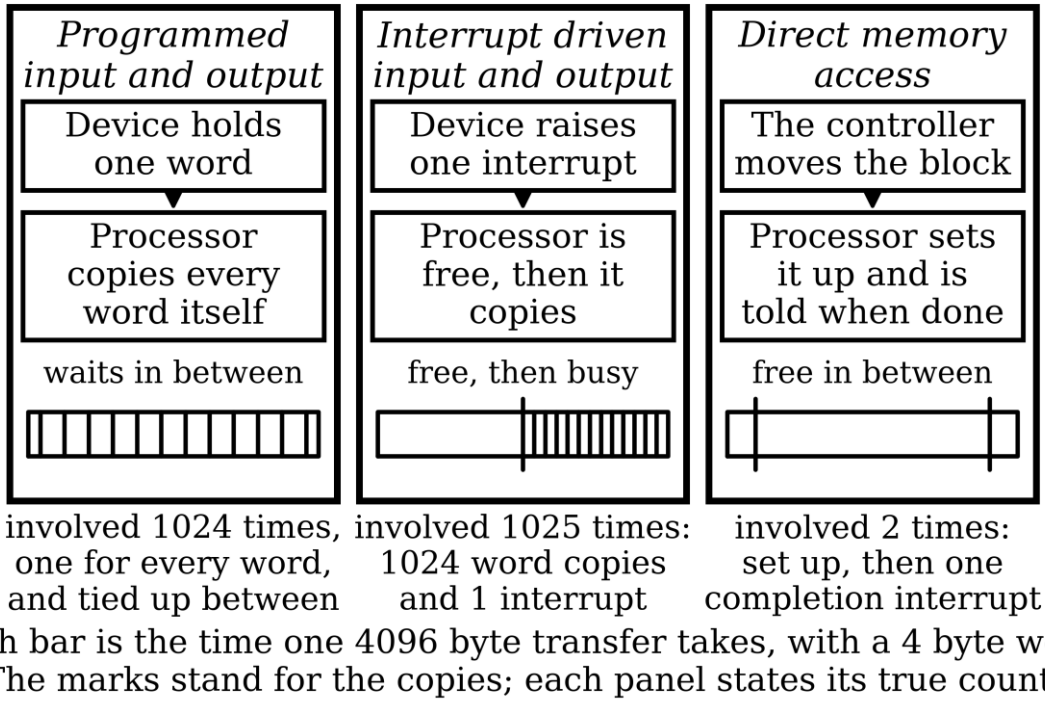


Figure 3.7: *The three ways of moving one block, where each mark on a bar is a moment when the processor is in the work.*

Interrupt-driven transfer buys back the waiting, and only direct memory access buys back the copying.

The Master Key: *Programmed transfer costs an involvement for every word, interrupts remove the waiting, and only direct memory access removes the copying.*

REAL Question 24

A message went round the staff group at Malampaka one evening, three lines long, forwarded from somebody's notes. By morning it had reached four schools. Two of its three lines are wrong, and the one that is right looks the dullest of the three.

Three statements about the difference between a mode switch and a context switch are given. Identify the correct one, explain the fault in each of the others, and correct them.

Statement A. A mode switch changes the privilege the processor is running at while the same process stays on it; a context switch takes one process off the processor and puts a different one on.

Statement B. Every system call causes a context switch, because the kernel takes the processor over.

Statement C. A mode switch is the more expensive of the two, because changing privilege means saving the whole of the program's memory.

Model Answer

Statement A is the correct one: a mode switch changes only the privilege the processor is running at and leaves the same process on it, while a context switch replaces the running process with a different one.

Why A is right: when a program makes a system call the privilege rises and the kernel then works on behalf of the very process that asked. That process is still the one on the processor and the registers are still its own. A context switch is a different event, in which one process's state is saved and another's is loaded, and a later section takes it further.

The fault in B: it treats entering the kernel and changing process as one act. A system call causes a mode switch. It causes a context switch only if the kernel then has to make the process wait, for a disk read for instance, or if the scheduler decides that some other process should run next.

B corrected: every system call causes a mode switch, and a system call causes a context switch only when the process cannot continue immediately or is preempted. A call that merely asks the system what the time is returns to the same process, on the same processor, with the same registers.

The fault in C: the costs are the wrong way round, and no memory is copied by either event. A mode switch saves a small amount of processor state and is over in a few hundred instructions; a context switch saves and reloads a whole register set and leaves the caches full of the wrong data.

C corrected: a context switch is the more expensive of the two, because it changes whose work the processor is doing and every trace of the previous process has to be put away and restored later. Neither switch saves memory contents; memory stays where it is and is reached through the tables the kernel keeps.

The question that settles any such claim: ask which process is on the processor once the event is over. If it is the same one, that was a mode switch. If it is a different one, that was a context switch. Every context switch contains a mode switch, because the kernel must be entered to perform it, but the reverse does not hold.

The line to distrust in any forwarded note is the one that makes entering the kernel and changing process the same thing.

The Master Key: *A mode switch changes what the processor may do, and a context switch changes whose work it is doing.*

HOT Question 25

At the Vikindu depot they gave every attendant a key to the main valve, and the queue moved twice as fast. It moved that way for eleven weeks. Then one attendant turned the wrong valve, and nothing moved at all.

Moving a service from user space into the kernel makes it faster. Argue whether operating systems should therefore run more services in kernel mode, and reach a judgement.

Model Answer

Move a service into the kernel only when it is small, settled and crossed so often that the crossings dominate what it costs, and leave it in user mode otherwise, because the speed is bought with containment: a fault in kernel mode takes the machine down, while a fault in user mode takes one process down.

Where the speed actually comes from: a service in user space is reached by a system call, and every call costs a mode switch, a check of the arguments and a copy of the data across the boundary. Inside the kernel the service reads the structures directly and none of that is paid. The saving is real, and it is paid back on every single call.

The case for moving services inward: a service crossed thousands of times a second pays that toll thousands of times a second. Graphics, networking and file handling have all been moved inward on real systems for exactly this reason, and on a machine whose whole job is to serve files the difference is measurable rather than theoretical.

The case against, which is about failure and not about speed: kernel code has no containment around it. A fault in user mode ends one process, the kernel reclaims its memory and the machine carries on serving everybody else. The same fault in kernel mode can overwrite any memory on the machine, and the usual outcome is that everything stops together.

The second cost, which is trust: everything inside the kernel is trusted absolutely, so each service moved in enlarges the body of code that has to be correct. Device drivers are the standing example: code written elsewhere, shipped with a piece of hardware, running with complete privilege on a machine whose owner has never read it.

The questions to ask of one service: how often is the boundary crossed, how large and how stable is the code, and what does its failure cost if it runs where nothing can contain it. A small, settled, heavily crossed service passes all three. A large service that changes with every release passes none of them.

The verdict: reject the general claim. Move a service inward where the crossings dominate its cost and the code is small and stable enough to be trusted with everything on the machine, and keep it in user mode where it is large, new, written by somebody else, or able to fail without ending the machine's usefulness. Which of these choices a system makes for its whole kernel is the subject of the next section.

Speed here is bought with the size of the failure, and that is the currency the argument has to be settled in.

The Master Key: *A fault in user mode costs one process, and the same fault in kernel mode costs the machine.*

HOT Question 26

The conveyor at Lamadi takes nine minutes to set up and thread. For a full lorry that is nothing. The morning it was started to move a single crate, two men stood watching it for the whole nine minutes.

Direct memory access frees the processor from copying data. Argue whether it is therefore always the better choice, and reach a judgement.

The same machine as before: a 4 byte word, and one processor involvement moves one word under programmed transfer.

Programming the controller and taking the completion interrupt cost the processor about as much work as copying 64 words.

Model Answer

Direct memory access is the better choice above a break-even size and the worse choice below it, and with a set-up worth about 64 word copies that break-even is 256 bytes on this machine, so a 4096 byte block repays the set-up sixteen times over while a single keystroke never repays it at all.

The case for, put at full strength: on a 4096 byte block the processor is involved twice under direct memory access against 1024 times under programmed transfer. While the controller works the processor runs other programs, so the machine gets a second thing done inside the same interval, and the saving grows with every byte the block gains.

The set-up cost is fixed, and small transfers are where that matters: programming the controller means writing the memory address, the byte count and the direction, starting the device, and later servicing the completion interrupt. If all that costs what copying 64 words costs, the break-even is 64 words, which is 256 bytes. Below that size, copying the words directly is the cheaper thing to do.

The two comparisons that show the flip: the 4096 byte block is 1024 words, sixteen times the break-even, so direct memory access wins comfortably and would still win if set-up cost twice as much. A single character from a keyboard is one word: programmed transfer costs one involvement, and direct memory access would cost the equivalent of 64.

Bus contention, which the counting hides: the controller and the processor share one path to memory. While a block is moving the controller takes memory cycles for itself, so the processor, free of the copying, still runs more slowly on anything that touches memory. Part of what was saved is handed back that way.

The costs that only direct memory access has: the controller works with physical addresses, so the kernel must hold the buffer still in memory for the length of the transfer, and data written into memory behind the processor can leave a stale copy in the cache that has to be discarded. Programmed transfer has neither problem.

The verdict: reject the claim that it is always better. Prefer direct memory access for any transfer above the break-even, which is 256 bytes on this machine, and for anything block-structured such as a disk or a network interface. Prefer programmed transfer below it, for a keystroke, a status byte or one register read, where the set-up costs more than the work it saves.

A mechanism that saves per byte is worth having only when there are bytes enough for it to save on.

The Master Key: *Direct memory access repays its set-up above about 256 bytes and wastes it below.*

Operating system architectures

This section lays out how the inside of an operating system can be organised. It restores the layered model, names a real system against every architecture, and gives advantages and disadvantages for each rather than for one. It also puts the process scheduler where it belongs in a microkernel, in kernel space beside inter-process communication and basic memory management.

BINDER Question 27

At Ngudu the district office keeps land records, birth registers and trading licences in one back room, so a clerk fetches any file in seconds. One leaking roof would soak them all at once. Most operating systems are built that way.

Explain the monolithic architecture. State its advantages and its disadvantages, and name a real operating system built this way.

Model Answer

A monolithic operating system places every kernel service in a single program that runs in kernel space and shares one address space, so any service can call any other directly.

What sits inside the kernel: The process scheduler, the memory manager, the file system, the device drivers, inter-process communication and the network stack all live in kernel space together, and applications in user space reach them through one interface rather than talking to each service separately.

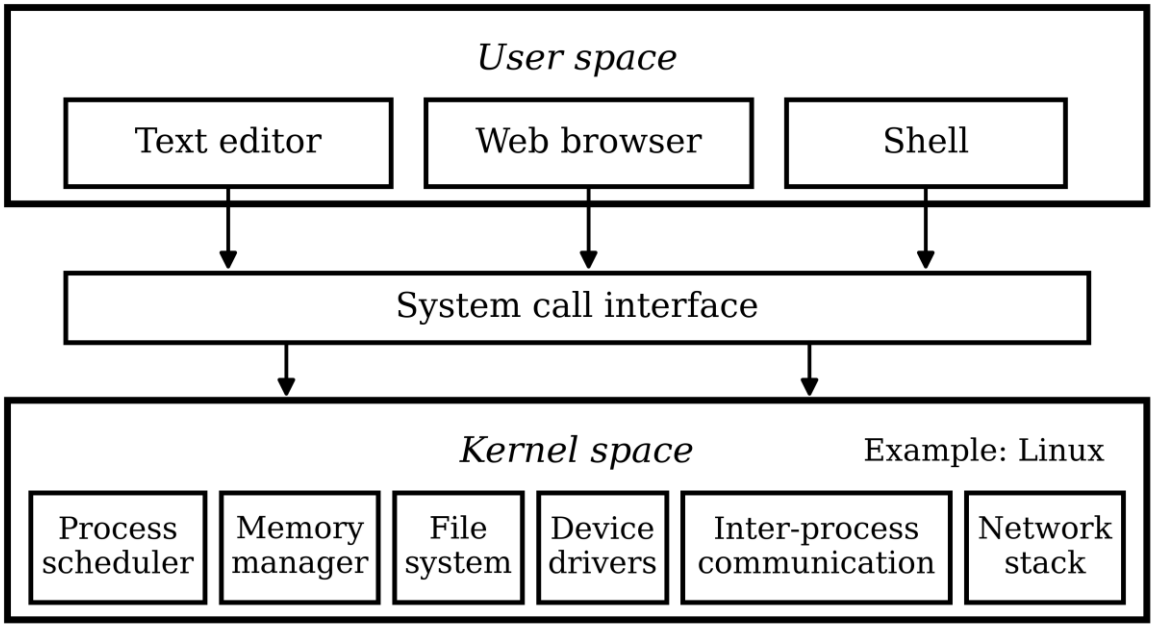


Figure 3.8: *A monolithic kernel. Every service sits in one address space in kernel space, and applications reach all of them through a single interface.*

Why it is fast: When the file system needs the disk driver it makes an ordinary procedure call inside the same address space. Nothing is copied and no boundary is crossed, so a monolithic kernel gives the highest throughput of any of these designs.

A second advantage: Because everything shares one address space, the file system and the memory manager use the same cached disk blocks without copying them anywhere. A microkernel has to hand those blocks across a boundary, so the shared cache is a monolithic advantage rather than an accident.

What it costs: There is nothing between the parts, so a defective driver can overwrite the scheduler's data and bring the whole machine down. The trusted code is the entire kernel, and one changed line means rebuilding and restarting all of it.

The modular refinement: Linux is monolithic and modular. A driver is compiled as a loadable module and linked into the running kernel when the device appears, so the machine gains hardware without a rebuild. The module still runs in kernel space with the same reach as everything else there, so modularity buys flexibility, not isolation.

Monolithic design trades isolation for speed, and it remains the most widely used shape of kernel.

The Master Key: *One address space, every service inside, nothing between them: quick to call and total in failure.*

BINDER Question 28

The borehole at Sumve fills a tank, the tank feeds the ward taps, and the taps fill the basins. No nurse speaks to the pump attendant. A whole operating system was once built on that rule alone, and the discipline outlived the machine.

Explain the layered architecture. State the rule that governs what a layer may call, its advantages and its disadvantages, and name a real system.

Model Answer

A layered operating system divides the kernel into ordered levels, and the rule that governs it is strict: a layer may call only the layer immediately below it, never a layer above it and never one further down.

The stack from the bottom up: Hardware sits at the bottom. Above it come processor allocation, then memory management, then operator communication, then input and output management, and user programs on top. Five numbered layers stand above the hardware, and each has exactly one legal call downward, which is five call paths in all.

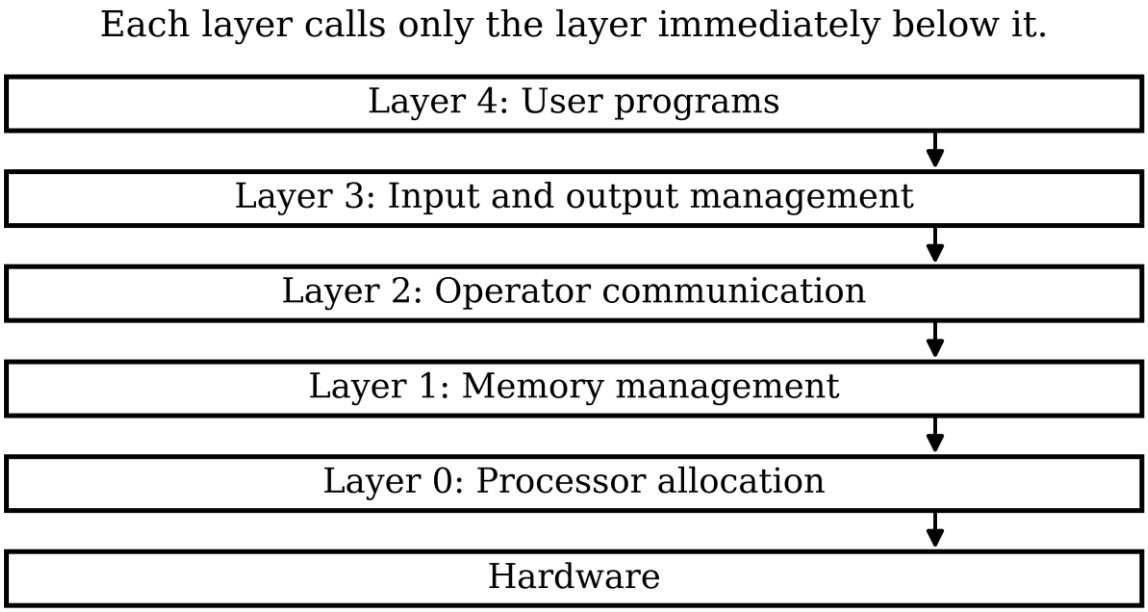


Figure 3.9: *The layered architecture. Each layer calls only the layer immediately below it, from the hardware at the bottom to user programs at the top.*

Why the rule pays: A layer can be checked as soon as the layers beneath it are known to work, because it depends on nothing else. When something fails, the fault is in the layer that broke its contract, and the search stops there instead of running across the whole kernel.

The price of the discipline: A request from a user program reaches the disk only by descending every layer between, and each crossing costs a call and often a copy. Deciding what belongs on which level is harder, because a memory manager that must read from disk and a disk driver that must be given memory cannot both be the lower one.

Real systems: THE, built at Eindhoven, was the first, and its levels were ordered so that each could be proved correct once the level below it was. MULTICS carried the same idea into hardware as a set of protection rings, with the innermost ring holding the most trusted code.

Layering buys checkable structure and pays for it in speed, which is why its rule outlived its kernels.

The Master Key: *A layer calls only the layer immediately below it, and that single restriction is the whole architecture.*

BINDER Question 29

At the Malya livestock yard the clerk in the booth does three things: writes the ticket, weighs the beast, takes the fee. Every trader stands outside and passes a note in. What stays inside the booth decides what survives a failure.

Explain the microkernel architecture. State precisely what remains in kernel space and what moves out, give its advantages and disadvantages, and name a real system.

Model Answer

A microkernel keeps in kernel space only what cannot work anywhere else, which is the process scheduler, inter-process communication and basic memory management, and moves every other service into user space as an ordinary server process.

Why those three stay: The process scheduler must be kernel-resident, because the code that takes the processor away from a running process cannot itself be one of the processes queueing for it. Inter-process communication stays because it is how every other service is reached, and basic memory management stays because it builds the address spaces that keep the servers apart.

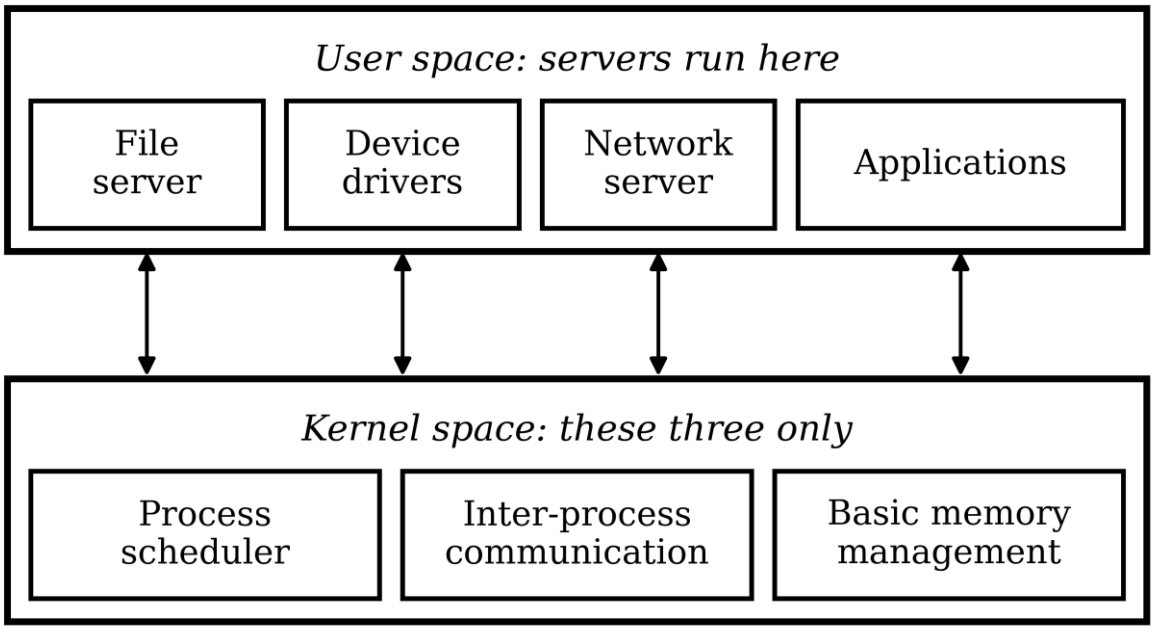


Figure 3.10: *The microkernel. The process scheduler, inter-process communication and basic memory management stay in kernel space; the file server, the drivers and the network server run in user space.*

What moves out: File systems, device drivers and networking become user space servers, each in its own address space. A request from an application is a message that goes through the kernel to the right server, and the reply comes back the same way. A common error is a drawing that leaves the scheduler outside with them.

The advantages: The trusted code shrinks to a body small enough to review line by line and, in some systems, to prove correct. A failed driver is a failed process, so it can be restarted while the machine keeps running, and a new service is started rather than compiled into the kernel.

The disadvantages: Every call that used to be a procedure call is now a message with two boundary crossings, and a request passing through several servers pays that cost each time, so throughput falls. Faults are harder to follow, because one operation is now spread across several processes.

Systems built this way: MINIX 3 runs its device drivers as user space processes and restarts one that dies without disturbing the rest. QNX is used where a machine must not stop, in vehicles and industrial control, and its kernel is small enough to be certified for safety work.

The microkernel buys containment with messages, and the bill is paid in throughput.

The Master Key: *Scheduler, communication and basic memory management stay in; file systems, drivers and networking go out.*

REAL Question 30

The bus stand at Nyambiti offers several routes to the same lakeside town: fast and rough, slow and certain, one that changes vehicles halfway, one that mixes both. Nobody chooses without knowing what each costs. Kernels are chosen on the same kind of grid.

Compare the monolithic, layered, microkernel and hybrid architectures on performance, reliability, ease of extension, size of the trusted code and portability.

A server at Nyambiti is measured under two builds of the same system.

The monolithic build runs 21 million lines of code in kernel space.

Device drivers, file systems and networking account for 18 million of those lines.

The microkernel build runs those three as user space servers and changes nothing else.

Model Answer

The four architectures differ on the same five measures, and every difference follows from one thing: where the boundary between kernel space and user space is drawn, and how often a running system has to cross it.

Criterion	Monolithic	Layered	Microkernel	Hybrid
Performance	Highest	Falls with each level crossed	Lowest, a message per crossing	Near monolithic
Reliability	One fault stops the machine	Fault localised to a layer	Failed server restarts alone	Depends where the fault is
Ease of extension	Load a module	Insert at the right level	Start a server	Either, service by service
Size of trusted code	The whole kernel	The whole stack	Smallest	Large again
Portability	Low	High, machine code at the bottom	High, small kernel to port	Moderate

Table 3.8: *The four architectures against five measures. Each row is one measure and each column is one architecture.*

Performance: Monolithic is fastest, because a service calls another as a procedure inside one address space. Layered pays for every level a request descends. Microkernel pays a message for each service crossing, which is the largest cost of the four. Hybrid pays it only where a service was left outside.

Trusted code on this server: Take the monolithic build first. All 21 million lines run in kernel space, so every one of them is trusted, and a fault in any line can reach any other part of the kernel.

What the move leaves behind: Now move the drivers, the file systems and the networking out. Subtracting 18 million from 21 million leaves 3 million lines in kernel space, and the other 18 million become user space servers the kernel no longer has to trust.

Reading the fraction: Dividing 18 by 21 gives 0.857, so six lines in every seven leave the trusted code. That ratio is the whole of the microkernel argument on reliability, and it is why a failed driver need not take the machine with it.

Ease of extension: A modular monolith loads a module into the running kernel. A microkernel starts another server and needs no kernel change at all. A layered system is hardest, because the new code must go on the right level and everything above it must be checked again.

Portability, and the hybrids: Layered and microkernel systems concentrate machine-dependent code at the bottom, so a port touches little, while a monolithic kernel spreads those assumptions widely. Windows NT and macOS are hybrids, with a microkernel core and the file systems, drivers and network stack pulled back into kernel space for speed.

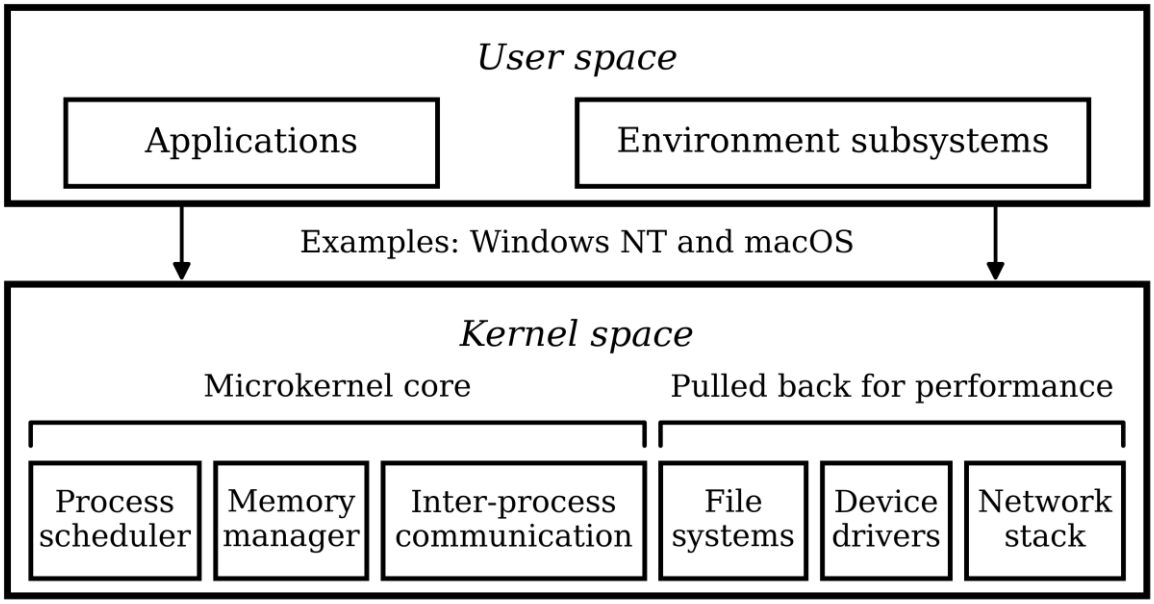


Figure 3.11: *A hybrid kernel. A microkernel core keeps the scheduler, the memory manager and inter-process communication, and the costly services are pulled back beside them.*

The four designs are one trade made at different points: the more a system moves out of kernel space, the less it must trust and the more it must copy.

The Master Key: *Every difference between these architectures follows from where the boundary sits and how often it is crossed.*

REAL Question 31

A printed card is propped on the service counter of the computer shop at Ushirombo, listing what a small kernel keeps for itself. Two of its three lines are wrong, and the shop has been repeating them to customers all season.

Three statements about what a microkernel keeps in kernel space are printed on the card. Identify the correct one, explain the fault in each of the others, and correct them.

The card on the counter reads:

Statement A: in a microkernel the process scheduler runs in user space, beside the file server and the drivers.

Statement B: in a microkernel the process scheduler, inter-process communication and basic memory management stay in kernel space.

Statement C: in a microkernel the device drivers stay in the kernel, because reaching hardware needs privilege.

Model Answer

Statement B is the correct one, and the other two fail in opposite directions: A empties kernel space of the one thing that cannot leave it, and C keeps inside the very services a microkernel exists to move out.

Why B is right: Those three services are the ones a microkernel cannot export. Scheduling shares out the processor, communication carries every request, and basic memory management builds the separate address spaces the servers live in. Remove any one of them and the rest of the system has nothing to run on.

The fault in A: A scheduler in user space would be one of the processes waiting to run, and a process cannot hand the processor to itself or take it from another. Students are often taught, wrongly, that the scheduler sits outside with the servers, and a drawing that shows it there describes a system that could never start.

Correcting A: The corrected line reads: in a microkernel the process scheduler stays in kernel space with inter-process communication and basic memory management, while the file server and the drivers run outside it as user space processes.

Component	Where it runs	Why it sits there
Process scheduler	Kernel space	Only kernel-resident code can take the processor from a running process
Inter-process communication	Kernel space	Every request to every server travels through it
Basic memory management	Kernel space	It builds the address spaces that keep the servers apart
File system	User space	A fault stops one server and not the machine
Device drivers	User space	The largest and least trusted body of code, and restartable
Network stack	User space	Replaceable without rebuilding the kernel

Table 3.9: *What a microkernel keeps in kernel space and what it moves out, with the reason for each.*

The fault in C: Drivers are exactly what a microkernel moves out, and they are moved because they are the largest and least trustworthy part of a kernel. A driver does not need to be inside the kernel to reach its device; it needs the kernel to give it access to that device and to nothing else.

Correcting C: The corrected line reads: in a microkernel the device drivers run in user space as separate processes, each granted access to its own device, so a driver that fails is restarted without stopping the machine. MINIX 3 does exactly this.

The test to apply: Ask of any service whether the kernel could still run without it. Scheduling, communication and address spaces fail that test, so they stay. A file system, a driver and a network stack all pass it, so a microkernel puts them outside where a fault is survivable.

Two of the three lines invert the design they describe, and the corrected pair now agrees with the drawing.

The Master Key: *What stays in a microkernel is what the kernel cannot run without, and everything else is a process.*

HOT Question 32

The school at Bulyanhulu keeps its records on one server. When it stopped last term, the office copied the whole register by hand for the rest of the day. The next machine will be chosen on what happens when a driver fails.

A school is choosing between an operating system with a monolithic kernel and one with a microkernel for a server that must stay up. Argue both and reach a judgement.

The school at Bulyanhulu runs one server for records, printing and the network.

On the monolithic build a driver fault stops the machine, and a reboot restores service in 180 seconds.

On the microkernel build the same fault stops one user space server, which is restarted in 2 seconds.

The site logs six driver faults a year.

The microkernel build completes 12 per cent less work per hour, and the server runs at 30 per cent of its capacity.

Model Answer

Choose the microkernel here, because this server is nowhere near its capacity ceiling and the school pays for a stoppage in lost office hours rather than in lost throughput.

The case for the monolithic kernel: It is the faster of the two on identical hardware, its drivers are the ones the manufacturers actually check, and one address space means one thing to install and one thing to update. For most machines that is the end of the argument.

The case for the microkernel: A driver here is a process, so a fault kills the driver and not the machine. Service returns in 2 seconds instead of 180, and the work in progress on the other servers survives, because nothing else was in the failing address space.

The downtime over a year: Six faults on the monolithic build cost 6 times 180, which is 1080 seconds of stopped service. The same six faults on the microkernel build cost 6 times 2, which is 12 seconds.

Reading the difference: The gap is 1080 minus 12, which is 1068 seconds, or 17.8 minutes of office time a year. Set against a whole school year that is small, and it is the honest size of the reliability advantage on this site.

What the throughput costs: The server runs at 30 per cent of capacity. Losing 12 per cent of its work per hour raises that to 0.30 divided by 0.88, which is 0.34, so the machine works at 34 per cent of capacity and is still nowhere near its limit.

What neither number covers: A machine that stops in the middle of writing to disk can leave a file half written, and repairing that costs more than the reboot did. Containment is worth most where the work in progress is expensive, which on a records server it is.

The judgement, and what would reverse it: Take the microkernel while the machine has spare capacity, because unused throughput is worth nothing and a stopped office is worth a great deal. Reverse the verdict the moment the server is worked near its ceiling, where 12 per cent is the difference between finishing the day's work and carrying it over.

The decision is not about which design is better but about which of the two costs this site actually pays.

The Master Key: *Weigh what a stoppage costs against what the throughput is worth, and let the spare capacity decide.*

HOT Question 33

At Nyakanazi the tarmac from the west meets the murram road north, and every lorry changes its habits at the junction: speed on one, care on the other. Kernel designers made the same swap, and the purists among them call it a surrender.

Hybrid kernels take from both designs. Argue whether this is sound engineering or an admission that neither pure design works, and reach a judgement.

Model Answer

Hybrid kernels are sound engineering, because the cost of a boundary is not the same at every crossing, and a design that places each service according to how often it is crossed is measuring rather than compromising.

The purist case at its strongest: A hybrid keeps the message interfaces of a microkernel but puts the file system, the drivers and the network stack back in kernel space. It pays for the interfaces and loses the one thing they were for, which was a small trusted code base. If messages were affordable the pure design would have won, and if they were not, the interfaces should have gone too.

What that case gets right: It is right about the trusted code. A hybrid trusts nearly as much code as a monolith does, and a defective driver inside macOS or Windows can still stop the machine. Anyone who claims a hybrid has the reliability of a microkernel is claiming something the design does not deliver.

Why the boundary is not uniform: Crossing a boundary once a minute to change a policy costs nothing measurable. Crossing it for every block a disk reads costs a great deal, because that crossing repeats with every block. Placing the disk path inside and leaving a rarely used service outside is arithmetic, not surrender.

The evidence from real systems: Windows NT began with more of itself outside kernel space and moved the window manager and graphics back in when the cost of the crossings showed. The XNU kernel under macOS joins a Mach core to a BSD kernel and keeps its drivers in kernel space. Both moved after measuring.

A hybrid is not a modular monolith: Modularity changes only when code is linked, and a loadable module still runs beside the rest of the kernel and shares everything with it. A hybrid keeps the message interfaces, so a service can be moved to the other side of the boundary when the measurement changes.

The judgement: Call it engineering when the placement follows a measurement and the interfaces are kept, so that a service can move again. Call it an admission of defeat when services are pulled back without measurement and the interfaces are abandoned, because that design has paid the price of a microkernel and kept the risk of a monolith.

The pure designs were arguments about where the boundary belongs, and the hybrids answered them one service at a time.

The Master Key: *A boundary is worth its cost where it is crossed rarely, and a hybrid is that judgement made service by service.*

HOT Question 34

In the records office at Sumve nobody knows which kernel the machine runs, and nobody needs to, until the printer dies at the busiest hour of the morning and takes the half typed list of names down with it.

Argue whether the architecture of an operating system makes any difference to the person using the machine, and reach a judgement.

Model Answer

Mostly it makes no difference at all, because the person at the keyboard sees applications, files and speed, and two machines of different architecture doing the same work feel the same. There are three places where it shows, and none of them happens on an ordinary day.

Why it is invisible in ordinary work: A program asks the operating system for the same services whatever shape the kernel has, and it gets them. Typing, printing, saving and opening a file behave identically, and the few per cent of speed that separates a tuned hybrid from a tuned monolith is below what anyone notices.

Recovery from a failed driver: This is where it shows first. On a monolithic kernel a defective printer driver takes the machine and the unsaved work with it. On a microkernel the driver is a process, so it is restarted, and the clerk sees a failed print job instead of a dead machine and a lost morning.

How quickly a new device is supported: This is the second place. A device whose driver must be compiled into the kernel waits for a kernel that contains it. A modular or microkernel system loads the module or starts the server while the machine keeps running, so a scanner bought in the morning works the same afternoon.

The third place, admitted honestly: Updates. Replacing a user space server does not stop the machine, while replacing a monolithic kernel does, so the architecture decides whether the office loses an afternoon to an update or never notices one happening.

What the user is doing	Does the architecture show	Why
Typing and saving a document	No	Every design offers the same services to the program
A printer driver fails	Yes	One design restarts a process, the other stops the machine
Plugging in a new device	Yes	A driver can be loaded or started, or must be built in
Installing an update	Sometimes	A user space server can be replaced while the machine runs

Table 3.10: *Where the architecture of a kernel reaches the person using the machine, and where it does not.*

What to ask instead: Nobody should choose a machine by the shape of its kernel, because the shape is not what is bought. The question that carries the same information is simpler, and it is what happens here when one part fails, and how soon this machine can run a device it has never met.

The judgement: The verdict is that architecture is invisible during ordinary work and decisive at the moments when work is lost or hardware is new, so judge a system by how it behaves at those moments and not by which diagram it matches.

Architecture is a question about bad days, and on a good day it answers nothing.

The Master Key: *The shape of a kernel is invisible until something fails or something new is plugged in.*

Processes, threads and the process control block

This section supplies what the common teaching leaves out entirely. It separates a program from a process and a process from a thread, draws the state model and its six transitions, opens the process control block, and puts a cost on a context switch.

BINDER Question 35

At the Lusahunga lorry stop one repair manual has lain on the shelf for six years. This morning three mechanics work from that single open copy, each on a different axle, each holding his own spanner. The book is not the work.

Distinguish a program, a process and a thread. State what several threads of one process share and what each keeps to itself.

Model Answer

A program is a passive file of instructions stored on disk, a process is that program in execution together with the memory and resources the operating system has given it, and a thread is a single flow of control inside a process, which is the unit the processor actually schedules.

The program is passive: A program is a file, and a file does nothing. The same program can lie unopened for years and can also back several processes at once: a clerk and a manager who each start the same weighing program on one machine are running two processes built from one file.

The process is active: A process is a program that has been admitted. It has an address space, a process control block, open files and a current state, and the operating system manages it. Two processes from one program have separate memory, so neither can read the other’s variables.

The thread is what runs: A thread is one path of execution through the code of a process. Every process begins with one thread and may create more, and each thread can be on the processor while the others are not. The process owns the resources; the thread does the running.

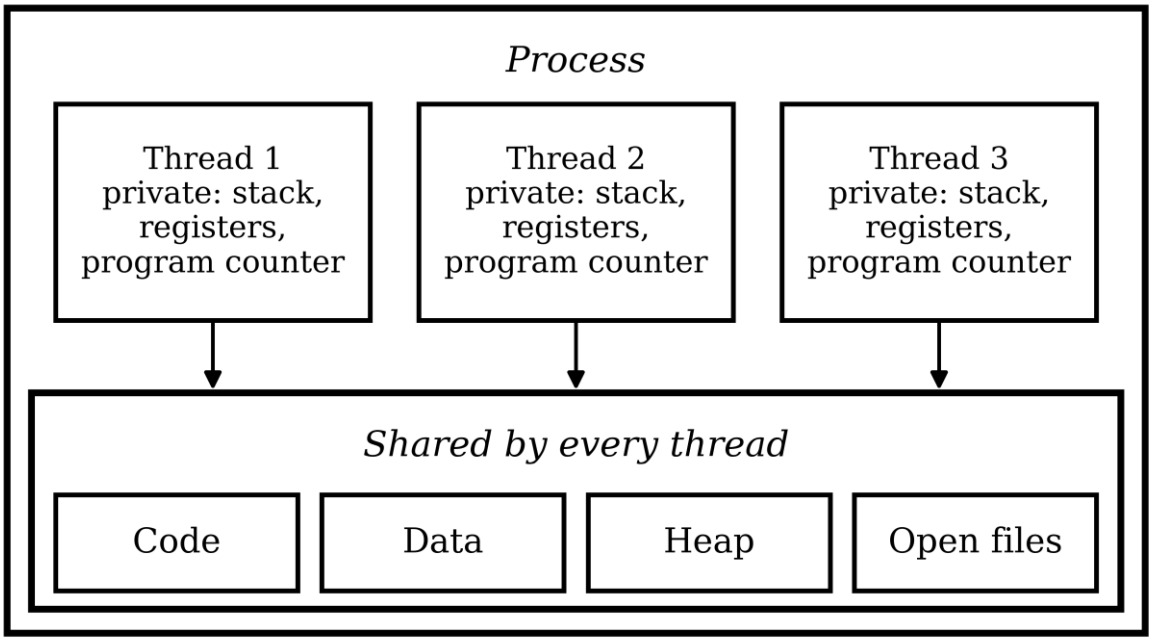


Figure 3.12: *One process holding three threads, showing what all of them reach and what each keeps.*

Item	Held where	Why it falls that way
Program code	Shared	Every thread runs instructions out of the same text.
Global data	Shared	The threads are cooperating on one body of data.
Heap	Shared	Memory taken at run time belongs to the process, not to a thread.
Open files	Shared	A file opened by one thread is readable by all of them.
Stack	Private	Each thread has its own chain of calls and its own local variables.
Registers	Private	Each thread holds its own working values.
Program counter	Private	Each thread is at a different instruction.

Table 3.11: *What the threads of one process share, and what each keeps to itself.*

Reading the division: The rule behind the table is short. Anything that says what the work is, is shared, and anything that says where a thread has reached is private. A stack, a set of registers and a program counter are exactly the things that differ between two threads at the same instant.

Name the level before you state the property, because a claim that is true of a thread is often false of the process that holds it.

The Master Key: *A program is a file, a process is that file running, and a thread is the part of it that the processor schedules.*

BINDER Question 36

The dispensary at Rulenge keeps four benches: outside the door, on the bench by the window, inside with the clinical officer, and along the wall waiting for a laboratory result. A patient may move between them, but not by every route.

Draw and explain the five-state process model. Name each transition and state what causes it.

Model Answer

A process occupies one of five states, new, ready, running, waiting or terminated, and only six transitions between them are legal.

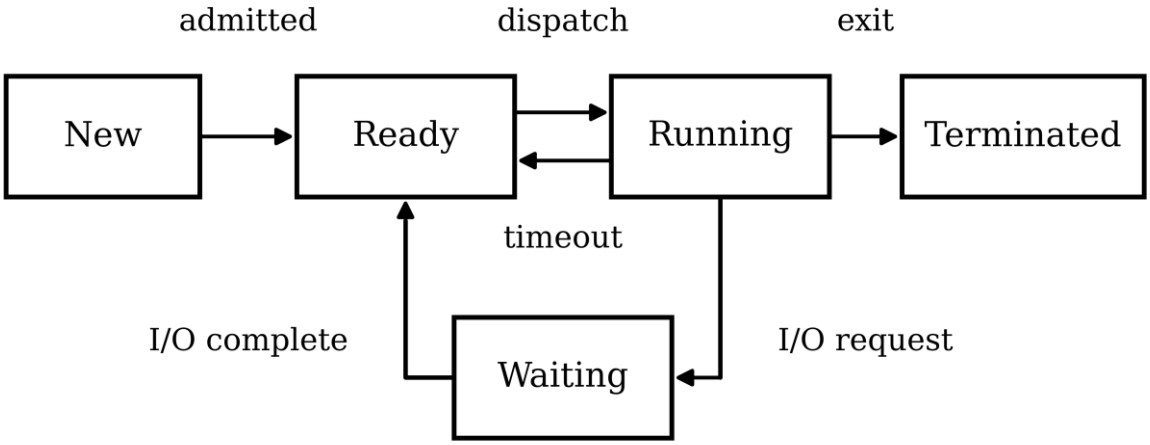


Figure 3.13: *The five-state process model, with the six transitions that are legal and no others.*

The five states: New is a process being created, before the operating system has admitted it. Ready means able to run and short of nothing but the processor. Running means on the processor now. Waiting means unable to run until some event happens. Terminated means finished, with its result not yet collected.

Admitted and dispatch: New to ready is admitted: the operating system has built the process control block and found memory, so the process joins the ready queue. Ready to running is dispatch: the process is chosen and given the processor. Dispatch is the only way into running.

Timeout and the request for input: Running to ready is timeout: the process has held the processor long enough, a timer interrupt returns control to the operating system, and the process goes back to the ready queue still able to run. Running to waiting is an I/O request: the process has asked for data and cannot continue until it arrives.

Completion and exit: Waiting to ready is I/O complete: the data has arrived, so the process becomes able to run again and rejoins the ready queue. Running to terminated is exit: the process has finished or has been killed, and its memory is released.

The two arrows that cannot exist: Ready to waiting is impossible, because a process that is not on the processor has executed nothing and so has requested nothing to wait for. Waiting to running is impossible, because the dispatcher takes only from the ready queue. Of the twenty ordered pairs of distinct states, fourteen must never be drawn.

Five states, six transitions, and every one of the six has a cause you can name.

The Master Key: *A process leaves the processor by timeout, by an I/O request or by exit, and it returns only through the ready queue.*

BINDER Question 37

At the Nyamirembe fish landing the clerk opens one ledger line for every boat: owner, crates brought in, ice taken, money owed. Lose that line and the boat is still on the beach, but nothing about it can be settled.

State what a process control block holds and explain why each item must be saved. Say where the operating system keeps it.

Model Answer

The process control block is the record the operating system keeps for one process, and it holds everything needed to stop that process at any instant and start it again as though nothing had happened.

Identity and state: The process identifier names the process, and every other table refers to it by that number. The state field says whether it is new, ready, running, waiting or terminated, which is how the operating system knows whether it is able to run.

The fields that let it resume: The program counter holds the address of the next instruction, so the process resumes instead of restarting. The registers hold half finished work, and that work is gone for ever if it is not copied out before another process is loaded.

What the process holds: Memory limits record the bounds this process may touch and are reinstated on every switch, so it cannot reach into another. The list of open files records each file and the position reached in it, so a read continues from where the process had got to.

How the process is treated: Accounting information records processor time used and the owner, which is how usage is charged and limits enforced. Scheduling information records the priority and the queues the process is on, and it travels with the process so that the choice of what runs next can be made.

Field	What it holds	Why it must be saved
Process identifier	The number that names this process	Every other table refers to the process by it.
Process state	New, ready, running, waiting or terminated	It says whether the process is able to run at all.
Program counter	The address of the next instruction	Without it the process would restart rather than resume.
CPU registers	Accumulators, index registers, the stack pointer	Half finished work lives in registers and is lost if not copied out.
Memory limits	The bounds of the memory this process may touch	They are reinstated so the process cannot reach another's memory.
List of open files	Each open file and the position reached in it	A read must continue from where the process had got to.
Accounting information	Processor time used, time limits, the owner	Usage is charged and limits are enforced against it.
Scheduling information	Priority and the queues the process is on	It is needed when the choice of what runs next is made.

Table 3.12: *The eight fields of a process control block and why each of them must survive a switch.*

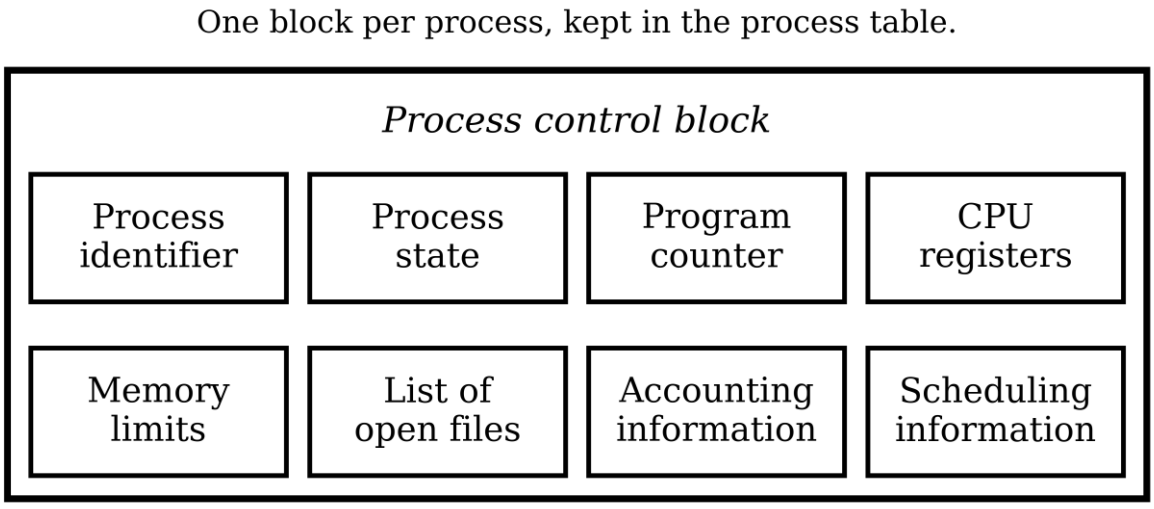


Figure 3.14: *One process control block, held with all the others in the process table.*

Where it is kept: Every block lives in kernel memory, in an array known as the process table, one entry for each process. It is never kept in the memory of the process it describes, which could then alter its own priority or its own memory limits.

Eight fields, in four pairs, and the process cannot be resumed if any one of them is missing.

The Master Key: *The process control block is what the operating system knows about a process, and a process that loses it cannot be resumed.*

BINDER Question 38

The mill at Bugene grinds maize until noon, then millet. Between the two the miller must empty the hopper, sweep the chute and reset the plates. For those eleven minutes the engine runs and nobody gets flour.

Explain what happens during a context switch. State what is saved, what is restored, and why the switch is pure overhead.

Model Answer

A context switch is the operation in which the operating system saves the whole state of the process now on the processor into its process control block, restores the saved state of another process, and lets that one run.

What is saved: The program counter and every processor register belonging to the outgoing process are copied into its process control block, and its state field is set to ready if it was timed out or to waiting if it asked for input. Nothing of it is left on the processor.

What is restored: The incoming process has its registers and its program counter loaded from its own block, and its memory limits are reinstated so that it addresses its own memory and no other. Its state field becomes running, and the processor continues from the instruction that process last reached.

Why it is pure overhead: During the switch the processor is fully occupied and neither process advances by a single instruction. The work is real, it is done by the operating system, and no part of it belongs to the job the user asked for. That is what overhead means here.

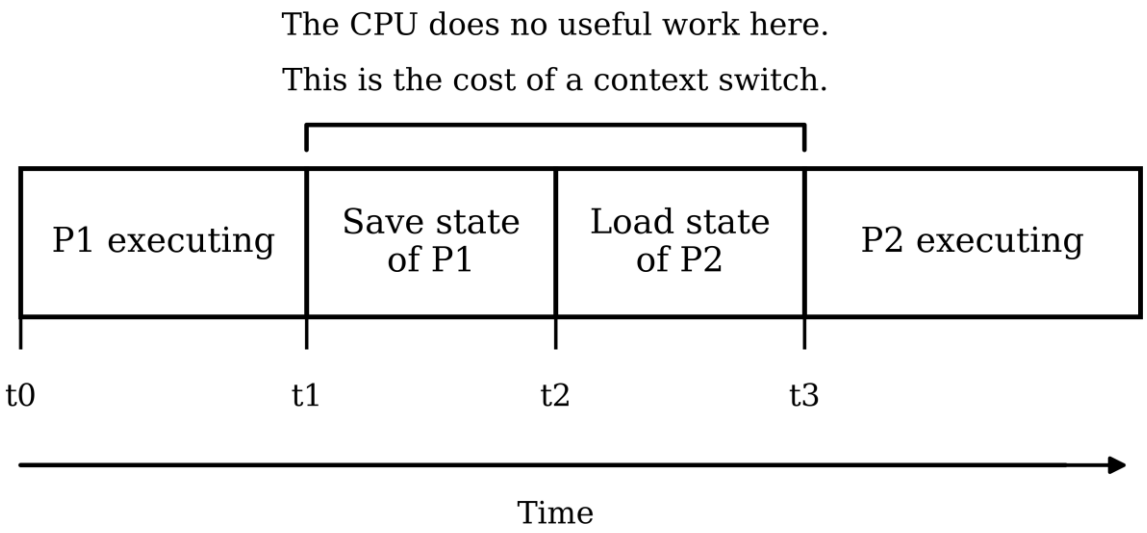


Figure 3.15: *A context switch on the timeline, with the stretch in which neither process advances.*

Reading the timeline: The bar for the first process ends, then two stretches follow in which the state of one is saved and the state of the other is loaded, and only after both does the second bar begin. The gap between the bars is the whole cost.

Putting a figure on it: Suppose one switch takes ten microseconds and a busy machine performs a thousand of them in a second. That is ten milliseconds of every second spent switching, one per cent of the machine given to work no user asked for.

The switch itself produces nothing, so an operating system is designed to make it short and to make it worth doing.

The Master Key: *A context switch saves one process into its block and loads another from its own, and no useful work is done while it happens.*

REAL Question 39

Coffee arrives at the Nkwenda buying post from six in the morning, and the clerk's computer weighs, records and prices every sack. One program does all three, and in an hour it stops and starts perhaps forty times.

The life of one process is described below, from launch to exit. Name every state it occupies and every transition it makes, in order, and say what caused each.

The clerk starts the weighing program, and the operating system builds its process control block and finds it memory.

Two other processes are ahead of it, so it does not reach the processor at once.

It runs, and the timer takes the processor back from it before the first sack is priced.

It runs again, and asks the disk for yesterday's price list.

The disk delivers the list, but the processor is busy with another process.

It runs a third time, prints the totals for the day and finishes.

Model Answer

The process occupies all five states and makes eight transitions, and the two that look alike, running to ready and running to waiting, are the pair this life is built to separate.

New, then admitted: The program becomes a process the moment the operating system starts building for it, and it is new while the process control block is being filled and memory found. When that is done it is admitted, which is the transition into the ready state.

Ready, then dispatch: In the ready state it is able to run and is short of nothing but the processor, which two other processes are using first. When it is chosen the transition is dispatch, and it becomes running.

The timeout: It is still computing when the timer interrupt arrives, so the operating system takes the processor back. That transition is running to ready, and the important word is ready: nothing is wrong with the process, it needs no event, and it would use the processor this instant if it were handed back.

The request for input: Dispatched a second time, it asks the disk for the price list. That transition is running to waiting, and here the process could not use the processor even if it were given it, because the data it needs has not arrived. This is what separates waiting from ready.

Back through the ready queue: When the disk delivers, the transition is I/O complete and it carries the process to ready, not to running. The processor is busy with another process, and an arrival from waiting has no claim on it, so the process joins the ready queue and is dispatched a third time.

Exit, and the count: It prints the totals and finishes, which is running to terminated by exit. The whole life is one admission, three dispatches, one timeout, one I/O request, one I/O complete and one exit, eight transitions in all, and the ready state is entered three times.

Every transition in that list has a cause, and no state is entered except through one of them.

The Master Key: *A process that was timed out is ready and a process that has blocked is waiting, and only one of the two can be handed the processor.*

REAL Question 40

In the logbook beside the estate pump at Kanyigo, a trainee has copied the state model in pencil, seven arrows in all. Two of them describe journeys no process has ever made, and one journey it makes daily is absent.

The logbook page below lists the seven arrows a trainee has drawn on a process state diagram. Identify every arrow that cannot happen, explain why each is impossible, and give the correct set of transitions.

New to ready, labelled admitted.

Ready to running, labelled dispatch.

Ready to waiting, labelled the process waits for input.

Waiting to running, labelled the input has arrived.

Running to waiting, labelled input requested.

Waiting to ready, labelled input complete.

Running to terminated, labelled exit.

Model Answer

Two of the seven arrows cannot happen, ready to waiting and waiting to running, and one transition that happens constantly, running to ready, is missing altogether.

Ready to waiting cannot happen: A process in the ready state is not on the processor and has therefore executed no instruction since it left it. It has issued no request, so there is nothing for it to wait for. Only a running process can ask for input, and only a running process can block.

The label gives the error away: The trainee has written that the process waits for input, which describes something a running process has just done. The mistake underneath is treating waiting as a general queue for anything that is not running, when it means one thing only: held up by an event that has not yet happened.

Waiting to running cannot happen: When the input arrives the process becomes able to run, which is not the same as being given the processor. The dispatcher takes only from the ready queue, and the processor may well be held by another process at that moment, so a completed operation returns the process to ready.

The arrow that is missing: Running to ready, the timeout, is not on the page. Without it a process could leave the processor only by asking for input or by finishing, so on this diagram one long calculation would hold the machine until it ended. The timer takes the processor back, and this arrow is how.

The correct set: Remove the two impossible arrows, keep the five that are right and add the missing one, and the count is six: new to ready by admission, ready to running by dispatch, running to ready by timeout, running to waiting by an I/O request, waiting to ready when that I/O completes, and running to terminated by exit.

A quick way to check any such drawing: Count the arrows leaving each state. Running has three, ready has one, waiting has one, new has one and terminated has none, which is six. Any state other than running with more than one arrow leaving it is wrong on the face of it.

The corrected drawing carries six arrows, one fewer than the trainee made, and not one of them is impossible.

The Master Key: *Only a running process can start waiting, and only a ready process can start running.*

REAL Question 41

At the Muganza ginnery two programs share one computer. One writes weighbridge readings all morning, two hundred megabytes of them; the other must summarise the lot before the lorries leave. How the second gets the first's numbers decides everything.

Explain the four mechanisms by which processes communicate. Choose one for the situation below and justify the choice against the alternatives.

Two processes run on one machine at the ginnery.

The first records weighbridge readings, about two hundred megabytes an hour.

The second summarises them and must see each reading within a second of it being written.

Both were written by the same team and run under the same account.

Model Answer

Processes communicate through pipes, shared memory, message passing and signals, and for this situation the right choice is shared memory, bought at the price of coordination that the mechanism itself does not provide.

Pipes: A pipe is a one way stream of bytes between two processes, usually a parent and its child. It is simple and needs no agreement beyond the order of the bytes, but every byte is copied into the kernel by the writer and out of it again by the reader.

Shared memory: A region of memory is mapped into the address space of both processes, so a write by one is visible to the other with no copying and no call into the operating system once the region is set up. It is the fastest mechanism there is, and the least protected.

Message passing: The processes exchange discrete messages through send and receive calls handled by the operating system. Each message is copied, which costs, but the boundaries of a message are kept and the two processes need share no memory, so the same code works across a network.

Signals: A signal is a notification and nothing more. One process tells another that something has happened, and the receiver runs a handler for it. A signal carries no data beyond its own number, so it can announce that a file is ready but cannot deliver the file.

Why shared memory wins here: Two hundred megabytes an hour through a pipe is four hundred megabytes of copying, into the kernel and out of it again, and message passing copies each reading in the same way. Shared memory copies nothing, and the reader sees a value the moment it is written.

What the choice costs: Shared memory hands over a region and nothing else. The two processes must agree between themselves when a reading is complete, or the summariser will read a half written record, and they must agree on where the newest reading lies. A signal can carry that announcement; the agreement itself has to be built.

The mechanism that moves the data fastest is also the one that leaves the most for the programmer to arrange.

The Master Key: *Pipes and messages copy through the operating system, shared memory does not copy at all, and signals carry news rather than data.*

HOT Question 42

Three fitters at the Rulenge mission workshop share one bench, one vice and one tray of tools. Bringing in a fourth takes a minute and nothing has to be bought. Then one of them knocks the tray into the pit.

Threads are cheaper to create and to switch between than processes. Argue whether a program should therefore always use threads, and reach a judgement.

Model Answer

No; use threads where the parts of one job must work on the same data and trust one another, and separate processes where a failure in one part must not be able to destroy the others, because what threads give up for their cheapness is isolation.

The saving is real: Creating a process means building an address space, a process control block and a table of open files. Creating a thread means a stack, a set of registers and a small structure inside a process that already exists, and a switch between two threads of one process leaves the memory mapping untouched.

What that is worth on a busy machine: Suppose a process costs five hundred microseconds to create and a thread fifty, and a server starts a worker for each of two thousand requests. The processes cost one second of the machine and the threads a tenth of a second, a ratio of ten to one before any work is done.

Sharing is the other half of the saving: Threads of one process share the heap and the open files, so passing a megabyte between them costs a pointer. Two processes must set up shared memory or copy the data through the operating system, and that setting up is code somebody has to write and get right.

What is given up: There is one address space, and the operating system protects memory between processes, not between threads. A thread that writes past the end of an array corrupts data another thread is using, and a fault serious enough to terminate the process terminates every thread in it.

Where that decides the design: A browser renders pages written by strangers, so each tab is given a process of its own and a page that crashes takes down one tab rather than the browser. A media player decoding one file into one window has no such boundary to defend, and uses threads throughout.

The cost that arrives later: Shared data has to be coordinated, and a fault of that kind shows itself only under some orderings, on one machine, once a week. A defect of that shape can cost far more than the second of creation time the threads saved.

The judgement: Use threads inside a trust boundary, where the parts of one job share data, are written by one team and would fail together anyway. Use processes across a trust boundary, where code of doubtful origin runs, or where the service must survive the failure of any one part.

Threads are cheaper because they share, and everything they share is something that can be damaged.

The Master Key: *Threads buy speed with shared memory, and isolation is what that memory costs.*

HOT Question 43

In the transport office at Lusahunga a trip sheet stays in the tray for weeks after the lorry has come home, because only the manager may sign it off. Another lorry's manager has left the firm entirely.

A zombie process holds an entry in the process table after it has finished. Argue whether zombie and orphan processes represent a failure of the operating system or of the programmer, and reach a judgement.

Model Answer

Neither is a failure of the operating system: an orphan is a case the system is designed to handle and does handle, while a zombie is the visible trace of a parent that has not done its part.

What a zombie is: A zombie is a process that has finished and released its memory but still holds its entry in the process table, because that entry carries its exit status and nobody has collected it. It uses no processor time and no memory; it uses an entry.

What an orphan is: An orphan is a process still running whose parent has terminated first. The operating system reassigns it to the first process started at boot, which becomes its new parent and collects it when it ends. Nothing is lost and nothing accumulates.

Why the entry is kept at all: The exit status is a result and the parent is entitled to it, because it says whether the child succeeded and a shell or a build tool acts on it. If the system discarded the status the moment the child ended, a parent that asked a moment later would find nothing.

Why a zombie is the programmer's: A parent that creates children and never collects their status leaves one entry behind for each, and the process table is a fixed size. Suppose it holds thirty thousand entries and a server leaks ten a second: it is full in fifty minutes, and then nothing new can start at all.

The case for blaming the system: It could be argued that a design which lets an ordinary mistake exhaust a table is a poor design, and that the system should discard a status nobody has asked for after some interval. The argument has force, and the answer is that the system cannot know that nobody will ask.

What the system does do: The accumulation is bounded, because when the parent itself ends its zombies are orphaned, adopted by that first process and cleared at once. A zombie survives only as long as the parent that ignored it, which is why the trouble belongs to long running programs.

The judgement: An orphan is the operating system working, a designed handover that costs nobody anything. A zombie is the programmer failing, and the system is right to let it show, because the alternative is discarding a result its owner may still want. The verdict turns on the life of the parent: untidy in a program that ends soon, and a fault that will stop the machine in one that runs for months.

The system keeps the entry because the result in it is not the system's to throw away.

The Master Key: *An orphan is adopted and cleared, while a zombie waits for a parent that never asks.*

CPU scheduling I: criteria, FCFS and shortest job first

This section makes scheduling computable. It states the criteria a scheduler is judged against, separates the three schedulers that share the name, and works first come first served and shortest job first all the way through to averages that can be checked.

BINDER Question 44

At the Buhongwa bus stand the clerk counts departures an hour; a passenger counts only the minutes she stood in the sun. Both judge the same timetable, and they disagree. A scheduler is judged by five numbers, and two of them pull opposite ways.

State the five criteria by which a scheduling algorithm is judged, define each one precisely, and say which two of them pull against each other.

Model Answer

A scheduling algorithm is judged by five measurable criteria, CPU utilisation, throughput, turnaround time, waiting time and response time, and the two that pull against each other are throughput and response time.

CPU utilisation: the proportion of time the processor spends executing a process rather than standing idle. It is the one criterion the operating system wants driven as high as it will go, and a scheduler that leaves the processor idle while a ready process waits has already failed on it.

Throughput: the number of processes completed per unit of time. It is a rate and not a duration, so it is quoted per second on short work and per hour on long work, and it is the criterion a batch installation cares about above the others.

Turnaround time: the whole time a process is in the system, which is its completion time minus its arrival time. It counts every unit the process spent running, queueing for the processor and waiting on a device, so it is the figure the owner of the work feels.

Waiting time: the time a process spends in the ready queue with work to do and no processor to do it on, which is its turnaround time minus its burst time. It is the only one of the five a scheduling algorithm can genuinely shorten, because the burst itself is fixed by the work.

Response time: the time from a process arriving to the first moment it reaches the processor, which is its first time on the processor minus its arrival time. On an interactive machine this, and not turnaround, is what the person at the keyboard notices.

Throughput and response time are the pair in conflict, because answering every new arrival quickly means handing the processor over more often, and no process finishes during a handover.

The Master Key: *Name the criterion before you compute anything, because one schedule can be the best by waiting time and the worst by throughput.*

BINDER Question 45

The Igoma dispensary has three people deciding who moves. One admits patients at the gate, one calls the next name into the consulting room, and one clears the veranda by sending people home to return later. Nobody remembers the third.

Distinguish the long-term, the medium-term and the short-term scheduler. State what each one decides and how often each one runs.

Model Answer

Three schedulers share the name and differ in what they decide and how often they run, in that the long-term scheduler admits a job into memory, the short-term scheduler chooses which ready process runs next, and the medium-term scheduler removes a process from memory and brings it back later.

The long-term scheduler: chooses which jobs waiting in the job pool are admitted, loaded into memory and turned into processes. It runs seldom, on the scale of seconds or minutes, because it acts only when a job is submitted or a process leaves. Its duty is to hold the degree of multiprogramming, the number of processes resident in memory, at a level the machine can carry.

The short-term scheduler: chooses which of the processes already in the ready queue gets the processor next, and it runs on the scale of milliseconds, every time the running process finishes, requests input or output, or otherwise gives the processor up. Because it runs so often it must itself be quick, since the time it takes is time no process is using.

The medium-term scheduler: is the one most often left out. It removes a whole process from memory, writes it out to disk and brings it back when memory allows, which is swapping, and section 10 works that through. It runs on the scale of seconds and lowers the degree of multiprogramming when memory is oversubscribed.

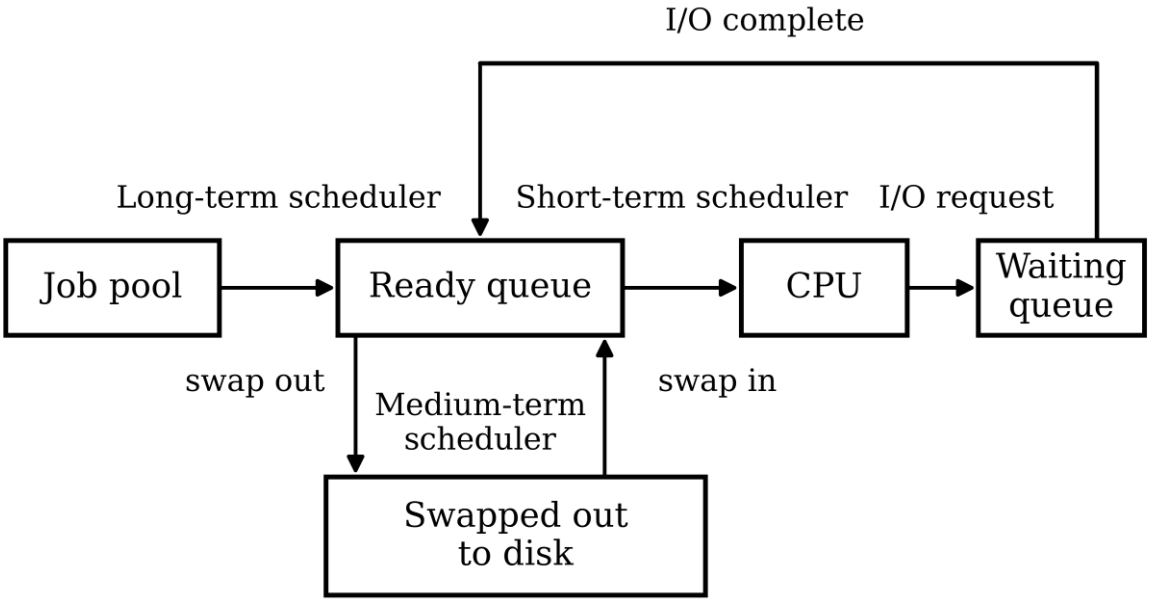


Figure 3.16: *Where each of the three schedulers acts, with the medium-term scheduler drawn doing its whole job, out to disk and back again.*

Scheduler	What it decides	How often it runs	Also called
Long-term	Which jobs are admitted to memory	Seconds to minutes	Job scheduler
Medium-term	Which process leaves memory for disk	Seconds	Swapper
Short-term	Which ready process runs next	Milliseconds	CPU scheduler

Table 3.13: *The three schedulers separated by what they decide and how often they run.*

Reading the levels: a job crosses from the job pool into the ready queue once, under the long-term scheduler. It crosses from the ready queue to the processor many times over its life, under the short-term scheduler. It may also drop out of memory to disk and come back, and that crossing belongs to the medium-term scheduler alone.

Name the medium-term scheduler whenever the difficulty is too many processes for the memory available.

The Master Key: *Long-term admits, short-term chooses from the ready queue, and medium-term swaps a process out and back.*

BINDER Question 46

At the Solwa lorry park a foreman chooses which lorry loads next, and a gateman lifts the barrier. The foreman decides in a moment; the gateman takes ninety seconds to move the chain. The delay belongs to the gateman, not to the choice.

Explain what the dispatcher does and define dispatch latency. Distinguish the dispatcher from the scheduler.

Model Answer

The dispatcher is the part of the operating system that actually hands the processor to the process the short-term scheduler has already chosen, and dispatch latency is the time that handover takes, measured from the instant the outgoing process stops to the instant the incoming one executes its first instruction.

What the dispatcher does: three things, in order. It saves the state of the outgoing process and restores the state of the incoming one, it puts the processor back into user mode, and it jumps to the stored instruction address so that the incoming process resumes exactly where it last stopped rather than at its beginning.

Dispatch latency defined: the time from stopping one process to starting the next, and it is pure overhead, because no process is doing useful work inside it. It is small on any modern machine, but it is paid on every single handover, so an algorithm that changes process often pays it often.

The scheduler chooses, the dispatcher hands over: the short-term scheduler is a decision and the dispatcher is an action. The scheduler names one process out of the ready queue; the dispatcher makes that name true by putting the process on the processor. Neither one can do the other's work, and a system needs both of them.

A common error: students are often taught, wrongly, that the dispatcher decides which process runs next. It decides nothing. Give the decision to the short-term scheduler and the handover to the dispatcher, and be clear which of the two the latency belongs to, because dispatch latency is the cost of the action and not of the decision.

Keep the two apart, since the difference between choosing and handing over is the whole of this idea.

The Master Key: *The scheduler picks the process, the dispatcher puts it on the processor, and dispatch latency is what the putting costs.*

REAL Question 47

Four maize growers reach the Didia posho mill at different minutes and are served strictly in the order they arrive. The first has seven sacks; the third has one. The grower with a single sack watches the whole morning go by.

Four processes arrive with the burst times given below. Draw the Gantt chart for first come first served, compute the waiting time and the turnaround time of each process, and give both averages.

P1 arrives at time 0 and needs 7 units of processor time.

P2 arrives at time 2 and needs 4 units.

P3 arrives at time 4 and needs 1 unit.

P4 arrives at time 5 and needs 4 units.

There is one processor, and a process that is given it keeps it until its burst is finished.

Model Answer

Under first come first served the four processes run in the order they arrive, P1 from 0 to 7, P2 from 7 to 11, P3 from 11 to 12 and P4 from 12 to 16, giving an average waiting time of 4.75 units and an average turnaround time of 8.75 units.

Building the chart: first come first served compares no burst times at all. P1 arrives at 0 and runs to 7. P2 arrived at 2 and has been in the ready queue since, so it takes the processor at 7 and runs to 11. P3 follows from 11 to 12, P4 from 12 to 16, and the four bursts of 7, 4, 1 and 4 fill 16 units with the processor never idle.

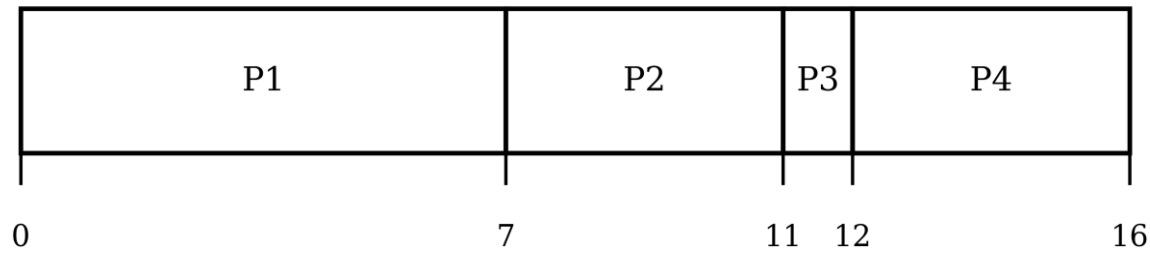


Figure 3.17: *First come first served on the four processes, with every boundary of the schedule ticked and labelled.*

P1 and P2: P1 completes at 7 and arrived at 0, so its turnaround is 7 minus 0, which is 7, and its waiting time is 7 minus its burst of 7, which is 0. P2 completes at 11 and arrived at 2, so its turnaround is 11 minus 2, which is 9, and its waiting time is 9 minus 4, which is 5.

P3 and P4: P3 completes at 12 and arrived at 4, so its turnaround is 12 minus 4, which is 8, and its waiting time is 8 minus 1, which is 7. P4 completes at 16 and arrived at 5, so its turnaround is 16 minus 5, which is 11, and its waiting time is 11 minus 4, which is 7.

The two averages: add the four waiting times and divide by four. They are 0, 5, 7 and 7, which total 19, and 19 divided by 4 is 4.75 units. Do the same with the turnarounds, which are 7, 9, 8 and 11, total 35, and 35 divided by 4 is 8.75 units.

Process	Arrival	Burst	Start	Finish	Turnaround	Waiting
P1	0	7	0	7	7	0
P2	2	4	7	11	9	5
P3	4	1	11	12	8	7
P4	5	4	12	16	11	7
Average					8.75	4.75

Table 3.14: *Every figure for the four processes under first come first served, read off the chart above it.*

What P3 pays: P3 needs a single unit of processor time and waits 7 units to get it, seven times its own burst. Nothing in first come first served can see that, because the algorithm reads arrival times and never looks at a burst time when it chooses.

The chart, the table and the averages must all be read off one schedule, or one of the three is wrong.

The Master Key: *Under first come first served the running order is the arrival order, so a short process that arrives late pays for every long one ahead of it.*

REAL Question 48

The Mwadui mill tries something different on the same four customers: whenever the grinder falls idle, the smallest load waiting goes on next. The grower with one sack is home before noon. Somebody paid for that, and the averages say who.

Using the same four processes, draw the Gantt chart for non-preemptive shortest job first, compute the same figures, and say precisely why the averages differ from those of first come first served.

P1 arrives at time 0 and needs 7 units of processor time.

P2 arrives at time 2 and needs 4 units.

P3 arrives at time 4 and needs 1 unit.

P4 arrives at time 5 and needs 4 units.

The shortest burst among the processes that have already arrived is chosen next, and once chosen a process runs to the end of its burst.

Model Answer

Under non-preemptive shortest job first the same four processes run P1 from 0 to 7, P3 from 7 to 8, P2 from 8 to 12 and P4 from 12 to 16, which brings the average waiting time down from 4.75 units to 4.0 and the average turnaround time from 8.75 units to 8.0.

Why P1 still runs first: at time 0 the only process that has arrived is P1, and the algorithm chooses the shortest burst among those already arrived, not the shortest of all four. P1 therefore runs from 0 to 7 exactly as before, and the two schedules part company only at 7.

The choice at time 7: P2, P3 and P4 have all arrived by then, with bursts of 4, 1 and 4. The shortest is P3, so P3 runs from 7 to 8. At time 8 P2 and P4 both need 4 units, the tie goes to the earlier arrival, and P2 runs from 8 to 12, leaving P4 from 12 to 16.

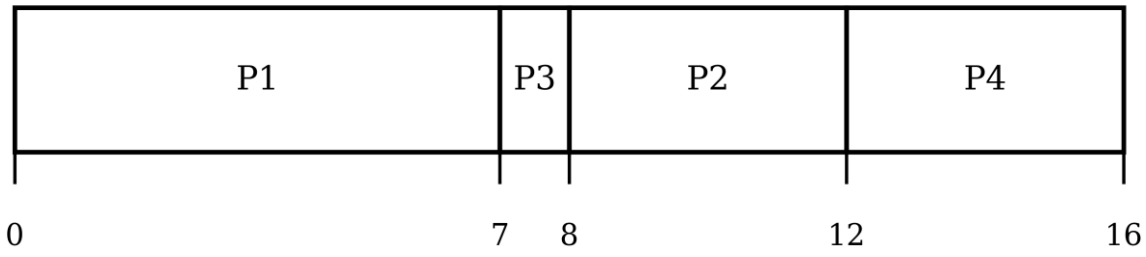


Figure 3.18: *The same four processes under non-preemptive shortest job first, drawn on the same time scale as the chart before it.*

P1 and P3: P1 completes at 7 and arrived at 0, so its turnaround is 7 and its waiting time is 7 minus 7, which is 0. P3 completes at 8 and arrived at 4, so its turnaround is 8 minus 4, which is 4, and its waiting time is 4 minus 1, which is 3.

P2 and P4: P2 completes at 12 and arrived at 2, so its turnaround is 12 minus 2, which is 10, and its waiting time is 10 minus 4, which is 6. P4 completes at 16 and arrived at 5, so its turnaround is 11 and its waiting time is 7, both unchanged.

Process	Arrival	Burst	Start	Finish	Turnaround	Waiting
P1	0	7	0	7	7	0
P3	4	1	7	8	4	3
P2	2	4	8	12	10	6
P4	5	4	12	16	11	7
Average					8.0	4.0

Table 3.15: *The same four processes under shortest job first, listed in the order the processor took them.*

Why the averages fall: one swap did all of it. Running P3 before P2 lengthens the wait of P2 from 5 units to 6, a cost of 1, and shortens the wait of P3 from 7 units to 3, a saving of 4. Total waiting falls from 19 units to 16 and the average with it. Putting a short burst in front of a long one always trades a small addition for a larger subtraction.

The two charts share their first bar and their last, and everything the algorithm changed happened between time 7 and time 12.

The Master Key: *Shortest job first lowers the average by moving short bursts forward, and it can only choose among processes that have already arrived.*

REAL Question 49

A tractor needing twenty four minutes of welding reaches the Ngokolo yard first, and two three minute jobs queue behind it. Reverse the three and the welder still finishes at the same hour, yet the yard empties far sooner.

One long process is at the head of the queue and two short ones are behind it. By computing the average waiting time twice, show what the order alone costs under first come first served, and name the effect.

P1 needs 24 units of processor time.

P2 needs 3 units.

P3 needs 3 units.

All three are ready at time 0, and the processor serves them strictly in the order the queue holds them.

Model Answer

With the long process at the head of the queue the average waiting time is 17 units, and with the two short processes in front of it the average is 3 units, so the order alone costs 14 units of average waiting on identical work, and the effect has a name, the convoy effect.

The expensive order: P1 runs from 0 to 24, P2 from 24 to 27 and P3 from 27 to 30. P1 waits nothing, P2 waits 24 units and P3 waits 27, so the three waiting times total 51 and the average is 51 divided by 3, which is 17 units.

The cheap order: put the two short processes first and P2 runs from 0 to 3, P3 from 3 to 6 and P1 from 6 to 30. P2 waits nothing, P3 waits 3 units and P1 waits 6, so the three total 9 and the average is 9 divided by 3, which is 3 units.

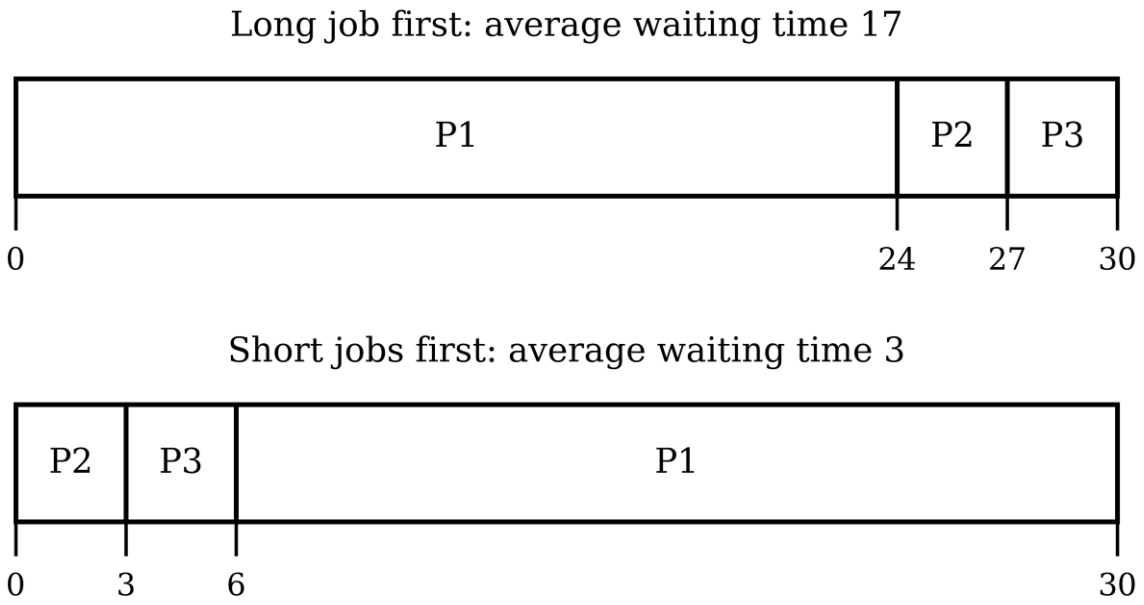


Figure 3.19: *The same three processes in both orders on one time scale, each strip labelled with the average waiting time it produces.*

Nothing else changed: the bursts are the same three numbers, every process is ready at time 0, and the processor is busy for 24 plus 3 plus 3, which is 30 units, in both charts. The last process finishes at 30 either way. Only the order of the bars moved, and the average waiting time fell by 14 units.

Why it is called a convoy: one heavy lorry on a single lane road holds a line of light vehicles behind it and none of them can pass. Under first come first served one long process does the same to every short process behind it, for the whole of its burst, because the algorithm has no way to let a short process through.

Where it hurts most: the short processes are usually the ones that ask for a device and give the processor straight back. While the long process computes they cannot reach the processor to issue those requests, so the disk and the printer stand idle behind a busy processor and the whole machine slows.

An average in this topic is not a property of the processes alone; it is a property of the order the algorithm puts them in.

The Master Key: *The convoy effect is one long process holding every short process behind it, and reordering the very same work is what removes it.*

REAL Question 50

A duty roster pinned above the bench at the Nyamilama workshop carries a table of job lengths and, beneath it, a chart drawn from that table. The chart runs one hour past the point where the three jobs can possibly end.

The roster below gives a table of job lengths and a chart drawn from it. Show that the two disagree, identify which of them is wrong, and give the corrected chart.

Duty roster, Nyamilama workshop. Three jobs, all ready at hour 0, taken in the order listed, one bench free.

Job P1 needs 3 hours.

Job P2 needs 6 hours.

Job P3 needs 5 hours.

The chart drawn underneath the roster shows P1 from 0 to 3, P2 from 3 to 10, and P3 from 10 to 15.

Model Answer

The chart is wrong and the table is right, because the chart gives P2 seven hours where the table says six, so every boundary after the first is an hour late and the chart ends at 15 when the three jobs account for only 14, and the corrected chart runs P1 from 0 to 3, P2 from 3 to 9 and P3 from 9 to 14.

The check that catches it: every job is ready at hour 0 and the bench is never free, so the chart must end at the sum of the three job lengths. That sum is 3 plus 6 plus 5, which is 14. The chart ends at 15. An hour has appeared that no job on the roster asked for.

Finding the bar that is wrong: measure each bar against its row. P1 runs 3 minus 0, which is 3 hours, and the table says 3. P2 runs 10 minus 3, which is 7 hours, and the table says 6. P3 runs 15 minus 10, which is 5 hours, and the table says 5. Only P2 disagrees, and its extra hour is the one the total gained.

Which of the two is wrong: the table of job lengths is the data and the chart is drawn from it, so a disagreement between them is a fault in the drawing and not in the work. Correct the chart to the table. Correcting the table to the chart would invent an hour of work the bench never had to do.

The corrected chart: P1 occupies 0 to 3, P2 occupies 3 to 9 because 3 plus 6 is 9, and P3 occupies 9 to 14 because 9 plus 5 is 14. The four boundaries are 0, 3, 9 and 14, and the last of them equals the total of the job lengths, which is the check the drawn chart failed.

What the error did to the figures: on the drawn chart P3 appears to wait 10 hours, and on the corrected chart it waits 9. The corrected waiting times are 0, 3 and 9, which total 12, and 12 divided by 3 is an average waiting time of 4 hours. Read off the faulty chart, two of the three jobs would have been given the wrong wait.

Add the burst times and compare that total with the right hand edge of the chart before trusting anything else on it.

The Master Key: *A Gantt chart drawn from a table must end at the sum of the bursts, and where it does not, the chart is the thing to correct.*

REAL Question 51

The logbook at the Mwadui pump house records only when each of three jobs finished, never how long any of them took. The endings are there in one neat column, and the order of running is known. Nothing else is.

Three processes ran under first come first served and their completion times are given. Work backwards to the burst time of each one and to the average turnaround time.

Three processes ran to completion under first come first served on one processor, in the order P1, P2, P3.

P1 arrived at time 0 and finished at time 4.

P2 arrived at time 2 and finished at time 10.

P3 arrived at time 3 and finished at time 18.

The processor never stood idle once P1 had begun.

Model Answer

Working backwards from the finishing times, the bursts are 4 units for P1, 6 units for P2 and 8 units for P3, and the average turnaround time is 9 units.

The rule that unlocks it: under first come first served a process starts either when it arrives or when the process before it finishes, whichever of the two is later. Here the processor never idles, so every process after the first starts at the exact instant its predecessor finished, and a burst is then simply the finish time minus that start.

The burst of P1: P1 arrived at 0 and was first in the queue, so it started at 0 with nothing ahead of it. Its burst is its finish minus its start, which is 4 minus 0, giving 4 units, and because it never waited its turnaround and its burst are the same number.

The burst of P2: P1 finished at 4 and P2 had already arrived at 2, so P2 started at 4 and not at 2. Its burst is 10 minus 4, which is 6 units. The two units between its arrival and its start are waiting time, not work.

The burst of P3: P2 finished at 10 and P3 had arrived at 3, so P3 started at 10. Its burst is 18 minus 10, which is 8 units, and it stood in the ready queue for the seven units between arriving and starting.

The average turnaround: turnaround is completion minus arrival, so P1 gives 4 minus 0, which is 4, P2 gives 10 minus 2, which is 8, and P3 gives 18 minus 3, which is 15. Those three total 27, and 27 divided by 3 is 9 units.

The check: the three bursts total 4 plus 6 plus 8, which is 18, and 18 is exactly where the last process finished. Every unit of processor time between 0 and 18 is accounted for, so no burst has been left out or counted twice.

The direction of the arithmetic changes but the definitions do not, and everything here came out of completion minus arrival and turnaround minus burst.

The Master Key: *Given arrival and completion times under first come first served, each start is the previous finish and each burst is finish minus start.*

HOT Question 52

The Buhongwa print bureau serves the smallest job first and its average waiting time is the best on the street. A customer with a five hundred page binding has been waiting since Tuesday. The policy that produced the best average produced him too.

Shortest job first gives the lowest average waiting time of any non-preemptive algorithm. Argue whether it is therefore the right algorithm to use, and reach a judgement.

Model Answer

Shortest job first does produce the lowest average waiting time available to any algorithm that lets the process it has chosen run to the end of its burst, and it is still the wrong scheduler for a general purpose machine, because it can starve a long process for as long as short ones keep arriving, and because it needs a number, the length of the next burst, that nobody can supply in advance.

The case for it, put properly: the claim is not that shortest job first is fast but that it is optimal on one criterion. Move a shorter process in front of a longer one and the total waiting time always falls, because the shorter one saves more than the longer one loses. That mechanism is the whole of it, and it is what took the four processes of question 47 from an average wait of 4.75 units to 4.0.

The first objection, starvation: a long process is chosen only when it is the shortest thing waiting, so a steady arrival of short work can hold it in the ready queue indefinitely. The algorithm contains nothing that ever rescues it, because it never looks at how long a process has already waited. The repair, letting a waiting process gain ground as it ages, sits with the priority algorithms in section 7.

The second objection, the number does not exist: the algorithm needs the length of the next burst before that burst has run, and no operating system knows it. Real systems estimate it from the recent past, averaging previous bursts with the most recent weighted heaviest, then act on the estimate. An algorithm that is optimal on a number it must guess is optimal only as far as the guess holds.

Where it is genuinely right: a batch installation, where work is submitted with a declared run time and the queue drains overnight. The number the algorithm needs is written on the submission, so it is not a guess, and starvation is bounded because no new work arrives once the queue has closed. Payroll and billing runs are scheduled exactly this way.

The judgement: use shortest job first where burst length is declared rather than guessed and where the queue empties, and do not use it as the scheduler of an interactive system. On such a system the verdict rests on the two objections together, since the number is a guess and a wrong guess can leave a long process waiting without end.

Its optimality is real and narrow, and the two things that break it are the two things a general purpose machine has most of, unpredictable work and long processes.

The Master Key: *Shortest job first is optimal on average waiting time, unusable without a burst length it cannot know, and unsafe for long processes.*

HOT Question 53

Twenty two machines in the Ngokolo Secondary computer room run short typing and browsing work all afternoon, while three long print jobs wait in the same queue. The teacher wants the room to feel quick. Quick for whom is the whole question.

A school computer room runs short interactive work and a few long print jobs on the same machines. Argue which scheduling criterion should be optimised there, and reach a judgement.

Model Answer

Optimise response time in that room and accept that the print jobs will finish later than they otherwise would, because the value of a short interactive burst is destroyed by delay, while a print job that finishes at half past four instead of a quarter past is noticed by nobody.

What the room actually contains: two populations of work with different needs. Dozens of short bursts, a keystroke echoed, a line reformatted, a file saved, each measured in milliseconds and each watched by a person. Against them a handful of long print jobs that nobody watches, measured in minutes, whose only requirement is to finish before the room closes.

The case for throughput: throughput counts completed processes per hour and is the easiest criterion to report. It is the wrong one here. A scheduler tuned for throughput runs the long print jobs without interruption, because uninterrupted work finishes fastest, and every keyboard in the room feels dead while it does so.

The case for turnaround time: turnaround is what the print jobs care about and it is a defensible objective in a queue of long work. In this room it is the wrong average to minimise, because a typing burst has a turnaround of milliseconds and a print job one of minutes, so the mean is dominated by the work nobody is waiting on.

Why response time wins here: response time is the gap between an arrival and its first moment on the processor, which is precisely the gap the person at the keyboard experiences. The interactive work is the majority of what the room runs and the whole of what its users judge it by, and a room where a keystroke appears at once feels quick.

What is being given up: the turnaround time of the print jobs, and some throughput with it. Holding response time low means the processor changes hands more often, and no process finishes during a handover, so the work completed per hour falls. That is the price, and in a teaching room it is worth paying.

The judgement: optimise response time, and protect the print jobs only so far as guaranteeing that they finish before the room closes. Were the same machines left running payroll overnight with nobody present, the verdict would reverse and throughput would win, which shows that the criterion follows the users and not the hardware.

Choose the criterion from who is waiting, not from what happens to be running.

The Master Key: *Optimise response time wherever a person is watching, and pay for it in turnaround and throughput on the work nobody watches.*

CPU scheduling II: preemption, quanta and priority

This section adds the thing that makes an interactive machine possible, taking the processor back. It classifies every algorithm in the chapter as preemptive or not, runs the same workload at two different quanta, and shows what ageing does to a process that would otherwise never run.

BINDER Question 54

At the Salawe maize store the weighing scale is handed from one seller to the next. On Monday nobody may be interrupted; on Tuesday the clerk takes the scale back after two minutes. Same sacks, same scale, and two completely different afternoons.

Define preemptive and non-preemptive scheduling. Classify first come first served, shortest job first, shortest remaining time first, round robin and priority scheduling, and state what makes each one what it is.

Model Answer

Scheduling is preemptive when the operating system may take the processor from a running process before its burst is finished, and non-preemptive when the process keeps it until the burst ends or it blocks, which puts first come first served and shortest job first on the non-preemptive side, shortest remaining

time first and round robin on the preemptive side, and priority scheduling on both, because it exists in two forms.

The four moments of choice: the running process finishes, the running process blocks for a device, a new process joins the ready queue, or a timer expires. A non-preemptive algorithm chooses at the first two and nowhere else, so the last two are exactly what preemption adds.

First come first served: is non-preemptive because arrival order is its only input, and arrival order cannot change while a process is running. Nothing occurring during a burst can alter a choice already made.

The two shortest first algorithms: apply one comparison at different moments. Shortest job first picks the shortest burst among the processes that have arrived and lets it run out, so it is non-preemptive. Shortest remaining time first repeats that comparison at every arrival and hands the processor to a shorter newcomer, so it is preemptive.

Round robin: is preemptive by construction, because what ends a turn is a timer and not the process. A process still holding work when its quantum expires is returned to the ready queue unfinished, and that return is preemption.

Priority scheduling belongs on both sides: In its non-preemptive form a better priority arrival waits until the running process finishes. In its preemptive form it takes the processor at once. An answer naming only one of the two has classified half an algorithm.

Algorithm	Preemptive	What ends a turn on the processor
First come first served	No	The burst finishes or the process blocks
Shortest job first	No	The burst finishes or the process blocks
Shortest remaining time first	Yes	A process arrives with a shorter remaining time
Round robin	Yes	The quantum expires
Priority, non-preemptive	No	The burst finishes or the process blocks
Priority, preemptive	Yes	A process arrives with a better priority

Table 3.16: *The five algorithms classified, with the event that takes the processor away in each case.*

Classify any algorithm by asking one question of it, whether anything can take the processor from a running process before its burst is done.

The Master Key: *Preemption is the power to take the processor back part way through a burst, and only first come first served and shortest job first lack it outright.*

BINDER Question 55

The clerk at Igalula gives each customer three minutes at the counter, then rings a bell. Yesterday she tried thirty seconds and served nobody; today she tried an hour and the line never moved. Somewhere between the two lies the right figure.

Explain what a time quantum is and describe what goes wrong when it is set too large and when it is set too small.

Model Answer

A time quantum is the longest single run the operating system allows a process before a timer takes the processor back, and it fails in two directions, because a quantum larger than every burst preempts nobody and turns round robin back into first come first served, while a quantum close to the cost of one handover spends the machine on handing over rather than on work.

What the quantum actually is: a fixed length of time, the same for every process in a plain round robin, chosen by the operating system and enforced by a timer. It is not a property of any process and it takes no account of how much work that process still has left to do.

Set too large: the longest burst among the four processes of this chapter is 7 units, so a quantum of 7 or more preempts nobody at all. The schedule that comes out is P1, then P2, then P3, then P4, which is the arrival order and therefore first come first served exactly. Round robin has not been switched off; it has simply never had cause to act.

Set too small: every expiry costs a handover, and a handover is overhead. Suppose one costs 1 unit. At a quantum of 4 the processor spends 5 units to deliver 4 units of work, so 20 per cent of it is lost. At a quantum of 1 it spends 2 units to deliver 1, and 50 per cent is lost.

The shape of the choice: response time improves as the quantum falls, but only until overhead begins to eat the gain, after which response time and throughput worsen together. A workable quantum therefore sits well above the cost of a handover and a little above the length of a typical interactive burst.

Say which of the two failures a proposed quantum would produce before arguing about the number itself.

The Master Key: *A quantum longer than every burst hands back first come first served, and a quantum near the cost of a handover hands back nothing at all.*

BINDER Question 56

At the Malongwe dispensary the nurse calls emergencies first, and a man with a cut hand has been on the bench since eight. By noon the clerk adds a rule of her own: every hour of waiting moves you one place up.

Explain priority scheduling, define starvation and explain how ageing prevents it. Define priority inversion and say how it differs from starvation.

Model Answer

Priority scheduling gives the processor to the ready process with the best priority number, and throughout this chapter a smaller number means a higher priority, so its characteristic failure is starvation, a low priority process left ready and never chosen while better work keeps arriving, and its repair is ageing, which adds a step of priority for every fixed period a process has waited.

Where the number comes from: an internal priority is computed by the operating system from something it can measure, such as memory held or the ratio of processor time to time spent on input and output. An external priority is set from outside the machine, by rank or by who paid.

Starvation defined: indefinite postponement, meaning a process that is ready to run and is never selected. It is not merely a long wait, because a long wait ends by itself. Starvation has no end while better priority work keeps arriving, and a starved process can stay ready until the machine is restarted.

How ageing removes it: the scheduler improves the priority of every waiting process by one step for each period it has gone without running. A process admitted at the worst priority therefore climbs, reaches the best priority after a bounded number of steps, and must then be chosen. Waiting itself earns priority.

Priority inversion: a separate fault, in which a high priority process is made to wait for a low priority one because the low one holds a resource that only a single process may hold at a time. A middling process that preempts the holder stretches the wait further. The repair is priority inheritance, lending the holder the priority of the process it is blocking until it releases.

Starvation comes from the priority order alone and strikes the worst number, while priority inversion needs a shared resource and strikes the best one.

The Master Key: *Ageing turns waiting into priority, which is the one thing a fixed priority scheme can never do for a starving process.*

REAL Question 57

The Puge milling shed keeps the same four customers as yesterday, but the miller now stops mid sack whenever someone lighter walks in. Two customers are interrupted before they finish, and the chart of the morning grows two extra bars.

Using the same four processes as the previous section, draw the Gantt chart for shortest remaining time first, compute the waiting time and the turnaround time of each process, and compare the averages with those of non-preemptive shortest job first.

P1 arrives at time 0 and needs 7 units of processor time.

P2 arrives at time 2 and needs 4 units.

P3 arrives at time 4 and needs 1 unit.

P4 arrives at time 5 and needs 4 units.

At every instant the processor goes to the process with the least time still to run, and an arrival that needs less than the running process takes the processor from it immediately.

Model Answer

Under shortest remaining time first the four processes run P1 from 0 to 2, P2 from 2 to 4, P3 from 4 to 5, P2 again from 5 to 7, P4 from 7 to 11 and P1 again from 11 to 16, which gives an average waiting time of 3.0 units against 4.0 under non-preemptive shortest job first, and an average turnaround time of 7.0 against 8.0.

The two interruptions: at time 2 P2 arrives needing 4 units against the 5 P1 still has, so P1 loses the processor with work outstanding. At time 4 P3 arrives needing 1 unit against the 2 P2 has left, so P2 loses it in turn. Those two moments are the entire difference from the non-preemptive run.

Where nothing happens: at time 5 P4 arrives needing 4 units while P2 has only 2 left, so the comparison fails and P2 keeps the processor to the end of its burst at 7. The rule is not that every arrival preempts, but that an arrival preempts when it is shorter than what remains.

The order after time 7: P2 finishes at 7 and the choice is between P4, which needs 4, and P1, which needs 5. P4 goes first and runs from 7 to 11, then P1 takes the processor back at 11 and runs the last 5 of its 7 units through to 16.

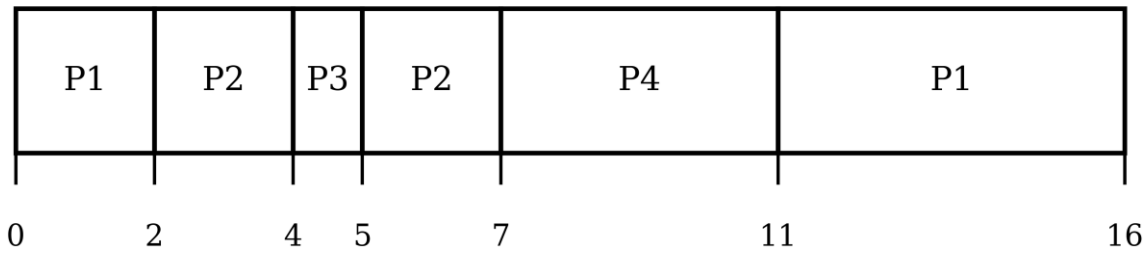


Figure 3.20: *Shortest remaining time first on the four processes, with P1 and P2 each appearing twice because each was preempted once.*

P1 and P2: P1 completes at 16 and arrived at 0, so its turnaround is 16 and its waiting time is 16 minus its burst of 7, which is 9. P2 completes at 7 and arrived at 2, so its turnaround is 5 and its waiting time is 5 minus 4, which is 1.

P3 and P4: P3 completes at 5 and arrived at 4, so its turnaround is 1 and its waiting time is 1 minus 1, which is 0. P4 completes at 11 and arrived at 5, so its turnaround is 6 and its waiting time is 6 minus 4, which is 2.

The two averages: the waiting times are 9, 1, 0 and 2, which total 12, and 12 divided by 4 is 3.0 units. The turnaround times are 16, 5, 1 and 6, which total 28, and 28 divided by 4 is 7.0 units. Both averages are 1.0 unit better than the non-preemptive run of the same work.

Process	Arrival	Burst	Completion	Turnaround	Waiting
P1	0	7	16	16	9
P2	2	4	7	5	1
P3	4	1	5	1	0
P4	5	4	11	6	2
Average				7.0	3.0

Table 3.17: *Every figure for the four processes under shortest remaining time first, read off the chart above it.*

Six bars for four processes: P1 and P2 are each preempted once, so the chart carries 4 bars plus 2, which is 6. A preemptive chart showing one bar per process is either a schedule in which nothing was preempted or a drawing error, and counting the bars is the fastest way to tell.

Who paid for the improvement: P1 waited 0 units under shortest job first and waits 9 here. The average fell because three short processes were pushed forward, and the long process absorbed that movement entirely.

The chart, the table and the averages all come from one schedule, so a disagreement between any two of them is an error.

The Master Key: *Shortest remaining time first re-runs the shortest job comparison at every arrival, and every preemption it makes adds one more bar to the chart.*

REAL Question 58

Two minutes each, then the next in line, and round again until everybody is done. That is the rule at the Ipole grinding mill, and the customer who arrived first with the heaviest load is the one who goes home last.

Using the same four processes, draw the Gantt chart for round robin with a time quantum of two units, compute the waiting time and the turnaround time of each process, and give the averages.

P1 arrives at time 0 and needs 7 units of processor time.

P2 arrives at time 2 and needs 4 units.

P3 arrives at time 4 and needs 1 unit.

P4 arrives at time 5 and needs 4 units.

A turn on the processor lasts at most 2 units, after which an unfinished process goes to the back of the ready queue.

A process that arrives while another is running joins the ready queue before the process whose turn has just been ended.

Model Answer

At a quantum of 2 the schedule needs nine slices, P1 from 0 to 2, P2 to 4, P1 to 6, P3 to 7, P2 to 9, P4 to 11, P1 to 13, P4 to 15 and P1 to 16, so P4 completes at 15 and P1, which arrived first, completes last at 16, with an average waiting time of 5.0 units, an average turnaround time of 9.0 and an average response time of 1.5.

Where the queue convention decides the chart: P4 arrives at 5, while P1 is part way through the turn that ends at 6. At that instant P4 is put into the ready queue ahead of P1, giving P3, P2, P4 and then P1. Enqueue the preempted P1 ahead of P4 instead and the later half of the chart changes, so the convention is not a detail.

Why P1 finishes last: a burst of 7 taken 2 units at a time needs 4 turns, three of them full and a last one of 7 minus 6, which is 1 unit. That fourth turn cannot begin until every other process has had its share, so it lands at 15 and P1 leaves the system at 16, having arrived before all the others.

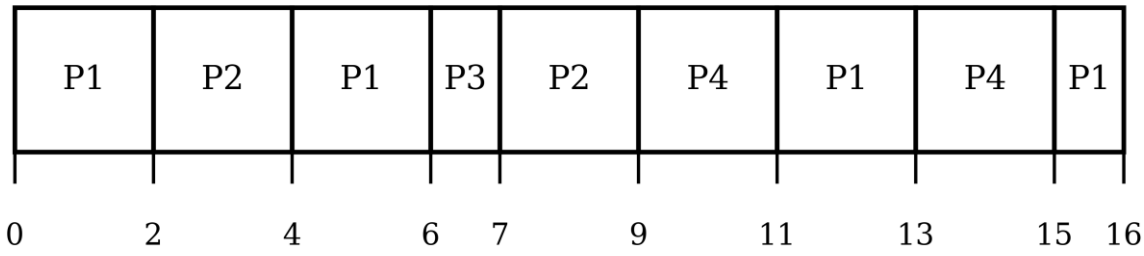


Figure 3.21: *Round robin at a quantum of two units, in which no bar is longer than two and P1 occupies four separate bars.*

P1 and P2: P1 completes at 16 and arrived at 0, so its turnaround is 16 and its waiting time is 16 minus 7, which is 9. P2 completes at 9 and arrived at 2, so its turnaround is 7 and its waiting time is 7 minus 4, which is 3.

P3 and P4: P3 completes at 7 and arrived at 4, so its turnaround is 3 and its waiting time is 3 minus 1, which is 2. P4 completes at 15 and arrived at 5, so its turnaround is 10 and its waiting time is 10 minus 4, which is 6.

The three averages: the waiting times are 9, 3, 2 and 6, which total 20, and 20 divided by 4 is 5.0 units. The turnarounds are 16, 7, 3 and 10, totalling 36, so the average is 9.0. First turns come at clock 0, 2, 6 and 9, so the response times are 0, 0, 2 and 4, which total 6 and average 1.5 units.

What the averages are buying: 5.0 units of average waiting is worse than the 4.75 of first come first served and the 4.0 of shortest job first. Round robin is not competing for that average. What it buys is the average response time of 1.5 units, since no process waits longer than a few quanta for its first turn.

Nine slices for four processes is the price of the rule, and the response time is what the price bought.

The Master Key: *Round robin trades average waiting time for early first turns, and at a small quantum the longest process is the one that finishes last.*

REAL Question 59

Double the turn at the Usinge mill from two minutes to four and the queue behaves differently at once. Fewer handovers, less shuffling, and one grower who arrived late now stands much longer before the machine first turns for him.

Run the same four processes under round robin with a quantum of four units. Compute the averages and explain precisely what changed against the quantum of two and why.

P1 arrives at time 0 and needs 7 units of processor time.

P2 arrives at time 2 and needs 4 units.

P3 arrives at time 4 and needs 1 unit.

P4 arrives at time 5 and needs 4 units.

A turn on the processor now lasts at most 4 units, and the queue convention is unchanged, so an arrival is enqueued before the process whose turn has just been ended.

Model Answer

At a quantum of 4 the same work needs only five slices, P1 from 0 to 4, P2 to 8, P3 to 9, P1 to 12 and P4 to 16, which lowers the average waiting time from 5.0 units to 4.5 and the average turnaround from 9.0 to 8.5, halves the handovers from 8 to 4, and raises the average response time from 1.5 units to 3.25.

Only one process is preempted: P1 uses a full turn of 4 and still has 7 minus 4, which is 3 units left, so it is sent back to the ready queue. P2, P3 and P4 all have bursts of 4 or less and each finishes inside a single turn. Four processes therefore produce five bars, not nine.

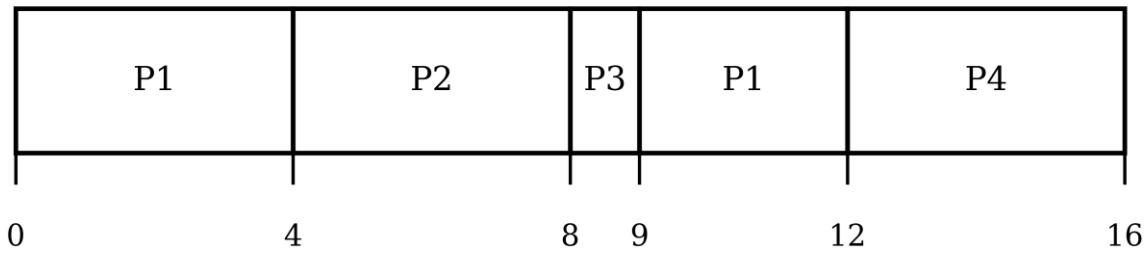


Figure 3.22: *Round robin at a quantum of four units on the same four processes, drawn on the same time scale as the chart before it.*

The four waiting times: P1 completes at 12, so its waiting time is 12 minus 0 minus 7, which is 5. P2 completes at 8, giving 8 minus 2 minus 4, which is 2. P3 completes at 9, giving 9 minus 4 minus 1, which is 4. P4 completes at 16, giving 16 minus 5 minus 4, which is 7.

The averages: the waiting times 5, 2, 4 and 7 total 18, and 18 divided by 4 is 4.5 units. The turnarounds are 12, 6, 5 and 11, which total 34, so the average is 8.5 units. First turns come at clock 0, 4, 8 and 12, so the response times are 0, 2, 4 and 7, which total 13 and average 3.25 units.

Fewer handovers: five slices sit end to end with 4 handovers between them, against nine slices and 8 handovers at the smaller quantum. The same work, unchanged in every burst and every arrival, is delivered with half as many changes of process, and each change avoided is overhead not paid.

What the late arrival pays: P4 arrives at 5 and first reaches the processor at 12, a response time of 7 units, where at a quantum of 2 it waited only 4. Every process ahead of it now takes a longer turn, and P1 takes two of them, so the process that joins the queue last is the one that feels the larger quantum.

Why the averages fell here: the four bursts are 7, 4, 1 and 4, and a quantum of 4 is at least as long as three of them. When most processes finish inside one turn, round robin behaves like the non-preemptive run and its turnaround improves. That improvement is a property of this workload, not a general rule about larger quanta.

Raising the quantum improved three numbers and worsened the one that matters most to a person at a keyboard.

The Master Key: *A larger quantum buys fewer handovers and lower averages, and it charges the whole of that saving to whichever process arrives last.*

REAL Question 60

Five jobs reach the Nyahua workshop at opening time, each with a number on its docket, and the lowest number is welded first. One docket carries a five. It is still on the hook when the workshop closes.

Five processes arrive together with the priorities given below. Compute the average waiting time under non-preemptive priority scheduling, identify the process that starves and show what ageing does to it.

All five processes arrive at time 0, and a smaller priority number means a higher priority.

P1 needs 10 units of processor time and has priority 3.

P2 needs 1 unit and has priority 1.

P3 needs 2 units and has priority 4.

P4 needs 1 unit and has priority 5.

P5 needs 5 units and has priority 2.

Once a process is given the processor it runs to the end of its burst.

Model Answer

Priority order puts the five processes on the processor as P2, P5, P1, P3 and finally P4, giving an average waiting time of 8.2 units, and the process that starves is P4, which needs a single unit of processor time and waits 18 units to get it, eighteen times its own burst.

Building the order: every process is present at time 0, so the algorithm simply takes them in order of priority number, smallest first. P2 has 1, P5 has 2, P1 has 3, P3 has 4 and P4 has 5. Bursts do not enter the choice at any point, which is what separates this from shortest job first.

The completion times: P2 runs from 0 to 1, P5 from 1 to 6 because 1 plus 5 is 6, P1 from 6 to 16 because 6 plus 10 is 16, P3 from 16 to 18, and P4 from 18 to 19. The processor never idles, so the last completion equals the total of the five bursts.

The waiting times: every process arrived at 0, so each waiting time is simply its completion minus its own burst. That gives 0 for P2, 1 for P5, 6 for P1, 16 for P3 and 18 for P4. The five total 41, and 41 divided by 5 is an average waiting time of 8.2 units.

Process	Burst	Priority	Start	Completion	Turnaround	Waiting
P2	1	1	0	1	1	0
P5	5	2	1	6	6	1
P1	10	3	6	16	16	6
P3	2	4	16	18	18	16
P4	1	5	18	19	19	18
Average					12.0	8.2

Table 3.18: *The five processes in the order priority scheduling takes them, with the starved process on the last row.*

Why this is starvation and not bad luck: P4 is last only because its number is the largest, and nothing in the algorithm ever revisits that number. Add one more process of priority 1 every few units and P4 is never reached at all. Here the queue empties, so P4 runs at 18; in a system that keeps receiving work it would not run.

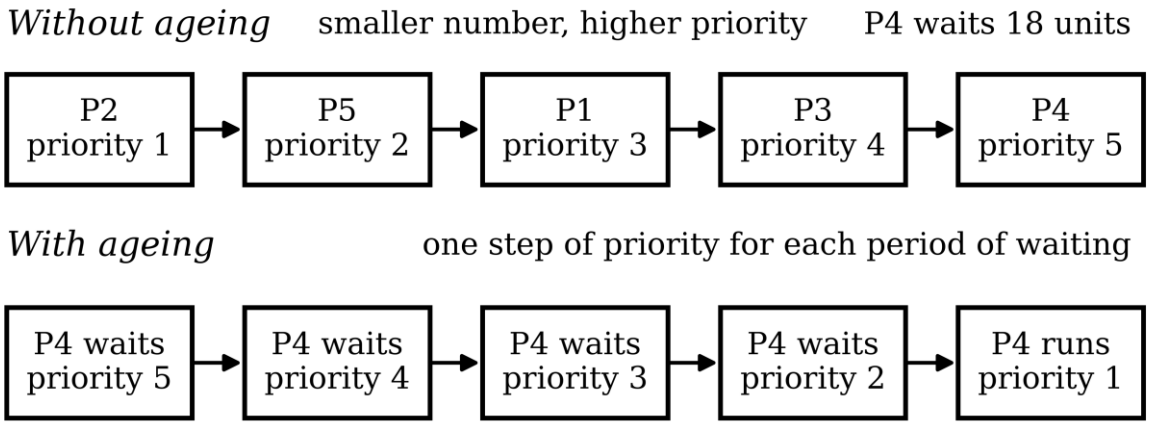


Figure 3.23: *The order the five priorities produce, and beneath it the same starved process carried up one step at a time by ageing until it may run.*

What ageing does to P4: each period of waiting improves its priority by one step, so P4 climbs from 5 through 4, 3 and 2 to 1, which is 5 minus 1, or 4 steps. Run the same algorithm with P4 now at priority 1 and the order becomes P2, P4, P5, P1, P3, so P4 starts at 1 and waits a single unit instead of 18.

What it cost the others: P5, P1 and P3 are each pushed back by the single unit P4 takes, so the waiting times become 0 for P2, 1 for P4, 2 for P5, 7 for P1 and 17 for P3, totalling 27. The average falls from 8.2 units to 5.4, a fall of 2.8, so rescuing the starved process improved the mean rather than damaging it, because the work moved forward was tiny.

A single unit of work waiting 18 units is the signature of starvation, and the repair cost the rest of the queue one unit each.

The Master Key: *Priority scheduling reads only the priority number, so without ageing the worst number waits for everything else in the system.*

REAL Question 61

A noticeboard outside the computer room at Puge Secondary carries three typed statements about round robin, left there by last year's class. Two of them are wrong, and one of the two is wrong in a way that changes every chart drawn from it.

Three statements about round robin are pinned on the noticeboard below. Identify the one that is correct, explain the fault in each of the others, and give a corrected version of both.

Noticeboard, outside the computer room at Puge Secondary. Three statements about round robin scheduling.

Statement A. Round robin is a non-preemptive algorithm, because every process is allowed to finish the turn it has been given.

Statement B. The size of the time quantum makes no difference to the number of context switches; it changes only the order in which the processes run.

Statement C. A process that has not finished when its quantum expires goes to the back of the ready queue, behind any process that arrived while it was running.

Model Answer

Statement C is the correct one, statement A confuses finishing a turn with finishing a burst and so classifies round robin as the opposite of what it is, and statement B has the truth exactly reversed, because the quantum sets the number of context switches and left the order of first turns unchanged on the very workload of this section.

Why C is right: it states both halves of the rule, that an unfinished process is returned to the back of the ready queue, and that any process which arrived during the expired turn is placed ahead of it. That second half is the convention every trace in this chapter uses, and it decides where the bars fall from time 6 onwards.

The fault in A: finishing a turn and finishing a burst are different things. Preemption means the processor is taken from a process before its burst is done, and that is exactly what the expiry of a quantum does. A timer, not the process, ends the turn, which makes round robin preemptive by construction.

A corrected: round robin is a preemptive algorithm, because a process that still has work when its quantum expires loses the processor and is returned to the ready queue unfinished. Only first come first served and non-preemptive shortest job first let a chosen process run its burst to the end.

The fault in B: the same four processes give nine slices and 8 handovers at a quantum of 2, and five slices and 4 handovers at a quantum of 4. Doubling the quantum halved the switches, since 8 divided by 4 is 2. Meanwhile the order in which the four processes first reached the processor was P1, P2, P3, P4 in both runs, unchanged.

B corrected: the size of the quantum sets the number of context switches directly, because a burst longer than the quantum is cut into as many turns as it needs. What the quantum changes least is the order of first turns, which follows the arrival order while every process is still waiting for its first slice.

Why A is the dangerous one: a reader who believes A draws one bar per process and produces a chart identical to first come first served, so every completion time, every waiting time and both averages come out wrong together. B misleads about cost; A destroys the schedule itself.

Two of the three statements can be tested against a chart in under a minute, and the third is the classification the other two depend on.

The Master Key: *Round robin is preemptive and its quantum fixes the number of context switches, which is the opposite of what both faulty statements claim.*

HOT Question 62

Every evening the Malongwe telecentre runs twenty machines on one processor, and every keystroke has to feel immediate. Make the turn too short and the evening goes on swapping between people. Make it too long and the twentieth person notices.

A machine serves twenty interactive users through one processor under round robin. Argue how the time quantum should be chosen there, and reach a judgement.

Model Answer

Choose the quantum so that a typical interactive burst finishes inside one turn and the last of the twenty users still reaches the processor quickly, which on twenty users means a quantum of a few tens of milliseconds, and never a quantum chosen as a bare number without checking both of those conditions against the machine it will run on.

The first condition, one turn per request: an interactive burst is short, a few milliseconds of work between one keystroke and the next screen update. If the quantum comfortably exceeds that burst, the request is answered within a single turn and the user waits once. If the quantum is smaller, the same request is cut into several turns and the user waits several times over for one reply.

The second condition, the last user in the round: with twenty users all ready, nineteen turns stand ahead of the last of them. At a quantum of 100 milliseconds that user waits 19 times 100, which is 1900 milliseconds, nearly two seconds before anything appears. At 20 milliseconds the same wait is 380 milliseconds, and at 10 milliseconds it is 190.

The floor underneath both conditions: a handover is not free. If one costs a millisecond, then at a quantum of 10 milliseconds the processor gives up 1 millisecond in every 11, and at a quantum of 1 millisecond it gives up 1 in every 2, so half the machine disappears into changing process.

Why a single number is not an answer: the same 20 milliseconds is generous where a handover costs a few microseconds and ruinous where it costs several milliseconds, and it is right for typing and wrong for a screen redraw. The quantum belongs to the workload and the machine together, never to the algorithm alone.

The judgement: set the quantum just above the burst that most interactive requests need, then check that the number of users multiplied by the quantum keeps the last of them inside about half a second, and lift it if the

handover cost is large. On twenty users that lands near 20 milliseconds, and the rule, not the figure, is what transfers to the next machine.

What it costs when the rule is broken: too small and throughput falls while everybody still feels fast, because the processor is busy handing over rather than working. Too large and the machine drifts towards first come first served, one long request holds the round, and the twentieth user waits seconds for a keystroke.

The quantum is the one parameter in this chapter that has to be measured on the machine it will run on rather than argued from first principles.

The Master Key: *Set the quantum a little above the typical interactive burst and far above the cost of a handover, then confirm that the last user in the round is still served quickly.*

HOT Question 63

The Ipole health centre stopped using one queue. It now runs three, a quick one for dressings, a longer one for consultations, and a slow one for the rest. Nobody has asked yet whether the extra bother pays.

A multilevel feedback queue is far more complex than round robin. Explain how it works, argue whether that complexity is justified, and reach a judgement.

Model Answer

The complexity is justified on a general purpose machine whose workload mixes interactive and compute bound processes and cannot be predicted in advance, and it is not justified anywhere the workload is known and uniform, because everything a multilevel feedback queue buys is the ability to discover, at run time, which of those two kinds each process is.

The mechanism: several ready queues are kept, each with its own priority and its own quantum, the highest queue having the shortest. A new process enters at the top. A process that uses a whole quantum without giving the processor up is moved down one level, where it will next receive a longer turn less often, and the bottom queue is usually served first come first served.

Its simpler cousin: a multilevel queue also keeps several queues, but assigns each process to one of them permanently, by kind, so that system, interactive and batch processes never mix. It is cheaper and it cannot adapt, and a process placed in the wrong queue at creation stays there for life.

What demotion is really measuring: a process that surrenders the processor before its quantum expires was waiting on a person or a device, and one that runs a whole quantum out was computing. The algorithm separates interactive from compute bound work without being told which is which, and without the burst length shortest job first cannot obtain.

The case against, put fairly: there are four things to choose, the number of queues, the quantum of each, the rule that demotes and the rule that promotes, and a poor choice is worse than plain round robin. The scheduler runs at every handover, so its own cost is paid constantly, and its behaviour is hard to predict.

Why one quantum cannot serve both kinds of work: the same four processes gave an average response time of 1.5 units at a quantum of 2 and 3.25 at a quantum of 4, while the average waiting time moved the other way, from 5.0 to 4.5. One quantum is either good for response or good for the longer work. Several queues with different quanta answer exactly that.

The judgement: use it on a general purpose machine running a workload nobody declared in advance, the case a desktop or a shared server presents every day, and prefer round robin or a static priority scheme on an embedded controller, a batch installation with declared run times, or anywhere the response must be guaranteed rather than merely good on average.

The complexity is not decoration; it is the price of learning at run time what shortest job first has to be told in advance.

The Master Key: *A multilevel feedback queue earns its complexity only where the workload is mixed and unknown, and wastes it wherever the workload is already understood.*

Process synchronisation: race conditions, semaphores and the bounded buffer

This section is about what happens when two processes touch the same thing at the same time. It shows a race condition as an interleaving, states the three requirements any solution must meet, and puts semaphores where they belong, which is here and not under deadlock.

BINDER Question 64

At the Kirando fish landing two clerks share one tally sheet for crates. Each reads the total, adds his own load and writes it back. Both are careful, both are honest, and the count still comes out short.

Define a race condition, and explain why the result of a race depends on the order in which the instructions of two processes interleave.

Model Answer

A race condition is a situation in which the result produced by two or more processes depends on the order in which their instructions happen to interleave, so the same processes working on the same data can give different answers on different runs.

Why one addition is three instructions: A processor cannot add one to a value held in memory in a single indivisible step. It loads the value into a register, adds to the register and stores the register back, and the switch from one process to another may fall between any two of those three steps.

Reading the two orders: The counter starts at 5 and each process adds one. In the upper order P completes all three of its steps before Q begins, so Q reads 6 and the counter ends at 7. In the lower order both read 5, both compute 6, and the second write puts 6 back over 6.

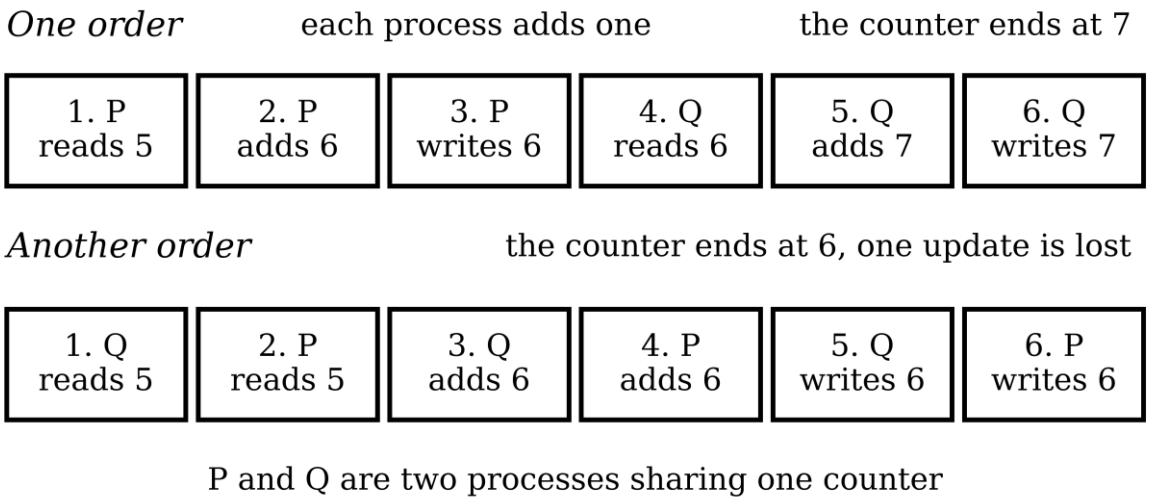


Figure 3.24: *The same six steps in two orders. Only the order changes, and the order decides the answer.*

One update is overwritten: Nothing is corrupted and no instruction misbehaves. The lower order loses an increment because Q read the counter before P had written its new value, so Q wrote back a total that was already out of date. This is a lost update, the commonest form a race takes.

The order is not the programmer's to choose: Both orders are legal. The processor may be taken away from a process between any two instructions, and which order actually happens depends on the timing of that moment, so the fault can hide through a thousand correct runs and appear on the next one.

What puts data at risk: Two things together: the data is reachable by more than one process, and at least one of them writes to it. Processes that only read the same value can never race, which is why a shared count is dangerous and a shared constant is not.

A race condition is a property of the interleaving, not of either process examined on its own.

The Master Key: *If changing the order changes the answer, the code was never safe to share.*

BINDER Question 65

The ramp at Karema jetty takes one lorry at a time, and a second waits at the top of the slope. The rule is easy to say and hard to enforce, because when the ramp clears, somebody still has to decide who goes next.

State the critical-section problem, and state and explain the three requirements that any solution to it must satisfy.

Model Answer

The critical-section problem is the problem of designing a protocol by which processes that share data can guarantee that no two of them are executing their critical sections at the same time, and any solution must satisfy three requirements, mutual exclusion, progress and bounded waiting.

The four parts of the code: A process that touches shared data has an entry section in which it asks to go in, a critical section in which it works on that data, an exit section in which it lets others in, and a remainder section of ordinary work, after which it comes round again.

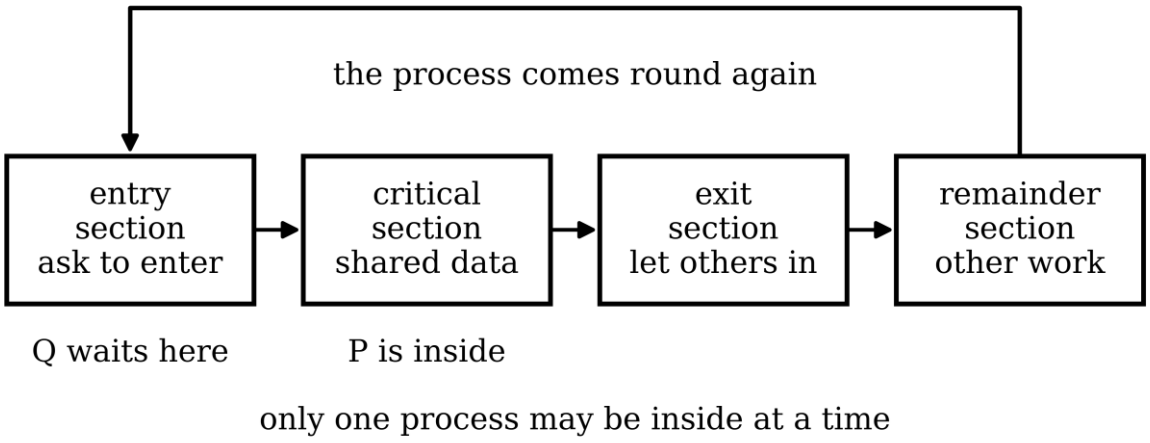


Figure 3.25: *The four sections a process passes through, with one process inside and one held at its entry section.*

Mutual exclusion: If one process is executing in its critical section, no other process may be executing in its own critical section on the same data. This is the requirement the problem exists for, and it is the one that most attempted solutions do get right.

Progress: If no process is inside its critical section and one or more wish to enter, only those not in their remainder section may take part in choosing which goes next, and the choice cannot be put off indefinitely. A protocol that leaves a free critical section unentered has failed on progress.

Bounded waiting: There must be a limit on how many times other processes may enter their critical sections after a process has asked to enter and before it is let in. Without that limit one process can be passed over again and again while the others take turns, which is starvation.

What no solution may assume: Nothing may be assumed about the relative speeds of the processes or about how many processors there are, because a protocol that works only while the two run at similar speeds will fail on the machine it was not written for.

Mutual exclusion on its own is not a solution, since a protocol that admits nobody at all satisfies it perfectly.

The Master Key: *One process inside, a decision always reached, and nobody shut out for ever.*

BINDER Question 66

At the Ikola godown four handcarts serve the whole quay, and the storekeeper hangs four tokens on a nail. Take a token, take a cart, and hang the token back when you finish. The office key hangs there too, under a different rule.

Explain what a semaphore is, distinguish a counting semaphore from a binary semaphore, describe the two operations defined on a semaphore, and distinguish a mutex from a binary semaphore.

Model Answer

A semaphore is an integer variable shared between processes that may be inspected and changed only by two indivisible operations, wait and signal, and it is counting when it may take any value from zero upwards and binary when it may hold only zero or one.

The wait operation: Wait decrements the semaphore, and if no unit is free the calling process is held until one is. The test and the decrement are indivisible, so no second process can slip between them and take the unit the first has just claimed.

The signal operation: Signal increments the semaphore and, if any process is being held on it, releases one of them. A process that has finished with a resource must signal, and a wait left without its matching signal holds the next process for ever.

A counting semaphore: It is initialised to the number of identical units of a resource, four handcarts or five buffer places, and its value is the number of units still free. When it reaches zero, the next process to wait on it is held until a unit comes back.

A binary semaphore: It holds only zero or one and is initialised to one. The process that finds it at one takes it down to zero and enters its critical section, and every later arrival is held at its wait until that process signals, so a binary semaphore enforces mutual exclusion.

A mutex is owned: A mutex also takes two values, but the process that locks it is recorded as its owner and only that owner may unlock it. A binary semaphore has no owner, so any process may signal it, and that is precisely why a semaphore can pass a signal from one process to another while a mutex cannot.

Point of comparison	Binary semaphore	Mutex
Values it may hold	Zero or one	Locked or unlocked
Owner	None is recorded	The process that locked it
Who may release it	Any process may signal it	Only its owner may unlock it
What it is used for	Mutual exclusion, or signalling between two processes	Mutual exclusion only
Value it starts at	One as a lock, zero as a signal	Unlocked

Table 3.19: *A binary semaphore and a mutex compared. Ownership is the distinction that decides which one you may use.*

Used carelessly, semaphores can bring a set of processes to a permanent halt, a failure called deadlock, which the next section takes up in full.

The Master Key: *A semaphore counts what is free, and a mutex remembers who is holding it.*

REAL Question 67

At Kipili the boarding count for the lake boat shows on one shared screen. One clerk has a group of five to add, the other a group of three, and each reads, adds and writes back. Three different totals can end up there.

Two processes add to a shared counter as set out below. Show the interleavings of their steps that produce a wrong total, and state every final value the counter can hold.

The counter starts at 40.

P reads the counter, adds 5 to the value it read, then writes that value back.

Q reads the counter, adds 3 to the value it read, then writes that value back.

Each step is indivisible, and a switch between the two processes may fall between any two steps.

Model Answer

The counter can finish at 48, 45 or 43, and only 48 is right: of the twenty possible interleavings of the six steps, two produce it and the other eighteen lose one of the two additions.

Why there are twenty orderings: Each process keeps its own three steps in order, so an interleaving is settled by choosing which three of the six positions belong to P. That is 6 times 5 times 4, divided by 3 times 2 times 1, which is 20 orderings in all.

The two orderings that are right: P reads, adds and writes, and only then does Q read, add and write: Q reads 45 and finishes at 48. The other correct order is the mirror of it, Q first and P second, where P reads 43 and finishes at 48 as well.

What the correct orders have in common: In each of them one process has written before the other reads. The moment a read happens before the other process has written, the value read is already stale, and the addition made to it will be carried out on an out of date total.

The nine orderings that finish at 45: Both processes read 40 and P writes last. A representative ordering is P reads, Q reads, Q adds, Q writes, P adds, P writes. P writes 40 plus 5, which is 45, over the 43 that Q had just left there, so the addition of 3 is lost.

The nine orderings that finish at 43: Both processes read 40 and Q writes last. A representative ordering is P reads, P adds, Q reads, Q adds, P writes, Q writes. Q writes 40 plus 3, which is 43, over the 45 that P had just left there, so the addition of 5 is lost.

Counting the wrong orderings: Twenty orderings less the two correct ones leaves 18 that give a wrong total, and they divide evenly, 9 finishing at 45 and 9 at 43. Nine in ten of the possible orderings of this pair of processes are wrong, which is not a rare accident.

How large the error is: The total is short by whatever the overwritten process was adding. 48 less 45 is 3, the addition Q lost, and 48 less 43 is 5, the addition P lost. The error is never the size of the counter, only the size of one update, which is why it passes unnoticed.

Putting the three steps of each process inside a critical section removes 18 of the 20 orderings and leaves only the two that give 48.

The Master Key: *A lost update is not a damaged number but a correct number written on top of a newer one.*

REAL Question 68

In the Mishamo sorting shed a bench between the leaf man and the packer holds five trays and no more. One fills, one empties, and neither watches the other. Three counts keep the bench honest, and dropping any one breaks it differently.

A producer and a consumer share a buffer of five places. Name the semaphores the solution needs, give the initial value of each, and explain exactly what goes wrong if any one of them is left out.

Model Answer

Three semaphores are needed: empty, a counting semaphore initialised to 5, full, a counting semaphore initialised to 0, and mutex, a binary semaphore initialised to 1, and leaving out any one of the three breaks the solution in its own particular way.

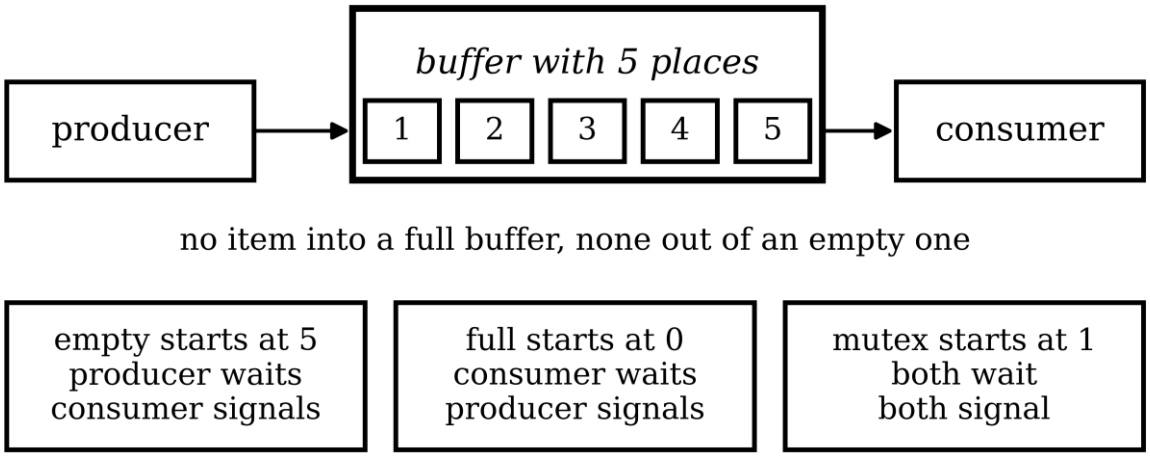


Figure 3.26: *A producer, a consumer, a buffer of five places, and the three semaphores that hold the arrangement together.*

What empty counts: Empty counts free places. The producer waits on it before placing an item and the consumer signals it after taking one away, so it starts at 5 because all five places are free. When it reaches zero the producer is held, which is the buffer telling it there is nowhere to put anything.

What full counts: Full counts items waiting to be taken. The consumer waits on it before removing an item and the producer signals it after placing one, so it starts at 0 because the bench is bare. The two counts always total the number of places: with three items present, empty is 2 and full is 3.

What the mutex protects: The mutex guards the buffer structure itself, the position at which the next item goes and the position from which the next is taken. It starts at 1 because exactly one process may be manipulating those positions at a time, and both parties wait on it and both signal it.

Semaphore	Type	Starts at	Waited on by	Signalled by
empty	Counting	5	Producer	Consumer
full	Counting	0	Consumer	Producer
mutex	Binary	1	Both	Both

Table 3.20: *The three semaphores of the bounded buffer, with the initial value of each and the party that waits on it.*

Leave out empty: Nothing now holds the producer back when every place is taken. It writes a sixth item over a place whose item the consumer has not yet collected, so an item is lost silently and the consumer never learns that anything went missing. This is overflow of a bounded buffer.

Leave out full: Nothing holds the consumer back when the bench is bare. It removes from an empty place and takes whatever that place last held, so it either consumes rubbish or consumes the previous item a second time. This is underflow, and it is a different failure from overflow.

Leave out the mutex: The counts stay right and the buffer neither overflows nor underflows, but nothing keeps the two apart while they update the shared positions. Add a second producer and both can read the same insertion position, write into the same place and lose an item. The failure here is a race condition on the buffer structure, not a shortage of places.

Each of the three does one job, so each omission fails in its own way: overflow without empty, underflow without full, and a race without the mutex.

The Master Key: *Two counting semaphores count places and items, and the mutex protects the counting itself.*

REAL Question 69

A printed handout lies on a bench outside the Katumba dispensary, three lines about locks typed in a neat column. One line is exactly right. The other two are the mistakes almost everyone makes here, and each is the reverse of the other.

The handout carries the three statements below. Identify the one that is correct, explain the fault in each of the other two, and give a corrected version of each.

A. A mutex has an owner, and only the process that locked it may unlock it.

B. A semaphore always makes a waiting process busy wait, spinning on the processor until the value changes.

C. A binary semaphore and a mutex are the same thing under two names, so either of them may be released by any process.

Model Answer

A is correct, and B and C are both wrong: a semaphore need not busy wait at all, because a blocking implementation puts the waiting process on a queue, and a binary semaphore is not a mutex, because a mutex is owned and a binary semaphore is not.

Why A is right: Ownership is recorded when the lock is taken, so the system can refuse an unlock attempted by anybody else. That refusal turns a common programming mistake into an error the machine reports rather than a lock silently released by a process that never held it.

The fault in B: It mistakes one implementation for the definition. A semaphore is an integer with two indivisible operations, and how a process waits on it is a matter of implementation, not of what a semaphore is. Nothing in the definition mentions the processor at all.

What a blocking semaphore actually does: Wait places the process on a queue belonging to that semaphore and hands the processor to another process, so the waiting process consumes no processor time whatever. Signal takes one process off that queue and returns it to the ready queue, where the scheduler can dispatch it.

Where the belief in B comes from: The simplest way to present a semaphore is a loop that reads the value over and over until it changes, which is a spinlock. That version does exist and is used for very short waits, but it is one implementation of the idea and not the idea itself.

B corrected: A semaphore may be implemented with busy waiting, but a blocking implementation puts the waiting process on the semaphore queue and gives the processor to another process.

The fault in C: Both hold two values and both can enforce mutual exclusion, so the resemblance is real, but a mutex records an owner and a binary semaphore does not. The second half of the statement is the damage, because releasing a mutex you do not hold is exactly what ownership exists to stop.

C corrected: A binary semaphore and a mutex both take two values, but only the mutex has an owner, and only the process that locked it may unlock it, while a binary semaphore may be signalled by any process.

Two of the three lines needed correcting, and they were wrong in opposite directions, one taking away the queue a semaphore has and the other taking away the owner a mutex has.

The Master Key: *A mutex is owned and a semaphore is not, and neither of them has to spin.*

HOT Question 70

At the Mtowisa mill a driver leaves his lorry running while the weighbridge clears, because starting the engine cold costs him more than two minutes of idling. Somewhere past two minutes that reasoning quietly turns around.

A spinlock keeps a processor busy doing nothing while it waits for a lock to be released. Argue whether busy waiting is ever the right choice, and reach a judgement.

A context switch on this machine costs about 10 microseconds.

The critical section the lock protects runs for about 2 microseconds.

Model Answer

Busy waiting is right when the expected wait is shorter than the cost of blocking and resuming, and wrong otherwise, so on this machine a spinlock pays while the wait stays under about 20 microseconds and there is more than one processor to run the lock holder.

What spinning costs: Every cycle spent reading a lock that has not changed is a cycle no process uses for work. On a loaded machine that is throughput given away, and the loss grows with the length of the wait, because the spinner keeps the processor for the whole of it.

What blocking costs: Blocking is not free either. The process must be moved out of running and onto the semaphore queue, its context saved, another process dispatched, and the whole exchange reversed when the lock comes free. That is two context switches, 10 plus 10, which is 20 microseconds before any work resumes.

The comparison in numbers: Spinning through this critical section costs about 2 microseconds of processor time. Blocking costs 20, ten times as much, so a process that spins here finishes sooner and leaves the machine better off. The break-even expected wait is 20 microseconds, the price of the pair of switches.

Why one processor changes the answer: On a single processor the holder of the lock is not running while the spinner spins, because the spinner has the only processor there is. It burns the very resource the holder needs to finish and release, and nothing improves until the spinner is preempted. That is close to indefensible.

Where spinning is genuinely right: Inside the kernel on a multiprocessor, guarding a short update to a shared structure, and in code that is not permitted to block at all. There the alternative is not a cheaper wait but no wait that is allowed, so a spinlock held for a handful of instructions is the correct construct.

Busy waiting never disappears entirely: The indivisible test-and-set on which wait itself is built spins at the hardware level for a few cycles. Blocking implementations do not abolish busy waiting; they push it down to a level where the wait is a few instructions long.

The judgement: Busy wait when two conditions hold together, that the expected wait is reliably shorter than a pair of context switches and that another processor can run the lock holder meanwhile; block in every other case. The verdict turns on the length of the critical section, and a spinlock on a single processor fails both tests at once.

The question is never whether spinning wastes time, since it always does, but whether it wastes less than the switch it saves.

The Master Key: *Spin only when the wait is shorter than the two switches that blocking would cost.*

HOT Question 71

The Karema boatyard keeps its one caulking iron in a locked cage, and the storekeeper holds the key. He can refuse to lend it twice over, and he does. What he cannot do is stop a man reaching over the counter for it.

Argue whether responsibility for correct synchronisation should rest with the programmer or with the operating system, and reach a judgement.

Model Answer

The operating system is responsible for providing synchronisation that works and the programmer is responsible for using it, and the division falls exactly where knowledge of the program falls, because no kernel can see which data a program shares or where its critical sections begin and end.

What only the system can supply: Indivisibility is not something a programmer can build alone. Making a test and a decrement inseparable needs a hardware instruction and kernel support, and so does a queue of waiting processes that the scheduler honours. Semaphores and mutexes rest on that foundation.

The case for the system taking the responsibility: For anything the system itself owns, it already does. Two processes writing to one file are serialised by the file system whether either asked or not, and a device is handed out by its driver, not by agreement between the programs that want it.

Where the power of the system stops: A counter in shared memory looks to the kernel like any other memory. Which variables are shared and which statements form one indivisible unit of work are facts about the meaning of the program, and nothing in the kernel can read that.

The case for the programmer taking the responsibility: Placing every wait and its matching signal, keeping the critical section short enough that others are not held needlessly, and choosing a mutex where there is an

owner and a semaphore where there is not, are design decisions. Only the author knows what the program means.

Why leaving it wholly to the programmer is hard: The failures are silent. Of the twenty orderings seen earlier, 18 are wrong, yet a program can run correctly for months, because a machine that never happens to interleave badly reports nothing at all, and no amount of running the program will show the fault.

The middle ground that has actually worked: Higher level constructs move the burden without moving the responsibility. A monitor puts the mutual exclusion into the language, so the compiler places the lock around every procedure and the programmer cannot forget it. The tool is still the system's, but it is made much harder to misuse.

The judgement: Providing the tools is the operating system, guaranteeing their correct use is the programmer, and the two must not be confused. Judge the system on whether its operations are indivisible and hard to misuse, and the programmer on whether every access to shared data lies inside a critical section. The condition that decides it is knowledge, and only the author of the program has it.

A system that supplied no synchronisation would make correctness impossible, and a system that promised it outright would be promising something it cannot see.

The Master Key: *The system can make a lock impossible to break, but only the programmer knows what needs locking.*

Deadlock: the four conditions, resource graphs and the Banker's algorithm

This section teaches deadlock as it actually works, which means teaching the four conditions as conditions and prevention as their negation. It draws the graphs that make a circular wait visible, runs the Banker's algorithm to a safe sequence, and separates detection from avoidance.

BINDER Question 72

At the Laela maize mill one grinder and one weighing scale serve forty farmers. Two arrive together, one takes the grinder, one takes the scale, and neither will let go. Nothing is milled that morning, and four things had to be true first.

State the four necessary conditions for deadlock and explain each one. State what follows from the word necessary.

Model Answer

Deadlock requires four conditions to hold at the same moment, mutual exclusion, hold and wait, no preemption and circular wait, and because every one of them is necessary, making any single one impossible makes deadlock impossible.

Mutual exclusion: at least one resource is held in a non-shareable mode, so only one process may use it at a time and any other process that asks for it must wait. A tape drive and a printer behave this way. A file opened for reading only does not, because many processes can read it at once.

Hold and wait: a process is holding at least one resource and is at the same time waiting to acquire further resources that other processes are holding. It keeps what it already has while it queues for what it still lacks, and it gives nothing back while it queues.

No preemption: a resource cannot be taken away from the process holding it. It is released only voluntarily, by that process, once that process has finished with it. The operating system has no power to reclaim it in the meantime, however long the reclaiming would save.

Circular wait: a closed chain of processes exists in which each one is waiting for a resource held by the next, and the last in the chain is waiting for a resource held by the first. Two processes are enough to close such a chain, and the chain may be far longer.

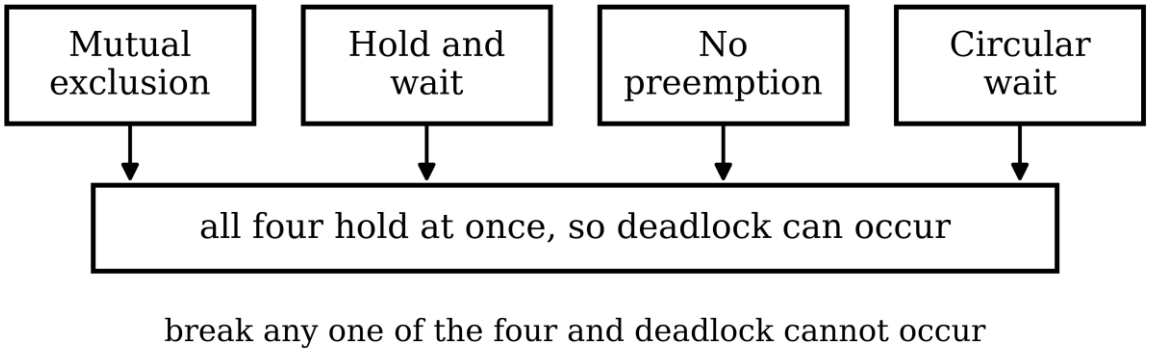


Figure 3.27: *The four conditions, drawn feeding one outcome, because deadlock follows from all four together and never from one of them alone.*

What necessary means: all four hold together or there is no deadlock, so a system exhibiting three of them is not deadlocked and never will be until the fourth appears. That is why they are called conditions and never strategies. Breaking any one of the four is sufficient on its own.

Name all four whenever deadlock is asked about, because an account that stops at three has not described a deadlock at all.

The Master Key: *Mutual exclusion, hold and wait, no preemption and circular wait, all four at once or no deadlock.*

BINDER Question 73

The Galula co-operative chalks a wall chart each morning. An arrow from a member to a machine means he wants it; the other way means he holds it. Tuesday the arrows closed a loop, and whether that mattered depended on the hullers.

Explain what a resource-allocation graph shows and how its two kinds of edge differ. State when a cycle in one means deadlock and when it does not.

Model Answer

A resource-allocation graph shows which processes hold which resources and which processes are waiting for which, and a cycle in it means deadlock only where every resource type has a single instance, because with several instances per type a cycle is necessary but not sufficient.

What the vertices carry: processes are drawn one way and resource types another, and inside each resource type a dot is drawn for every instance the system owns of it. A type with three identical printers carries three dots, and that count is what decides how a cycle in the graph must be read.

What the two edges mean: a request edge runs from a process to a resource type and says that the process has asked and is waiting. An assignment edge runs from one instance of a resource back to a process and says that the process holds it. A granted request becomes an assignment edge pointing the other way.

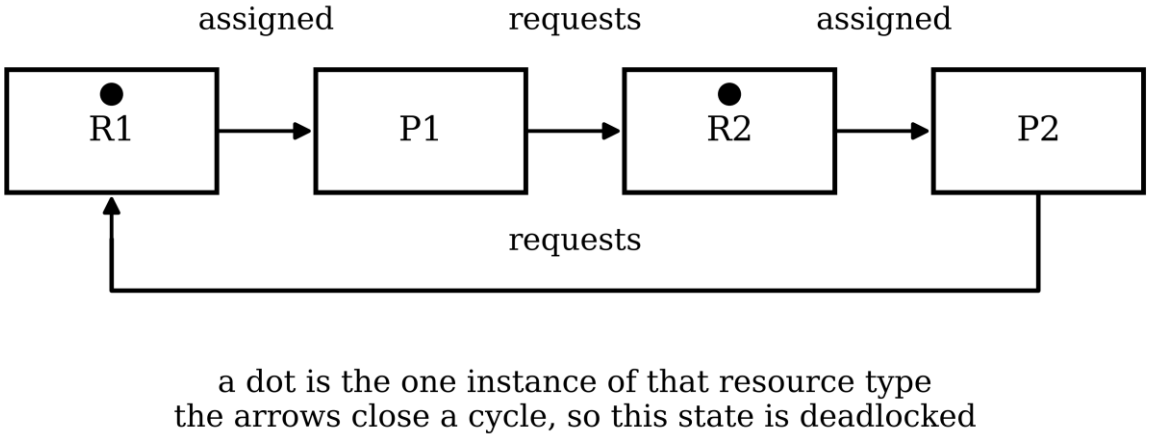


Figure 3.28: *One instance of each type, and the arrows close a loop. P1 holds R1 and wants R2, P2 holds R2 and wants R1, so the state is deadlocked.*

One instance of each type: a cycle is then necessary and sufficient, so finding one is finding a deadlock. Every process in the loop waits on the single instance of a type, and the only process that could release that instance is itself in the loop waiting. Two requests and two assignments close the shortest such loop.

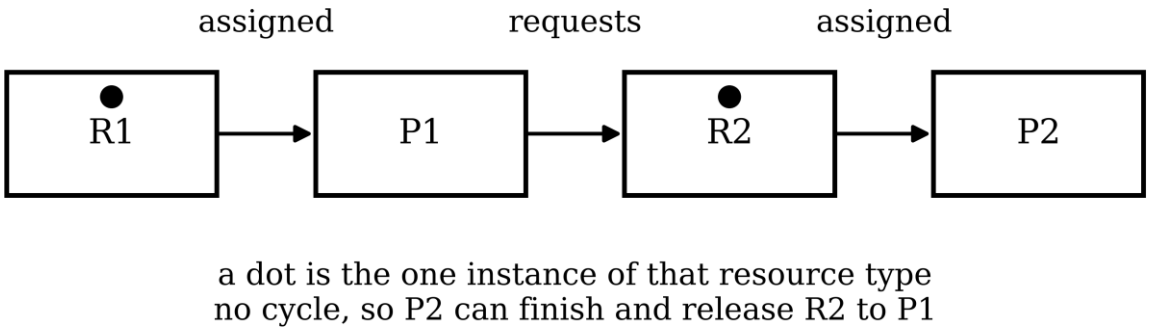


Figure 3.29: *The same graph with the closing request removed. There is no cycle, so P2 runs to completion, releases R2, and P1 gets what it was waiting for.*

Several instances of a type: a cycle is then necessary but not sufficient, so a loop on the chart proves nothing by itself. A process outside the loop may hold another instance of a type inside it, and when that process finishes and releases its instance, a waiting process in the loop is satisfied and the loop breaks.

Count the instances before reading the cycle, because the same loop of arrows is a deadlock in one graph and an ordinary queue in another.

The Master Key: *A cycle is deadlock only where each resource type has one instance; with several it is necessary but not sufficient.*

BINDER Question 74

The Songwe crossing runs on four habits. One clerk turns back a lorry without both permits, one asks where it is going before waving it in, one walks the yard at noon counting lorries that have not moved, and one keeps a tractor.

Distinguish deadlock prevention, avoidance, detection and recovery. State what each one requires to be known in advance, and name the routes recovery can take.

Model Answer

Prevention, avoidance, detection and recovery are four separate strategies, in that prevention removes one of the four conditions by design, avoidance grants a request only when the state that results is still

safe, detection allows deadlock to happen and then looks for it, and recovery is what is done once detection has found one.

Prevention: makes one of the four conditions structurally impossible, so it needs nothing at all known in advance about any process. Its price is paid by every process on every acquisition, forever, whether or not a deadlock was ever going to occur.

Avoidance: needs the maximum claim of every process on every resource type declared before that process starts. Given those claims it grants a request only if the resulting state is safe, and the Banker's algorithm is the standard test of safety.

Detection: needs nothing declared in advance, but needs the operating system to keep the allocation and request matrices and run a search over them periodically. Students are often taught a list of deadlock strategies with detection left off it, which is why detection and avoidance are so often confused; they are separate, and detection acts after the fact.

Recovery: takes one of four routes once a deadlock is confirmed. Terminate every deadlocked process at once, terminate them one at a time until the cycle breaks, preempt resources from a chosen victim and give them to another process, or roll a process back to an earlier checkpoint and restart it from there.

Strategy	What it does	What must be known in advance	When it acts
Prevention	Negates one of the four conditions by design	Nothing about any process	Before, always
Avoidance	Refuses a request that would leave an unsafe state	Every maximum claim	At each request
Detection	Searches the allocation state for a deadlock	Nothing about any process	After the fact
Recovery	Terminates, preempts or rolls back to break it	A choice of victim	After detection

Table 3.21: *The four strategies separated by what each does, what each needs declared in advance, and when each acts.*

Detection and recovery are a pair, in that detection without recovery only names the problem and recovery without detection has nothing to act on.

The Master Key: *Prevention designs deadlock out, avoidance steers around it, detection finds it afterwards, and recovery is what breaks it.*

REAL Question 75

At the Chiwanda tea factory the manager tried four rules in one season. Share the weighbridge, issue every tool at the gate or none, let a foreman take back an idle trolley, number the sheds one to nine. Each rule cost him something.

For each of the four necessary conditions, state the negation that prevents deadlock and the price that negation charges.

Model Answer

Prevention works by making one of the four conditions impossible, so every prevention strategy is the negation of a condition and never the condition itself, and each of the four negations charges a price in utilisation, in throughput or in the freedom of the programmer.

Where the confusion begins: students are often taught, wrongly, that mutual exclusion, hold and wait and no preemption are themselves strategies for preventing deadlock. They are three of the four conditions under which deadlock becomes possible, so a system that has them has the makings of a deadlock rather than a defence against one.

Negating mutual exclusion: make the resource shareable wherever its nature allows it, as a read-only file or a spooled printer queue is shareable. The price is reach, in that many resources cannot be shared at all without corrupting them, so this negation removes only part of the problem and never the whole of it.

Negating hold and wait: require every resource a process will need to be claimed at once, or none of them granted, so a process never holds one while queueing for another. The price is utilisation, because resources sit idle inside a process that is not yet using them, and a process needing many resources may wait a very long time for them to be free together.

Negating no preemption: allow a held resource to be taken back and returned to its owner later, so a process refused a new resource loses the ones it already holds and restarts when all are available. The price is that only resources whose state can be saved and restored qualify, such as registers and memory; a half-finished print job cannot be preempted safely.

Negating circular wait: number the resource classes and require every process to acquire them in ascending order, which makes a closed chain arithmetically impossible. The price falls on the programmer, who must know the numbering and honour it, and on any process whose natural order of use differs, since it must take a low-numbered resource early and hold it longer than it needs.

Condition	The negation that prevents deadlock	The price it charges
Mutual exclusion	Make the resource shareable where its nature allows it	Reaches only the resources that can be shared
Hold and wait	Require every resource to be claimed at once, or none	Idle resources and long waits before starting
No preemption	Allow a held resource to be taken back and returned later	Only works where state can be saved and restored
Circular wait	Number the resource classes and require ascending acquisition	Binds the programmer and holds resources early

Table 3.22: *Each condition against the negation that prevents it and the price that negation charges.*

Circular wait is the condition prevention attacks most often in real systems, because ascending order is cheap to state and cheap to check.

The Master Key: *Prevention is the negation of a condition, so state the negation and never the condition itself.*

REAL Question 76

The Ifisi savings group keeps one tin and five members with loans running. The treasurer hands out nothing unless she can name an order in which every member still repays in full. Some mornings she can name one, and some mornings she cannot.

For the system described, compute the Need matrix, determine whether the state is safe, and give a safe sequence with the working shown.

Three resource types, A, B and C. Five processes, P0 to P4.

Allocation, as A B C: P0 is 0 1 0, P1 is 2 0 0, P2 is 3 0 2, P3 is 2 1 1, P4 is 0 0 2.

Maximum, as A B C: P0 is 7 5 3, P1 is 3 2 2, P2 is 9 0 2, P3 is 2 2 2, P4 is 4 3 3.

Available, as A B C: 3 3 2.

Model Answer

The state is safe, and one safe sequence is P1, P3, P4, P0, P2, obtained by subtracting Allocation from Maximum to get Need and then repeatedly finding a process whose Need is met by the resources in hand.

Need is Maximum minus Allocation: taken cell by cell, this gives Need of 7 4 3 for P0, 1 2 2 for P1, 6 0 0 for P2, 0 1 1 for P3 and 4 3 1 for P4. Need is what a process may still demand, so it is what the safety test must be able to meet.

Process	Allocation A B C	Maximum A B C	Need A B C
P0	0 1 0	7 5 3	7 4 3
P1	2 0 0	3 2 2	1 2 2
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1

Table 3.23: *Allocation and Maximum as given, with the Need matrix obtained by subtracting one from the other cell by cell.*

The rule the search applies: start with Work equal to Available, which is 3 3 2. Find any unfinished process whose Need is no greater than Work in every column, assume it finishes, and add the whole of its Allocation to Work. Repeat until every process is marked finished or none will fit.

P1 goes first: its Need of 1 2 2 is met by Work of 3 3 2 in all three columns, so P1 can be assumed to finish and return its Allocation of 2 0 0. Work becomes 5 3 2. P0 could not have gone first, because its Need of 7 4 3 exceeds Work in every column.

P3 then P4: P3 has Need 0 1 1 against Work 5 3 2, so it finishes and returns 2 1 1, making Work 7 4 3. P4 has Need 4 3 1 against that, so it finishes and returns 0 0 2, making Work 7 4 5.

P0 then P2 finish the sequence: P0 has Need 7 4 3 against Work 7 4 5, which fits exactly in the first two columns, so P0 finishes and returns 0 1 0, making Work 7 5 5. P2 has Need 6 0 0 against that and finishes last, returning 3 0 2 and leaving Work at 10 5 7, which is every resource the system owns.

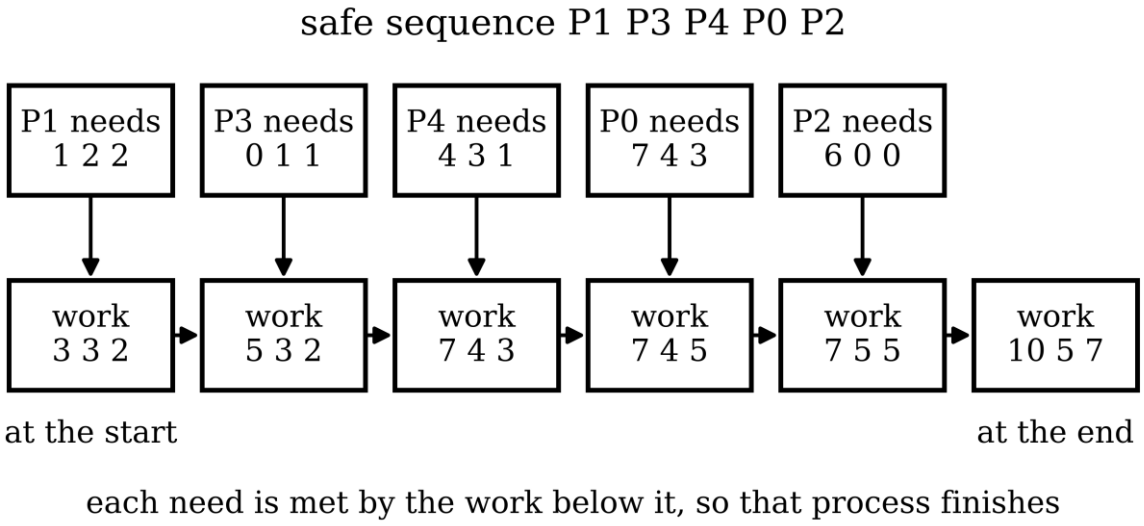


Figure 3.30: *The safe sequence with Work growing at each step, because what a finishing process hands back is exactly what lets the next one finish.*

What safe means here: a state is safe when at least one such sequence exists. It is a guarantee that even if every process now demanded its full Maximum, some order exists in which all of them are served, so granting from this state cannot strand the system.

A different search order may find a different safe sequence, and one sequence is all that safety requires.

The Master Key: *Need is Maximum minus Allocation, and a state is safe when one order exists in which every Need is met from the Work in hand.*

REAL Question 77

At Ipinda the coffee store lends drying trays, twelve of them, to three growers who each know the most they will ever ask for. The storeman hands out one more tray, and no order of finishing now works. Nothing has stopped yet.

Show that the state described is unsafe, and explain the difference between an unsafe state and a deadlocked one.

One resource type with twelve identical instances. Three processes, P1, P2 and P3.

Maximum: P1 is 10, P2 is 4, P3 is 9.

Allocation: P1 is 5, P2 is 2, P3 is 3.

Available: 2.

Model Answer

The state is unsafe because no order of finishing exists, in that only P2 can be met from the two instances in hand and the four instances free after it finishes fall short of both remaining Needs, yet nothing is deadlocked, because no process is yet waiting on an instance that another process holds.

Need and Available: Need is Maximum minus Allocation, giving 5 for P1, 2 for P2 and 6 for P3. Ten of the twelve instances are allocated, so Available is 2, and that is the Work the safety search begins with.

Only one process fits: P2 has Need 2 against Work 2, so P2 alone can be assumed to finish. P1 needs 5 and P3 needs 6, and neither is met by the two instances in hand, so neither can be placed first in any sequence.

And then the search stops: P2 finishes and returns its 2, so Work becomes 4. P1 still needs 5 and P3 still needs 6, and 4 meets neither. The search ends with two processes unfinished, which means no safe sequence exists and the state is unsafe by definition.

One grant earlier it was safe: had P3 been holding 2 rather than 3, Available would have been 3, Work after P2 finished would have been 5, and P1 with a Need of 5 would have been met, giving the safe sequence P2, P1, P3. A single granted instance is what removed the guarantee.

Unsafe is not deadlocked: unsafe describes a state, and it means only that the operating system can no longer guarantee that every process will finish. Deadlocked describes an event that has already happened, in which processes hold resources and wait on one another and none can proceed. No process here is waiting on another.

Why the distinction matters: all three of these processes may well run to completion, because a process need not ever claim its full Maximum. Every deadlocked state is unsafe, but most unsafe states never deadlock, which is precisely why avoidance is conservative: it refuses requests that would probably have been harmless.

Avoidance keeps the system inside the safe states, so it refuses far more than it needs to and never has to recover from anything.

The Master Key: *An unsafe state means deadlock has become possible, not that it has happened.*

REAL Question 78

A page of notes goes round the Kiwira staff room, photocopied so often the last line has gone grey. It lists five ways of preventing deadlock, and one case worked through. Three of the five prevent nothing, and the case cannot occur.

Identify everything that is wrong with the list of five strategies and with the worked example, and give the correct account of both.

Deadlock preventive strategies: mutual exclusion, hold and wait, no preemption, semaphores, critical sections.

Worked example: a process holds processor time while it waits for memory to be freed, and a second process holds the memory while it waits for processor time, so the two deadlock.

Model Answer

The list and the example are both wrong, in that three of the five entries are necessary conditions misfiled as cures, the other two are synchronisation tools that cause deadlock more often than they prevent it, the fourth condition is missing altogether, and the example cannot deadlock because processor time is preemptible and is released the moment a process blocks.

Three entries are conditions: mutual exclusion, hold and wait and no preemption are three of the four conditions under which deadlock becomes possible. Listing them as preventive strategies inverts the topic, because prevention is the negation of a condition and never the condition itself, as question 75 sets out in full.

The fourth condition is missing: circular wait is not on the list at all, and it is the condition prevention attacks most often in practice, by numbering the resource classes and requiring every process to acquire them in ascending order. A list of conditions that names three of the four cannot describe a deadlock.

Semaphores and critical sections are tools: they are synchronisation mechanisms, taught in section 8, and what they do is enforce mutual exclusion. Enforcing mutual exclusion supplies the first condition rather than removing it, and two processes each waiting on a semaphore the other holds is the standard way a deadlock comes to be written.

The example cannot deadlock: processor time is preemptible and is not held across a wait. A process that asks for memory it cannot have does not keep the processor; it leaves the running state at once, moves to waiting, and the dispatcher gives the processor to the next ready process. Nothing is held, so hold and wait fails.

What a correct example needs: two non-preemptible resources and two processes, each holding one and requesting the other. A tape drive and a printer will do, or two files locked in opposite order. Only then do all four conditions hold at once, which is what an example of deadlock has to demonstrate.

Correct the list by replacing all five entries with the four negations, and correct the example by choosing resources the operating system cannot take back.

The Master Key: *A process cannot deadlock while holding the processor, because the processor is preemptible and is released the moment the process blocks.*

HOT Question 79

The Laela clinic asks every patient at the gate how long he will need inside. Some cannot say, some guess high so as not to be turned away, and the queue is planned on those guesses. An operating system asks processes the same.

Deadlock avoidance requires every process to declare its maximum resource needs in advance. Argue whether a general-purpose operating system should implement it, and reach a judgement.

Model Answer

A general-purpose operating system should not implement avoidance, and the verdict turns on the declaration rather than on the cost of the algorithm, because a process whose demands are decided at run time cannot state its maximum honestly and the conservative declaration it gives instead makes almost every state look unsafe.

The case for avoidance: it never enters a state from which deadlock is reachable, so nothing is ever killed, rolled back or lost. It is also less brutal than prevention, since it refuses only the individual requests that would endanger safety instead of restructuring how every process acquires everything.

What it demands in return: every process must declare its maximum claim on every resource type before it begins, the number of resource types and instances must be fixed and known, and no process may ever exceed what it declared. None of those three is a small demand on a machine that runs whatever a user starts.

The declaration is the fatal one: an interactive process cannot know how many files it will open or how much memory it will allocate, because those depend on what the user does next. A programmer required to declare will declare the worst case, and worst-case claims inflate every Need until the safety test refuses requests that would have been harmless.

The cost of running it: the safety search runs on every request and touches each process against each resource type for each candidate in the sequence. With eight resource types and two hundred processes that is 320000 cell comparisons per request, on a path taken thousands of times a second. This is a serious objection, but it is the second one.

Where avoidance does belong: a closed system with a fixed process set makes the declaration honest and the process count small. An embedded controller, a hard real-time system and a database transaction manager all qualify, and the last of those runs a safety test in practice under a different name.

The judgement: do not implement avoidance in a general-purpose kernel. Implement it where the process set is fixed and the claims are known truthfully, and elsewhere prevent circular wait by ordering the lock classes, which costs nothing at run time and asks the programmer rather than the process to supply the knowledge.

The algorithm is sound and the arithmetic is affordable; it is the honest maximum claim that a general-purpose machine cannot supply.

The Master Key: *Avoidance fails on general-purpose machines because the declaration it needs is a guess, and a conservative guess refuses safe requests.*

HOT Question 80

The Ifisi welding shop owns one generator and one habit. When two jobs jam the line and nothing has moved for ten minutes, the foreman switches the generator off and on again. In eleven years he has never found out why.

Most production operating systems detect no deadlocks and prevent none, leaving the user to restart. Argue whether ignoring deadlock is defensible, and reach a judgement.

Model Answer

Ignoring deadlock, which is the approach commonly named after the ostrich, is defensible on a general-purpose machine and indefensible on one whose failure costs more than a restart, and the verdict turns on how often deadlock actually occurs weighed against what the machinery to handle it would cost every process all of the time.

What ignoring actually means: no acquisition rules, no maximum claims, no periodic search of the allocation state. If a set of processes deadlocks, the system carries on serving everything else, and the person at the machine ends the offending process or restarts the machine.

The case for ignoring it: deadlock over kernel resources is rare, while the cost of prevention is paid by every process on every acquisition forever, and the cost of detection is a repeated pass over the allocation matrices whether or not anything is wrong. Paying continuously to handle a rare event is a poor trade, and general-purpose machines fail more often for other reasons anyway.

The case against ignoring it: the cost is moved from the designer onto the user, and the failure mode is silent. A hung application looks exactly like a slow one, so the user has no way to tell what has happened, waits, and then destroys unsaved work by force. Nothing in the machine ever records that a deadlock occurred.

What actually decides it: the frequency of the event against the cost of the machinery, and then the cost of a restart. On a desktop or a phone a restart costs minutes and the event is rare, so the trade is sound. On a switch carrying traffic, a controller in a plant or a database serving a bank, a restart costs far more than the detection pass ever would.

Where the line is actually drawn: production kernels ignore deadlock at the kernel level and let subsystems handle it locally, which is why a database engine keeps a wait-for graph, detects cycles among its transactions and aborts a victim, while the kernel underneath it does nothing of the kind. The two decisions are consistent, because the costs differ.

The judgement: ignoring deadlock is defensible for a general-purpose kernel and not defensible for a subsystem that holds long-lived locks or for real-time control. Ignore it where the event is rare and the restart is cheap, and pay for detection and recovery the moment either of those two stops being true.

The approach is a deliberate engineering choice about where cost should fall, not an oversight, and it should be argued as one.

The Master Key: *Ignore deadlock only where it is rare and a restart is cheap, and detect it wherever either of those fails.*

HOT Question 81

At the Songwe river landing two boats are locked bow to bow in a channel too narrow to pass, while a third has waited behind larger boats since dawn. One is stuck for good. The other is only ever next in line.

Compare deadlock with starvation. Argue which of the two is the more serious problem for an operating system to leave unsolved, and reach a judgement.

Model Answer

Deadlock is a set of processes each waiting for a resource another member of that set holds, so none can ever proceed, while starvation is one runnable process repeatedly passed over that could proceed at once if the policy chose it, and starvation is the more serious to leave unsolved because it cannot be detected.

Deadlock stated precisely: a set of two or more processes, each holding at least one resource and waiting for one held by another member of the set. All four conditions hold at that moment, no member will ever proceed without intervention from outside, and the resources they hold are withheld from everything else in the machine.

Starvation stated precisely: a single process is runnable and indefinitely denied a resource that is repeatedly given to others. Nothing whatever prevents it running; the policy simply never selects it, as a stream of shorter arrivals does under shortest job first and a stream of higher-priority arrivals does under priority scheduling.

The differences that matter: deadlock involves a set and starvation a single process. In deadlock the resource is held by a peer and is unavailable; in starvation it is free and is handed elsewhere. Deadlock is permanent by construction, while a starved process may be served the moment the workload thins.

Detectability decides it: deadlock has an exact test, because a cycle in the wait-for graph is a definite structure and the deadlocked state is stable, so the search cannot race the system and report something that has already changed. Starvation has no exact test at all, since a starved process looks identical to one that is about to be served.

What follows from that: declaring starvation requires an arbitrary threshold on waiting time, and no threshold is right for every workload. So a deadlock left unsolved is still findable and breakable at any later moment, while a starvation left unsolved is invisible, and the machine degrades steadily without ever reporting that anything is wrong.

The judgement: starvation is the more serious to leave unsolved, even though deadlock is the more severe failure when it occurs. Deadlock is rare, detectable and recoverable; starvation is common under any priority policy, undetectable, and cured cheaply by ageing, so omitting the cure has no defence.

Deadlock is the worse event and starvation is the worse omission, and an answer that treats the two as the same failure loses both distinctions.

The Master Key: *Deadlock is permanent and detectable, starvation is temporary in principle and undetectable, which is why leaving starvation unsolved is the greater fault.*

Memory management: allocation and fragmentation

This section covers memory before paging arrives. It separates the address a program uses from the address the hardware puts on the bus, runs the four placement policies over one common request stream so their differences show, and measures both kinds of fragmentation.

BINDER Question 82

Itungi port stacks crates for one consignment and numbers them one to forty in the order they were loaded. The hold has bay numbers of its own, and no crate keeps its own number once it is aboard. Something must do the conversion.

Distinguish a logical address from a physical address, explain when address binding can happen, and state what the memory management unit and the relocation register do.

Model Answer

A logical address is the one the processor generates as a program runs, a physical address is the one that reaches the memory unit, and the memory management unit converts the first into the second by adding the relocation register at every reference.

The logical address: is generated by the processor and counted from zero inside the space of the program itself. Every jump and every variable reference in the compiled code is logical, so a program compiles without knowing where in memory it will run.

The physical address: is the one that appears on the memory bus and selects a real location in the memory unit. The program never sees it. Two copies of one program running at once hold identical logical addresses and different physical ones, so they do not collide.

Compile time and load time binding: fix the physical address early. Compile time binding writes absolute addresses into the code, so the program must be recompiled to start anywhere else. Load time binding leaves the code relocatable and the loader fixes the addresses once, after which the process cannot be moved.

Execution time binding: defers the decision to every reference, because the hardware forms the physical address as the logical one is issued. Only this scheme lets a running process be moved, which is what compaction and swapping need: a process written out to the backing store may return somewhere else.

The hardware that performs the conversion: is the memory management unit, which holds a relocation register loaded with the base address of the running process. It adds that base to every logical address the processor issues, so a logical address of 346 in a process based at 14000 reaches memory as 14346.

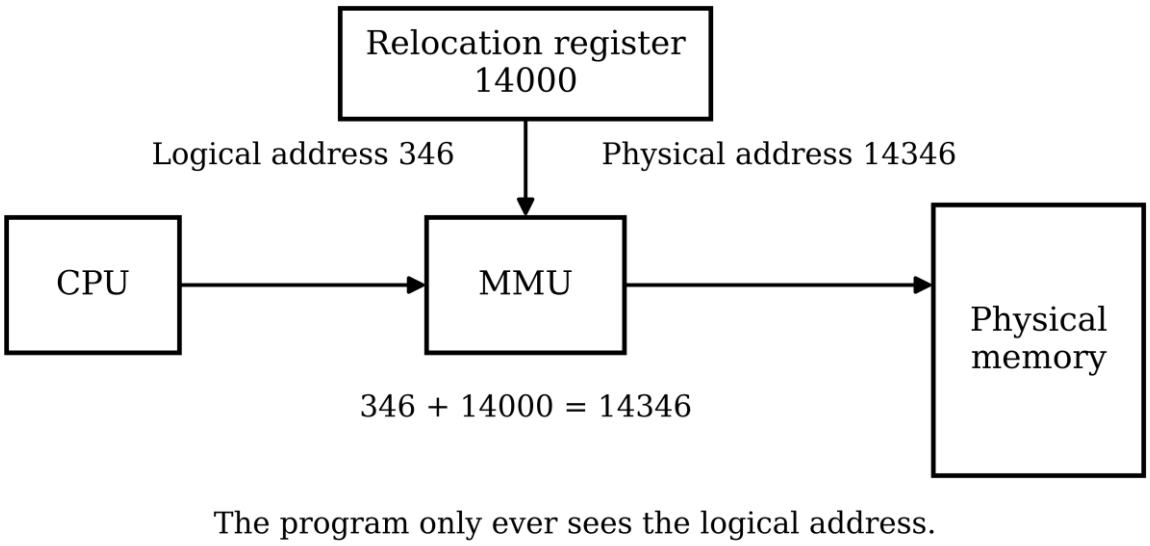


Figure 3.31: *The relocation register adds the base of the running process to every logical address the processor issues.*

Only execution time binding lets a process be moved after it has started, and every technique in this section that shifts memory about depends on it.

The Master Key: *The program counts from zero, the hardware adds the base, and only what the hardware adds can change while the process runs.*

BINDER Question 83

At Igurusi the rice godown is divided into three bays of equal size by concrete walls. A load that fills two thirds of a bay leaves the rest of that bay useless to anyone else. The alternative is to knock the walls down.

Explain fixed partitioning and variable partitioning, and state one advantage and one disadvantage of each.

Model Answer

Fixed partitioning divides memory into partitions whose boundaries are decided before any process arrives, while variable partitioning cuts each partition to the size of the process that asks for it and lets the boundaries move as processes come and go.

How fixed partitioning works: memory is divided at start up into partitions of a chosen size, and one process occupies one whole partition whatever it actually needs. The number of partitions fixes how many processes can be resident, and a process larger than the largest partition cannot run at all.

What fixed partitioning is worth: the advantage is simplicity, because allocation means finding a free partition and the operating system keeps nothing more than a short table. The disadvantage is the space wasted inside each partition, which belongs to the process holding it and cannot be lent to another.

How variable partitioning works: the operating system keeps a list of free blocks and cuts from one of them exactly as much as the arriving process asked for. What is left of that block stays free, and when a process ends its space is returned and merged with any free neighbour.

What variable partitioning is worth: the advantage is that nothing is wasted inside an allocation, since the partition is the size of the request. The disadvantage is that free space breaks into scattered gaps as processes come and go, and the free list must be searched and maintained on every request.

Reading the two columns: three partitions of 400K holding processes of 240K, 180K and 280K leave 160K, 220K and 120K unused, 500K in all, and no other process can reach it. Under variable partitioning the same three processes leave that same 500K free in one block, which a fourth process can take.

Feature	Fixed partitioning	Variable partitioning
Boundaries decided	Before any process arrives	When the process is allocated
Size of an allocation	The whole partition	Exactly what was requested
Waste produced	Internal, inside the partition	External, between partitions
Bookkeeping kept	A short table of partitions	A free list, searched and merged
Largest process it can run	The largest partition	The largest free block

Table 3.24: *Fixed and variable partitioning compared on the five things that separate them.*

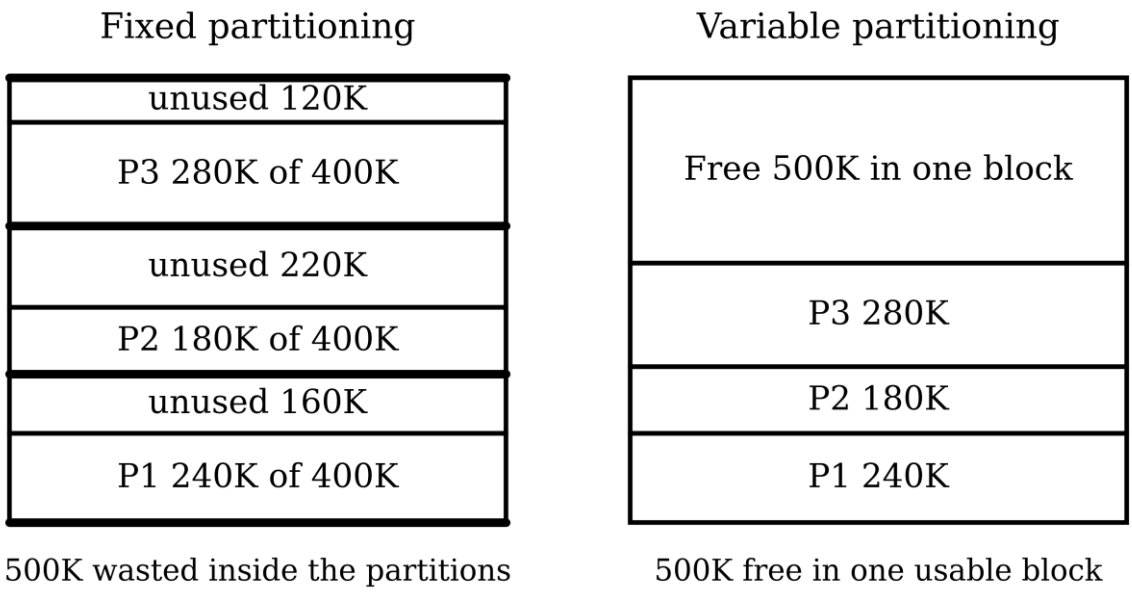


Figure 3.32: *The same three processes under fixed and under variable partitioning, and where the free space ends up.*

The choice is between simple bookkeeping with the waste locked inside the partitions and exact allocation with the waste scattered between them.

The Master Key: *Fixed partitioning sets its boundaries before it knows the processes; variable partitioning sets them afterwards and pays for it in bookkeeping.*

BINDER Question 84

At Madibira the seed store keeps two rooms with door frames already built to size, and a long open shed where sacks are set down wherever they fit. Both end the season with wasted floor, and the waste is not of the same kind.

Distinguish internal fragmentation from external fragmentation, state what causes each, and say which allocation scheme suffers from which.

Model Answer

Internal fragmentation is unused space inside a block that has already been allocated to a process, and external fragmentation is free space lying in the gaps between allocated blocks, too broken up to serve a request that the total by itself could satisfy.

Where internal fragmentation comes from: fixed partitioning. A process is given a whole partition and uses part of it, and the remainder belongs to that process, appears on no free list, and is returned only when the process ends. A 450K process in a 600K partition wastes 150K, a 400K process wastes 200K, and the two together lose 350K.

Where external fragmentation comes from: variable partitioning. Each allocation is cut to size, so nothing is wasted inside it, but the gaps left as processes end are scattered up and down memory. Gaps of 130K, 150K and 160K come to 440K free, and a 200K process still cannot be placed, because the largest single gap is 160K and falls 40K short.

The distinction the common teaching loses: students are often taught, wrongly, that fragmentation is one phenomenon with one cure. The two kinds fall on opposite sides of the allocation boundary, internal waste inside an allocation and external waste outside every allocation, so a measure that cures one leaves the other exactly as it was.

What removes each of them: internal fragmentation goes only when the system allocates what was asked for instead of a whole partition, which means giving up fixed partitioning. External fragmentation is removed by compaction, which slides the allocated blocks together, and by paging, which follows in the next section.

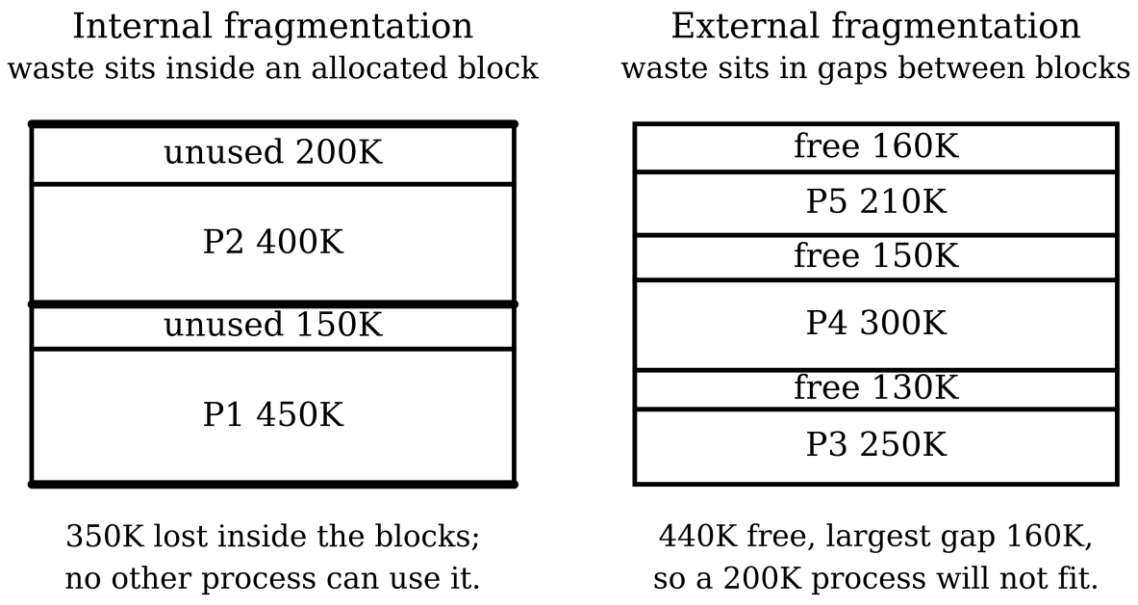


Figure 3.33: *Waste inside allocated blocks on the left, waste between allocated blocks on the right.*

Say which side of the allocation boundary the waste lies on and the cause names itself, because inside means a fixed block and between means a variable one.

The Master Key: *Internal waste is inside an allocated block and comes from fixed partitioning; external waste is between allocated blocks and comes from variable partitioning.*

REAL Question 85

At Ubaruku the grain sheds stand in a row, and the clerk leaves each arriving load in the first shed with room in it. By the fourth lorry the row holds hundreds of spare bags, and not one of those bags of room helps.

Show where first fit places each of the four processes and state what happens to any process that cannot be placed.

Memory holds five free blocks, in this order from the lowest address: 100K, 500K, 200K, 300K and 600K.

Four processes arrive, in this order: 212K, 417K, 112K and 426K.

First fit searches from the lowest address and takes the first block large enough.

Model Answer

First fit puts the 212K process in the 500K block, the 417K process in the 600K block and the 112K process in what is left of the 500K block, and it strands the 426K process, which is refused while 959K of memory stands free.

How the search runs: first fit begins at the lowest address on every request and takes the first block whose size is at least the size of the request. It stops the moment it finds one, so the low blocks are examined on every request and the high blocks only when the low ones fail.

Placing the 212K process: the 100K block is too small, so the search moves to the 500K block, which is large enough. The process goes there, and 500 less 212 leaves 288K of that block still free.

Placing the 417K process: the search restarts at the bottom. The 100K block, the 288K remnant, the 200K block and the 300K block are all too small, so the process is placed in the 600K block, and 600 less 417 leaves 183K.

Placing the 112K process: the search restarts at the bottom again, passes the 100K block and reaches the 288K remnant of the 500K block. Placing it there leaves 176K, and the five free blocks now stand at 100K, 176K, 200K, 300K and 183K.

Stranding the 426K process: no single block will hold it. The free blocks add to 959K, more than twice what the process asked for, but the largest of them is 300K and falls 126K short, so the request is refused and the process waits for memory to be released.

What the refusal measures: external fragmentation, stated as a number. Of the 959K free, only 300K is reachable by any one request, so 659K of real memory is beyond the reach of everything. The memory is not short of space; it is short of space in one piece.

Process	Block chosen	Size before	Left in that block
212K	Block 2	500K	288K
417K	Block 5	600K	183K
112K	Block 2 remnant	288K	176K
426K	None, refused	Largest is 300K	Not placed

Table 3.25: *First fit over the four requests, with the remainder left in each block.*

Free blocks 100, 500, 200, 300, 600 K in that order
Requests 212, 417, 112, 426 K in that order

Policy	212K	417K	112K	426K	Placed
First fit	block 2	block 5	block 2	stranded	3 of 4
Best fit	block 4	block 2	block 3	block 5	4 of 4
Worst fit	block 5	block 2	block 5	stranded	3 of 4
Next fit	block 2	block 5	block 5	stranded	3 of 4

Best fit places all four requests.
The others strand the 426K request.

Figure 3.34: *The four placement policies over the same free blocks and the same four requests.*

First fit is quick because it stops at the first block that fits, and on this stream that speed costs it the last process.

The Master Key: *A refusal with memory left over is not a shortage but fragmentation, and first fit produced one in four requests.*

REAL Question 86

At Igawa one lorry park works each arrival into the smallest space that will take it. The park across the road aims for the widest gap instead. By evening only one of the two has room left for the trailer that comes in last.

Using the same free blocks and the same processes, show where best fit and where worst fit place each process, and state which of the three policies places all four.

The free blocks are again 100K, 500K, 200K, 300K and 600K, and the processes again 212K, 417K, 112K and 426K.

Best fit takes the smallest block that is large enough.

Worst fit takes the largest block available.

Model Answer

Best fit places all four processes and worst fit strands the 426K process exactly as first fit did, so best fit is the only one of the three policies that gets every process into memory.

How best fit searches: it examines the whole free list on every request and takes the smallest block that is at least as large as the request. Stopping early is impossible, because a tighter block may lie anywhere further along the list.

Best fit placing the four: the 212K process takes the 300K block and leaves 88K, the 417K process takes the 500K block and leaves 83K, and the 112K process takes the 200K block and leaves 88K. The 600K block is untouched until the 426K process arrives, and it fits there with 174K to spare.

How worst fit searches: it also examines the whole list, and takes the largest block, on the argument that the remainder left behind will be big enough to be worth something to somebody else.

Worst fit placing the four: the 212K process takes the 600K block and leaves 388K, the 417K process takes the 500K block and leaves 83K, and the 112K process takes the 388K remnant and leaves 276K. The 426K process is then refused, because the largest block remaining is 300K.

The result worth remembering: best fit places four processes and worst fit three, the same three first fit managed. First fit and worst fit each leave 959K free, best fit leaves 533K with every process resident, and the 426K difference between those totals is precisely the process the other two could not place.

Why best fit won here: it spent the small blocks on the small requests and kept the 600K block whole for the request that needed all of it. Worst fit did the reverse, breaking the largest block first for a 212K process that the 300K block would have taken comfortably.

Process	Best fit block	Left	Worst fit block	Left
212K	Block 4, 300K	88K	Block 5, 600K	388K
417K	Block 2, 500K	83K	Block 2, 500K	83K
112K	Block 3, 200K	88K	Block 5 remnant	276K
426K	Block 5, 600K	174K	None, refused	Not placed

Table 3.26: *Best fit and worst fit over the same four requests, side by side.*

On this stream the policy that looks the most miserly is the one that finds room for everything.

The Master Key: *Best fit preserves large blocks by consuming the tight ones, and worst fit destroys the largest block first and pays for it at the final request.*

REAL Question 87

At Utengule the coffee drying beds are filled in a circuit, and the man with the rake carries on from where he stopped rather than walking back to the first bed. His legs are grateful. The beds at the low end are another matter.

Explain how next fit differs from first fit. Using the same free blocks, show where next fit places the first two processes and compute the total free memory remaining afterwards.

The five free blocks are again 100K, 500K, 200K, 300K and 600K, in that order.

Next fit resumes its search from the block where the last placement was made, wrapping round to the lowest address if it reaches the top.

Model Answer

Next fit differs from first fit in one respect only, which is where the search begins, since first fit starts at the lowest address every time while next fit resumes from the block that took the previous process.

Placing the 212K process: next fit begins at the lowest address, because nothing has been placed yet. The 100K block is too small and the 500K block takes the process, leaving 288K, and the search now rests on that block rather than returning to the bottom.

Placing the 417K process: the search resumes at the 288K remnant instead of the 100K block. That remnant, the 200K block and the 300K block are all too small, so the process is placed in the 600K block, and 600 less 417 leaves 183K there.

The free memory remaining: the five blocks held 1700K between them at the start and the two processes took 629K, so 1071K is free. Read block by block instead, the free space is 100K, 288K, 200K, 300K and 183K, which comes to the same 1071K.

Where the two policies part: the third request shows the difference. First fit walks back to the bottom and puts the 112K process into the 288K remnant, while next fit carries on from the 600K block and puts it into the 183K remainder, leaving 71K there and the 288K remnant untouched.

The consequence of the moving start: next fit stops rescanning the low addresses, so the average search is shorter and no one block is broken by every request in turn. The cost is that small holes at the low end are passed over for long stretches, and the large block at the top is cut up sooner than first fit would cut it.

The two policies differ only in where the search starts, and that alone changes which blocks get broken and in what order.

The Master Key: *Next fit is first fit with a memory of where it stopped, which spreads the allocations out and leaves the low holes unvisited.*

REAL Question 88

A stores ledger at Matema is ruled into three columns, and three statements about wasted space have been copied into it in a careful hand. One of the three is right. The other two have been carried from ledger to ledger for years.

Three statements about fragmentation and compaction are given. Identify the correct one, explain the fault in each of the others, and correct them.

Statement A. Internal fragmentation is the free space in the gaps between allocated blocks, and external fragmentation is the space left unused inside an allocated block.

Statement B. Compaction cures external fragmentation by moving the allocated blocks together so that the free space forms one run, and it is possible only when addresses are bound at execution time.

Statement C. Compaction cures internal fragmentation as well, because sliding the processes together recovers the unused space inside each partition.

Model Answer

Statement B is the correct one: compaction gathers the allocated blocks at one end of memory so that the scattered free space becomes a single run, and it can be done only if the addresses of a running process may be changed, which is what execution time binding provides.

Why B is right: external fragmentation is free space broken into gaps, and moving the allocated blocks together is the direct cure, because the gaps merge as the blocks slide past them. The condition attached is real, since a process bound at compile time or at load time holds addresses that would be wrong in its new place.

The fault in A: the two definitions have been swapped. Waste inside an allocated block is internal and free space in the gaps between allocated blocks is external, and the swap loses the cause of each along with the name, because the first comes from fixed partitions and the second from variable ones.

A corrected: internal fragmentation is unused space inside a block already allocated to a process, produced by fixed partitioning, and external fragmentation is free space in the gaps between allocated blocks, produced by variable partitioning.

The fault in C: compaction cannot touch internal fragmentation, because that waste is inside the allocation and travels with it. A process holding a 600K partition and using 450K of it holds the same 600K after the move, and the 150K inside is no more available to anybody else than it was before.

C corrected: compaction cures external fragmentation only. Internal fragmentation is removed by allocating exactly what was requested rather than a whole fixed partition, which is what variable partitioning does and what fixed partitioning by its nature cannot do.

A cure is specific to the kind of waste it addresses, and compaction addresses only the waste that lies between allocations.

The Master Key: *Compaction merges the gaps between blocks and never recovers a single byte from inside one.*

HOT Question 89

The hardware counter at Igurusi keeps offcuts of timber, and the storeman always cuts from the piece that leaves the smallest scrap. The shelf is filling with scraps too short for anything. He has served every customer so far, and the shelf is losing.

Best fit is named for choosing the block that fits best. Argue whether the name is deserved, and reach a judgement.

Model Answer

The name is deserved for the choice best fit makes and not for the memory it leaves behind, and the judgement turns on how tight memory is, because best fit earns the name when requests are large against the blocks available and first fit is the better policy when the free list is long and the requests are small.

The case for the name: on this request stream best fit is the only one of the three policies to place all four processes. Putting the 212K process in the 300K block and the 112K process in the 200K block kept the 500K

and 600K blocks whole, and the 426K process that first fit and worst fit both refused went in with 174K to spare.

The case against, in the holes it leaves: the remainders best fit leaves are by construction the smallest available. After the four placements the free blocks are 100K, 83K, 88K, 88K and 174K, and the three slivers come to 259K of memory that almost no future request will be small enough to use.

The case against, in the search: best fit cannot stop early, since a tighter block may lie anywhere further down the list. Over these four requests it examines twenty blocks where first fit examines fourteen, and the gap widens as the free list lengthens, which is exactly what happens as memory fragments.

What the name actually promises: best fit is a local rule, and its name describes one decision rather than an outcome. It looks only at the request in front of it, so a different arrival order changes the ranking, and no placement policy is best over every stream that memory might be given.

Whether the cost can be reduced: keeping the free list sorted by size lets the best fit search stop at the first block that fits, which removes the scanning penalty. The sort has to be maintained on every allocation and every release, so the cost moves from the search to the bookkeeping rather than disappearing.

The judgement: the name is fair as a description of the choice and overstated as a promise about the result. Prefer best fit where memory is tight and a refused process is expensive, because placing everything is worth the longer scan; prefer first fit where the free list is long and allocations are frequent, because the search is paid for on every request while the placement advantage appears only when the blocks are barely large enough.

Best fit placed every process here, and the slivers it left behind are the price of that success.

The Master Key: *Best fit chooses well and leaves badly, so judge it on whether a refused process costs you more than a longer search.*

HOT Question 90

The workshop at Igawa clears its floor once a season, sliding every machine to one wall so that the open space lies in one piece. Nothing is made that day. The foreman wants to know what the quiet day bought him.

Compaction removes external fragmentation by moving every process to one end of memory. Argue whether it is worth its cost, and reach a judgement.

Model Answer

Compaction is worth its cost when a process has actually been refused and the free total could have held it, and it is not worth running to a timetable, because the whole of memory is copied to buy a benefit that the next few releases start to undo.

What compaction buys: under first fit the four requests left 959K free with the largest block at 300K and the 426K process refused. Compaction slides the three resident processes together and the free space becomes one run of 959K, into which the refused process fits with 533K still free behind it.

What compaction costs: every byte of every process that moves has to be copied, and here that is the 741K the three resident processes occupy. The processor and the memory do the copying, so the processes being moved cannot run while it happens, and a system that compacts all of memory runs nothing at all meanwhile.

What makes it possible at all: execution time binding and the relocation register. Moving a process is a matter of copying its bytes and reloading its base, because every address it issues is formed afresh from that register as it runs. Under compile time or load time binding the same move would leave every address inside the process wrong.

How long the benefit lasts: the single run of free memory starts breaking up again as soon as processes end and new ones are placed, so the cost is not paid once and forgotten. A system that compacts whenever a small gap appears spends a large share of its time copying memory that nobody has asked for.

The alternative that is not the same thing: swapping writes a whole process out to the backing store and brings it back later, possibly to a different place, which frees a large contiguous region without sliding anything. A common error is to describe swapping and then call it virtual memory, when swapping moves a whole process and demand paging, which follows, moves one page.

The judgement: compact when a request has been refused that the free total could serve, and when the system will then run long enough to earn the pause back; do not compact on a timer. The two decisive facts are that execution time binding is what makes the move legal and that the processor does nothing else while it copies.

Compaction is a real cure with a stated price, and paging, which follows in the next section, removes the external fragmentation that makes it necessary.

The Master Key: *Compaction is affordable when it rescues a refused process, wasteful when it runs to a timetable, and legal only under execution time binding.*

Paging, segmentation and virtual memory

This section takes memory apart into fixed pieces and then lets a program run without all of it present. It works an address translation in both binary and decimal, traces three replacement policies over one reference string, and shows the case where more memory makes things worse.

BINDER Question 91

At the Isyonje tea factory the drying trays are all one size. A picker's basket is emptied across four trays, whichever four are free, and the last tray holds only a handful. Memory is managed the same way, and the handful has a name.

Explain paging. Distinguish a page from a frame, explain what the page table holds, and state which kind of fragmentation paging removes and which kind it introduces.

Model Answer

Paging divides a process's logical address space into equal fixed-size pages and physical memory into frames of the same size, so a process can occupy any set of free frames and needs no unbroken run of memory.

Page and frame: a page is a fixed-size division of the logical address space; a frame is a division of physical memory of exactly the same size, commonly 4096 bytes. A page is what the program has, a frame is what the machine has, and the words are never interchangeable.

Two fields in one address: the hardware reads a logical address as a page number followed by an offset. The page number selects a row of the page table; the offset says how far into the page the byte sits, and it crosses to physical memory untouched.

What the page table holds: one entry for every page of the process, holding the frame that page occupies. Beside it sit control bits: a valid bit saying whether the page is in memory at all, and protection bits saying whether it may be written. Every process has its own table.

The translation lookaside buffer: the page table lives in memory, so a plain lookup would cost two memory accesses for every reference, one to read the entry and one to fetch the data. A small cache of recently used page and frame pairs, the translation lookaside buffer, removes the second access on most references.

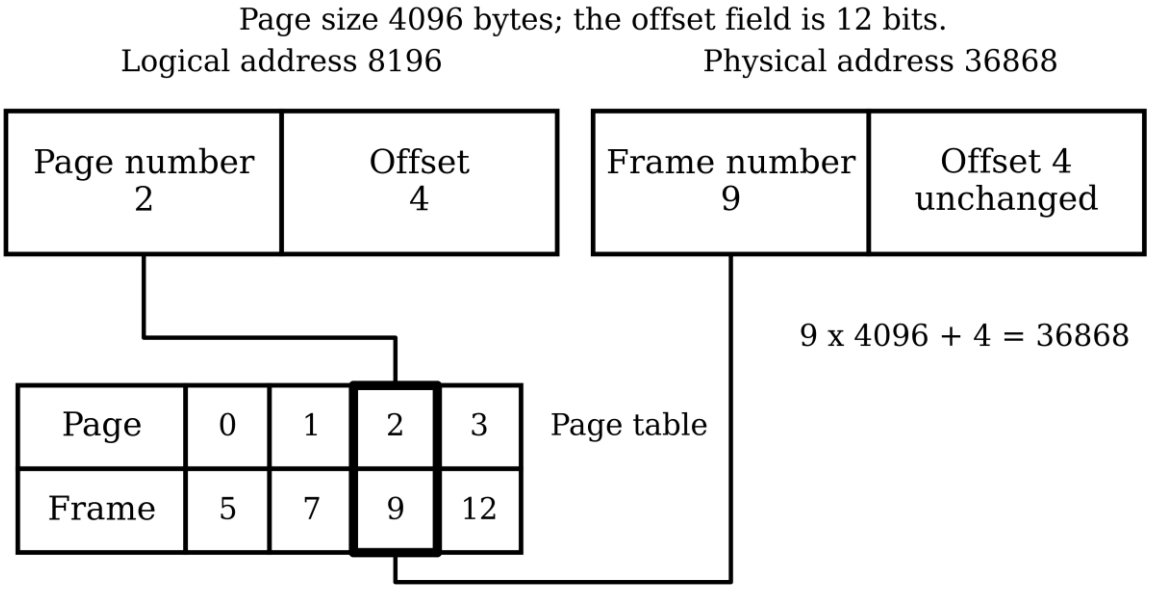


Figure 3.35: *A logical address split into a page number and an offset, the page number replaced by a frame number from the page table, and the offset carried across unchanged.*

Fragmentation removed and fragmentation added: external fragmentation goes, because a process no longer needs one unbroken run of memory and any free frame will do. Internal fragmentation arrives in its place, in the last page of each process, which is rarely filled exactly; the average loss is half a page.

The page table is what makes the scattering invisible, because the program still sees one continuous address space.

The Master Key: *A page is logical and a frame is physical, and paging trades external fragmentation for a small internal loss in the last page.*

BINDER Question 92

A Kibena carpenter keeps three racks: long timber, short offcuts, and boards. Nothing is cut to a standard length; each rack is as deep as its contents need. Memory can be divided that way, and the table that finds a piece holds two numbers.

Explain segmentation, state what a segment table holds, and state precisely how segmentation differs from paging.

Model Answer

Segmentation divides a program into variable-length segments that match its own logical parts, a code segment, a data segment and a stack segment, and addresses a byte by naming the segment and the offset inside it.

A segment is a unit the program recognises: the divisions are not arbitrary. A compiler produces a segment for code, one for global data and one for the stack, and each is as long as its contents require. Two segments of one program are almost never the same size.

What the segment table holds: one row for each segment, carrying a base and a limit. The base is where the segment starts in physical memory and the limit is its length. A page table entry needs only a frame number, because every page is the same size; a segment table needs both.

Translating a segmented address: the offset is checked against the limit before anything else. For a code segment based at 3200 with a limit of 1200, offset 900 is legal and the byte sits at 4100; offset 1300 is beyond the limit and the hardware traps instead of reading it.

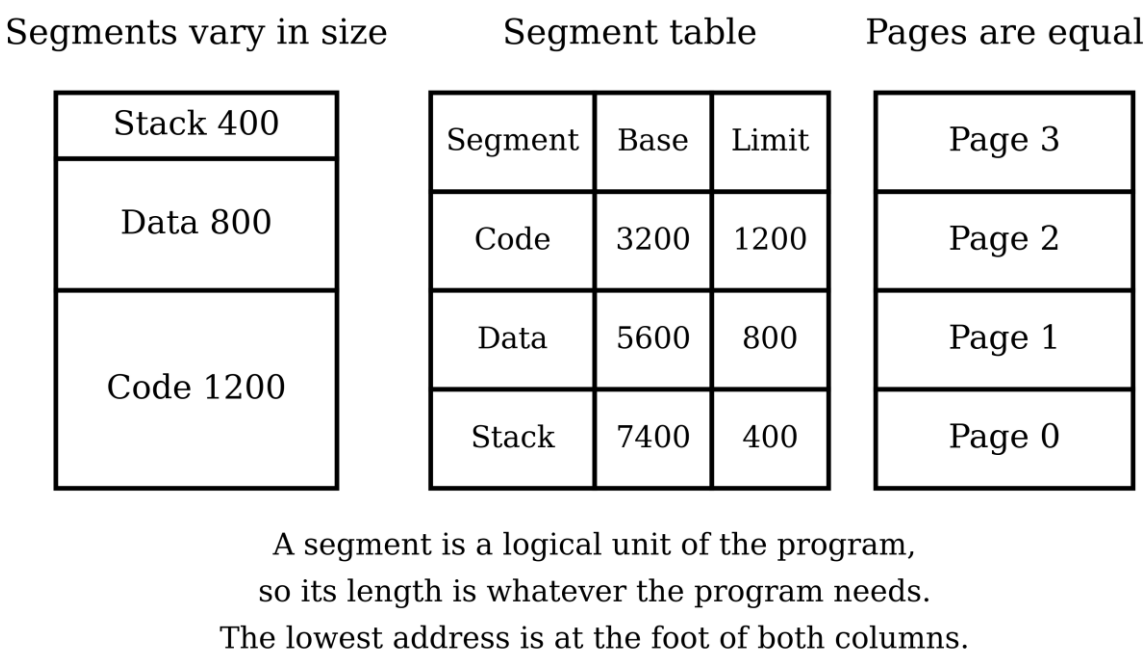


Figure 3.36: *Three segments of unequal length against four pages of equal length, with the segment table carrying a base and a limit for each segment.*

Who sees the division: paging is invisible to the program, which still sees one flat run of addresses, and the operating system chooses the page size. A segmented address is two numbers the program itself supplies, so the division follows the structure the programmer wrote.

Where it costs: segments vary in size, so they are placed the way variable partitions are, and external fragmentation comes back with them. Paging never suffers that, because a frame is a frame. Segmentation buys protection and sharing at the granularity of a logical unit, and pays in holes.

Point of difference	**Paging**	**Segmentation**
Size of the unit	Equal and fixed by the hardware	Variable and fixed by the program
Who divides the program	The operating system	The compiler and the programmer
Seen by the program	No, one flat address space	Yes, a segment number and an offset
One table entry holds	A frame number	A base and a limit
Fragmentation produced	Internal, in the last page	External, between segments

Table 3.27: *Paging and segmentation compared on the five points that separate them.*

Paging answers the question of where to put a program; segmentation answers the question of what the parts of a program are.

The Master Key: *A page table entry holds a frame number and a segment table entry holds a base and a limit, and the limit is what makes the check possible.*

BINDER Question 93

The Uwemba dispensary orders no medicine it has not been asked for. A prescription arrives, the storeman walks to the back, and the patient waits. Nothing waits on the shelf. A program can be run the same way, and the waiting has a name.

Explain demand paging, define a page fault, and list in order the steps the operating system takes to service one.

Model Answer

Demand paging brings a page into memory only when a process actually refers to it, and a page fault is the trap the hardware raises when the page referred to is marked not present in the page table.

What is resident and what is not: a process starts with almost nothing in memory. Each page table entry carries a valid bit; a page that has never been fetched, or has been taken away again, has that bit clear, and the entry points at a copy on disk rather than at a frame.

The first three steps of servicing a fault: the hardware traps to the operating system and the faulting instruction stops. The operating system checks the reference against the address space of the process and terminates it if the reference is illegal. It then finds a free frame, or selects a victim and writes it back if it was modified.

The last three steps: the operating system schedules a disk read of the wanted page into that frame, and the process waits while the transfer runs. When it completes, the page table entry is updated with the frame number and the valid bit is set. The faulting instruction is then restarted.

Virtual memory, and why swapping is not it: virtual memory lets a process run in a logical address space larger than the physical memory holding it, only part resident at any moment. Swapping moves a whole process out and later brings all of it back; demand paging moves one page and the process runs on those it still holds. Students are often taught, wrongly, that it is simply disk used as extra memory; without a page table, a translation and a fault there is none of it.

A page fault is not an error; it is the ordinary way a page arrives, and only an illegal reference ends the process.

The Master Key: *Demand paging fetches one page on a fault and leaves the rest of the process running, which is what separates it from moving a whole process out.*

REAL Question 94

At the Ilembula water office every meter reading is filed as a book number and a line within that book. The clerk never searches the whole store; two numbers land her on the row. The hardware translates an address by exactly the same trick.

A system has a page size of 4096 bytes and a process refers to logical address 8196, and page 2 of that process is held in frame 9. Compute the page number, the offset and the physical address, showing the working in decimal and in binary.

Page size: 4096 bytes

Logical address referred to: 8196

Page table entry: page 2 is held in frame 9

Model Answer

The reference falls in page 2 at offset 4, that page is held in frame 9, and the physical address is 36868, which is the frame number multiplied by the page size with the offset added on.

The page size fixes the two fields: 4096 is 2 to the power of 12, so the offset needs exactly 12 bits to count every byte of a page, from 0 to 4095. Every bit above those 12 is the page number. A page size that was not a power of two would make this impossible, which is why every real page size is one.

The page number in decimal: divide the logical address by the page size and take the whole part. 8196 divided by 4096 is 2 with something left over, so the reference lies in page 2. Multiplying back, 2 times 4096 is 8192, which is where page 2 begins. Nothing about the page table has been used yet.

The offset in decimal: the offset is what is left over. 8196 minus 8192 is 4, so the byte wanted is the fifth byte of page 2. The same figure comes out of the division as its remainder, and it must be smaller than the page size or the arithmetic is wrong.

The same split in binary: 8196 in binary is 10 0000 0000 0100. Cut off the low twelve bits and they read 0000 0000 0100, which is 4, the offset. What remains above the cut reads 10, which is 2, the page number. No division is done at all, which is why the hardware can do it in one step.

The lookup and the physical address: the page table says page 2 is held in frame 9. The page number is replaced by the frame number and the offset crosses unchanged, because a frame is the same size as a page. In decimal, 9 times 4096 is 36864, and 36864 plus 4 is 36868.

The physical address in binary: frame 9 is 1001. Write those bits in front of the unchanged twelve offset bits and the physical address reads 1001 0000 0000 0100, which is 36868 in decimal. The two routes agree, and they must, because joining the fields is the same operation as multiplying by the page size and adding.

A page fault would have shown itself at the lookup, because an entry marked not present cannot supply a frame number.

The Master Key: *The offset is never translated, so only the page number is looked up and only the page number changes.*

REAL Question 95

A Kifanya lorry has room for three drums and the loader must throw one off to take another on. He can drop the drum loaded first, the drum untouched longest, or the drum nobody will call for again. Three rules, three different lorries.

For the reference string and the three frames given, trace first in first out, least recently used and optimal replacement, and give the number of page faults under each.

Reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Frames available: 3

All three frames start empty

Model Answer

First in first out faults 15 times, least recently used 12 times and optimal replacement 9 times over the same twenty references, so the policy alone changes the fault count by six.

The three rules: first in first out evicts the page that has been resident longest, whatever it has been doing. Least recently used evicts the page that has gone unreferenced for longest. Optimal evicts the page whose next reference lies furthest ahead, which requires knowing the rest of the string.

The faults nobody can avoid: the first three references arrive while all three frames are empty, so 7, 0 and 1 fault under every policy. Those three are compulsory, and the contest between the policies is fought over the seventeen references that follow.

Where first in first out goes wrong: at the sixth reference it evicts page 0 to make room for 3, because 0 has been resident longest. Page 0 was used one step earlier and is wanted again at the very next reference, so the eviction buys nothing and costs a fault. Age is not use.

Where least recently used does better: at that same sixth reference it evicts page 1 instead, because 0 has just been touched and 1 has not been touched since the third reference. The reference to 0 that follows is then a hit. Looking backwards is a good guess at looking forwards, and over the whole string it saves three faults.

What optimal does with the same frames: at the eighth reference, with 2, 0 and 3 resident and 4 arriving, it evicts 0, because 2 is wanted immediately, 3 two steps later and 0 not until the eleventh reference. Nine faults is the floor for this string on three frames.

Reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7
FIFO		0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0
			1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	1
15 faults	F	F	F	F		F	F	F	F	F	F			F	F			F	F	F
	7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
LRU		0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
			1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	7
12 faults	F	F	F	F		F		F	F	F	F			F		F		F		
	7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7
Optimal		0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0
			1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1
9 faults	F	F	F	F		F		F			F			F				F		

Figure 3.37: *The three policies traced over the same twenty references on three frames, with F marking each page fault.*

Reading the counts: fifteen faults in twenty references is a rate of 0.75, twelve is 0.6 and nine is 0.45. The string and the frame count are identical throughout, so every part of that difference belongs to the replacement rule and to nothing else.

Step	Ref	FIFO frames	F	LRU frames	F	Optimal frames	F
1	7	7	F	7	F	7	F
2	0	7 0	F	7 0	F	7 0	F
3	1	7 0 1	F	7 0 1	F	7 0 1	F
4	2	2 0 1	F	2 0 1	F	2 0 1	F
5	0	2 0 1		2 0 1		2 0 1	
6	3	2 3 1	F	2 0 3	F	2 0 3	F
7	0	2 3 0	F	2 0 3		2 0 3	
8	4	4 3 0	F	4 0 3	F	2 4 3	F
9	2	4 2 0	F	4 0 2	F	2 4 3	
10	3	4 2 3	F	4 3 2	F	2 4 3	
11	0	0 2 3	F	0 3 2	F	2 0 3	F
12	3	0 2 3		0 3 2		2 0 3	
13	2	0 2 3		0 3 2		2 0 3	
14	1	0 1 3	F	1 3 2	F	2 0 1	F
15	2	0 1 2	F	1 3 2		2 0 1	
16	0	0 1 2		1 0 2	F	2 0 1	
17	1	0 1 2		1 0 2		2 0 1	
18	7	7 1 2	F	1 0 7	F	7 0 1	F
19	0	7 0 2	F	1 0 7		7 0 1	
20	1	7 0 1	F	1 0 7		7 0 1	

Table 3.28: *The frames after each reference under the three policies, where F marks a page fault and a blank marks a hit.*

Optimal cannot be run on a live machine, but it fixes the floor that the other two are measured against.

The Master Key: *Same string, same frames, three different fault counts, so the replacement policy is a real engineering choice and not a detail.*

REAL Question 96

On the back of a Matembwe calendar someone has drawn a chart: frames along the bottom, faults down the side, and a line that only ever falls. It is drawn with confidence and it is wrong, and one string of page numbers proves it.

It is widely believed that adding memory frames always reduces page faults. Using the reference string given, show that this is false under first in first out, and name the effect.

Reference string: 1 2 3 4 1 2 5 1 2 3 4 5

Run the same string twice, once on three frames and once on four

All frames start empty

Model Answer

On this string first in first out takes 9 faults with three frames and 10 with four, so the fourth frame costs a fault rather than saving one, and the effect has a name, Belady's anomaly.

Three frames: the first seven references all fault, the eighth and ninth are hits, the tenth and eleventh fault and the twelfth is a hit, which is nine faults in twelve references. Nothing unusual happens here; first in first out is behaving exactly as described.

Four frames: the extra frame delays the first eviction, so the fifth and sixth references are hits where before they were faults. From the seventh reference onward every single reference faults, and the run ends on ten. Two faults were saved early and three were lost later.

Why more can be worse: first in first out picks its victim by age alone. At the seventh reference, three frames evict page 4, which is not wanted again until the eleventh reference; four frames evict page 1, which is wanted at the very next reference. The larger memory made the worse choice.

Reference	1	2	3	4	1	2	5	1	2	3	4	5
	1	1	1	4	4	4	5	5	5	5	5	5
FIFO, 3 frames		2	2	2	1	1	1	1	1	3	3	3
			3	3	3	2	2	2	2	2	4	4
9 faults	F	F	F	F	F	F	F			F	F	
	1	1	1	1	1	1	5	5	5	5	4	4
FIFO, 4 frames		2	2	2	2	2	2	1	1	1	1	5
			3	3	3	3	3	3	2	2	2	2
				4	4	4	4	4	4	3	3	3
10 faults	F	F	F	F			F	F	F	F	F	F

3 frames give 9 faults, 4 frames give 10.
More memory, more faults. This is the anomaly.

Figure 3.38: *The same twelve references under first in first out on three frames and on four, where the larger memory faults more often.*

The property that rules it out: a policy is called a stack policy when the pages held with n frames are always among the pages held with n plus one. Least recently used has that property, so extra frames can never hurt it. First in first out does not, and that is the whole explanation.

The same string under least recently used: it takes 10 faults on three frames and 8 on four, so the fourth frame saves two. That is the ordinary behaviour students expect, and it is worth noticing that on three frames least recently used is here the worse of the two policies.

Frames	Faults under first in first out	Faults under least recently used
3	9	10
4	10	8

Table 3.29: *The same twelve references at two frame counts, where only first in first out gets worse when a frame is added.*

What the anomaly does not mean: adding frames usually does reduce faults, and on a long realistic string it almost always will. What fails is the guarantee. Buying memory is a good bet under first in first out and a certainty only under a stack policy, and the difference matters when a system is being sized.

The chart on the calendar is right about the trend and wrong about the rule.

The Master Key: *More frames cannot increase faults under least recently used, but under first in first out they can, and Belady's anomaly is the proof.*

HOT Question 97

The Mgololo sawmill manager knows, after the fact, exactly which order the day's logs should have been cut in. He could not have known it at seven in the morning. A rule you can only apply afterwards still tells you something worth having.

Optimal replacement cannot be implemented, because it requires knowledge of future references. Argue what use it is to the designer of an operating system, and reach a judgement.

Model Answer

Optimal replacement is worth computing even though it can never be installed, because it is the only way to know how much of a fault count belongs to the policy and how much belongs to the workload.

The case against it: an operating system cannot see the future. To choose the page whose next use is furthest away, it would have to know every reference the process has not yet made, and a general purpose system has no way of obtaining that. As a running policy it is not merely difficult, it is impossible.

It can be computed after the event: a recorded reference string is not the future, it is the past, and optimal can be run over it as easily as any other policy. Every measurement of a real policy is made this way, on traces captured from real programs and replayed.

Without it a fault count means very little: a policy that takes 12 faults on a string may be excellent or careless, and the number alone cannot say which. Against a floor of 9 it is three above the best obtainable, and against the 15 that first in first out takes it is three better. The floor supplies the scale.

It settles arguments about where to spend effort: if a cheap policy already sits close to the floor on the workloads that matter, a cleverer one cannot repay its cost, and the effort belongs elsewhere, in more frames or in a faster device. If the gap is wide, the policy is the thing to improve.

What is installed instead: real systems approximate least recently used, because a true one needs the hardware to timestamp every reference. A reference bit for each page, cleared periodically and inspected before eviction, gets most of the benefit for very little cost, and optimal is what tells the designer how much benefit is left unclaimed.

The judgement: optimal replacement is not a candidate for installation and was never offered as one; its role is as a lower bound, and it should be used exactly that way. Quote a policy's faults against the optimal count for the same string and the same frame count, and never on their own.

A bound that cannot be reached is still the only honest yardstick, which is why optimal is computed and never installed.

The Master Key: *Optimal is a measuring instrument rather than a policy, and a fault count quoted without it has no scale.*

HOT Question 98

The Isyonje records clerk has one desk and a store room down the corridor. Every enquiry needs a file he has just put back, so he spends the day walking and answers almost nothing. The desk is not the problem.

A machine spends more time paging than executing. Define thrashing and the working set, argue whether the fault lies with the operating system, with the hardware or with the user, and reach a judgement.

A memory access takes about 100 nanoseconds

Servicing one page fault takes about 8 milliseconds

Many processes are admitted and the processor is nearly idle

Model Answer

Thrashing is the state in which the resident processes hold so few frames between them that almost every reference faults, and the judgement is that the operating system carries the fault, because it is the only party that can see the working sets and act on them.

The working set: the working set of a process is the set of distinct pages it has referenced in the most recent window of references. Over the last ten references of the string traced earlier, that set is 0, 1, 2, 3 and 7, so the process needs five frames to run without faulting constantly, and three leaves it two short.

Why the cost is so violent: a memory access takes about 100 nanoseconds and a page fault about 8 milliseconds, which is 80000 memory accesses. One fault in every 80000 references doubles the average reference. One fault in every hundred makes it more than eight hundred times slower than the memory itself.

The case against the user: the person at the machine started the programs, and starting fewer would end the thrashing at once. But no user is shown the working set of anything, and the memory demand of a program is not visible from its window. Blaming the user asks for a judgement the user has no information to make.

The case against the hardware: more memory would cure it and a faster device would soften every fault. Both are true and neither is a defence, because a machine of any size can be thrashed by admitting enough processes, and a machine that thrashes with many processes admitted will run steadily with fewer, on exactly the same hardware.

The case against the operating system: the classic collapse is caused by the response of the operating system itself. Seeing the processor idle, a scheduler that concludes it should admit more processes takes frames away from processes already short of them, and the fault rate climbs again. Global replacement makes it worse, because a fault in one process steals a frame from another.

The judgement and the cure that works: the operating system is answerable, because only it can measure each working set and only it can act. The cure is to give every admitted process at least its working set of frames and to suspend a process whose working set no longer fits, which lowers the degree of multiprogramming until the fault rate falls.

Buying memory and closing programs both help, but only the operating system can tell how much of either is needed.

The Master Key: *Thrashing is cured by allocating each process its working set and admitting fewer processes, not by running the ones already admitted more often.*

HOT Question 99

A Uwemba builder stacks bricks in identical pallets and keeps doors, frames and glass in rooms of their own. The pallets make the yard easy to fill; the rooms make it possible to lock the glass away. Neither arrangement does both jobs.

Argue whether a modern operating system should divide memory by paging, by segmentation, or by both, and reach a judgement.

Model Answer

A modern operating system should page, and should borrow from segmentation only where a logical division earns its keep, because paging is the only one of the two that removes external fragmentation and allocates a frame without searching.

What paging solves: the allocation problem. Any free frame will take any page, so there is no search for a hole of the right size, no external fragmentation and no compaction. The cost is a small internal loss in the last page of each process, which is bounded and predictable.

What paging cannot say: a page is an arbitrary window of 4096 bytes with no meaning of its own. It may hold the end of one function and the start of another, so a permission attached to it is attached to an accident of address rather than to a part of the program.

What segmentation solves: the meaning problem. A code segment can be marked read only and shared between every process running the same program, and a stack segment can be allowed to grow, because each is a whole logical unit with a length of its own. Protection and sharing become natural.

Why segmentation alone fails: segments vary in size, so allocating them is the variable partitioning problem again, with the same holes and the same need for compaction. A system built on segments alone trades a bounded internal loss for an unbounded external one, and that is a poor exchange.

Segmentation with paging: the combination pages each segment, so the program keeps its logical divisions and the allocator keeps its fixed frames. The price is two lookups before the data access, one in the segment table and one in the page table, and a hit in the translation lookaside buffer removes both.

The judgement: page always, and segment only for what segments uniquely provide, which is protection and sharing at the boundary of a logical unit. In practice that is delivered by protection bits in the page table rather than by full segments, so the honest answer is paging as the base with the idea of segmentation kept and its machinery mostly dropped.

Segmentation alone is the arrangement no serious system chooses, because the fragmentation it brings back is the fragmentation paging was invented to remove.

The Master Key: *Page for allocation and segment for meaning, and when only one of the two can be had, page.*

File systems, device management and disk scheduling

This section covers what the operating system does with the things that outlive a process. It gives the structures a file system is built from, explains the difference between naming a file and finding its blocks, and computes head movement for five disk-scheduling algorithms over one request queue.

BINDER Question 100

At the Kihesa fuel depot every drum carries a card giving its owner and its ninety litres, while the drum says nothing. A file carries a card of that kind, and one line on it belongs to nobody until the file is opened.

State the attributes an operating system keeps for a file and the operations it provides on one. Explain what the open-file table holds and why it exists.

Model Answer

An operating system keeps seven attributes for a file, its name, identifier, type, location, size, protection and time and owner stamps, provides create, write, read, reposition, delete and truncate on its contents, and keeps an open-file table for the state that belongs to an opening rather than to the file.

An attribute is not the contents: The attributes are what the file system records about a file, and they sit in the directory entry rather than in the data. Only the name is chosen by a person; the identifier is the number the system uses internally.

Attribute	What it records	Operation	What it does
Name	The readable string a person chooses	Create	Makes a directory entry and finds space
Identifier	The number the file system uses internally	Write	Puts data at the current position and advances it
Type	What the contents are, by extension or header	Read	Copies data from the current position and advances it
Location	Where the blocks are on the device	Reposition	Moves the position without transferring anything
Size	The current length, and often the space allocated	Delete	Frees the blocks and removes the directory entry
Protection	Who may read, write and execute it	Truncate	Empties the contents but keeps the attributes
Time and owner	When it was made, changed and read, and by whom	Open and close	Build and discard the open-file entry

Table 3.30: *The seven attributes a file system keeps and the operations it provides, set side by side.*

Reading the operations: Create makes a directory entry and finds space, write and read transfer at the current position and advance it, reposition moves that position without transferring anything, delete frees the blocks and removes the entry, and truncate empties the contents but keeps every attribute.

What the open-file table holds: Opening a file copies its directory information into a system-wide table with one entry per open file, holding the location of the blocks, the size and a count of the processes that have it open. Each process keeps a small table of its own pointing into that one.

Why the position is not an attribute: The current position belongs to the opening and is held in the per-process table. Two processes reading one file at the same time each have a position of their own, and if the position were an attribute of the file, one reader would drag the other backwards.

What closing does: Close discards the per-process entry and decreases the count in the system-wide one. When that count reaches zero the entry goes, and anything the opening changed, the size and the time of last writing, is written back.

Two people opening one file share one set of attributes and keep two positions, and that is why the open-file table exists.

The Master Key: *Attributes belong to the file and the position belongs to the opening, so the open-file table is where the difference is kept.*

BINDER Question 101

The Ipogolo produce office keeps one book of trader names. Two women both trade as Neema, and the clerk cannot tell whose sacks are whose. Directories were built in four stages, and each stage fixed exactly one fault in the last.

Explain the single-level, two-level, tree and acyclic-graph directory structures. State one limitation of each of the first three that the next one solves.

Model Answer

A single-level directory holds every file in one list, a two-level directory gives each user a list of their own, a tree lets a directory hold further directories, and an acyclic graph lets one file be reached by two paths, each structure removing one limitation of the structure before it.

Single level, and where it breaks: Every file on the volume is in one list under one root, which is simple to search and impossible to share out. Two files cannot hold the same name, so the first person to call a file report has taken that name from everybody else on the machine.

Two level, and where it breaks: Each user is given a directory beneath a master file directory, and a file is now named by its owner and its name together, so two people may both keep a report. What no user can do is reach into another user directory, so nothing at all can be shared.

The tree, and where it breaks: Any directory may hold further directories, which gives the path names every modern system uses, absolute from the root or relative to a working directory. One file now has exactly one path, so two projects needing one shared list must each keep a copy the other never sees change.

The acyclic graph: A directory entry may point at a file that another directory also points at, so one file is reached by two paths and one copy serves both projects. Deletion becomes the hard part, because the file may only go when the last link has gone, which is why a count of links is kept.

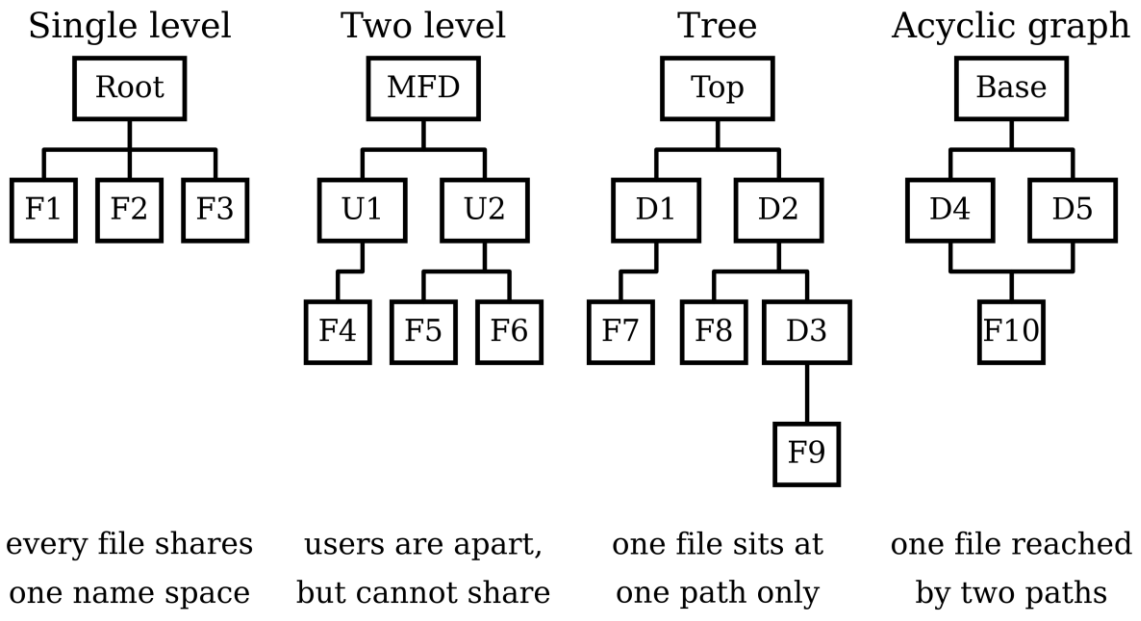


Figure 3.39: *The four directory structures, each one lifting a restriction that the structure to its left cannot lift.*

Each structure is the one before it with a single restriction removed, and the price of removing it is paid in bookkeeping.

The Master Key: *Name collisions force the second level, sharing between users forces the tree, and one file under two names forces the graph.*

BINDER Question 102

A carpenter at Nzihi stacks forty planks in one unbroken pile; his neighbour leaves his in gaps all over the yard, each with a chalk note saying where the next one lies. Both find their timber. Only one finds plank thirty at once.

Explain contiguous, linked and indexed file allocation. State the advantage and the disadvantage of each, and say which one a file allocation table and which one an inode implements.

Model Answer

Contiguous allocation puts a file in one run of consecutive blocks, linked allocation scatters the blocks and stores in each the address of the next, indexed allocation gathers every address into one index block, and a file allocation table is linked allocation with the pointers lifted into one table while an inode is indexed allocation.

Contiguous: The directory holds a start block and a length, so any block of the file is the start plus an offset and one seek reaches it. The cost is external fragmentation, and the final size must be known when the file is created.

Linked: The directory holds only the first block and each block carries the address of the next, so a file grows into any free block anywhere and no space is stranded. Reading block 30 means reading the 29 before it, and one damaged pointer loses everything after it.

Indexed: One block is set aside to hold nothing but addresses, so block 30 is found by reading the index and going straight there, with no external fragmentation and no chain to follow. The index block costs space even for a tiny file, and its size caps the file until it is nested.

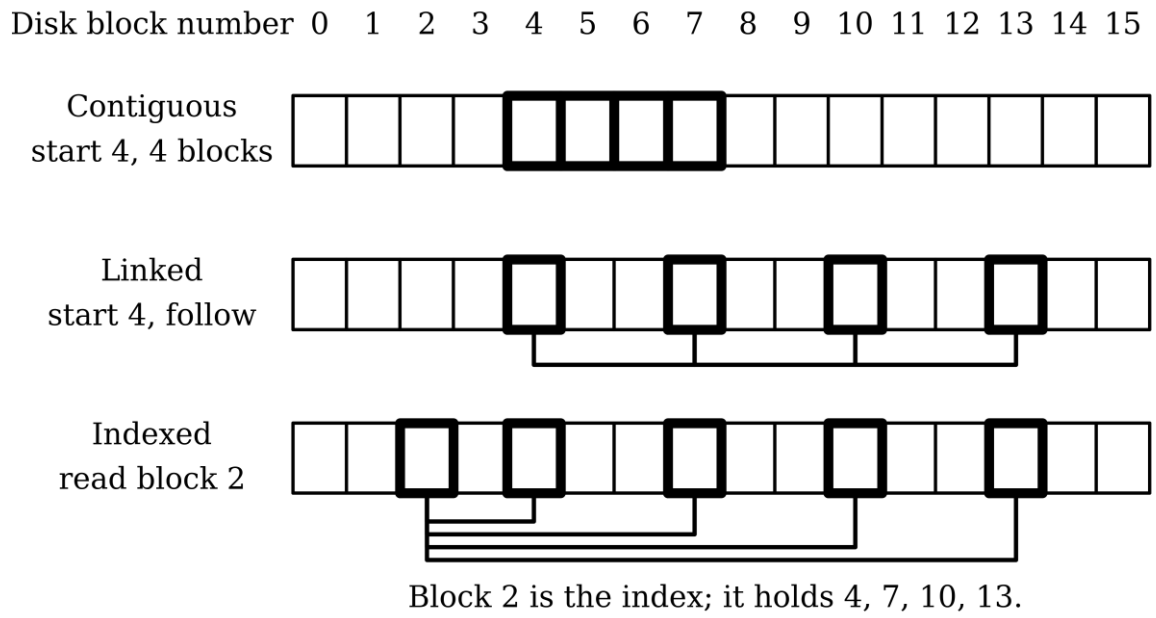


Figure 3.40: *The same four data blocks held contiguously, held as a chain, and reached through an index block.*

What a file allocation table really is: It is linked allocation with the pointers taken out of the data blocks and gathered into one table at the front of the volume. The chain is unchanged, but a table small enough to hold in memory is followed without touching the disk.

What an inode really is: It is indexed allocation. The inode holds the attributes, 12 direct block addresses, then a single, a double and a triple indirect pointer. With 4096 byte blocks and 4 byte addresses one index block holds 1024 addresses, so 12 direct pointers and one indirect block already reach 1036 blocks.

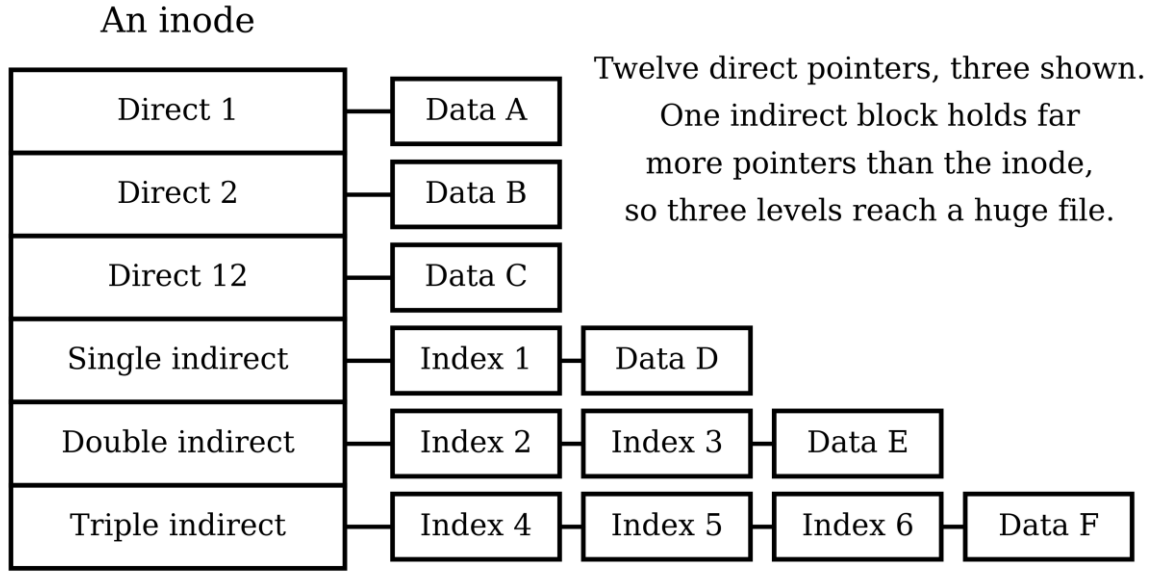


Figure 3.41: *An inode, with direct pointers to data and three levels of indirection for a file too large for them.*

Method	Advantage	Disadvantage	Where it is used
Contiguous	One seek reaches any block, so sequential and direct access are both fast	External fragmentation, and the size must be known at creation	Optical volumes, where nothing grows after writing
Linked	No external fragmentation, and the file grows into any free block	Direct access has to follow the chain, and a lost pointer loses the rest	A file allocation table, with the pointers collected
Indexed	Direct access with no external fragmentation and no chain	The index block costs space and caps the file until it is nested	The inode of a Unix file system

Table 3.31: *The three allocation methods, what each one buys, what each one costs, and where each is met in practice.*

All three answer one question, which blocks hold this file, and differ only in where the answer is written.

The Master Key: *A file allocation table is linked allocation with its pointers collected into one table, and an inode is indexed allocation.*

REAL Question 103

The Mwakitolyo office has one attendant and eight taps along a kilometre of pipe. He goes to each in the order the requests reached him, and by noon has walked four times the length of the pipe. A disk head can do worse.

A disk has 200 cylinders and the head is resting at cylinder 53. For the request queue given below, compute the total head movement under first come first served and under shortest seek time first.

The disk has 200 cylinders, numbered 0 to 199.

The read and write head is resting over cylinder 53.

Eight requests are waiting, in the order 98, 183, 37, 122, 14, 124, 65, 67.

The head is travelling towards cylinder 0. Neither of these two algorithms consults that direction, but the sweeping algorithms of the next question do.

Count head movement in cylinders only, and ignore rotational delay and transfer time.

Model Answer

First come first served serves the eight requests in the order they arrived and moves the head 640 cylinders, while shortest seek time first always takes the nearest outstanding request and moves it 236 cylinders, saving 404 cylinders of travel on exactly the same work.

What a cylinder is: A disk is a stack of platters, each with a head, and the tracks at one radius on every surface together form a cylinder. Moving the heads between cylinders is the seek, waiting for the wanted sector to come round is the rotational delay, and only the seek is under the control of a scheduling algorithm.

First come first served, move by move: The head goes from 53 to 98, which is 45 cylinders, then to 183, which is 85, then back to 37, which is 146, then to 122, which is 85, then to 14, which is 108, then to 124, which is 110, then to 65, which is 59, then to 67, which is 2.

Adding the first come first served moves: The first four moves, 45, 85, 146 and 85, come to 361 cylinders. The last four, 108, 110, 59 and 2, come to 279. The two parts together give 640 cylinders for the whole queue.

Shortest seek time first, move by move: From 53 the nearest waiting request is 65, which is 12 cylinders away, then 67 at 2, then 37 at 30, then 14 at 23, then 98 at 84, then 122 at 24, then 124 at 2, and last 183 at 59.

Adding the shortest seek moves: The first four, 12, 2, 30 and 23, come to 67 cylinders. The last four, 84, 24, 2 and 59, come to 169. Together they give 236 cylinders, and the queue is served in the order 65, 67, 37, 14, 98, 122, 124, 183.

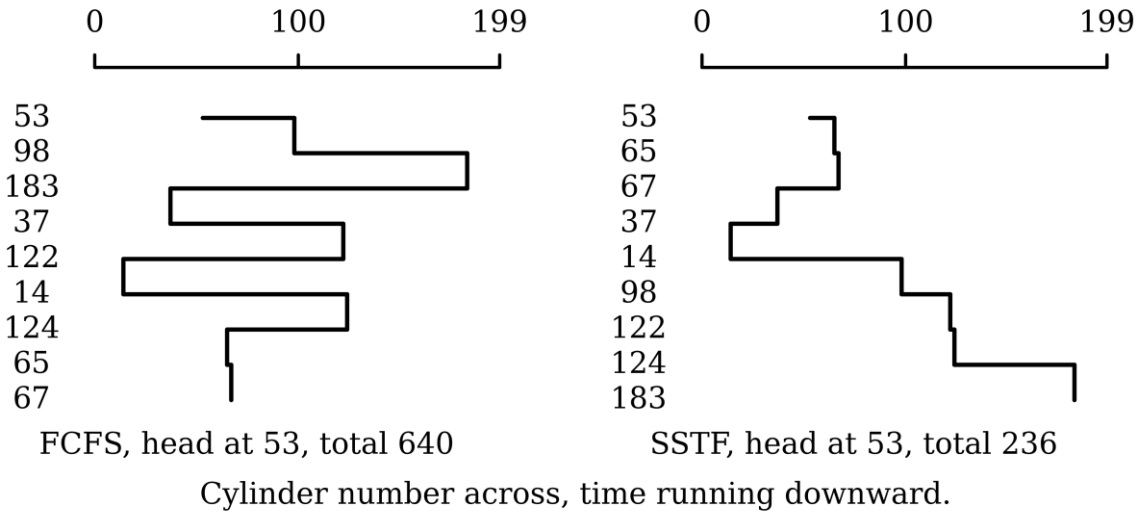


Figure 3.42: *The path of the head under each policy, cylinder number across and time running downward, from the same start at 53.*

Where the saving comes from: First come first served reads the queue and never the platter, so it crosses the whole disk whenever two requests in a row lie on opposite sides of the head. Its three longest moves, 146, 110 and 108, come to 364 cylinders between them, which is more than shortest seek time first spends on the entire queue.

Both policies serve all eight requests, and the whole difference between 640 and 236 is the order in which they were taken.

The Master Key: *First come first served is judged by arrival order and shortest seek time first by distance from where the head now stands.*

REAL Question 104

A Hungumalwa conductor works the aisle from back to front, taking every fare he passes, then walks the whole bus without collecting and starts at the back again. Everyone waits the same. Whether it costs him steps depends on one thing.

For the same disk and the same queue, with the head moving towards cylinder 0, compute the total head movement under SCAN, C-SCAN and LOOK, and rank all five algorithms by total head movement.

The disk is the one before: 200 cylinders numbered 0 to 199, with the head at cylinder 53.

The waiting requests are 98, 183, 37, 122, 14, 124, 65, 67.

The head is travelling towards cylinder 0, so SCAN and LOOK both sweep downward first and then turn.

SCAN runs on to the edge of the platter before turning; LOOK turns at the last request in the direction of travel.

C-SCAN serves in one direction only, and in the form used here that direction is upward: it sweeps to cylinder 199, jumps back to cylinder 0, and serves nothing during the jump.

Count head movement in cylinders only.

Model Answer

Sweeping downward, SCAN moves the head 236 cylinders and LOOK moves it 208, while C-SCAN serving upward moves it 382, so ranked by total head movement over this queue the order is LOOK first, then SCAN and shortest seek time first together on 236, then C-SCAN, and first come first served last on 640.

SCAN going down: The head takes 37 at 16 cylinders and 14 at 23, runs on to 0 at a further 14, then turns and sweeps up through 65, 67, 98, 122, 124 and 183, adding 65, 2, 31, 24, 2 and 59. The nine moves come to 236

cylinders. The run down to 0 serves nothing, since the lowest request was 14, but SCAN goes there because a sweep is defined by the edge of the platter and not by the queue.

LOOK going down: LOOK serves the requests in exactly the same order but turns at 14, the last request below the head, instead of running on to the platter edge. It goes 16, 23, then 51 straight up to 65, then 2, 31, 24, 2 and 59, which is 208 cylinders.

What those 28 cylinders are: The difference between 236 and 208 is 28, which is the 14 cylinders down to 0 and the 14 back up again. LOOK can never travel further than SCAN in the same direction, and it travels less whenever no request sits at the edge of the platter.

C-SCAN going up: Serving upward it takes 65, 67, 98, 122, 124 and 183, which is 12, 2, 31, 24, 2 and 59, then runs to 199 for a further 16, jumps the full 199 back to 0, and picks up 14 and 37 on the way out. That is 382 cylinders. Nothing is served during the jump, which is what makes the wait at the top of the platter much like the wait at the bottom.

Why the direction has to be stated: Swept upward instead, the same queue from the same starting cylinder gives SCAN 331 rather than 236, which is 95 cylinders more. A disk-scheduling question that names the queue and the starting cylinder but not the direction of travel has no single answer, and a total quoted without one is unfinished.

Rank	Algorithm	Direction	Cylinders moved	What it buys
1	LOOK	Downward first	208	The least travel here, because it turns at the last request
2	SCAN	Downward first	236	A bounded wait, since every cylinder is crossed each sweep
2	Shortest seek time first	Not used	236	The shortest next move every time, with no bound on any wait
4	C-SCAN	Upward only	382	Waiting time nearly equal at every cylinder on the platter
5	First come first served	Not used	640	Strict arrival order, and nothing else at all

Table 3.32: *All five algorithms over the one queue, ranked by total head movement, with SCAN and shortest seek time first tied.*

The ranking holds for this queue and this direction, and changing either of them changes the numbers in the fourth column.

The Master Key: *LOOK turns at the last request, SCAN turns at the platter edge, and C-SCAN never turns at all but jumps back.*

REAL Question 105

In the workshop at Kaisho a service card lives in a plastic wallet on the bench, three lines long, and every fitter trusts it. One of the three lines is sound and two have never been true at all.

Three statements about disk scheduling are given below. Identify the one that is correct, explain what is wrong with each of the others, and write a corrected version of each.

The service card in the plastic wallet reads:

A. SCAN and LOOK visit the requests in the same order, so the two always give the same total head movement.

B. Shortest seek time first always chooses the nearest request, so no request can be left waiting indefinitely.

C. C-SCAN serves requests in one direction only and returns without serving anything on the way back, so that waiting time is more nearly equal at every cylinder.

Model Answer

C is the sound line; A is wrong because LOOK turns at the last request while SCAN runs on to the edge of the platter, and B is wrong because shortest seek time first can leave a far cylinder waiting for as long as fresh requests keep arriving near the head.

Why C is right: C-SCAN treats the platter as a loop. It serves only while travelling one way, then returns without stopping, so a request just missed by the head waits one sweep rather than two, and the longest wait is roughly the same wherever the request lies.

What C costs, so the claim is not free: On this queue C-SCAN moves 382 cylinders against 208 for LOOK, which is 174 cylinders more. Uniform waiting is bought with that return jump, and a statement that C-SCAN equalises waiting is only correct because the jump serves nothing.

What A gets right: The first half of A is true on this queue. Sweeping downward, both SCAN and LOOK serve 37, 14, 65, 67, 98, 122, 124 and 183 in that order, so the eight requests really are visited in the same sequence by the two algorithms.

Where A fails: The total counts distance travelled, not requests served. SCAN also runs from 14 down to 0 and back up again, which is 14 cylinders each way and 28 in all, so it moves 236 against 208 for LOOK on the very queue that produced the same order.

A, corrected: SCAN and LOOK serve the requests in the same order, but SCAN also travels to the edge of the platter, so its total is equal to or greater than the LOOK total in the same direction, and never less.

Where B fails, and B corrected: Nearest means nearest to where the head is now, and that changes with every service, so a request at 183 need never become the nearest while requests keep arriving near 53. Corrected, B reads that shortest seek time first can starve a distant request precisely because it always chooses the nearest one.

Two of the three lines were plausible because each stated a true property and then drew a conclusion the property does not support.

The Master Key: *Same order of service does not mean same distance travelled, and always choosing the nearest request is what makes starvation possible.*

HOT Question 106

The Sao Hill yard has one crane and a driver who always lifts the nearest log. Fresh loads arrive beside the crane all morning. Nothing about his rule is broken, and the pile at the far fence will still be there on Friday.

Shortest seek time first gives lower average head movement than SCAN on most request queues. Argue which of the two a busy file server should use, and reach a judgement.

Model Answer

A busy server should use SCAN, or its practical form LOOK, and keep shortest seek time first only for a disk whose queue empties between bursts, because the deciding factor is not average distance but whether a far cylinder can be left unserved indefinitely, and that becomes possible the moment the queue stops emptying.

The saving is real: On the queue worked through above, shortest seek time first moves the head 236 cylinders against 640 for first come first served, a saving of 404. Choosing the nearest request exploits the fact that a file lies in nearby blocks, so the next request usually is close.

And it is not always a saving: On that same queue SCAN sweeping downward also totals 236, so the two are level, and LOOK does better still at 208. The advantage claimed for shortest seek time first is an average over many queues, not a promise about any one of them.

What nearest actually means: Nearest is measured from where the head is standing at this instant, and the head moves every time a request is served. Distance is therefore not a property of a request at all, which is why a request can be nearest at one moment and never nearest again.

Where that becomes starvation: A request at cylinder 183 is served only when nothing closer is waiting. If work keeps arriving around cylinder 53, nothing closer is ever absent, and the request at 183 waits without limit. No rule inside shortest seek time first can notice this, because the algorithm has no memory of how long anything has waited.

The load at which it starts to matter: The threshold is the point where the queue stops emptying. A disk that completes 120 requests a second offered 100 a second is 83 per cent busy, has 20 requests a second of spare capacity, and drains many times a minute; offered 118 it is 98 per cent busy with 2 to spare, and the queue is effectively never empty.

What SCAN gives instead: SCAN crosses every cylinder on every sweep, so the longest any request can wait is one sweep, at worst two traversals of a 200 cylinder platter, which is 398 cylinders of travel. That is a bound, and the whole point about shortest seek time first is that it has none.

The judgement: Prefer shortest seek time first on a lightly loaded disk, where the queue empties between bursts and no request can be overtaken for long. Prefer SCAN, in practice LOOK, once arrivals approach the service rate, because a bounded worst case is worth more to a server than a slightly shorter average.

The average is what the disk costs the installation, and the worst case is what one waiting user experiences.

The Master Key: *Shortest seek time first optimises the next move and guarantees nothing, while SCAN gives up a little distance to bound the longest wait.*

HOT Question 107

The Nzihi dispensary keeps its register in a locked cabinet and its rule on a note beside it, that assistants may read and only the clinician may write. The note is obeyed by everyone who reads it. The cabinet is obeyed by everyone.

Argue whether file permissions should be enforced by the operating system or by the applications that open the files, and reach a judgement.

Model Answer

The operating system must enforce them, because an application can only apply a rule to the path that passes through itself, and anybody who reaches the file by another program walks past the rule without breaking it, so a check placed in an application is advice while the same check placed in the kernel is a boundary.

What a file permission is: A permission names who may do what. The common scheme has three classes, the owner, the group and everybody else, and three rights, read, write and execute, giving 9 bits. Each class has 8 combinations, so the 9 bits carry 512 settings in all.

When the check happens: The rights are tested once, when the file is opened, and the result is recorded in the open-file entry. Every later read and write is allowed by that entry rather than tested again, which keeps the check cheap enough to apply to every file.

The case for the application: An application understands the meaning of the data and the file system does not. A records system can allow a clerk to see the names in a register but not the payments, a distinction the file system cannot express at all, because the smallest thing it can protect is a whole file.

The rest of that case: An application can also apply rules that have no place in a file system, such as allowing changes only during working hours, only from one office, or only when a second person approves. It can also log who looked at what.

Why none of that is enforcement: Those checks run in user mode inside a program the user chose to run. Copy the file with a different tool, open it in an editor, or read the raw device, and the checks are not bypassed so much as never invoked, and a rule skipped by not running its program protects nothing.

Why the kernel check cannot be avoided: Every route to the data goes through the open system call, and that runs in kernel mode inside the operating system itself. There is no way to reach the blocks without asking, and no way to ask without being checked, so the rule applies to every program equally.

The judgement: Enforce permissions in the operating system, and let applications add further restrictions on top of them, never in place of them. The test that settles it is whether a rule can be avoided by not running the program that applies it, and one that can is not a permission.

The operating system cannot see inside a file, so an application still has work to do, but behind a boundary rather than instead of one.

The Master Key: *A check is enforcement only when there is no route to the data that avoids it, and only the kernel sits on every route.*

HOT Question 108

The Ipogolo cashier writes what she is about to do in a day book, does it, then rules the line off. Ten seconds a transaction, every transaction. A file system keeps the same book, and the argument is over how much goes in it.

Journalling writes every change twice so that a file system can recover from an interrupted write. Argue whether the cost is worth paying, and reach a judgement.

Model Answer

It is worth paying for metadata and usually not for data, because journalling the file system structures costs a small extra write and removes an hour or more of checking after a failure, while full data journalling writes every byte twice and halves the useful write rate to protect against something most applications can detect for themselves.

What a journal is: A journal is a reserved area of the volume. The change a file system intends to make is written there first and committed, then applied to the file system proper, and only then is the journal entry cleared. After a failure the entries are replayed if committed and discarded if not.

Why an interrupted write is dangerous: One logical operation touches several structures. Creating a file writes a directory entry, marks blocks used in the free-space map and writes an inode. Power lost between them leaves the structures disagreeing, with blocks marked used that nothing points at, or worse, one block claimed by two files.

What the alternative costs: Without a journal the only repair is a consistency check of the whole volume. A 500 gigabyte volume read at 100 megabytes a second takes 5000 seconds, which is 83.3 minutes before the service can be offered again, and the check may still have to guess at what a half-finished operation intended.

What metadata journalling costs: Only the small structural records are written twice. A journal record of 128 bytes against a 4096 byte data write adds about 3.1 per cent to the bytes written, and the records are sequential, so they land in one place instead of scattering the head across the platter.

What full data journalling costs: Every byte of file content is written twice, once to the journal and once to its home. A disk that sustains 120 megabytes a second of useful writing now sustains 60, and on a write-heavy server that is not an overhead but a halving of the machine.

What each one actually guarantees: Metadata journalling promises that the file system is structurally sound after a failure, not that any file has its contents. Full data journalling promises the contents too, so a record is either wholly written or wholly absent, and never a torn half of each.

The judgement: Journal the metadata always, because 3.1 per cent buys back 83 minutes of downtime and a volume nobody can trust. Journal the data only where a partly written record would be accepted as real and cannot be detected afterwards, which is why a till or a patient record earns it and a media store does not.

The cost of a journal is paid on every ordinary day and the benefit is collected on the one day the power fails.

The Master Key: *Metadata journalling protects the file system and full data journalling protects the file, and only the second one doubles the writing.*

Chapter Four

DATABASES AND DATABASE MANAGEMENT SYSTEMS

Almost every organisation you will ever work in keeps its records in a database, and almost every one of them began by keeping those records in files. The move from one to the other is the subject of this chapter, and it is worth more marks than any other topic in your year. It is also the topic where the practical paper asks you to sit down and write statements that must run.

The chapter begins with what a database is and what a management system adds to it, then builds a design from an entity relationship diagram, normalises it, and proves the normalisation is sound. From there it turns to SQL: creating tables that defend themselves, changing rows, querying one table and then several, grouping, joining and nesting. The last third covers what happens when a database is in real use, where transactions, concurrency and indexes decide whether it works, and closes on security, on reaching a database from a program, and on the distributed and non relational systems that sit beyond a single server.

Three habits earn marks in this topic and nothing else does. Name the construct before you state the property, because most lost marks in database questions are a true fact attached to the wrong thing. Write out the statement rather than describing it, because the practical paper accepts nothing else. And check your own work against the data in front of you, since almost every claim in this subject is decidable and a claim you can decide is a claim you can be certain of.

From files to a database: what a DBMS actually adds

This section draws the line between a collection of files and a database, and says exactly what a management system puts between the two. It defines data against information, sets out the services a DBMS provides, and introduces the three schema architecture and the data independence it buys. Everything later in the chapter stands on this distinction.

BINDER Question 1

The fee clerk at Kisongo Secondary keeps three books. A student who changes stream is crossed out in one, rewritten in another, and missed entirely in the third. By June the three books disagree about who has paid.

Define a database, and explain how the database approach differs from the file based approach.

Model Answer

A database is a shared, organised collection of related data, stored once and described in a way that lets many different applications draw on it, while a file based system gives each application its own private file.

The data is integrated: In the file based approach the same fact is written into as many files as there are uses for it. In a database each fact is stored once, and every user reaches that one copy. The fee office, the head and the auditor read the same student record rather than three versions of it.

The description lives with the data: A spreadsheet knows nothing about what its columns mean; the meaning sits inside whichever program opens it. A database holds the definition of every table and column alongside the data itself, so a new program can discover the structure instead of being told it.

Access goes through one gate: Files are opened directly by whatever program wants them, so each program carries its own rules. In a database every read and every write passes through the management system, which is what makes a single set of rules possible at all.

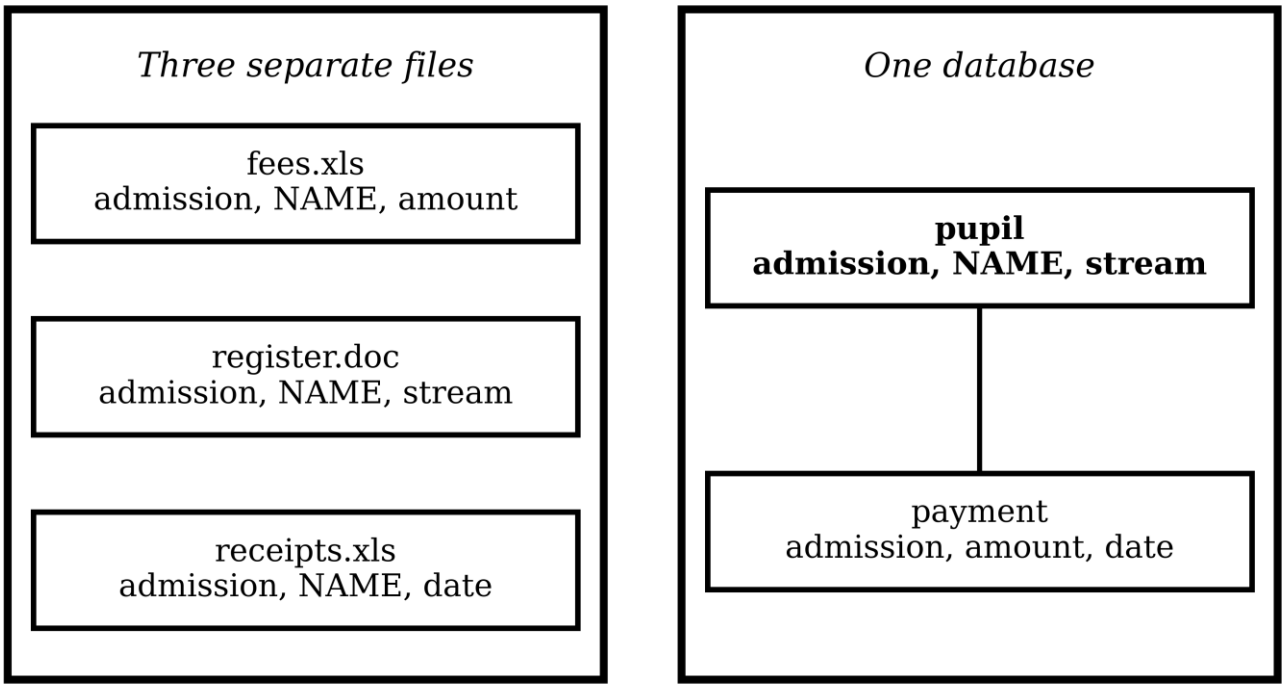


Figure 4.1: *The same student's name written into three files, and the same name held once in a database.*

The consequence is consistency: Because the fee clerk's three books each hold their own copy of a student's stream, correcting one does not correct the others, and by June they disagree. Storing the stream once removes the possibility of disagreement rather than merely reducing it.

The difference is not that a database is larger or faster, but that it holds each fact once and describes it in one place.

The Master Key: *A database stores each fact once and stores its description with it; files store each fact wherever it is needed.*

BINDER Question 2

Hundreds of payment rows fill a spreadsheet at Makuyuni. The head does not want the rows. She wants one sentence: which streams are behind, and by how much. The spreadsheet cannot answer her.

Distinguish data from information, and explain why a database stores data rather than information.

Model Answer

Data is raw recorded fact with no interpretation attached, and information is data that has been selected, combined or summarised so that it answers a question someone actually asked.

Data is the recorded fact: A row saying that admission K0142 paid a sum on a given date is data. It is true, it is stored, and on its own it settles nothing. Every payment row in the office spreadsheet is data in exactly this sense.

Information answers a question: The head's question is which streams are behind. Answering it means grouping the payments by stream, totalling each group, and comparing each total against what was due. The answer is information, and it did not exist anywhere until the question was asked.

Why the database stores the data: A database holds the payment rows, not the summary, because the summary is only one of many that might be wanted. Storing the totals instead would answer this question and destroy the ability to answer the next one, which may be by student, by month or by method of payment.

The summary is derived on demand: The management system derives information from stored data when it is asked to, using the query language. Because the derivation happens at the moment of asking, the answer reflects every payment recorded up to that moment rather than up to whenever a summary was last rebuilt.

Store the fact and derive the summary, because a stored summary is out of date the moment the next payment arrives.

The Master Key: *Data is what happened; information is the answer to a question about what happened, and a database stores the first to serve the second.*

BINDER Question 3

A padlocked cupboard at Ngaramtoni holds the fee ledgers. Only the bursar has the key, which is the whole of the security arrangement. When she travels, nobody can issue a receipt, and nobody can check one either.

State the services a database management system provides, and explain the purpose of each.

Model Answer

A database management system is the software layer between the user and the stored data, and it provides definition, manipulation, control of access, protection of correctness, recovery and concurrency, none of which a file based system provides at all.

Definition and manipulation: The system lets a designer declare what tables exist and what each column may hold, and lets a user insert, change, delete and retrieve rows without knowing how any of it is stored. These two services are what the data definition and data manipulation languages exist to express.

Controlled access: Rather than one key to one cupboard, the system grants named privileges to named users, so the head may read totals while only the bursar may record a payment. Access can therefore be shared without being surrendered, which the cupboard cannot manage.

Integrity and recovery: The system enforces declared rules, refusing a payment against an admission number that does not exist, and it restores the database to a correct state after a failure by replaying or undoing work that was in progress. A spreadsheet interrupted by a power cut offers neither.

Concurrency: Several users may read and write at the same time, and the system keeps their work from interfering. This is the service that is hardest to imitate by hand and the one whose absence causes the quietest errors, because nothing appears to go wrong at the moment it does.

Service	What it prevents
Definition	Columns whose meaning lives only in one programmer’s head
Manipulation	Every program writing its own file handling code
Controlled access	One key standing for the whole of security
Integrity	A payment recorded against a student who does not exist
Recovery	A failure part way through leaving the records wrong
Concurrency	Two users overwriting each other in silence

Table 4.1: *The six services, and the failure each one exists to prevent.*

Each service exists because the file based alternative fails in a way somebody has already paid for.

The Master Key: *A DBMS provides definition, manipulation, controlled access, integrity, recovery and concurrency.*

BINDER Question 4

The office at Kisongo moved its records to a faster disk over the holidays. On Monday every report was broken, because each one had been written to read the old file layout directly.

Describe the three schema architecture and explain the two kinds of data independence it provides.

Model Answer

The three schema architecture separates a database into an external level of individual user views, a conceptual level describing the whole database once, and an internal level describing how it is physically stored, so that a change at one level need not disturb the level above it.

The external level: Each user or application sees only the part of the database it needs, presented in the shape that suits it. The bursar's view shows fees owed, the head's shows totals by stream, and neither is aware of the other or of what else the database holds.

The conceptual level: One community description of the whole database sits beneath the views: every entity, every attribute and every relationship, stated once and without reference to how any of it is stored. This is the level a designer works at and the level normalisation acts on.

The internal level: Beneath that lies the physical reality of files, pages, record layouts and indexes. It decides how quickly things are found and how much room they take, and it is the only level that changes when a disk is replaced.

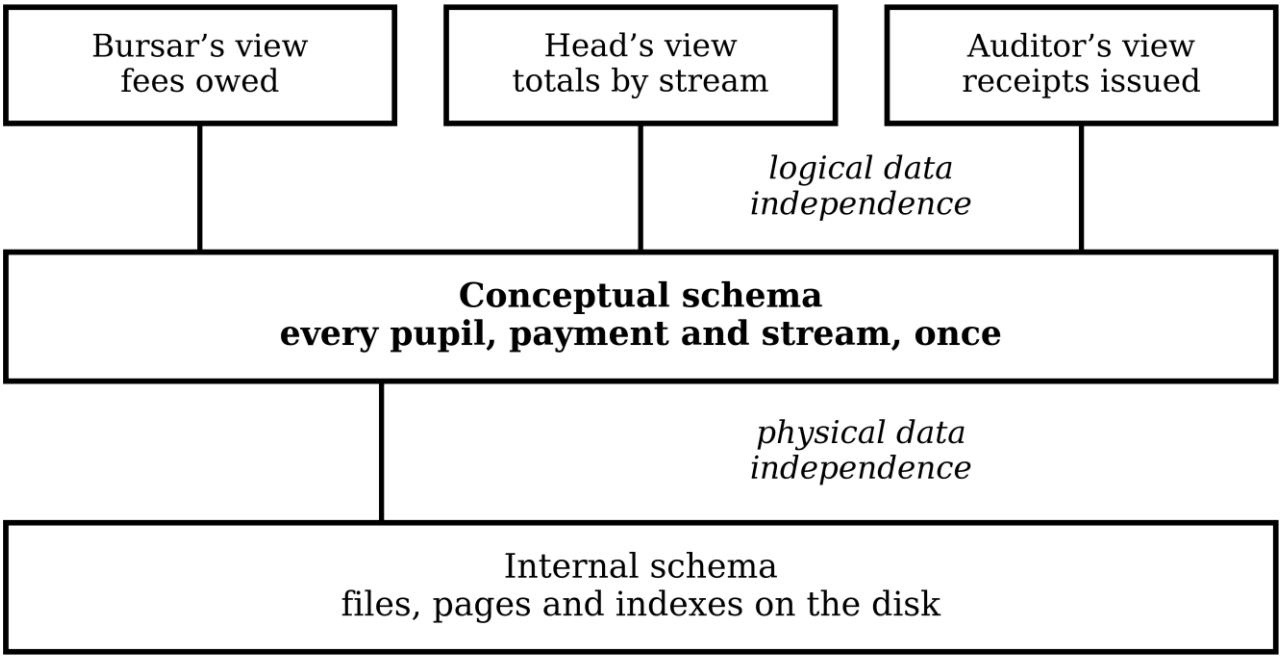


Figure 4.2: *The three levels, and the two mappings that absorb change between them.*

Physical data independence: Changing the internal level, by moving to a faster disk or adding an index, leaves the conceptual level untouched. The office reports broke only because they had been written to read the file layout directly, going round the architecture rather than through it.

Logical data independence: Adding a column or a table to the conceptual level leaves existing external views unchanged, so applications that never asked for the new column carry on working. This is the harder of the two to achieve, and the more valuable when it is achieved.

The architecture is worth having precisely because the two mappings absorb change that would otherwise reach every program.

The Master Key: *External, conceptual and internal; physical independence protects the design from the disk, logical independence protects the programs from the design.*

REAL Question 5

Two clerks at Makuyuni open the same fee spreadsheet on Monday morning, each on her own machine. Both correct the same student's balance. Both save. At noon one of the two corrections has vanished.

Trace what happened to the missing correction, name the problem, and explain how a database management system prevents it.

Model Answer

The second clerk to save overwrote the first clerk's work, because each had opened her own copy of the whole file and saved the whole of it back, and this is the lost update problem.

What each clerk actually held: Opening the spreadsheet copies the entire file into the machine's memory. From that moment each clerk is working on a private copy that knows nothing about the other, and neither copy is told when the file on the shared drive changes.

The order of events: The first clerk saves, and the shared file now carries her correction. The second clerk then saves the copy she opened before that happened, which still holds the original balance for that student and her own correction for another. Writing it back restores the old value.

Why nothing warns anybody: Both saves succeeded. No rule was broken, no error appeared, and the file is perfectly readable afterwards. The only evidence is that a balance somebody remembers correcting is wrong again, which may not be noticed for weeks, and by then nobody can reconstruct which of the two figures was the corrected one.

What the management system does instead: A database does not hand out copies of the file. Each clerk changes one row through the system, which serialises the two changes so that the second is applied to the result of the first rather than to a picture taken before it. Where the two genuinely conflict, one clerk is made to wait until the other has finished, or her change is refused and she is told so.

The name to use: This is the lost update, the first of the concurrency problems, and it is the reason concurrency appears in the list of services a management system provides. Note that it is not caused by the clerks working carelessly and cannot be cured by asking them to work carefully.

The correction was not deleted by anyone; it was overwritten by a copy that predated it.

The Master Key: *Two writers holding private copies of the same file produce a lost update, and only serialised access through one system prevents it.*

REAL Question 6

A card taped above the counter at Ngaramtoni lists three things the office believes about its new database. Two of the three are wrong, and the one that is right is the one the staff trust least.

Identify which of the three statements on the card is correct, say precisely what is wrong with each of the other two, and correct them.

A. A database removes all redundancy, so no fact is ever stored twice.

B. A database stores the description of its own data alongside the data.

C. A database is simply a very large and well organised spreadsheet.

Model Answer

Statement B is correct, and A and C are both wrong: A overstates what a database does about redundancy, and C mistakes a difference of kind for a difference of size.

Why B is right: A database holds the definition of every table, column and constraint in a catalogue that is itself part of the database and can itself be queried. This is what allows a program to discover the structure rather than be told it, and it is the property that most sharply separates a database from a file, whose meaning lives inside whichever program opens it.

What is wrong with A: A database controls redundancy; it does not remove it. A foreign key deliberately repeats a value so that two relations can be joined, and a designer may repeat more than that on purpose where a report would otherwise be too slow. The correct claim is that a database eliminates uncontrolled redundancy, meaning redundancy nobody planned and nothing keeps in step.

What is wrong with C: Size is not the difference. A spreadsheet has no declared types, no keys, no constraints, no controlled access, no recovery and no concurrency control, and it hands out whole copies of itself to be edited privately. A database with three rows still has every one of those things, and a spreadsheet with three million rows still has none of them.

Why the right one is trusted least: Staff meet the catalogue only indirectly, so the property that matters most is the one they never see. Redundancy and size are visible every day, which is why the two visible claims are the two that acquire confident wrong versions, and why a card written from experience alone gets two of its three statements wrong.

Correct A to controlled redundancy and C to a difference of kind, and B needs no correction at all.

The Master Key: *A database controls redundancy rather than removing it, and it differs from a spreadsheet in kind, not in size.*

REAL Question 7

The bursar at Kisongo describes her system with some pride. Fees in one workbook, receipts in another, the stream register in a third, and a fourth she keeps at home in case the others are lost.

Identify four distinct problems with this arrangement, and explain what causes each one.

Model Answer

The arrangement suffers from uncontrolled redundancy, data inconsistency, isolation of related data and a total absence of security and recovery, and every one of the four follows from the same cause, which is that each file is owned by the use that created it.

Uncontrolled redundancy: A student's name and admission number appear in all four files, because each file was built to be usable alone. Nothing records that these are the same student, and nothing keeps the four copies in step.

Data inconsistency: Redundancy on its own is only wasteful. It becomes an error the first time one copy is updated and the others are not, which happens whenever a student changes stream or corrects a spelling. After that the system holds two different answers to the same question and no way to say which is right.

Isolation of related data: A question spanning two files, such as which students have a receipt but no recorded payment, cannot be answered without opening both and matching them by hand. The relationship between the files exists in the bursar's understanding and nowhere in the system, so it cannot be checked, enforced, or handed to anybody else.

No security and no recovery: Anybody who can open a workbook can change any figure in it, and the copy kept at home is protection against loss of the file but not against loss of a single correct value. A failure part way through a set of related changes leaves the files disagreeing with no record of what was in progress.

The single cause: Each file belongs to the task that created it rather than to the school. Once ownership sits with the task rather than with the data, duplication is unavoidable, because every new task needs its own copy of whatever it refers to, and everything else on this list follows from the duplication.

The fourth workbook is a backup of the arrangement, not a repair of it.

The Master Key: *File based systems fail through redundancy, inconsistency, isolation and the absence of security and recovery, all because each file is owned by a task.*

HOT Question 8

A tailoring shop near Makuyuni keeps its customers in a notebook. A consultant offers to move the notebook into a database for a fee larger than the shop's monthly takings. The owner asks whether she needs it.

Assess whether a database is always preferable to a file based arrangement, and reach a judgement.

Model Answer

A database is not always preferable, and the condition that decides it is whether the data is shared, related or long lived; where it is none of the three, the file is the better engineering.

The case for the database: Every argument in this section holds. Redundancy is controlled, consistency is enforced, related data can be queried together, access is granted rather than surrendered, and failures are recoverable. None of these advantages goes away because an organisation is small.

The case against, on cost: A management system has to be bought or installed, configured, backed up and understood. Somebody must design the schema and somebody must maintain it. For a shop with one notebook and one reader, that cost buys services nobody in the shop will ever call on.

The case against, on fit: Some data is genuinely not relational. A single sequence of readings, a set of images, a log written once and read in order: these have no relationships to enforce and no concurrent writers to serialise, and forcing them into tables adds work without adding safety.

The three tests that decide it: Ask whether more than one person or program writes the data, whether the data has relationships that must be kept true, and whether it must survive long enough to be corrected. Each yes moves the balance decisively towards a database, and the first yes usually settles it.

The judgement: The tailoring shop should keep the notebook, because one writer, no relationships and a short life mean the database services would be paid for and never used. A school fee office should not, because all three tests are yes and the third alone is enough.

What changes the answer: The moment a second person begins writing, or the moment the notebook starts referring to something kept elsewhere, the shop's answer flips. The right time to build the database is when the first of the three tests turns, not when the pain arrives.

Sharing, relationships and longevity decide it, and a database bought without any of the three is a cost with no return.

The Master Key: *Prefer a database once the data is shared, related or long lived; below that threshold a file is the honest choice.*

HOT Question 9

Two changes reach the office at Ngaramtoni in the same week. The disks are replaced, and a new column is added to hold a student's guardian. Only one of the two forces every report to be rewritten.

Compare physical and logical data independence, and judge which is harder to achieve and which is worth more.

Model Answer

Physical data independence is easier to achieve and logical data independence is worth more, because the physical level is hidden by the architecture itself while the logical level is what applications were written against.

What each one protects: Physical independence protects the conceptual design from changes in storage, so replacing a disk or building an index changes nothing above it. Logical independence protects existing external views from changes in the conceptual design, so adding the guardian column leaves reports that never mentioned it alone.

Why physical is easier: Nothing above the internal level ever names a file, a page or an index. The management system is the only thing that does, so a change there is absorbed by one piece of software that was built to absorb it. The independence comes free with the architecture.

Why logical is harder: External views are written against the conceptual design in detail: named tables, named columns, an assumed shape. Adding a column is usually safe, but splitting a table, renaming a column or changing a relationship is visible to anything that referred to what changed.

Where views earn their keep: The remedy is to let no application read the base tables directly, giving each one a view instead. When the conceptual design changes, the view is redefined to present the old shape from the new tables, and the application never learns that anything happened.

Why logical is worth more: Disks are replaced rarely and by one team. Designs change constantly, because the organisation keeps thinking of new things to record, and each change threatens every program written so far. The independence that protects against the frequent event is the one that pays.

The judgement: Physical independence should be expected and is largely a property of using a database at all. Logical independence has to be designed for, by defining views early and never letting an application reach past them, and it is the one worth spending effort on.

The disks were the easy change; the new column was the one the architecture had to earn its keep on.

The Master Key: *Physical independence comes with the architecture; logical independence must be designed for with views, and it is the one that pays.*

Database models, historical and current

A database model is the answer to one question: what shape does the stored data take? Eight models are named in this section, from the hierarchical and network models that came first, through the relational model that displaced them, to the object oriented, document, key-value, wide-column and graph families that sit beside it

now. Two things are often listed among them and belong elsewhere, and this section says where: an entity relationship diagram is a notation for design, and a flat file is what an office has before it has chosen a model at all. Every example here is drawn from a district land register.

BINDER Question 10

Before a single parcel is written down, the office at Mererani has to settle what a record is allowed to look like, what may be done to one, and what makes one wrong. Nothing can be entered until those three are fixed.

Explain what a database model is, and state what any model must specify.

Model Answer

A database model is the logical arrangement a database management system uses to hold data and the relationships between it, and any model must specify three things: the structures data may be held in, the operations allowed on those structures, and the constraints that decide whether a stored state is valid.

The structures it allows: Every model names the thing data goes into and the thing that joins one to another. The hierarchical model has records and parent links, the relational model has relations of rows and columns, and a graph model has nodes and edges. Naming the structure fixes what can be represented at all.

The operations it defines: A model must also say what may be done with those structures, whether that is following a link from one record to the next or applying an operator to a whole relation and getting a relation back. A description of shape with no operations describes a filing cabinet, not a database.

The constraints it can enforce: The third part is the rules the model can state about a valid database, such as which values identify a record and which references must lead somewhere. A model is judged partly by which of those rules it can express rather than leave to whoever writes the programs.

A model is neither a product nor a drawing: MariaDB is a product and the relational model is what it implements, which is why other products implement the same model. A model is not the diagram made while designing either, and that is the reason a diagram never belongs in a list of models, however often it is put in one.

Choose the model and you have chosen what the register can hold and what it can refuse, before a single parcel is entered.

The Master Key: *A database model specifies the structures data is held in, the operations allowed on them, and the constraints that keep a stored state valid.*

BINDER Question 11

The old parcel files at Naberera hang on a wire, each plot beneath the single ward it belongs to. A plot on a boundary belongs to two wards, and the wire has no way of saying so.

Describe the hierarchical and network models, and explain the single structural difference between them.

Model Answer

The hierarchical model arranges records in a tree in which every record has exactly one parent, and the network model arranges the same records in a graph in which a record may have several parents, and that one permission is the whole structural difference between them.

The hierarchical model: Records sit in levels, each linked downwards to its children and upwards to the single parent that owns it. A district owns its wards, a ward owns its parcels, and a program reaches a parcel by starting at the root and following the links down.

The network model: The same records and the same links, with one restriction lifted, since a record may be owned by more than one parent. A parcel on a boundary is stored once and reached from either ward, so a many to many relationship is held directly instead of being broken up first.

The difference, stated exactly: It is the number of parents a record may have, one against several, and nothing else. A common error is to call the network model a tree in which a record has several parents. A tree is defined by every node having exactly one parent, so a structure allowing several is not a tree; it is a graph.

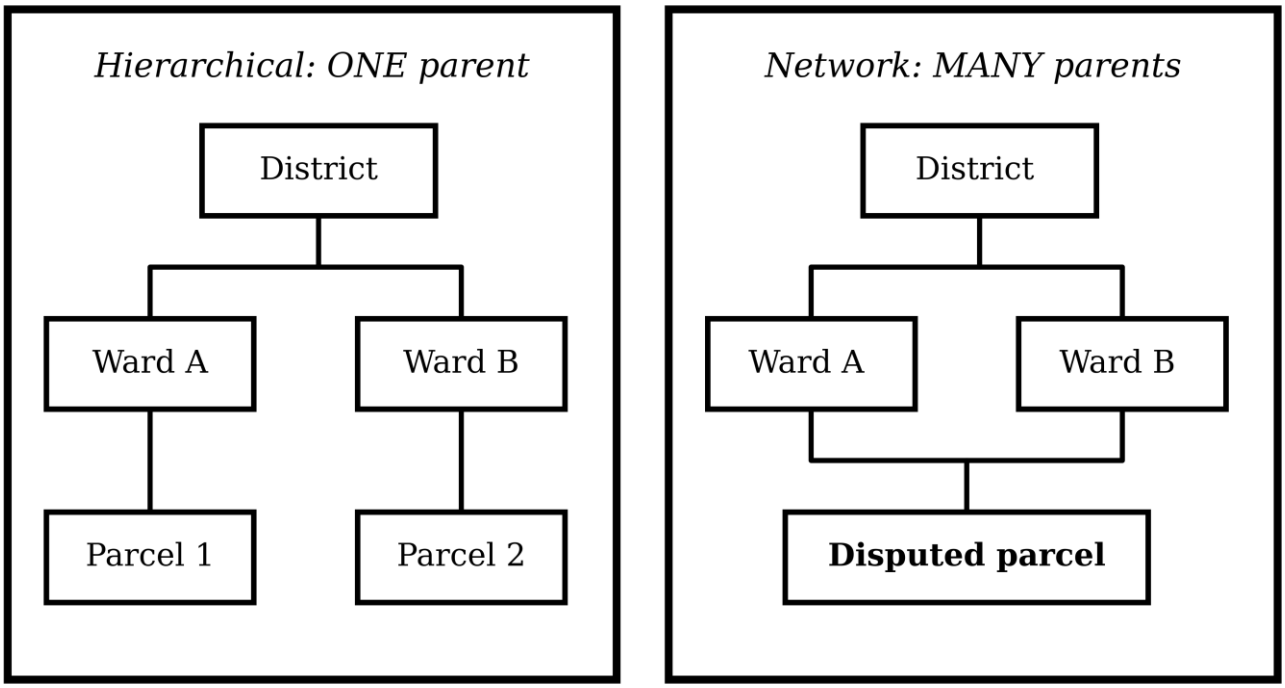


Figure 4.3: *One parcel under a hierarchy, which allows a single parent, and under a network, which allows two.*

What the two share: Both are navigational. A program must know the path and walk it, so a question the links were never built for is answered by reading everything or by rebuilding the structure, and the programs end up written against the storage rather than against the data.

One parent or several is the difference, and everything else the two models do they do alike.

The Master Key: *The hierarchical model is a tree with one parent per record, and the network model is a graph, because several parents are what make it one.*

BINDER Question 12

Nobody at Galapo missed the old chart of which record hung beneath which. The new register answered a question the office had never asked before, and nothing about the way the parcels were stored had to change to answer it.

Describe the relational model and explain why it displaced the two that came before it.

Model Answer

The relational model holds all data in relations, which are named tables of rows and columns with no pointers anywhere, and it displaced the hierarchical and network models because a request names the data wanted rather than the route to it.

What a relation holds: A relation is a set of rows over named columns, and every value in it is single. Nothing is stored to say where a row lives or what it is linked to, and two relations are joined because a value in one also appears in the other. Order of rows and columns carries no meaning.

Access by value, not by path: A request names the parcels it wants by their properties, and the system decides how to find them. Nothing in the request mentions a link or a file, so the stored arrangement can be changed without one request being rewritten.

Why the older two gave way: A new question needs no new structure, only a new request against the relations already there. A many to many relationship needs no owner set, only a further relation holding the pairs. And programs stopped carrying a map of the database inside them, which is what had made every change expensive.

A claimed advantage that is not one: Students are often taught, wrongly, that access becomes faster as the data grows. It does not, under any model, because searching more rows takes longer than searching fewer. What a database can do is stop the cost rising as fast as the data by keeping an index, and an index is never free.

The first relational systems were the slower ones and they still won, because the route was the thing nobody wanted to maintain.

The Master Key: *The relational model stores everything in relations and is asked by value, so a new question needs a new request rather than a new structure.*

BINDER Question 13

A survey report for one plot at Mererani will not go into a row. It carries a boundary description of uncertain length, the names of whoever was present, and photographs of the corners, and every attempt to force it into columns loses something.

Describe the object oriented model, and the document, key-value and graph families, and say what problem each answers.

Model Answer

The object oriented model stores data and the operations that act on it together, as objects with identity and inheritance, while the families after it each hold one shape well: a nested record, a value fetched by its key, rows whose columns need not match, and a network of things and links.

The object oriented model: An object carries its own identity, its attributes and the methods that act on it, and one class may inherit from another. The problem it answers is the survey report: a rich structure a program holds in one piece and then has to take apart to store and rebuild to use.

Document stores: A document store keeps that whole structure as one self describing record, with fields nested inside fields. The problem it answers is records of the same kind that do not all carry the same fields, where a relation needs columns most rows leave empty.

Key-value stores: A key-value store keeps a value under a key and hands it back when the key is given, without looking inside the value. The problem it answers is fetching one thing quickly where nothing within the value ever has to be searched.

Wide-column stores: A wide-column store keeps rows whose columns are gathered into families, and one row need not carry the columns the next one carries. The problem it answers is a very wide and mostly empty table spread across many machines.

Graph stores: A graph store keeps nodes and edges, and an edge is a stored thing in its own right with properties of its own. The problem it answers is data whose worth lies in the connections, such as tracing a plot through every transfer of ownership.

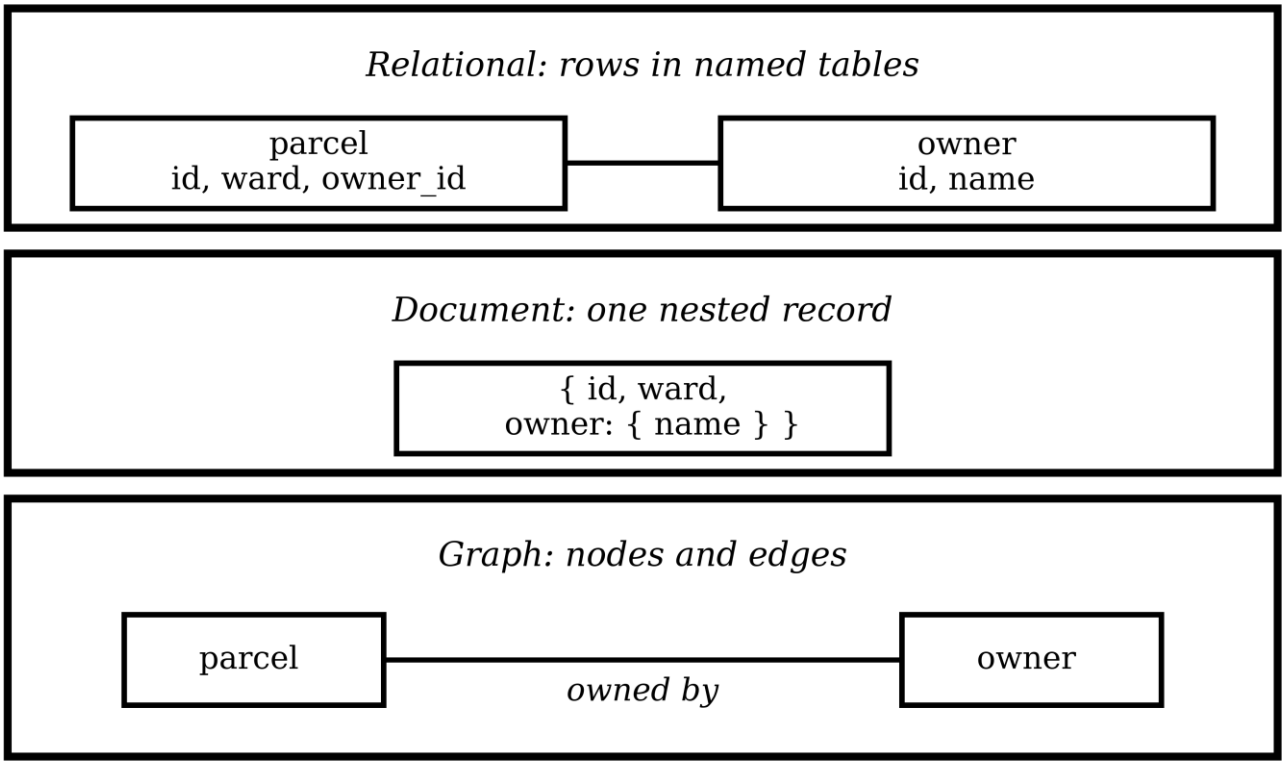


Figure 4.4: *The same parcel and its owner stored as related rows, as one nested document, and as two nodes joined by an edge.*

None of these four replaces the relational model; each is narrower, and better only at the shape it was built for.

The Master Key: *Objects keep data with its methods, documents keep one nested record, key-value pairs keep a value under its key, wide columns keep uneven rows, and graphs keep the links themselves.*

REAL Question 14

Four facts are wanted for every plot in the district: which district, which ward, who owns it, and whether anyone is arguing over it. The office at Naberera has money for one system and must decide what shape to build it in.

A district land register is to hold, for each parcel, the district, the ward, the owner and any dispute. Work out which model suits it, and justify the choice against two rejected alternatives.

Model Answer

The relational model suits this register, because every fact it holds is a regular attribute of a parcel, the same wards and owners are shared by many parcels, and the office asks its questions across all parcels at once rather than down one.

What the data actually looks like: Each parcel carries the same four facts, and three of them are shared with other parcels, since many parcels lie in one ward and one owner may hold several parcels. Only the dispute is optional, and a parcel with no dispute is the ordinary case rather than an exception.

Why the relational model fits: One relation for parcels, one for wards, one for owners and one for disputes, with each shared fact stored once and referred to by value. A question nobody planned for, such as which parcels in Galapo are disputed and owned from outside the district, is then answered without altering anything stored.

The first rejected alternative, a hierarchy: It commits the register to one direction, district and then ward and then parcel. That serves questions asked downwards, and the office also asks by owner and by dispute, which means walking every branch to answer them. A parcel on a boundary has no place in it at all.

The second rejected alternative, a document store: One document for each parcel, with its ward and owner nested inside, reads well for a single parcel and duplicates everything else. An owner holding land in Mererani and Galapo sits inside both documents, so correcting a name means finding every document carrying it, which is the duplication the register exists to end.

What would change the answer: If each parcel also carried a survey whose fields differed from parcel to parcel, a document store would earn a place beside the relations for that part alone. The choice follows the shape of the data and the questions asked of it, and nothing else decides it.

Regular facts, shared by many parcels and asked about in sets, make the relational case here without any need to argue it.

The Master Key: *Where every record carries the same shared facts and the questions cross many records at once, the relational model fits and the navigational and document alternatives do not.*

REAL Question 15

A page typed for the new staff at Galapo lists three claims about database models. One of the three is exactly right, and it is the one that sounds least like the others.

Identify the true statement, say precisely what is wrong with each of the other two, and correct them.

A. The network model is a tree in which a record may have several parents.

B. The entity relationship model is a way of designing a database, not a way a database stores data.

C. A flat file is the simplest of the database models.

Model Answer

Statement B is the true one, and the other two are both wrong: A contradicts itself, because a structure whose records may have several parents is not a tree, and C offers the absence of a model as though it were one of the models.

Why B is right: An entity relationship model is a notation for designing a database before any product is chosen. No management system stores anything as entities and relationships; the diagram is converted into whatever the chosen model does store, which in practice means relations. Putting it beside the hierarchical, network and relational models puts a drawing among ways of storing data.

What is wrong with A: A tree is defined by every node having exactly one parent, under a single root. Allowing a record several parents takes that defining property away, so what the statement describes is not a tree but a graph. The sentence denies in its second half what it asserts in its first.

A corrected, in one line: The network model is a graph in which a record may have several parents. That is not a repair of the wording alone, because the several parents are the entire structural difference between the network model and the hierarchical one, and calling the result a tree hides the very difference the sentence was written to state.

What is wrong with C: A flat file declares no structures, defines no operations and enforces no constraints, so it specifies none of the three things a model must specify. It is not the simplest model but what an office has before it has any model, which is why a model is described as replacing it.

Why the two wrong ones go wrong together: Both come from wanting one tidy list. One makes the network model a variety of tree so the list looks orderly, and the other puts the starting point at the head of a list of destinations. Naming the defining property first settles both of them.

Correct A to a graph, strike C from the list of models, and B stands exactly as it is written.

The Master Key: *A tree allows one parent, so several parents make a graph, and a flat file is the absence of a model rather than the simplest one.*

REAL Question 16

One plot at the edge of Naberera lies in two wards at once, and both ward officers count it in their returns. The register is asked to record that as a fact rather than settle it as an argument.

A parcel lies across two wards. Trace what each of the hierarchical, network and relational models does with it, and say which of the three stores the fact without distortion.

Model Answer

The hierarchy must either lose one ward or hold the parcel twice, the network can hold it once with both wards as parents, and the relational model holds it once and records the two memberships as two rows, which is the one arrangement that stores the fact without distortion.

What the fact actually is: It is a relationship between one parcel and two wards, not a property of the parcel. The moment ward is treated as a property of a parcel, one value has to be chosen, and any structure built on one value per parcel forces that choice sooner or later.

Under the hierarchical model: One parent is the rule, so there are three ways out and all three distort. File the parcel under one ward and the other membership is lost. File a copy under each and the register holds two parcels where there is one, and the copies drift apart. Invent a parent for shared parcels and neither ward is recorded.

Under the network model: Two wards may own the same member, so the parcel is stored once and reached from either ward, and the fact is represented exactly. The cost is that the double ownership is part of the structure, declared before any data goes in, so a third ward means altering the structure rather than adding data.

Under the relational model: The parcel appears once in the parcel relation and each ward once in the ward relation, while a third relation holds one row for every membership, which is two rows here. A third ward is a third row. The rule that produces that third relation is taken up later in the chapter.

Which one holds it truthfully: The relational model, because the fact goes in as data rather than as structure. The hierarchy distorts it whichever way it turns, and the network holds it only where somebody foresaw the double ownership, while a further row costs nothing and nobody has to foresee it.

Two wards became two rows, and nothing else in the register had to move.

The Master Key: *A parcel in two wards is a relationship and not a property, and only a model that stores it as data rather than as structure holds it without distortion.*

HOT Question 17

A visiting trainer tells the Galapo office that tables are finished, and that everything serious now runs on documents and graphs. The district officer, whose register has not lost a row yet, asks what she is meant to do about hers.

Assess whether the relational model is now obsolete, and reach a judgement.

Model Answer

The relational model is not obsolete, and the condition that decides the matter is the shape of the data together with the questions asked of it: keep relations wherever the facts are regular, shared and asked about in sets, and reach for another model only where one of those three fails.

The case that it is finished: The complaints are real ones. A declared structure is an obstacle where the fields keep changing, joining across relations costs more as the relations grow, spreading one relational database across many machines is genuinely hard, and a great deal of new work is built on document and graph stores.

What the complaints actually reach: Each of them describes a kind of application rather than a defect in the model. A declared structure is an obstacle only where the facts are unknown, and where they are known it is the thing that keeps them correct while several programs write to the same data.

The case that it is not finished: Most institutional data is regular, shared and long lived, which is the case relations were designed for. The model states a constraint once and enforces it for every program, and its foundation is what lets a system rewrite a request and still be certain of returning the same answer.

The direction the alternatives have moved: The families that were offered as replacements have added declared validation, transactions and query languages borrowing heavily from SQL. Movement in that direction is not the behaviour of a family replacing a dead model, and those systems are taken up in their own right later in this chapter.

The judgement: Not obsolete, and not universal either. The relational model stays the default for a record of fact, and another model is chosen beside it where one of three things holds: the records are genuinely irregular, the questions are about paths of unknown length, or the volume must be spread across many machines.

What would change it: If the district register began keeping a scanned survey and unpredictable field notes for every parcel, that part would be better held as documents while the parcels, wards and owners stayed in relations. The verdict follows the data, and the register at Galapo has data that has not changed shape.

The model is old in the way arithmetic is old, and the register the trainer wants replaced should stay where it is.

The Master Key: *The relational model is not obsolete but is the default for regular shared data, and another model joins it only for irregular records, questions about paths, or volume that must be spread.*

HOT Question 18

Two proposals for rebuilding the register reach the district office at Mererani in one week. One is built of rows and keys, the other of nodes and edges, and both promise every answer the office has ever needed.

Judge whether a district land register should be built on a relational or a graph database, and state the condition that decides it.

Model Answer

Build the register on the relational model and put a graph beside it only if connections become the main business, because the condition that decides it is whether the register's ordinary questions are about sets of parcels or about chains of connection whose length is not known in advance.

The case for the graph: Land is full of connections. This parcel borders that one, this owner inherited from that one, and a dispute joins families linked through somebody else. A graph stores a connection as a thing in its own right, so a question following a chain of unknown length is natural in it and awkward anywhere else.

The case for relations: The register's daily work is not chains. It is the parcel, its ward, its owner and whether it is disputed, together with counts by ward, and those are questions about sets of parcels that all carry the same facts. Relations answer them directly and declare the rules that keep them true.

What the graph would cost here: Fewer people in a district office can operate one, and a register must be workable by whoever holds the post next and readable years afterwards. Constraints a relational system declares in a line are commonly left to application code in a graph store, so the register would defend itself less.

The question that settles it: Count the register's real questions and ask how many need a path whose length is unknown when the register is designed. If the answer is a handful, the paths are short and known in advance, and relations win on every other ground: familiarity, declared rules and the ability to answer in sets.

The judgement: Build it relationally. Should a tribunal later need to trace succession and adjacency across generations, load a graph from the register and ask the chain questions there, keeping the relational database as the record of fact. What must not happen is the register itself becoming a graph maintained by hand.

What would flip it: If tracing chains of succession became the register's main duty rather than saying who holds what today, the graph becomes the record and the relations become the copy taken from it. That is a change in the work of the office, not a change in what technology offers.

Ownership today is a statement about one parcel, and the register's first duty is that statement.

The Master Key: *Keep a land register relational unless its main questions are chains of unknown length, in which case a graph earns a place beside the relations rather than instead of them.*

The relational model: structure, keys and the three integrity rules

The relational model is what allows a database to be reasoned about rather than merely used, and its vocabulary is exact. This section defines a relation and its parts, works down the key family from the superkey to the alternate key, and closes on the three integrity rules that decide which stored values are allowed to exist at all. Every later section of this chapter, from the diagrams through to the SQL, is written in the words settled here.

BINDER Question 19

Every book at Endasak Secondary carries a stamped number inside its cover, and no two copies carry the same one. The register that lists them has a shape, and the parts of that shape have names most readers never learn.

Define relation, tuple, attribute and domain, and state what the degree and the cardinality of a relation are.

Model Answer

A relation is a named set of tuples over a fixed list of attributes, a tuple is one row of it, an attribute is one named column of it, and a domain is the set of values that a given attribute is permitted to take.

The relation and the table: A relation is the mathematical object: a set of tuples, with no duplicates in it and no order among them. A table is how a relation is stored and displayed by a management system. Say relation when you are designing and table when you are writing SQL, and do not treat the two words as one.

The attribute and its domain: An attribute is a named column of the relation, such as accession_no or shelf. Its domain is the set of values that column may hold: eight characters for an accession number, a date for borrowed_on, one of the shelf codes for shelf. A domain belongs to an attribute and not to any one row.

The tuple: A tuple is one row, holding exactly one value from the domain of every attribute, and those values belong together because they describe the same thing. In the library relation, one tuple is one physical copy standing on the shelves; it is not one title, and it is not one loan.

Degree and cardinality: The degree of a relation is the number of its attributes and the cardinality is the number of its tuples. A relation holding accession_no, isbn, title, author and shelf has degree five whatever else happens to it, while its cardinality rises by one with every copy the library accepts.

Degree is settled by the design and cardinality is settled by the data, which is why one of the two changes on a Monday morning and the other does not.

The Master Key: *A relation is a set of tuples over named attributes drawn from declared domains; its degree counts the attributes and its cardinality counts the tuples.*

BINDER Question 20

A number stamped inside the cover finds one book at Kifaru. The ISBN printed on the back finds it too, most of the time. Only one of them appears on the spine label, and the choice was not arbitrary.

Define superkey, candidate key, primary key and alternate key, and explain how the four are related to one another.

Model Answer

A superkey is any set of attributes whose values identify a tuple uniquely, a candidate key is a superkey with nothing in it that could be removed, the primary key is the candidate key the designer nominates, and every candidate key not nominated is an alternate key.

The superkey: A set of attributes is a superkey when no two tuples of the relation can agree on all of it. Adding a further attribute to a superkey leaves a superkey, so a relation with one identifying attribute already has many of them, and most of those are carrying passengers.

The candidate key: A candidate key is a minimal superkey, meaning that taking any attribute out of it destroys the uniqueness. In the library relation accession_no is a candidate key, while accession_no together with title is a superkey that is not one, because title is doing no work in it.

The primary key: The primary key is the one candidate key the designer chooses to identify tuples and to be pointed at from other relations. Nothing in the stored data makes that choice; the designer makes it, on stability, on brevity, and on the certainty that a value will always be there.

The alternate key: Every candidate key that was not chosen becomes an alternate key. It is still unique and it may still be declared unique, but it is not the identifier other relations refer to. In the figure below, accession_no is the primary key and isbn is the alternate one.

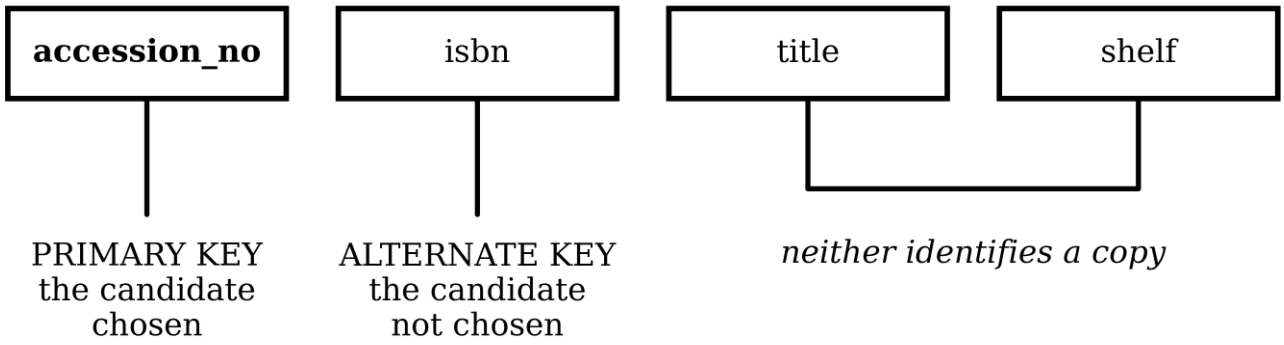


Figure 4.5: *Two candidate keys in one relation, one chosen as the primary key and one left as the alternate.*

The four names describe one nesting, since every primary key and every alternate key is a candidate key, and every candidate key is a superkey.

The Master Key: *A superkey identifies, a candidate key identifies minimally, the primary key is the candidate key chosen, and the rest are alternate keys.*

BINDER Question 21

A loan was recorded at Lembeni against a number nobody could find on any shelf. The borrower had gone, the book had not, and the line in the register pointed at a copy the library had never owned.

Explain what a foreign key is and state what it is for.

Model Answer

A foreign key is an attribute, or a set of attributes, in one relation whose values are required to match a primary key value in another relation, and it is there to record a relationship between the two and to keep that relationship true.

Where it sits and what it points at: The foreign key lives in the relation on the many side of the relationship. In the library, loan carries accession_no as a foreign key and it points at accession_no in book, which is the primary key there. The pointing runs one way only, and it is declared in loan.

The domains have to agree: A foreign key and the key it references are drawn from the same domain, because a value in one is compared against a value in the other. Holding accession_no as eight characters in both relations is not tidiness; a comparison across two different domains would mean nothing.

What it is for: It records the relationship without copying the whole of a book into every loan, and it lets the two relations be joined again when a report needs both. It also gives the management system something it can enforce, which is the second half of its purpose.

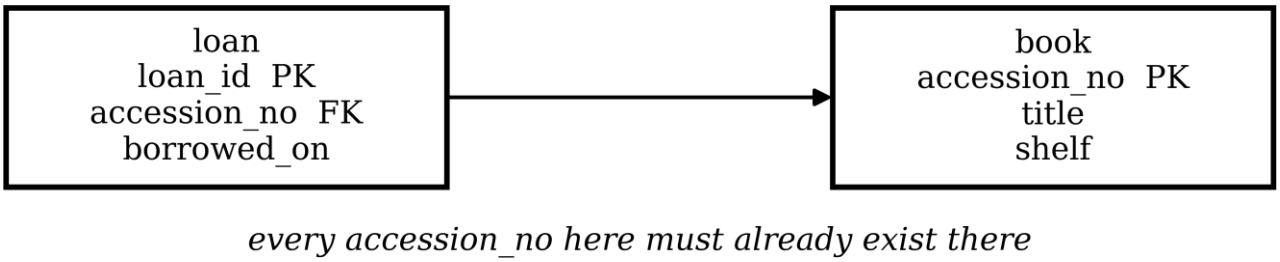


Figure 4.6: *A foreign key in the loan relation pointing at the primary key of the book relation.*

A foreign key may be null: Where the relationship is optional the foreign key may be left null, and that is not a breach of anything. What is forbidden is a value that is neither null nor present in the referenced relation, such as a loan standing against an accession number the library has never held.

A foreign key is a value borrowed from another relation, and its whole worth lies in the promise that the value is really there.

The Master Key: *A foreign key holds a value that must either be null or match an existing primary key value in the relation it references.*

BINDER Question 22

Nothing looked wrong with the register at Lembeni until a return was attempted. One line carried no accession number at all, and nobody could say which copy it described or whether that copy had ever come back.

State the three integrity rules, and explain what each one prevents.

Model Answer

The three integrity rules are entity integrity, which forbids any part of a primary key from being null, referential integrity, which requires every foreign key value to be null or to match an existing primary key value, and domain integrity, which requires every value to come from the declared domain of its attribute.

Entity integrity: No attribute of a primary key may be null, and in a key of several attributes not one of them may be. Students are often taught that the rule exists because the other attributes depend on the key. The reason is plainer than that: a null identifies nothing, so a tuple with a null there cannot be told apart from any other.

Referential integrity: A foreign key value must either be null or match a primary key value that exists now. This is what prevents the orphan row, a loan standing against a copy the library does not hold, which can be neither returned nor traced and which no report can account for.

Domain integrity: Every value must come from the declared domain of its attribute, so a date cannot appear where a shelf code belongs and a negative number cannot appear where a count belongs. This is the rule that the declared type of a column, and the constraints written beside it, exist to express.

Rule	What it prevents
Entity integrity	A tuple that no key value can identify
Referential integrity	A loan pointing at a copy the library does not hold
Domain integrity	A value the attribute was never meant to carry

Table 4.2: *The three rules in their settled order, and the failure each one exists to prevent.*

Where the three rules live: All three are declared once, in the schema, and are then applied to every write from every program and every person. A rule enforced inside one application is a rule the next application has never heard of.

Name them in that order, entity, referential and domain, and attach each one to the construct it governs: the primary key, the foreign key, and the attribute.

The Master Key: *Entity integrity protects the identifier, referential integrity protects the reference, and domain integrity protects the value.*

REAL Question 23

Somebody has pinned a note inside the returns box at Kifaru with three sentences about keys, copied from a friend's exercise book. Two of the three are wrong, and the one that is right is the one that sounds least like a definition.

Identify which of the three statements is true, say exactly what is wrong with each of the other two, and correct them.

A. A composite key is formed by combining two or more candidate keys.

B. A primary key may be a tuple, a field, or a set of tuples or fields.

C. Every candidate key is a superkey, but not every superkey is a candidate key.

Model Answer

Statement C is the true one; A is wrong because a composite key combines attributes and never keys, and B is wrong because a key is a set of attributes and never a set of tuples.

Why C is true: A candidate key is defined as a minimal superkey, so it is a superkey by construction. The reverse fails: accession_no with title added to it still identifies a copy and is therefore a superkey, but title can be dropped without any loss of uniqueness, so the set is not minimal and is not a candidate key.

What is wrong with A: A composite key is built from attributes, not from keys. The union of two candidate keys is a superkey, but it can never be a candidate key, because either half of it already identifies the tuple on its own and the union is therefore not minimal.

A corrected: A composite key is a key consisting of two or more attributes, where no smaller part of it identifies a tuple by itself. A relation holding one fine per loan per day is keyed by loan_id and charged_on together, since neither of those attributes is unique alone.

What is wrong with B: A tuple is a row. Calling a primary key a tuple, or a set of tuples, says that rows are identified by other rows, which is circular and leaves no way to write the key down at all. The looser word field passes for attribute in conversation; tuple for row does not pass inside a definition of a key.

B corrected: A primary key is one attribute, or a set of attributes, chosen from among the candidate keys to identify each tuple of the relation uniquely. Put that way, the definition names what a key is made of and what it does, and both halves can be checked against the relation in front of you.

The two false statements are both wrong about what a key is made of, and the true one is the only sentence of the three that says how two kinds of key stand to each other.

The Master Key: *A key is a minimal set of attributes, a composite key combines attributes rather than keys, and every candidate key is a superkey.*

REAL Question 24

The librarian at Endasak is asked which column will still identify a copy long after everyone now in the office has left. Two of the five look equally good today, and only one of them survives the question.

Identify every superkey and every candidate key of the relation below, then choose a primary key and justify the choice.

book(accession_no, isbn, title, author, shelf)

Each physical copy is stamped with its own accession_no on the day it reaches the library.

An ISBN names an edition, so two copies of one edition carry the same ISBN.

Model Answer

Only accession_no is a candidate key, so the superkeys are the sixteen sets of attributes that contain it, and accession_no is the primary key; isbn is a second candidate key, and the superkeys rise to twenty four, only if the library never holds two copies of one title.

Which attributes identify a copy: accession_no is stamped into one physical copy and issued once, so no two tuples can share it. The others repeat freely: two copies stand on the same shelf, one author writes several titles, and a title returns in a new edition with a new isbn beside it.

Every superkey: Any set containing accession_no is a superkey, since adding attributes to something that already identifies cannot stop it identifying. Four other attributes remain, so there are sixteen such sets, from accession_no alone up to the whole relation. No set without accession_no is a superkey while two copies of one edition are possible.

Every candidate key: Of those sixteen, only accession_no by itself is minimal, so it is the sole candidate key. Every other superkey carries at least one attribute that could be removed without loss, and a set that can be trimmed is not a candidate key at all.

What the ISBN assumption changes: isbn is a candidate key only if the library never holds two copies of one title, because two copies of one edition carry the same isbn and would destroy its uniqueness. Under that assumption there are two candidate keys, and the superkeys become the twenty four sets containing accession_no or isbn or both.

The primary key, and why: Take accession_no. The library issues it, every copy has one including donations printed before ISBNs existed, it never changes, and it identifies the copy rather than the edition. isbn is then the alternate key, and on the day a second copy of one title arrives, isbn would have failed as a primary key while accession_no is untouched.

A key that rests on an assumption the library cannot enforce is a key that fails on the day the assumption does.

The Master Key: *Superkeys are every set containing an identifying attribute, candidate keys are the minimal ones, and the primary key should be the one whose uniqueness the organisation itself guarantees.*

BUILD Question 25

Writing a loan against a book that does not exist is easy at Kifaru, since a page argues with nobody. A server can be made to argue instead, and it takes one clause in the declaration of the table.

Create the two tables of the lending schema so that the keys and the referential rule are declared rather than merely intended, then show the server refusing a loan against a book that does not exist.

book holds accession_no as its primary key, and a title that may not be null.

loan holds loan_id as its primary key, accession_no as a foreign key referencing book, and borrowed_on.

Attempt one loan against an accession number that book does not hold.

Model Answer

The two declarations below put the primary keys, the foreign key and the not null rule into the schema itself, so that the refusal which follows comes from the server rather than from anything the person typing was expected to remember.

```
 1  CREATE TABLE book (
 2    accession_no CHAR(8)     NOT NULL,
 3    title        VARCHAR(80) NOT NULL,
 4    shelf        CHAR(4),
 5    PRIMARY KEY (accession_no)
 6  ) ENGINE = InnoDB;
 7
 8  CREATE TABLE loan (
 9    loan_id      CHAR(6) NOT NULL,
10    accession_no CHAR(8) NOT NULL,
11    borrowed_on  DATE    NOT NULL,
12    PRIMARY KEY (loan_id),
13    FOREIGN KEY (accession_no) REFERENCES book (accession_no)
14  ) ENGINE = InnoDB;
15
16  INSERT INTO book (accession_no, title, shelf)
17  VALUES ('EN000431', 'Physics for Form Five', 'S12');
18
19  INSERT INTO loan (loan_id, accession_no, borrowed_on)
20  VALUES ('L00017', 'EN009999', '2026-04-14');
```

The server refuses it

```
ERROR 1452 (23000) at line 19: Cannot add or update a child row: a foreign key constraint
fails (`school_library`.`loan`, CONSTRAINT `loan_ibfk_1` FOREIGN KEY (`accession_no`)
REFERENCES `book` (`accession_no`))
```

Program 4.1: *Two tables whose keys are declared, and the loan the server will not take.*

What is declared, and where: Each table names its own primary key, so entity integrity is enforced on both of them. The foreign key clause inside loan names the attribute and the relation it references, and that is what turns referential integrity from an intention into a rule applied to every write.

Reading the refusal: The server names the schema, the child table and the constraint that stopped the write, and it rejects the whole row rather than storing part of it. Nothing was inserted, so loan still holds only rows whose accession number exists in book.

The one decision worth naming: accession_no is declared as fixed length characters in both tables rather than as an integer, because an accession number is a code and not a quantity, and an integer would strip a leading zero from it. The foreign key and the key it references must agree on that domain or they cannot be compared.

A rule that lives in the schema is enforced against every program that ever connects, including the ones nobody has written yet.

The Master Key: *Declare the keys and the reference in the table itself, and the server refuses the impossible loan without being asked to.*

HOT Question 26

Which number should a library build its records on: the one the publisher printed on the cover, or the one the school stamped inside it? Both of them identify something. Only one identifies a copy the school owns.

Judge whether a natural key such as an ISBN or a surrogate key such as an auto generated number makes the better primary key, and state the condition that decides it.

Model Answer

Prefer the natural key where its values are issued by an authority you can rely on, are guaranteed to be present and never change, and otherwise generate a surrogate and keep the natural value beside it as an alternate key.

The case for a natural key: A natural key already exists in the world, so nothing has to be invented, stored or explained to anybody. It is printed on the thing itself, which means a value can be checked against what it describes, and two systems holding the same natural value can be matched without a translation table between them.

The case for a surrogate: A generated number is short, uniform and under the database's own control. It is never absent, never reused and never revised by anybody outside, so no correction to a real world value can ripple out through the relations that reference it.

How a natural key fails: It fails when the outside world revises it. An identifier can be reissued, mistyped or lengthened, as ISBNs were, and a key value that changes has to be changed in every referencing relation at the same moment. It can also be missing, which entity integrity forbids outright.

How a surrogate fails: A surrogate makes every row unique, including two rows that describe the same copy, so the duplicate it hides is a duplicate it also permits. It carries no meaning, so a wrong reference cannot be caught by reading it, and two systems cannot be matched on it at all.

The condition that decides it: Ask who issues the value. Where your own organisation issues it, is obliged to issue one for every thing it records, and never revises it, the natural key is the better primary key. Where any of those three fails, take the surrogate and move on.

The judgement: The accession number wins here, because the library issues it, stamps it on arrival and never reissues it, while the ISBN fails all three of those tests at once. Whichever is chosen, the natural value should still be declared unique, since a surrogate allowed to hide a duplicate will eventually hide one.

The question is never which of the two values means more, but which value the organisation can promise to keep.

The Master Key: *Key on a natural value only where your own organisation issues it, always issues it and never changes it, and otherwise take a surrogate and keep the natural value as an alternate key.*

HOT Question 27

Two programs now write to the library database at Lembeni: the machine on the counter and a phone in the head's office. The counter checks that a book exists before it records a loan. The phone was written by somebody else.

Assess whether referential integrity should always be enforced by the database rather than by the application, and reach a judgement.

Model Answer

Referential integrity belongs in the database in every case where the referenced relation is in that database, and a check written into an application is worth having only in addition to the declared constraint, never in place of it.

What an application check cannot do: The program asks whether the book exists and then inserts the loan, and those are two separate steps. Between them another connection can delete the book, and the loan is written against nothing. The declared constraint is checked as part of the write itself, so no such gap exists.

One rule, or one rule for each program: The counter machine, the phone in the office, a bulk load at the start of the year and a person typing at the client all reach the same tables. A rule written inside one program governs the writes that pass through that program and no others at all.

The case for checking in the application: An application can say something a person understands, in the right language, before a form is ever submitted, and it can do so without a journey to the server. A raw server complaint naming a constraint is not a message to put in front of a borrower.

The real cost of the declared rule: A constraint costs a lookup on every insert and update, it makes rows harder to delete in the wrong order, and it makes a bulk load fail unless the parent rows go in first. Those costs are real, and every one of them is the rule doing its work.

Where the database genuinely cannot enforce it: The reference may point outside the database altogether, at a table held in another system, or at rows that have not arrived yet in a staging area. There the constraint cannot be declared, and the check has to live somewhere else until the data lands in one place.

The judgement: Declare it in the database always, and add the application check on top of it wherever a person needs a civil message. Where the parent really does live elsewhere, treat the missing constraint as a known risk with a stated repair, not as a design decision.

An application check is a convenience for the person typing, while the declared constraint is the only thing that makes the promise true.

The Master Key: *Enforce referential integrity in the database always, and in the application as well when the wording of the message matters, but never in the application instead.*

Conceptual, logical and physical design

A design does not arrive as a set of tables. It arrives as three answers taken in order: what the organisation records, what relations will hold those records, and how one named product will store them. This section separates the three tiers, says which decision belongs to each, and shows what goes wrong when a decision is taken at the wrong one.

BINDER Question 28

Nothing has been built yet at Hedaru. The desk opens next month, and nobody has agreed whether a ticket is one seat on one bus or a booking that covers a whole family. That question has to be settled first.

Explain what database design is, and describe the three tiers it passes through.

Model Answer

Database design is the work of deciding what a database will hold and how it will hold it, and it is carried out in three tiers: a conceptual model of the things the organisation records, a logical model of the relations that will hold them, and a physical model of how one named product will store those relations.

Design is a sequence of decisions: Each tier settles a different question and hands its answer down to the next. The conceptual tier settles what exists, the logical tier settles how it is structured, and the physical tier settles how it is stored. Nothing is decided twice.

The conceptual tier: The designer names the things the ticketing desk deals with, passenger, route, trip and ticket, and the relationships between them, and expresses that as an entity relationship diagram. No table, no data type and no product is named here.

The logical tier: Each entity becomes a relation with named attributes, every relation is given a primary key, and every relationship is carried by a foreign key. The model is now precise enough to argue about, and it still names no product.

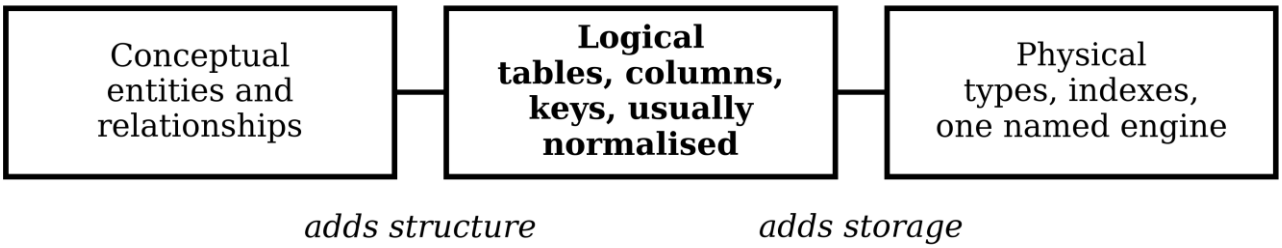


Figure 4.7: *What each design tier adds, and the decision that has been taken by the time it is finished.*

The physical tier: Now the product is named, every column is given a type and a length, indexes are chosen, and the storage engine and character set are settled. This is the first tier at which the design would actually run if somebody typed it out.

Not the three schema architecture: The three schema architecture describes a database already in use, its external, conceptual and internal levels all present at the same moment. The three design tiers are stages of work, each finished before the next begins. Only the word conceptual is shared.

The tiers differ by what has been decided, not by how much detail has been drawn.

The Master Key: *Conceptual settles what exists, logical settles how it is structured, and physical settles how it is stored.*

BINDER Question 29

Three drawings lie on the manager's table at Kisangara: one showing passengers and trips, one listing tables and columns, one giving the width of every column. Only one of them is a conceptual model.

Describe the conceptual model, and state what it must not contain.

Model Answer

The conceptual model is a statement of what the organisation records and how those things relate to one another, written as entities, their attributes and the relationships between them, and it contains no table, no data type, no length, no index and no product name.

What it contains: For the ticketing desk it names the passenger, the route, the trip and the ticket, gives each of them the attributes that belong to it, and states how they are joined: a route has many trips, a trip carries many tickets, and a passenger buys many tickets.

How it is expressed: It is normally drawn as an entity relationship diagram, because a drawing carries a relationship in one line where a sentence needs a paragraph. The diagram is the form; the model is the agreement about meaning that the diagram records.

What it must not contain: Nothing about storage belongs here. No column, no data type, no length, no index, no storage engine and no product name, because every one of those is an answer to a question this tier has not yet asked.

Why the exclusions are the whole point: The conceptual model is the one version of the design that the person doing the work can check. A clerk at Kisangara can say whether a ticket is ever valid for two trips; she cannot say whether a name should hold thirty characters. Fill the model with storage and you lose your only expert reader.

It outlives the product: Because it names nothing product specific, the same conceptual model survives a change of database. Replace the software in five years and this is the one document that does not have to be written again.

A conceptual model describes the world the desk works in, not the machine the records will live on.

The Master Key: *The conceptual model names entities, attributes and relationships, and names nothing that belongs to storage.*

BINDER Question 30

The sketch said passenger and trip. The first attempt at tables put the passenger's name inside every ticket row, so one traveller from Segera appears under more than one spelling of her own name. Something between the two drawings went wrong.

Describe the logical model, and explain what changes when a conceptual model becomes a logical one.

Model Answer

The logical model expresses the same facts as relations: every entity becomes a relation with named attributes, every relation is given a primary key, and every relationship is carried by a foreign key, without naming a database product.

Entities become relations: Passenger, route, trip and ticket each become a relation, and every attribute is placed in exactly one of them. An attribute drawn beside an entity now belongs to a named relation, which is what stops a passenger name being written again on every ticket row.

Relationships become foreign keys: A line on the diagram becomes a foreign key held by the relation at the many end, so trip carries the route it runs and ticket carries the trip it belongs to. A many to many relationship becomes a relation in its own right.

Keys are chosen, not discovered: One candidate key is selected as the primary key of each relation. Where nothing in the world identifies a row uniquely and permanently, an identifier is introduced for the purpose, and that choice belongs to this tier.

Normalisation is decided here: A logical model is normally normalised, and normalisation belongs to this tier. It is not what makes a model logical: a model is logical because it is a set of relations with declared keys, in whatever normal form it has been left. Stepping deliberately back from that, keeping a repeated column so a common report reads from one relation instead of three, is denormalisation, and it trades storage and the risk of disagreement for speed of reading.

What has still not changed: No type, no length and no index has been chosen. The same logical model would stand unaltered if the desk bought a different database tomorrow.

The conceptual model says what the desk records; the logical model says in what relations those records will be held.

The Master Key: *A logical model turns entities into relations and relationships into foreign keys, and normalisation is a decision taken there rather than the definition of it.*

BINDER Question 31

Ask the desk manager at Segera why the fare column is DECIMAL and she will say it was always like that. Somebody chose it once, deliberately, and the reason for it appears nowhere in the drawing.

Describe the physical model, and explain what a designer decides at this tier and nowhere else.

Model Answer

The physical model is the logical model expressed for one named database product, and it is the only tier at which a data type, a column length, an index, a storage engine, a character set and the placement of files on disk are decided.

Types and lengths: Every attribute is given a type the product understands and a length it will enforce. The fare becomes DECIMAL(10,2) because money must be exact, the postal code becomes CHAR(5) because it is a code and not a quantity, and the departure becomes TIME or DATETIME according to what the row records.

Access structures: Indexes are chosen here, on the columns the desk actually searches by. An index is never free: it makes reads faster, makes every write slower and takes room on disk, so each one is chosen against a query somebody has named.

Storage decisions: The storage engine, the character set and where the files live are settled here too, and settled for one product. Move the same logical model to another product and every one of these decisions is taken again from the beginning.

Nothing above it depends on it: Because these decisions are taken last, they can be changed last. Adding an index or moving the files alters nothing in the logical model, which is why a design reached in tiers can be tuned without being redesigned.

Why this is design and not typing: A type is a decision with consequences no diagram can show. An integer postal code silently loses a leading zero, and a departure typed as a date loses the time of day, which is the only part of a departure a passenger needs.

Everything a product argues about is settled at this tier, and nothing above it needs to know the answer.

The Master Key: *The physical model names the product and settles types, lengths, indexes, engine and storage, and no other tier may settle them.*

REAL Question 32

A clerk at Hedaru writes four words on the back of a ticket stub: passenger, route, departure, fare. That is the whole of the design so far, and every table the desk will ever need has to come out of it.

Take the ticketing desk from that conceptual sketch through to a logical model, and name the decision taken at each step.

Model Answer

The sketch holds four words and the finished logical model holds four relations, but they are not the same four: passenger, route and ticket become relations directly, departure becomes a relation of its own called trip, and fare becomes an attribute of the ticket.

Step one, decide what is an entity: The decision is whether a departure is an attribute of a ticket or a thing in its own right. Many tickets are sold for the same departure from Hedaru, and the desk must record that departure once, so it becomes an entity: a trip, meaning one running of one route at one stated time.

Step two, decide where the fare belongs: Fare could be an attribute of the route, since the route is what is being sold. It becomes an attribute of the ticket instead, because the amount actually charged can differ from the posted amount, and the desk has to be able to say what this passenger paid.

Step three, read the relationships: A route has many trips, a trip sells many tickets, and a passenger buys many tickets. Each of the three is one to many, so none of them needs a relation of its own, and the conceptual sketch is now complete.

Step four, choose a primary key for each relation: A ticket already carries a printed number, so ticket_no is chosen. A route is identified by route_code. Nothing about a passenger is both unique and permanent, so passenger_id is introduced for the purpose, and trip_id likewise.

Step five, place the foreign keys: Each relationship is carried by a foreign key held at the many end: trip carries route_code, and ticket carries trip_id and passenger_id. The decision here is which end holds the key, and the many end always does.

Relation	Attributes	Primary key	Foreign keys
passenger	passenger_id, name, phone, postal_code	passenger_id	none
route	route_code, origin, destination	route_code	none
trip	trip_id, route_code, departure	trip_id	route_code
ticket	ticket_no, trip_id, passenger_id, seat, fare	ticket_no	trip_id, passenger_id

Table 4.3: *The logical model the four words became, with the key chosen for each relation.*

What has deliberately not been decided: No type has been chosen. Whether fare is DECIMAL(10,2), whether departure is TIME or DATETIME, and whether anything is indexed all remain open, and they stay open until a product is named.

Each step took one decision, and every one of them was about meaning rather than about storage.

The Master Key: *Turning a sketch into a logical model means deciding what is an entity, where each attribute belongs, what identifies each relation, and which end carries the foreign key.*

REAL Question 33

Somebody at Segera has written three lines about design on the inside cover of the ticket book, where the next clerk will read them. Two of the three are wrong, and both wrong ones sound more confident than the right one.

Say which of the three statements is true, name exactly what is wrong with each of the other two, and correct each.

A. A logical model is one that has been normalised to third normal form.

B. A conceptual model names entities and relationships but chooses no data types.

C. A physical model is simply the logical model drawn more neatly.

Model Answer

Statement B is the true one: A mistakes a decision usually taken at the logical tier for the definition of that tier, and C mistakes a difference of content for a difference of neatness.

Why B is true: A conceptual model names the entities the desk deals with and the relationships between them, and it stops there. Data types belong to the physical tier, two steps later, and a conceptual model that has chosen one has already left its own tier.

What is wrong with A: Normalisation is a decision taken at the logical tier, not the property that makes a model logical. A model is logical as soon as it is a set of relations with named attributes and declared keys. A designer may leave it in second normal form, carry it past third to Boyce-Codd, or step back from third on purpose by denormalising, and in every one of those cases what she holds is still a logical model.

A, corrected: A logical model is a set of relations with declared primary and foreign keys, expressed without reference to any product, and it is usually, though not necessarily, normalised to third normal form.

What is wrong with C: The physical model carries information the logical model never had. It names a product, and gives every column a type, a length, an index where one is wanted, a storage engine and a character set. Two physical models built from one logical model, for two different products, can differ in almost every line and both be right.

C, corrected: A physical model is the logical model expressed for one named product, with a type, a length and an access structure decided for every column.

The question that separates the three: Ask what each model would refuse. The conceptual model refuses nothing about storage, because it says nothing about storage; the logical model refuses a relation with no key; the physical model refuses a value its declared type cannot hold.

A is the dangerous line of the three, because it is nearly true and is repeated by people who have built working databases.

The Master Key: *A logical model is defined by its relations and keys rather than by a normal form, and a physical model adds what the logical model never carried.*

REAL Question 34

A receipt printed at Kisangara shows a postal code with its first digit missing and a departure with no time on it at all. Both values went into the database exactly as the schema allowed. The schema is what is wrong.

The desk types the passenger's postal code as an integer and the departure as a date. Explain what goes wrong in each case, and give the correct type with the reason for it.

Model Answer

An integer postal code destroys any code beginning with a zero and refuses outright any code holding a letter or a space, and a date departure throws away the time of day; the code should be CHAR(5), and the departure should be TIME for a repeating timetable slot or DATETIME for one dated trip.

What the integer does to a leading zero: A leading zero is a character, not a quantity. Stored as an integer, the code 01034 becomes the number 1034, and nothing remains in the column to say a zero was ever there. Printed back on a receipt it is a different code, and posted to, it is a different place.

What it refuses altogether: Any code holding a letter, a space or a hyphen cannot go into an integer column at all. The column either rejects the value outright or converts it into something else, and neither of those is an outcome the clerk asked for.

The rule underneath both: A string of digits is a number only if you would ever do arithmetic on it. Nobody adds two postal codes, subtracts one phone number from another or takes an average of ticket numbers, so all three of them are text.

The correct type for the code: CHAR(5) is right where every code is exactly five characters, because a fixed width type states that rule and then enforces it. Where lengths genuinely vary, VARCHAR is right instead, and under either one arithmetic is impossible, which is the whole intention.

What the date does to the departure: A date holds a calendar day and no time of day. The morning and the afternoon departures from Kisangara on the same day become the same stored value, so the desk cannot tell one from the other, cannot count seats on either, and cannot print a departure on a ticket.

Choosing between TIME and DATETIME: This is not a matter of preference. If the row is a timetable slot that repeats every day, its departure is a time of day and the type is TIME. If the row is one dated running of a route, its departure is a day and a time together and the type is DATETIME. Decide what the row records, and the type follows from it.

Column	Typed as	What the type does to the value	Correct type
postal_code	INT	A leading zero disappears, and a letter or a space cannot be stored at all	CHAR(5)
departure, a repeating timetable slot	DATE	The time of day is lost, so every slot on one day looks alike	TIME
departure, one dated trip	DATE	The time of day is lost, so two trips on one day cannot be told apart	DATETIME

Table 4.4: *Two columns typed wrongly, what the type does to the value, and what the type should have been.*

Both faults are physical decisions, and both were taken as though they were matters of taste.

The Master Key: *Type a code as characters and a departure as what the row actually records, TIME for a repeating slot and DATETIME for a dated trip.*

HOT Question 35

The desk at Hedaru will have four tables and one user. A designer wants an afternoon to draw entities and relationships before writing anything. The owner asks what the drawing will tell him that the tables would not.

Assess whether the conceptual tier is worth the time it costs on a small system, and reach a judgement.

Model Answer

The conceptual tier earns its cost on a small system whenever anybody other than the designer has to agree on what the data means, and it earns very little when one person understands the work, builds the database alone and will maintain it alone.

The case for skipping it: A ticketing desk of four relations fits in one head. Drawing entities and relationships takes an afternoon that could have been spent building, the drawing is out of date the first time a column is added, and nobody at the desk will open it again.

The case for keeping it: The conceptual model is the only version of the design the person doing the work can read. The owner at Hedaru cannot check a foreign key, but he can say that a ticket is never valid for two trips, or that a group booking is one payment for many seats, and either sentence changes the schema.

A wrong entity does not cost less on a small system: A small design with the wrong entities is as wrong as a large one, and it fails at the same moment, when somebody asks a question the tables cannot answer. On paper the correction costs an eraser. After a month of selling tickets it costs a migration and a night of closure.

What it actually costs: The conceptual model of this desk is four entities and three relationships. The objection is almost never to the time, which is short, but to the formality, and formality is not what is being bought. What is being bought is one sentence everybody agrees to before anybody types.

The condition that decides it: Ask whether anyone besides the designer must agree on what the data means, now or later. A second user, a second developer, a handover, or an owner who will one day ask for a report all count as yes, and a single yes settles the matter.

The judgement: Draw it, and keep it small: entities, relationships and nothing else, on one side of one sheet, finished before any table is created. Skip it only for a single relation with no relationships, built and used by one person for one task that ends, because there is then nothing to relate and nobody to agree with.

The drawing is cheap, and what you are paying for is the agreement it records.

The Master Key: *Draw the conceptual model whenever a second person must agree on what the data means, and keep it to entities and relationships.*

HOT Question 36

The same design has to run on the school laboratory server, on a shop bought package at Kisangara, and on a phone application the owner is promised next year. One drawing, three products, and nobody has said which drawing travels.

Judge whether one logical model should be used unchanged across different database products or tailored to each, and state the condition that decides it.

Model Answer

Keep one logical model and let the physical model differ from product to product, because every difference between relational products is a physical difference; tailor the logical model itself only where the target cannot represent a relation with a declared key at all.

The case for one model: The logical model states the facts, the keys and the rules that join them, and none of those change because a different product was bought. One model means one place to change a rule, one document to hand over, and one description everybody argues from.

The case for tailoring: Products genuinely differ. Types are spelled differently and sized differently, automatic identifiers are declared differently, one product enforces a declared check while another accepts it and ignores it, and a storage engine chosen carelessly can drop referential integrity without saying so. A design blind to all this is found to be wrong at installation.

Where those differences actually live: Read that list again: every item on it is a type, a length, a spelling, a constraint mechanism or an engine. All of them are physical decisions, and holding them at the physical tier is exactly what the physical tier exists for. The logical model was never the thing that had to change.

The real cost of getting it wrong: Write one model down to whatever the weakest product supports and you pay for features on every other product and never use them. Write a separate logical model per product and you lose the one thing a logical model gives you, which is certainty that the two schemas mean the same.

The condition that decides it: The logical model changes only where the target cannot express what it says. A relational product always can. A store that holds no relations and enforces no keys cannot, and a design for that is a different logical model rather than a translation of this one.

The judgement: One logical model, and as many physical models as there are products. Hold the logical model in a form no product owns, relation names, attributes, primary keys and foreign keys, and derive each physical model from it rather than editing a copy of the last one.

The relations travel between products unchanged; the types never do.

The Master Key: *Use one logical model with one physical model per product, and change the logical model only where a product cannot represent a relation with a declared key.*

Entity-relationship modelling: entities, attributes, relationships

An entity relationship diagram is the drawing a design is argued over in, and it holds three kinds of thing: entity types, the attributes they carry, and the relationships between them. This section defines each of the three, sets out the five kinds of attribute and the three degrees a relationship may have, and reads Crow's Foot notation one end at a time. Every example is drawn from a dispensary stock book. What a pair of ends says about a relationship as a whole, and what becomes of a diagram once it is turned into tables, are taken up in the two sections after this one.

BINDER Question 37

The shelf label at Maramba reads BAY 3 and nothing more. In the stock book that word appears in a column beside every batch; on a drawing pinned at Mkomazi it has a box of its own, and neither version can be shown wrong.

Explain what an entity is, distinguish an entity type from an entity instance, and state the test that decides whether something deserves to be an entity at all.

Model Answer

An entity is a thing the dispensary holds facts about and can tell apart from others of its kind; an entity type is the named class of such things, such as batch, and an entity instance is one member of it, such as the batch of amoxicillin received on Tuesday.

What makes something an entity: An entity has existence of its own and facts worth recording. It may be something that can be handled, such as a batch of tablets, or something that cannot, such as a delivery, and both are entities in the same sense.

The type is what is drawn: A diagram shows entity types, never instances. One box labelled batch stands for every batch the dispensary has received and every batch it ever will. Drawing a box for each batch on the shelf makes a picture of one morning.

The instance arrives afterwards: An instance is one member of the type, told apart from the rest by the value of its key attribute. Instances keep arriving and leaving while the entity types stay as drawn, and a new batch changes nothing in the design.

The test that decides: Ask three things of a noun: whether facts are recorded about it beyond its own value, whether there can be many of it, and whether it can be identified without leaning on anything else. Three answers of yes make an entity type.

The argument at Maramba settled: Shelf fails the first question, since nothing is recorded about a shelf beyond its name, so it stays an attribute. Supplier passes all three, having a phone number, many batches from one supplier, and a name of its own.

The same word can be an entity type in one dispensary and an attribute in the next, and only the recording tells you which.

The Master Key: *An entity type is a class the design holds facts about, an instance is one member of it, and a noun with no facts of its own is an attribute.*

BINDER Question 38

A batch card at Mkomazi carries its number, its supplier, the days left before it expires and the shelves it stands on. Only one of those four is a plain value stored exactly as it is written there.

Describe the kinds of attribute an entity type may carry, and explain what each kind is for.

Model Answer

An attribute is a named property of an entity type, and the kinds differ in what one value may be: a simple attribute holds one indivisible value, a composite attribute is made of parts that each mean something alone, a derived attribute is computed rather than stored, a multivalued attribute may hold several values for one instance, and a key attribute identifies the instance.

Simple attributes: A simple attribute holds one value the design has no reason to break up. The batch number is one code and the quantity received is one number, and nothing inside either of them is ever wanted on its own.

Composite attributes: A composite attribute is one property made of parts, each meaningful by itself. A supplier held as name and phone is composite, and saying so is what allows the phone to be reached without pulling a string of text apart.

Derived attributes: A derived attribute is not stored anywhere; it is worked out from other stored values at the moment it is asked for. Days to expiry is the expiry date less today's date, and storing it would make it wrong by morning.

Multivalued attributes: A multivalued attribute may hold more than one value for a single instance at the same time. One batch kept on two shelves has two shelf values at once, which is not two batches and not one value that changed.

The key attribute: The key attribute is the one whose value identifies an instance, shown by underlining it or setting it in bold. The key family was settled earlier in this chapter; what a diagram adds is that every entity type must show one.

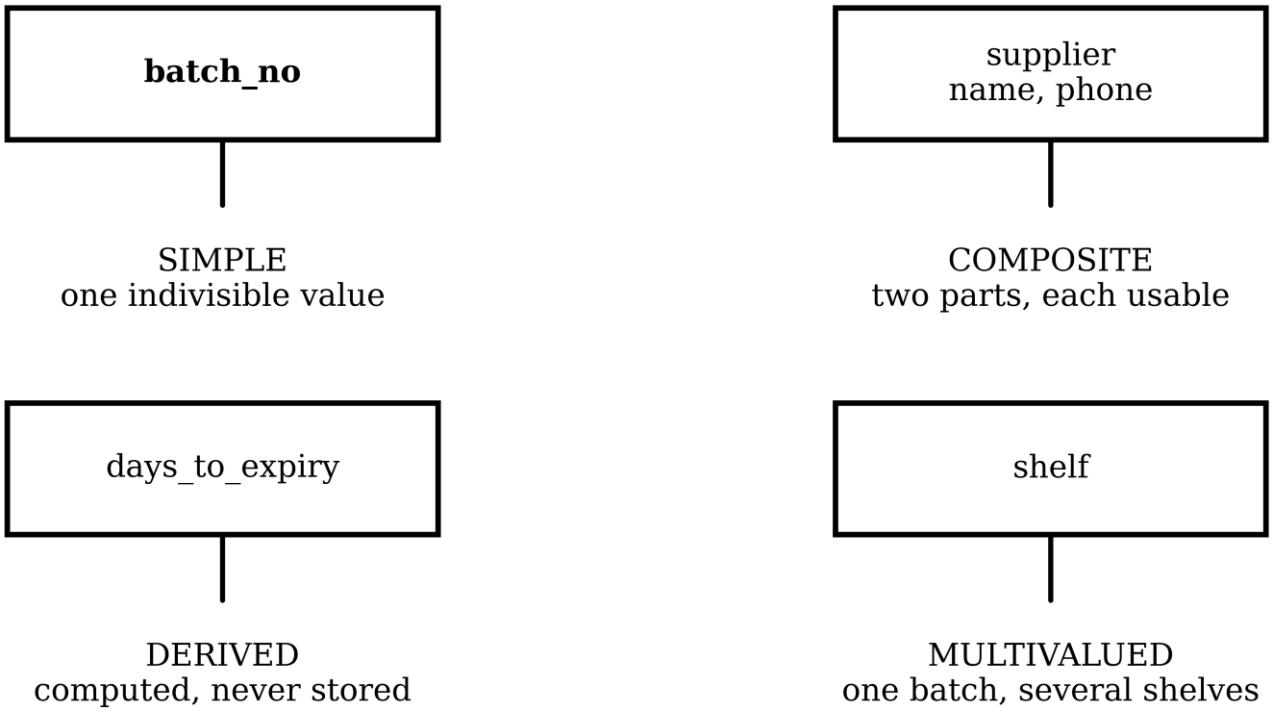


Figure 4.8: *Four attributes of one stock batch, each showing a different kind.*

Name the kind before you write the attribute down, because the kind decides whether a value is stored once, split, computed or repeated.

The Master Key: *Simple, composite, derived, multivalued and key are the five kinds, and the kind states what one value of that attribute is allowed to be.*

BINDER Question 39

Supplies, is that a verb or a box? Somebody asked it out loud at Hale while supplier, batch and patient were being drawn. One line on the sheet touches two boxes, another touches only one, and neither is a mistake.

Explain what a relationship is in an entity relationship model, and describe the degree of a relationship, giving an example of each degree.

Model Answer

A relationship is a named association between instances of entity types that the design must record, and its degree is the number of entity types taking part: one in a unary, two in a binary, three in a ternary.

A relationship is named, then read: The line carries a verb, and the design is checked by reading it back as a sentence: a supplier delivers a batch. A line nobody can turn into a sentence is one nobody has understood, and it is usually two relationships drawn as one.

Unary, also called recursive: One entity type takes part, and the association runs between two instances of it. A batch replaces a batch withdrawn after a recall, so the line leaves the batch box and returns to it, with a role name at each end saying which batch is which.

Binary: Two entity types take part, and almost every relationship in a stock book is of this degree. A supplier delivers a batch, and a batch holds one drug. Binary relationships are the easiest to read and to check, so a design gains wherever it can honestly use them.

Ternary: Three entity types take part in one association at once. A dispenser issues a batch to a patient is ternary, because the fact recorded is about the three of them together rather than about any pair taken on its own.

Degree counts entity types: It counts entity types, not instances, and not the attributes of a relation, which is a separate count sharing the word. Count the boxes the line touches, and a line returning to its own box has touched one.

Name the verb and count the boxes together, because a line whose sentence will not come out usually has its degree wrong.

The Master Key: *A relationship is a named association between entity instances, and its degree is the number of entity types in it: one, two and three for unary, binary and ternary.*

BINDER Question 40

A short bar, a small circle and three prongs like a bird's foot: that is the whole vocabulary at the end of a line on the drawing pinned up at Maramba. Read them in the wrong order and the diagram says the opposite.

Describe Crow's Foot notation, and explain how one end of a relationship line is read.

Model Answer

Crow's Foot notation draws every entity type as a named box and every relationship as a line between boxes, and each end of that line carries two marks: the one touching the entity gives its maximum and the one behind it gives its minimum.

What is on the drawing: A box for each entity type, named with a singular noun and holding its attributes, and a plain line for each relationship with a verb on it. Nothing else appears, which is why the drawing can be read by somebody who has never written SQL.

The three symbols: A short bar across the line means one. Three prongs opening against the box, the crow's foot that names the notation, mean many. A small circle means zero, and every end is built from those three.

Reading an end, in order: Take the mark touching the box first, because it is the maximum, then the one behind it, which is the minimum. A bar behind a foot reads as one or many, a circle behind a bar as zero or one, and those four are the whole notation.

An end describes the entity it touches: The pair of marks states how many instances of the entity they touch may take part, and says nothing at all about the entity at the far end of the line. Most misreadings begin here, and the cure is to put a finger on the box first.

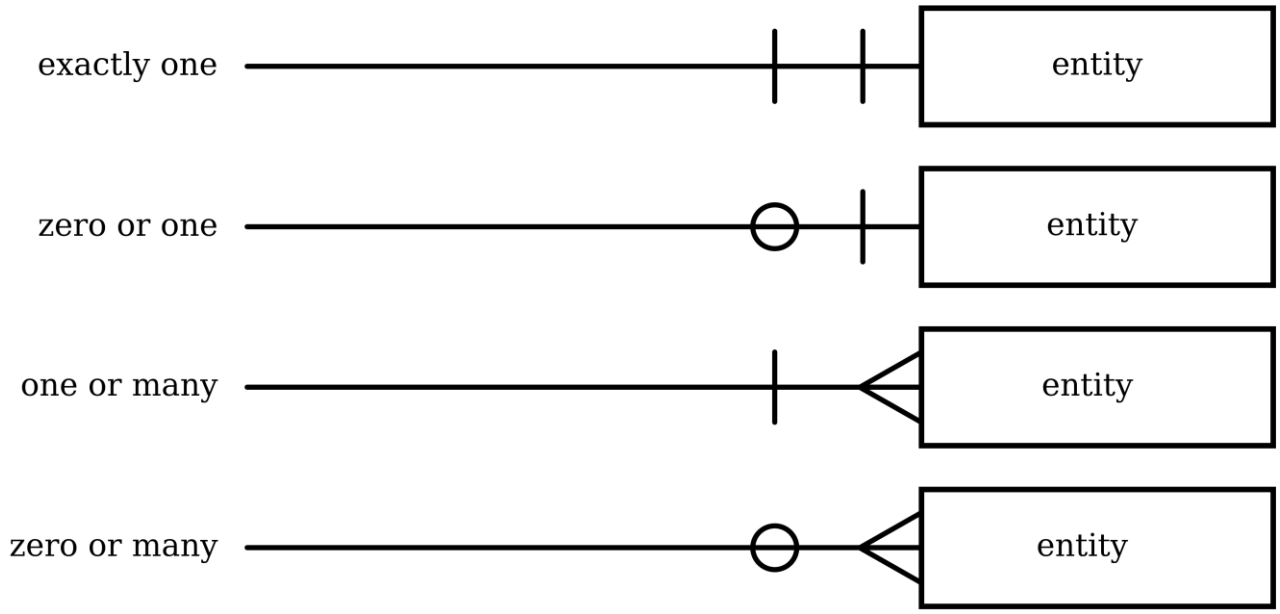


Figure 4.9: *The four end marks, and the rule that the mark touching the entity gives the maximum.*

One end at a time: Each end is read alone. What the two ends of one line say when taken together, and what a minimum of zero means for an instance standing unattached, belong to the next section and are not settled by the marks themselves.

Touching first, behind it second, and about the box it touches: three habits, and the notation stops being decoration.

The Master Key: *Every end carries two marks: the one touching the entity gives its maximum and the one behind gives its minimum, and both speak only of the entity they touch.*

REAL Question 41

Nine nouns come off a single line of the Mkomazi stock book, from batch number to storekeeper. Some are things the dispensary keeps facts about; the rest are only values, and the line itself gives no hint which is which.

The stock book at Mkomazi records, for every delivery, a batch number, the drug name, its strength, the supplier's name, the supplier's phone, the quantity received, the expiry date, the shelf or shelves the batch stands on, and the storekeeper who checked it in. Identify the entity types, say which of the remaining nouns are attributes rather than entities, and justify every decision.

Model Answer

Four entity types come out of that line, batch, drug, supplier and storekeeper, and the remaining five nouns are attributes, because each of them is a single value describing something else and the dispensary records nothing about any of them in its own right.

Batch, the thing the book is about: Every delivery creates a batch, and the batch carries facts that belong to nothing else: how much came, when it expires and where it went. Its batch number is the attribute that identifies it, not a second thing standing beside it.

Drug, separate from batch: The drug name and its strength describe the medicine rather than the delivery, and they repeat unchanged in every batch of it. Amoxicillin at one strength is one drug whether it arrives once or many times, so it is an entity type and each batch refers to it.

Supplier, a thing and not a name: A supplier has a name and a phone number, which already makes it composite, and many batches come from the same supplier. A changed phone number would otherwise have to be corrected on every line that named it, and facts of its own with many uses is exactly the entity test.

Storekeeper, on the same test: The person who checked the delivery in has facts of her own, will check in many more deliveries, and is known independently of any batch. She is an entity type, and the stock book line holds a reference to her rather than a description of her.

The nouns that stay attributes: Batch number, strength, quantity received and expiry date each hold one value about one other thing, and nothing is recorded about a quantity or a date on its own. A noun that only ever appears as a value is an attribute, however important the value is.

Shelf, the borderline one: As the book stands, a shelf is a label and nothing else, so it is a multivalued attribute of batch. Record which room a shelf is in, or the temperature it is kept at, and it becomes an entity type from that moment.

Every decision here was made by asking what the dispensary records about the noun, and not one of them by looking at the noun.

The Master Key: *A noun earns a box when facts of its own are recorded about it and there can be many of it; a noun that only ever appears as a value is an attribute.*

REAL Question 42

Three sentences about attributes reach a phone at Hale, forwarded from somebody's notes and passed on twice since. Two of the three are wrong in a way that sounds reasonable, which is very likely why they were worth forwarding at all.

Identify the statement that is true, say precisely what is wrong with each of the other two, and correct them.

A. A derived attribute is stored in the database and updated whenever its inputs change.

B. A multivalued attribute is one that may hold more than one value for a single entity instance.

C. A composite attribute is an attribute belonging to more than one entity.

Model Answer

Statement B is the true one, and the other two are wrong: A has a derived attribute stored, which is the one thing a derived attribute is not, and C uses composite for a relation between entities when the word describes the inside of a single attribute.

Why B is right: A multivalued attribute holds more than one value for one instance at the same time. A batch at Hale standing on two shelves has two values of shelf, and neither replaces the other. It is not the same batch entered twice, and it is not a value that changed last week.

What is wrong with A: Being computed rather than stored is the entire content of the word derived. Days to expiry is worked out from the expiry date and today's date when somebody asks for it, and it is held nowhere. Store it and it stops being derived, because it can then be right on the day it was written and wrong the next morning.

A, corrected: A derived attribute is not stored; it is computed from other attributes when it is asked for. That is more than a repair of the wording, since the reason for naming the kind at all is to mark the values the database is not obliged to keep in step with anything.

What is wrong with C: Composite says what one attribute is made of, not who owns it. A composite attribute breaks into parts that each mean something alone, such as a supplier held as name and phone. Every attribute belongs to exactly one entity type, and the same fact hanging under two of them is a design fault rather than a kind of attribute.

Why the two wrong ones fail together: Each takes a word about the inside of one attribute and points it outward: A at how the database keeps the value, C at which entity owns it. Ask what one value of the attribute looks like before reaching for the name of the kind, and both sentences correct themselves.

Correct A to computed and never stored, correct C to made of parts that each mean something, and B needs nothing done to it.

The Master Key: *A derived attribute is computed when asked for and never stored, and a composite attribute is one attribute made of parts that are each meaningful alone.*

REAL Question 43

A recall notice arrives at Maramba and one batch of amoxicillin comes off the shelf. The batch sent in its place is also a batch, which leaves the stock book needing to join one thing to another thing of exactly its own kind.

Model the fact that one batch replaces another batch withdrawn after a recall, and explain why a relationship from an entity type to itself is not a mistake.

Model Answer

It is modelled as a unary relationship on the batch entity type: one box, one line named replaces that leaves the box and comes back to it, and a role name at each end, and it is not a mistake because degree counts entity types and one entity type may associate with itself.

The fact to be recorded: A recall does not create a new kind of thing. The withdrawn batch and the one sent in its place are both batches, with the same attributes and the same kind of identifier, and what has to be stored is which of them stands in for which.

How it is drawn: One box labelled batch, and a line that leaves the box, turns and returns to the same box, with the verb replaces written on it. Each end of that line carries its own pair of marks and is read exactly as the end of any other relationship is read.

The role names are not decoration: With both ends on one box, the verb alone cannot say which end is which. So each end is labelled with the part it plays, replacement at one and replaced at the other. Without the two role names the line records that two batches are connected and loses which way round the connection runs.

Why it is not a mistake: A unary relationship associates two instances of one type; it does not join a thing to itself. No batch replaces itself, and nothing in the notation suggests one could. The box is a type, and the two ends of the line reach two different members of that type.

What avoiding it would cost: The usual escape is to invent a second entity type called replacement batch, which copies every attribute of batch and enters one physical batch under two names. A batch that replaced one batch and was later replaced itself must then be entered twice, and the two entries drift apart at the first correction.

The shape appears everywhere: A storekeeper supervises a storekeeper, a drug substitutes for a drug, and a delivery corrects an earlier delivery. Once the shape is recognised, the question is never whether the line may return to its own box but only what the two roles should be called.

One box and one line hold the fact; two boxes for one kind of thing would hold it twice and keep neither of them right.

The Master Key: *A relationship from an entity type to itself is unary, and the role names at its two ends are what say which instance plays which part.*

HOT Question 44

The storekeeper at Mkomazi knows what the four tables will be and wants to start typing them. A visiting designer asks for an afternoon and a blank sheet first, and cannot yet say what the drawing will show that the tables will not.

Assess whether an entity relationship diagram is worth drawing at all when the designer already knows what tables are wanted, and reach a judgement.

Model Answer

It is worth drawing wherever the design holds more than one entity type, because that is where relationships exist and a list of tables cannot state them; with one entity type, no relationship and one person who will ever touch the design, the drawing is honest waste.

The case for typing the tables: The tables are the thing that runs. A drawing costs an afternoon, and it stops being true the first time a column is added without anybody redrawing it. A diagram that disagrees with the database is worse than none, because somebody will believe it.

What a list of tables cannot say: A list of tables says which columns exist. It does not say which of them the designer holds to be a thing, which associations are claimed, or how many entity types each joins. Those are decisions, and a column list shows none of them.

The person who has to agree: The storekeeper at Mkomazi knows what a batch is and what a delivery is, and cannot read a CREATE TABLE statement. She can read boxes with nouns and lines with verbs, and she is the only person who can say the design has left something out.

Where the errors surface: Drawing forces every noun to be classified and every line to be named. A noun that fits in no box is usually two things written as one, and a line nobody can put a verb on is an association nobody has agreed. Both are cheaper to find on paper.

Knowing the tables is not knowing the design: A designer who already knows the tables usually knows them from the reports wanted rather than from the things recorded, and a design taken from today's reports answers today's questions and no others. The drawing asks what exists instead.

The judgement: Draw it, and the condition is the count of entity types: one entity type with no relationship and no second reader means the drawing tells nobody anything, while two are already enough to put a claim on the page that a column list cannot make.

What keeps it worth having: A diagram earns its keep only while it is true, so it is redrawn when the design changes and thrown away when nobody will redraw it. Keeping a stale drawing beside a working database is worse than never drawing one.

The afternoon buys a page the storekeeper can argue with, and an argument on paper is the cheapest one a dispensary will ever have.

The Master Key: *Draw the diagram once the design holds more than one entity type, because relationships and claims about what is a thing are what a list of tables cannot state.*

HOT Question 45

One patient, one batch and one dispenser meet at a counter at Hale for about a minute. The book has to hold all three names together, or it says nothing useful on the day a recall notice arrives.

Judge whether the ternary relationship in which a dispenser issues a batch to a patient should be modelled as it stands or broken into binary relationships, and state the condition that decides it.

Model Answer

Model it as it stands, and the condition that decides any such case is whether the binary relationships can be put back together to give exactly the facts recorded: where the pairs allow a combination that never happened, the fact was about the three together and splitting it destroys it.

The case for splitting it: Binary relationships are easier to draw, read and check, and many relationships that look ternary are not. Where the third participant is really an attribute of the association, or joins only one of the other two, the ternary is an accident of wording.

The three pairs written out: Split at Hale, the association becomes dispenser handles batch, dispenser serves patient, and batch reaches patient. Each of the three is true on its own, and that is exactly why the split is tempting; the difficulty is what the three of them together fail to say.

Putting the pairs back together: Suppose two dispensers are on duty and two batches of one drug are open, and each dispenser has handled each batch. For a patient served once, the pairs allow four dispenser and batch combinations and only one of the four happened. The pairs admit what never occurred, so the fact was ternary.

When the split is right: If the book recorded only who handled which batch, and separately who served which patient, with nothing tying the two together, then three binary relationships would hold everything there was. The question is what one recorded fact is about, never how many nouns appear in the sentence.

The cost of keeping it ternary: A three ended relationship is harder to read and easier to draw wrongly, and what becomes of it once the diagram is turned into tables is settled later in this chapter. That cost is worth paying only where the pairs lose something.

The judgement: Keep this one ternary, because a dispensary that cannot say which batch reached which patient from whose hand cannot answer a recall, which is the one question the book exists to answer. Split a ternary only where every fact it holds can be rebuilt from the pairs without inventing a single combination.

What would change the answer: If the dispensary stopped naming the dispenser and recorded only that a batch reached a patient, the relationship would fall to a binary of its own accord. Degree follows the fact recorded, and changes when the recording changes.

Three names in one sentence do not make a ternary; a fact that cannot be rebuilt from the pairs does.

The Master Key: *Keep a relationship ternary where the pairs would admit combinations that never happened, and split it into binaries wherever they would not.*

Cardinality, participation and weak entities

A cardinality says how many at most, and a participation says how few at least, and the two together are what an end of a relationship actually claims. This section states each of the three cardinalities in both directions, because one direction on its own does not separate one to many from many to many; it distinguishes total participation from partial; and it sets out the weak entity, which has no key of its own and is identified only inside the entity that owns it. One correction runs through the whole of it: in a one to many relationship the one end is at most one, and a definition that says at least one has quietly described a many to many instead. Every example here comes from a clinic appointment book.

BINDER Question 46

The argument at Mnyuzi is over one line in the book. A nurse says a patient has many appointments; the clerk says an appointment has one patient. Both are describing the same line, and neither has finished the sentence.

Define cardinality, and describe the three cardinalities a relationship may have.

Model Answer

Cardinality is the maximum number of occurrences of one entity that may be associated with a single occurrence of the other, and a relationship has one of three: one to one, one to many and many to many.

A cardinality is a pair of answers: The same question is asked at each end, and the name records both answers. One to many is the answer many in one direction set beside the answer one in the other, so half of it is not yet a claim about anything.

One to one, stated both ways: Each occurrence at one end relates to at most one at the other, and the same holds coming back. A patient at Mnyuzi holds at most one clinic card, and a clinic card belongs to at most one patient.

One to many, stated both ways: One occurrence at the first end may relate to many at the second, while each occurrence at the second relates to at most one at the first. One patient may hold many appointments; each appointment relates to at most one patient. The words at most carry the claim.

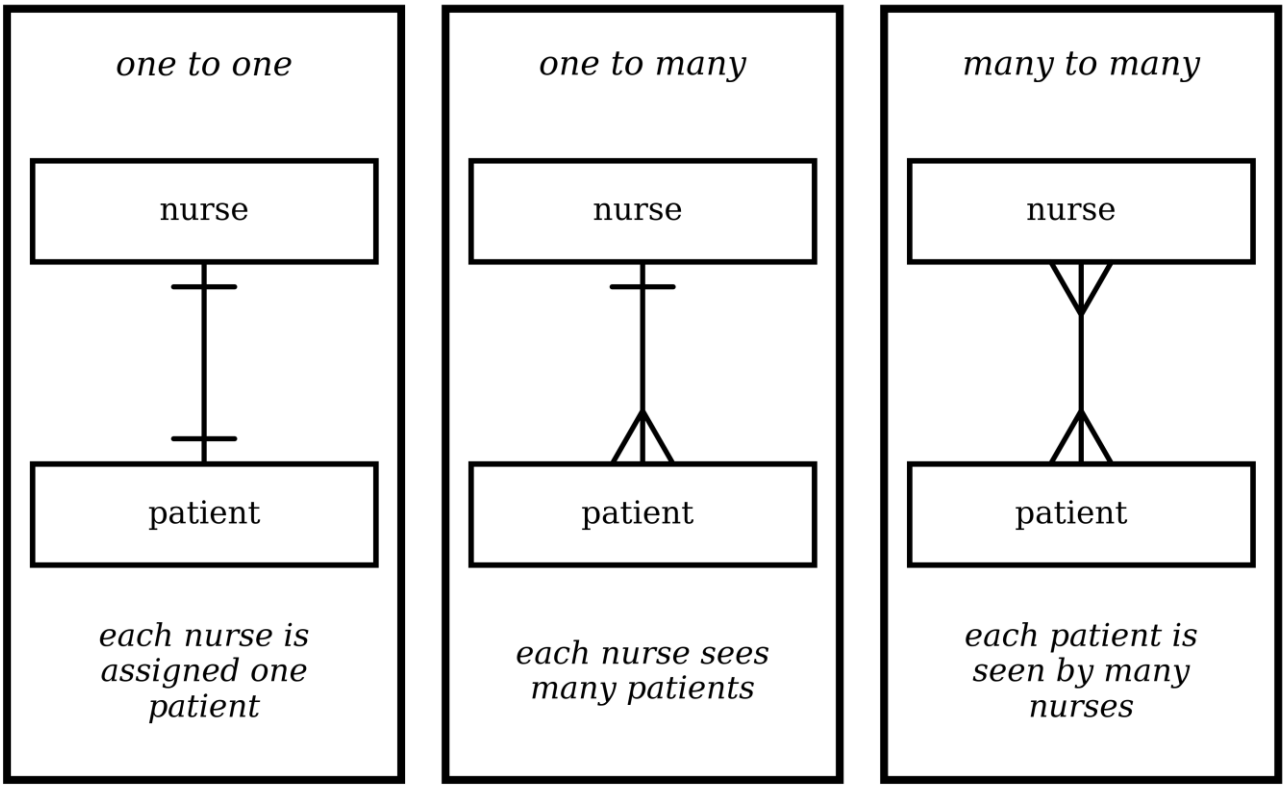


Figure 4.10: *The same two entities under each of the three cardinalities, so only the end marks change.*

Many to many, stated both ways: Many is the answer in both directions, as the third panel of Figure 4.10 shows. One patient is seen by many nurses over a year and one nurse sees many patients, so neither end can be pinned to a single occurrence. A shape like this must be resolved before tables are built from it.

Why one direction is never enough: The sentence a patient has many appointments is true of a one to many relationship and equally true of a many to many one, since it says nothing about the other end. Only the pair of answers separates them.

Say both directions and the name follows; say one and the reader is left to guess the other.

The Master Key: *A cardinality is a maximum in each direction, and one to one, one to many and many to many are the three pairs of answers.*

BINDER Question 47

A pencil line runs down the Msata appointment book where the clerk has ruled off patients with nothing booked. Half the page is names beside times. The other half is names beside empty space, and the book allows both.

Explain participation in a relationship, and distinguish total participation from partial.

Model Answer

Participation is the minimum number of occurrences an entity must take in a relationship: it is total when every occurrence of that entity must take part, and partial when an occurrence is allowed to stand outside the relationship altogether.

Participation fixes the minimum: Cardinality answers how many at most; participation answers how few at least. They are two different numbers about the same end, which is why an end carries two marks rather than one, and why reading only the maximum leaves half the rule unstated.

Total participation on an end: Every occurrence must take part, so no occurrence of that entity exists outside the relationship. Every appointment in the Msata book is written for a patient, and an appointment with no patient is not an unusual entry but an impossible one.

Partial participation on an end: An occurrence may stand alone. A patient registered this morning with nothing yet booked is a proper patient, so the patient end is partial. Partial does not mean rare; it means permitted, and one lawful case is enough to make an end partial.

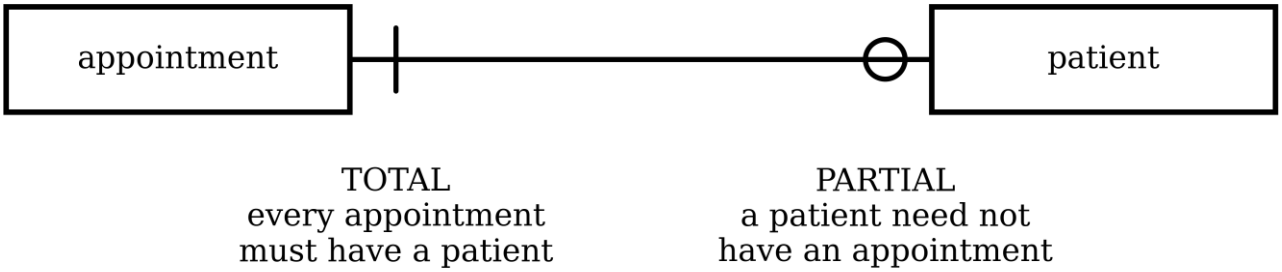


Figure 4.11: *Total participation on one end and partial on the other.*

The two ends are decided separately: Nothing about one end settles the other. In Figure 4.11 the appointment end is total and the patient end partial on the same line, and many designs go wrong by deciding the pair once and writing the same answer at both ends of it.

Why the minimum is worth the trouble: The maximum is usually obvious and the minimum is usually assumed. An assumed minimum is a rule kept only while people remember it, and the rows that break it arrive quietly, one at a time, in whatever the clinic starts doing next month.

The maximum tells you what the book may hold, and the minimum tells you what it may leave out.

The Master Key: *Cardinality is the maximum at an end and participation the minimum, total meaning every occurrence must take part and partial meaning an occurrence need not.*

BINDER Question 48

An appointment at Miono drew three reminders in one day, at eight, at noon and at four. The log names each of them by nothing but that time and the appointment it belongs to, and the clerk has never needed more.

Explain what a weak entity is, and describe its identifying relationship and its partial key.

Model Answer

A weak entity is an entity with no key of its own, identified only through the entity that owns it, so its identity is the owner's key together with a partial key that tells its occurrences apart inside that one owner.

Two tests, and both must hold: A weak entity cannot be told from its fellows by any attribute of its own, and it cannot exist apart from its owner. Dependence alone is not the test: an appointment cannot exist without a patient either, yet it carries an identifier of its own and is therefore strong.

The identifying relationship: The relationship joining a weak entity to its owner is called identifying, because it is what carries the owner's identity down to it. The weak entity takes part in that relationship totally, since an occurrence with no owner could not be named at all.

The partial key: A partial key separates occurrences within one owner and repeats freely across the whole entity. The time a reminder was sent tells the reminders for one appointment apart, and that same time stands against hundreds of other appointments, which is what makes it partial.

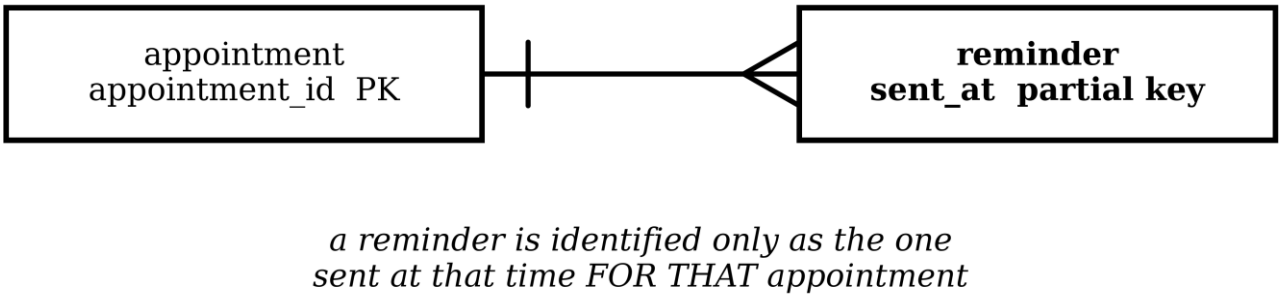


Figure 4.12: *A weak entity, identified only within the entity that owns it.*

The identity, written out: The full identity is the owner's key followed by the partial key, taken together: this appointment, and the reminder sent at that time. Either part alone names nothing, as Figure 4.12 shows, and that is the sense in which the entity is weak.

Weak is not a judgement on the entity; it is a statement that its name is only half its own.

The Master Key: *A weak entity borrows its owner's key through an identifying relationship and adds a partial key that separates its occurrences inside that owner.*

BINDER Question 49

The clinic officer at Mnyuzi is asked how many patients one nurse sees. She answers that it depends on the day, which is true and useless. The book needs the largest number the arrangement permits, not the usual one.

Explain how to decide the cardinality of a relationship from a description of how an organisation works, and give the two questions to ask.

Model Answer

Decide a cardinality by asking two questions, one in each direction: take a single occurrence at this end and ask at most how many occurrences at the other end it may relate to, then stand at the other end and ask the same thing coming back.

The first question: Take one patient at Mnyuzi and ask at most how many appointments she may hold. The answer is many, and it stays many whether she in fact holds one or several, because the question is about what the arrangement permits.

The second question: Now take one appointment and ask at most how many patients it may name. The clinic writes one name, so the answer is one. Many in the first direction beside one in the second is what makes the relationship one to many, and it also says which end is which.

Answer at most, never usually: What is wanted is the largest number the arrangement allows, not the number that commonly occurs. If an appointment may ever be written for a mother and her infant together, it may name many patients, however seldom that happens, and the cardinality changes with the one case.

Fix the relationship before counting: A nurse sees one patient at a time and many patients in a year, and both answers are right for different relationships. Name the relationship in a phrase the clinic would recognise before either question is asked.

Where the description is silent: A written description settles cardinality less often than it appears to. When it does not, ask the people who keep the book what has already happened, since a case that has occurred once settles the maximum and no argument about what should happen can.

Two questions, answered at most rather than usually, settle every cardinality you will be given.

The Master Key: *Ask at most how many at each end in turn, about what the arrangement permits rather than what it commonly does, and the pair of answers is the cardinality.*

REAL Question 50

A slip folded into the front of the appointment book at Miono carries three statements about the same relationship, one under another. Two of the three are wrong, and the staff have been working to all three since the new book opened.

Name the one true statement among the three, say exactly what is wrong with each of the other two, and correct them.

A. In a one-to-many relationship, each appointment relates to at least one patient.

B. In a one-to-many relationship, each appointment relates to at most one patient.

C. In a one-to-many relationship, each appointment relates to exactly one patient.

Model Answer

Statement B is the true one, because a cardinality states a maximum and at most one is exactly what the one end claims; A is wrong because at least one permits many and so describes a many to many, and C is wrong because exactly one quietly adds a minimum the cardinality never claimed.

Why B is right: A cardinality is a ceiling, and B puts that ceiling at one: an appointment relates to no more than one patient. Set beside the other direction, where one patient may relate to many appointments, that is the whole of what one to many asserts.

What at least one actually permits: At least one is a floor with no ceiling above it. A quantity that is at least one may be one, may be two and may be forty, so A allows an appointment to relate to forty patients. At most fixes a maximum and at least fixes a minimum, and the two belong to different halves of the design.

Why A defines the wrong relationship: An appointment that may relate to many patients, in a book where a patient may relate to many appointments, is a many to many relationship and nothing else. A therefore defines the very relationship one to many exists to be distinguished from, and the repair is one word, most for least.

Why C is not wrong about the maximum: Exactly one is at most one and at least one together. Its first half agrees with B and is right. Its second half is a separate claim, that no appointment may exist without a patient, and that claim belongs to participation rather than to cardinality.

Why the extra half of C may be false: Whether every appointment must name a patient is a fact about one clinic and not about one to many. Where the Miono clerk rules an outreach morning into the book before any names come in, an appointment stands with no patient, and C is false there while B stays true.

How to state it so it survives: Give the maximum in both directions first, at most one patient for an appointment and many appointments for a patient, then state the minimum at each end separately. Two sentences instead of one, and neither can be mistaken for the other.

One word carries it, since at most is the cardinality and at least is a different claim about a different thing.

The Master Key: *In a one to many relationship the one end is at most one, because at least one permits many and exactly one smuggles in a participation rule that may be false.*

REAL Question 51

A new clerk at Msata copies last year's diagram into the new book and changes nothing on it. By March the book holds sessions with nothing in them, and nobody can say whether that is a fault or something the diagram always allowed.

Work out the cardinality and the participation at each end of both relationships in this book, and justify each of the four decisions separately.

Every appointment at Msata is written for one named patient, and the clerk will not enter one without a name.

A patient may hold several appointments in the book, and a patient registered today usually holds none.

Every appointment is placed in exactly one session, a session being one nurse's clinic on one morning.

Sessions for the whole week are ruled in on Monday, and some are still empty when the week ends.

Model Answer

The book holds two relationships, a patient books appointments and a session holds appointments, and both are one to many with the many at the appointment end, while in both the appointment end is total and the other end partial.

The cardinality of books: A patient may hold several appointments, so the answer at the appointment end is many. An appointment is written for one named patient and no more, so the answer at the patient end is one. Many beside one makes it one to many, with the many at the appointment end.

The participation at each end of books: The clerk will not enter an appointment without a name, so no appointment stands outside the relationship and the appointment end is total. A patient registered today usually holds none, and that is an ordinary state rather than an error, so the patient end is partial.

The cardinality of holds: An appointment is placed in exactly one session, so the answer at the session end is one. A session is a whole morning and takes as many appointments as it has room for, so the answer at the appointment end is many. One to many again, and again the many is the appointment.

The participation at each end of holds: Exactly one session means at least one session, so every appointment takes part and that end is total. Sessions are ruled in on Monday and some are still empty when the week ends, so a session may take no part at all and the session end is partial.

Four decisions, four separate reasons: No decision here was taken twice. Each came from its own line of the description, and the word exactly did two jobs at once, fixing a maximum of one and a minimum of one, which is why that line had to be read twice before both were written down.

One occurrence of	At most, at the other end	Must it take part	The line that settles it
a patient, in books	many appointments	no, partial	a patient registered today usually holds none
an appointment, in books	one patient	yes, total	no appointment is entered without a name
a session, in holds	many appointments	no, partial	some sessions are still empty when the week ends
an appointment, in holds	one session	yes, total	every appointment is placed in exactly one session

Table 4.5: *The four decisions, each beside the line of the description that settles it.*

Four answers on two lines, and every one of them was fixed by a sentence somebody in the clinic had already said.

The Master Key: *Take each end of each relationship in turn and settle its maximum and its minimum separately, because the four answers are four decisions and no two of them are the same decision.*

REAL Question 52

Slot seven was cancelled at Miono, and slot seven still appeared on Wednesday, on Thursday and on Friday. Nobody had made a mistake. There are as many slot sevens in the book as there are sessions, and the number alone names none of them.

Identify the weak entity in this book, name its owner and its partial key, and explain what would go wrong if it were given a key of its own.

The book at Miono is ruled into sessions, and each session is divided into numbered slots, one to twenty four.

A slot is known to the staff as slot seven of Tuesday morning, and by nothing else.

The numbering starts again at one in every session, so every session has a slot seven.

When a session is cancelled its slots go with it, and a slot is never moved to another session.

Model Answer

The slot is the weak entity, the session owns it and the slot number is its partial key, so a slot is identified only as that number within that session and has no identity apart from it.

Why the slot is the weak entity: Nothing the slot carries tells one slot from another across the book. The number seven stands in every session and the slot has nothing else of its own, so the only way to reach one is through the session it sits in. The session has a key of its own and is therefore strong.

The relationship that identifies it: The relationship between session and slot is identifying, because it supplies the missing half of the slot's identity. The slot end takes part totally, since a slot outside a session could not be named, and one session holds many slots, so the slot end is also the many end.

The partial key and the full identity: The slot number is the partial key, unique inside one session and repeated in every other. The full identity is the session's key together with that number, and it is the pair, not either half of it, that picks out one slot in the book.

What breaks if the slot is made strong: Give the slot an identifier of its own and the number stops being controlled. Nothing then prevents two slot sevens inside one session, because uniqueness has moved to the new identifier, and nothing prevents a slot from being pointed at another session or at none, since the link becomes an ordinary reference that may be changed or left empty.

What is not the test: Depending on another entity does not by itself make an entity weak. An appointment cannot exist without a patient, yet it carries an identifier the clinic issues and is found by that alone. The test is whether the identity itself has to borrow the owner's key.

A slot number is not a name but half a name, and the session supplies the other half.

The Master Key: *An entity is weak when its identity has to borrow its owner's key, and its partial key separates its occurrences only inside that one owner.*

HOT Question 53

Two designers leave Mnyuzi with the same set of facts and different drawings. One says the many to many line is what the clinic told them and should stay. The other says the drawing is now a plan, and a plan cannot contain it.

Assess whether a many to many relationship should ever be left standing in a finished design, and reach a judgement.

Model Answer

A many to many relationship may stand in a drawing that only records what the clinic does, and may never stand in a design the tables are to be built from, and the condition that decides a case is whether the drawing is still a description or has become the plan.

The case for leaving it: It is the truthful shape of the fact. Many nurses see many patients at Mnyuzi, and a drawing that says so can be put in front of the staff and confirmed by them. Whatever is put in its place is an entity the clinic has no name for and cannot check.

The case against leaving it: Nothing stores a many to many directly, so a design that keeps one has left its most awkward decision undone. Worse, the pairing has facts of its own, the day a nurse saw a patient and the room they used, and a plain line between two boxes has nowhere to write them.

The tier settles most of the argument: A conceptual model exists to be shown and agreed, and a logical model exists to be built from, as the tiers were set out earlier in this chapter. A many to many is at home in the first and out of place in the second, and much of this quarrel is two people comparing drawings made for different purposes.

The test that decides one case: Ask whether the pairing carries any fact of its own. If it does, resolve it at once and at every tier, because a drawing that cannot hold a fact the clinic already records is not describing the clinic. If it carries nothing, it may stand until the drawing becomes a plan.

The judgement: Leave a many to many only in a conceptual model shown to the people it describes, and resolve it before that model becomes the thing tables are built from. A finished design containing one is unfinished, whatever else about it is right. How the resolution is done comes later in this chapter.

The excuse that is not a reason: Leaving it for the program to sort out is the one answer to refuse. The choice of how each pair is recorded then falls to whoever writes the first program, and the second program makes it differently, which is how one clinic ends with two records of the same visit.

A drawing may state the fact; a plan has to say where the fact will be kept.

The Master Key: *Leave a many to many only in a model meant to be read and agreed, and resolve it before that model becomes the plan the tables are built from.*

HOT Question 54

Nobody noticed the appointment with no patient against it until a reminder went out to nobody. The booking program at Msata had refused such a row since the book began. This one had arrived from a spreadsheet import, which refused nothing.

Judge whether total participation should be enforced by the database or left to the application, and state the condition that decides it.

Model Answer

Enforce total participation in the database wherever the rule can be judged from a single row, and leave it to the application only where it cannot, since the condition that decides the case is whether the minimum is visible in the row that carries the reference or only in rows somewhere else.

The half a database settles by itself: Total participation on the end that holds the reference is a rule about one row. Every appointment must name a patient, and the appointment either names one or does not, so requiring that reference to carry a value enforces the rule for every program and every import at once.

The half it cannot settle cheaply: Every patient must hold at least one appointment is a rule of the opposite shape. Nothing on the patient row shows whether it is kept, and the first appointment cannot be written before the patient exists, so the rule is broken for an instant every time a patient is registered.

The case for the application: The program can say why in the clerk's own words, can gather both facts before writing either, and can allow a half-finished booking on the screen that the finished record will never contain. A database can only refuse, and a refusal nobody can read is a refusal people work around.

Why the application alone is not enough: The program is one door among several. An import, a second program, a repair typed at the console and next year's replacement all reach the same rows, and a rule living in one of them is a habit rather than a constraint. The empty appointment came in through the door nobody guarded.

The judgement: Declare it in the database wherever one row settles it, and keep the check in the application as well, for the wording of the message and not for the safety. Where one row cannot settle it, do not force the matter with a rule that fires before the second row can exist; write both rows in one unit of work and check at the end of that.

What would change the answer: Only cost. Where a rule of the second shape has to be checked on every write of a large and busy relation, and breaking it produces a report that undercounts rather than a record that is wrong, checking it nightly is fair. The verdict follows the harm, not the tidiness.

The rule that survives is the one written where every door leads.

The Master Key: *Enforce a minimum the database can judge from one row, put the rest into one unit of work and check it there, and never leave either to a single program.*

Resolving many-to-many, and mapping a diagram to tables

A relationship with many on both sides cannot be stored as it stands, and what to do about it is the most useful thing in this part of the chapter. This section shows why a many-to-many has to be resolved into a relation of its own, and then sets out the rules that turn a finished diagram into a set of tables: where the foreign key goes for each cardinality, what becomes of a multivalued attribute, and how a weak entity is keyed. Every schema written later in the chapter is the output of these rules.

BINDER Question 55

A single column on the intake form at the Ngerengere store asks which member brought the sacks. Four names are squeezed into it, two of them abbreviated, and the weight each member brought is written nowhere at all.

Explain why a many-to-many relationship cannot be implemented directly in a relational database, and describe the associative entity that resolves it.

Model Answer

A many-to-many relationship cannot be implemented directly because a foreign key column holds one value in each row while each side needs to point at many rows of the other, and the resolution is an associative entity: a relation of its own holding the primary key of both sides.

What one column can hold: Put delivery_id inside member and each member is tied to one delivery. Put member_id inside delivery and each delivery names one member. Whichever side is chosen, the column has room for one value and the relationship needs many.

The associative entity: The resolution is a third relation, named for the event rather than for either side, holding the primary key of each. In the cooperative it is intake, and one row is the maize one member put into one load.

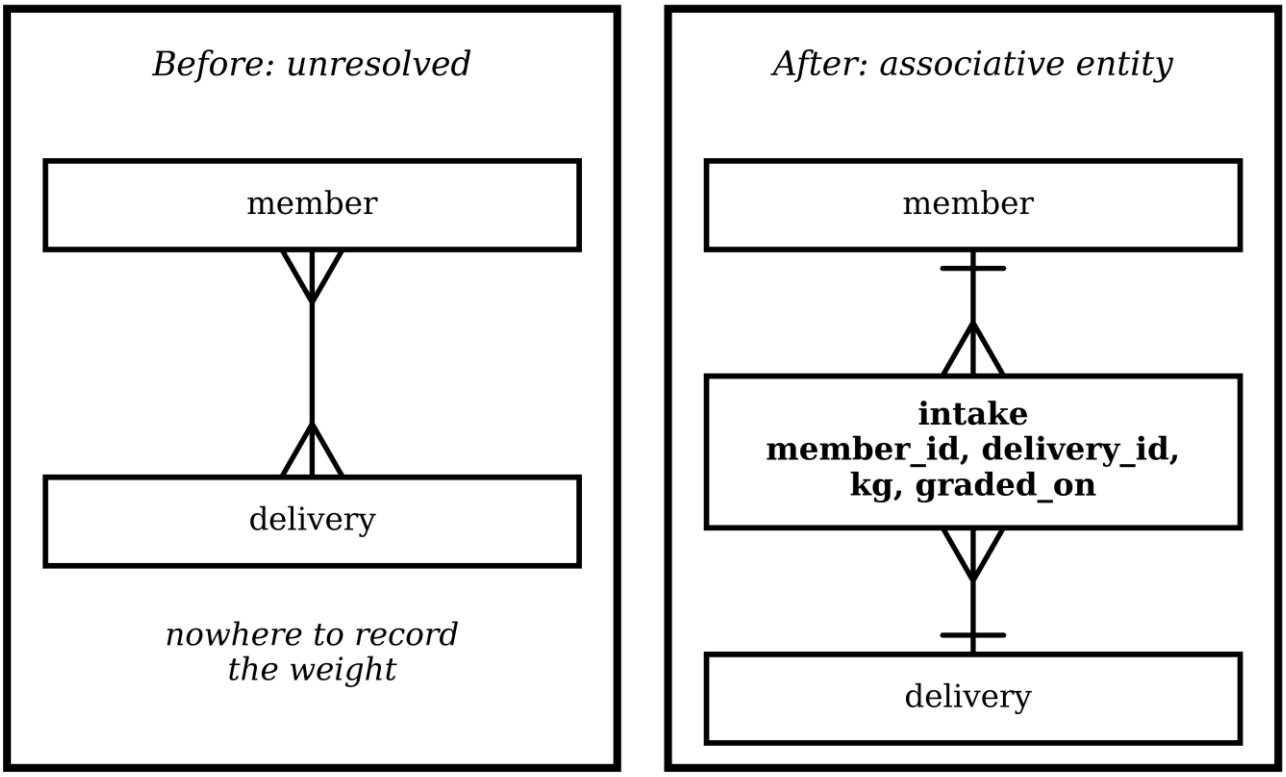


Figure 4.13: *A many-to-many before and after an associative entity is introduced.*

The attributes with nowhere else to live: The weight credited to a member in one load belongs to neither the member nor the load. Written into member it is one weight for the season; written into delivery it is one weight for everybody on the lorry. Until the pair is a relation there is nowhere to put it.

Not a trick of implementation: Students are often taught that resolution is something the database imposes on a design that was already finished. It is not: the diagram was incomplete, because it carried a relationship with attributes and no place to keep them.

What the diagram looks like afterwards: One many-to-many becomes two one-to-many relationships, each running from a strong entity into the associative entity, with the many end at the associative entity. A pairing once implied by a line is now a row that can be counted and dated.

The associative entity is not a way round the relational model; it is the relation the diagram was short of.

The Master Key: *A many-to-many needs an associative entity holding the key of each side, because a foreign key column has room for one value and the relationship's own attributes have nowhere else to live.*

BINDER Question 56

At Vikindu two people drew the same cooperative and produced different tables. The entities matched and the lines matched; the whole disagreement was about which side of each line the borrowed key column should end up on.

State the rules for mapping a completed diagram to tables, covering each cardinality, a multivalued attribute and a weak entity.

Model Answer

Every entity becomes a relation with its own primary key, and each relationship is then placed by its cardinality: the foreign key goes on the many side of a one-to-many, on either side of a one-to-one, and on neither side of a many-to-many, which needs a relation of its own.

One to many, where nothing is negotiable: The foreign key goes in the relation on the many side, holding the primary key of the one side. Each row there has exactly one partner, so one column is enough to name it.

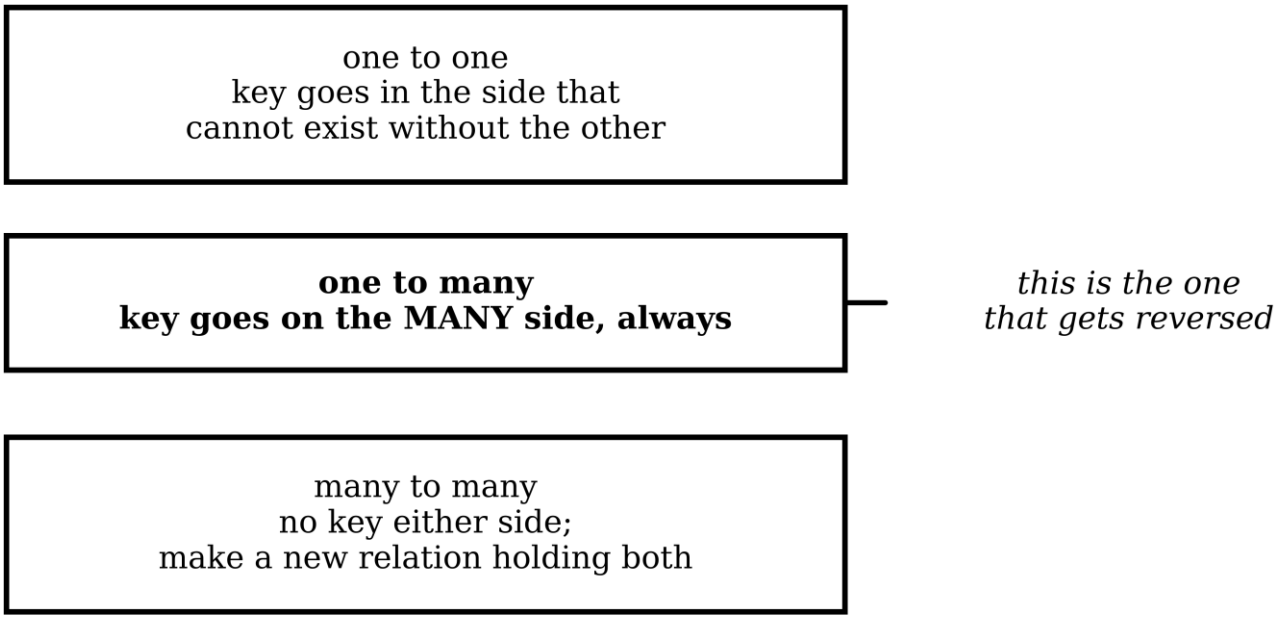


Figure 4.14: *Where the foreign key belongs for each cardinality.*

One to one, and the side to prefer: Either relation may carry it, so put it where the row always exists, meaning the side that cannot be there without a partner, and declare that column unique. The column is then never null, while on the optional side most rows would carry one.

Many to many, which neither side can hold: Create a relation holding the primary key of each side, keyed by those two columns together, and move every attribute of the relationship into it. The two original relations gain nothing at all.

A multivalued attribute: It leaves the entity for a relation holding the owner key and one column for the value, keyed by the two together. A member with three telephone numbers has three rows there and one row in member.

A weak entity: Its relation is keyed by the primary key of the owner plus its own partial key, and that owner key is a foreign key as well. The partial key is unique only inside one owner, and the composite key is what says so.

On the diagram	Where the key goes
One to one	Foreign key in either relation, declared unique; prefer the side whose row always exists
One to many	Foreign key in the relation on the many side, referencing the one side
Many to many	A new relation, keyed by the two foreign keys together
Multivalued attribute	A new relation, keyed by the owner key and the value together
Weak entity	Owner key plus partial key as the primary key, the owner key also a foreign key

Table 4.6: *The five mapping rules, and where the key goes in each.*

Every rule in the list puts the key where one value is enough, and creates a relation wherever one value is not.

The Master Key: *Every entity becomes a relation, the foreign key sits on the many side of a one-to-many, and a many-to-many, a multivalued attribute and a weak entity each need a relation with a composite key.*

REAL Question 57

A message forwarded from somebody's notes reached the committee at Ruvu with three schemas in it. Members take maize wherever the lorry is going, and every store fills up from whoever brings it, so two of the three cannot be right.

Each member may deliver to many stores and each store receives from many members. Identify which of the three schemas is correct, say exactly what each of the other two makes impossible, and correct them.

A. member(member_id, name, store_id) and store(store_id, village)

B. member(member_id, name) and store(store_id, village, member_id)

C. member(member_id, name), store(store_id, village) and delivery(member_id, store_id, kg, delivered_on)

Model Answer

Schema C is the correct one, because it gives the pairing of a member and a store a relation of its own; A pins each member to a single store, B pins each store to a single member, and neither A nor B has anywhere to record how much was delivered or when.

Why C is right: delivery holds member_id and store_id together, so one member may stand beside every store and one store beside every member. The pair is the key, and kg and delivered_on sit in the same row because they describe the delivery rather than either side of it.

What A makes impossible: store_id inside member gives each member one store and no more. A farmer who takes maize to Ngerengere in July and to Vikindu in August must have that column overwritten, so the July delivery stops existing, and the schema now states something false about the cooperative: that a member belongs to one store.

What B makes impossible: The same fault facing the other way. member_id inside store gives each store one member, so Ruvu can record maize from exactly one farmer and every other delivery to Ruvu has nowhere to go. A and B are one many-to-many written as a one-to-many, in the two directions available.

What neither of them can record: kg and delivered_on appear in neither A nor B, and there is no honest place to add them. A weight put in member is one weight for a member, a weight put in store is one weight for a store, and what the cooperative needs is one weight for each member at each store on each day.

Correcting A and B: Both are corrected the same way. Take the borrowed key column out of the entity that should never have held it and add the third relation, exactly as C has it. Where one member may deliver to the same store more than once in a season, delivered_on joins the key so the second load is a second row.

A and B are not two mistakes but one mistake facing two ways, and the cure for both is a relation neither of them has.

The Master Key: *Where both sides are many, no column on either side can carry the relationship, and the pairing needs a relation of its own with the attributes of the pairing inside it.*

REAL Question 58

Six tables came out of one diagram at Vikindu, and the committee had expected three. One extra carries a weight belonging to no single entity, one carries telephone numbers, and the last exists because each store numbers its stacks from one.

Map the cooperative described below to a full set of tables, naming every primary key and every foreign key, and saying which side each foreign key sits on and why.

The cooperative runs three stores, one at Ngerengere, one at Vikindu and one at Ruvu.

Each member lives in one village and may give the cooperative more than one telephone number.

A delivery is one lorry load arriving at exactly one store, and a store receives many deliveries.

One delivery carries maize from many members, and one member's maize travels in many deliveries; the weight credited to each member in a delivery and the date it was graded must both be kept.

Each store is divided into stacks, numbered from one within that store and meaningless outside it.

Model Answer

The three entities become member keyed by member_id, store keyed by store_id and delivery keyed by delivery_id, with store_id sitting in delivery as a foreign key; three further relations are then needed, intake for the many-to-many, member_phone for the multivalued attribute and stack for the weak entity.

The three entities and their keys: member holds member_id, name and village; store holds store_id and village; delivery holds delivery_id and arrived_on. Each of the three has an identifier the cooperative issues itself, so each becomes a relation with that identifier as its primary key and no foreign key yet.

The one to many, and the side the key sits on: A store receives many deliveries and a delivery arrives at one store, so store_id goes into delivery as a foreign key referencing store. It sits on the many side because a row of delivery has exactly one store to name, while a row of store would need as many delivery columns as there are lorries.

The many to many: intake holds member_id, delivery_id, kg and graded_on, with member_id and delivery_id together as its primary key and each of them also a foreign key. The weight and the date live here because they belong to the pairing, and one row is the maize one member sent in one load.

The multivalued attribute: member_phone holds member_id and phone, keyed by the two together and referencing member. The alternative that looks easier, a second and third telephone column inside member, fixes the number of telephones in the schema and makes one fact live in three places.

The weak entity: stack holds store_id, stack_no and sacks, keyed by store_id and stack_no together, with store_id referencing store. stack_no is a partial key: it counts from one inside each store, so it identifies a heap of maize only in the company of the store that holds it.

Why every foreign key sits where it does: Each one is in the relation whose row has a single partner. delivery has one store, intake has one member and one delivery, member_phone has one member, and stack has one store. No relation here holds a key belonging to something it has many of.

Relation	Primary key	Foreign keys
member	member_id	none
store	store_id	none
delivery	delivery_id	store_id references store
intake	member_id, delivery_id	member_id references member, delivery_id references delivery
member_phone	member_id, phone	member_id references member
stack	store_id, stack_no	store_id references store

Table 4.7: *The six relations, the primary key of each, and every foreign key it carries.*

Three entities on the diagram became six relations, and the three that were added carry facts the first three had no room for.

The Master Key: *Map each entity to a relation, put the foreign key of a one-to-many on the many side, and add one relation for each many-to-many, each multivalued attribute and each weak entity.*

REAL Question 59

The column was added to store, not to delivery, and for a week nothing looked wrong at Ruvu. Then a second lorry arrived, the clerk typed its number over the first, and the earlier load stopped existing.

A one-to-many relationship has been mapped with the foreign key on the one side. Explain exactly what breaks, and correct it.

store(store_id, village, delivery_id)

delivery(delivery_id, arrived_on)

Each delivery arrives at exactly one store, and each store receives many deliveries.

Model Answer

The column can hold one value where the relationship needs many, so store can name only one delivery at a time, and the design then either loses every earlier delivery or repeats the whole store row once per lorry; the correction is to take delivery_id out of store and put store_id into delivery.

What the column can hold: delivery_id inside store has room for one delivery identifier in each row of store. Ruvu receives many, so the identifier standing there is whichever was written last, and it was written over the one before it. The relationship is real and the column is not big enough to be told about it.

The first way it breaks, by losing rows: Keep one row per store and every delivery but the newest is gone from the schema. The delivery rows may still exist, but nothing connects them to a store any more, so a question as ordinary as which loads reached Ruvu in July has no answer the database can give.

The second way it breaks, by repeating rows: Keep every delivery instead and store must be repeated once for each of them. store_id then identifies nothing, since the same store appears many times, and the village is copied into every one of those rows, so correcting the village in one place leaves the copies disagreeing with it.

Why one side is right and the other never is: A row on the many side has exactly one partner, so one column holds all of it. A row on the one side has many partners and no single column will ever hold them. This is not a convention that could have gone the other way; it follows from a column holding one value.

The correction: Drop delivery_id from store and declare store_id in delivery, not null and referencing store. store keeps store_id and village and gains nothing, delivery names the store it arrived at, and the number of lorries a store may receive is now limited by nothing in the schema.

The two schemas look equally reasonable written down, and only one of them can hold the deliveries the cooperative actually made.

The Master Key: *A foreign key on the one side can record a single partner where the relationship has many, so it belongs on the many side, where each row has exactly one.*

BUILD Question 60

Twice on one morning at Ngerengere a member's maize goes onto the same lorry, and the clerk records it twice. The second row is refused before it lands, and the refusal comes from the shape of the key.

Create the three tables that resolve the many-to-many between members and deliveries, with the junction table carrying its own attributes, then show that its composite primary key refuses a duplicate pairing.

member holds member_id as its primary key and a name that may not be null.

delivery holds delivery_id as its primary key and the date the lorry arrived.

intake holds member_id and delivery_id together as its primary key, with kg and graded_on.

Then attempt the same member and the same delivery a second time.

Model Answer

The three declarations below turn one many-to-many into two one-to-many relationships through intake, whose primary key is member_id and delivery_id together, so the second attempt at the same pairing is refused by the key itself rather than by anyone remembering to check.

```
 1  CREATE TABLE member (
 2    member_id CHAR(5)     NOT NULL,
 3    name      VARCHAR(40) NOT NULL,
 4    PRIMARY KEY (member_id)
 5  ) ENGINE = InnoDB;
 6
 7  CREATE TABLE delivery (
 8    delivery_id CHAR(5) NOT NULL,
 9    arrived_on  DATE    NOT NULL,
10    PRIMARY KEY (delivery_id)
11  ) ENGINE = InnoDB;
12
13  CREATE TABLE intake (
14    member_id   CHAR(5)      NOT NULL,
15    delivery_id CHAR(5)      NOT NULL,
16    kg          DECIMAL(7,2) NOT NULL,
17    graded_on   DATE         NOT NULL,
18    PRIMARY KEY (member_id, delivery_id),
19    FOREIGN KEY (member_id)   REFERENCES member (member_id),
20    FOREIGN KEY (delivery_id) REFERENCES delivery (delivery_id)
21  ) ENGINE = InnoDB;
22
23  INSERT INTO member (member_id, name) VALUES ('M0041', 'Neema Kiluwa');
24  INSERT INTO delivery (delivery_id, arrived_on)
25  VALUES ('D0107', '2026-07-14');
26  INSERT INTO intake (member_id, delivery_id, kg, graded_on)
27  VALUES ('M0041', 'D0107', 480.00, '2026-07-14');
28  INSERT INTO intake (member_id, delivery_id, kg, graded_on)
29  VALUES ('M0041', 'D0107', 95.00, '2026-07-15');
```

The server refuses it

```
ERROR 1062 (23000) at line 28: Duplicate entry 'M0041-D0107' for key 'PRIMARY'
```

Program 4.2: *A junction table whose composite key refuses a second row for the same pairing.*

What the composite key says: Naming both columns in one primary key states that a member and a delivery may be paired at most once. The single declaration does two jobs: it identifies a row of intake, and it forbids the duplicate that would let one load be credited to a member twice.

Reading the refusal: The server names the duplicated pair and the key that rejected it, and it refuses the whole row rather than storing part of it. Nothing was written, so intake still holds one row for that member in that load, and the weight already credited is untouched.

The one decision worth naming: kg is DECIMAL because a weight is measured and has to add up exactly, and it sits in intake and nowhere else: a weight in member would be one figure for the season, and a weight in delivery would be one figure for everybody on the lorry.

The junction table is where the relationship keeps its own facts, and its key is where the cooperative keeps its rule about them.

The Master Key: *Key a junction table on its two foreign keys together, and a repeated pairing is refused by the schema instead of being found in the accounts later.*

BUILD Question 61

Stack one at Ngerengere and stack one at Vikindu are different heaps of maize in different villages. The number chalked on the post means nothing until somebody says which store it was written in, and the key has to say so too.

Create the table for the weak entity stack, keyed by the owner key together with the partial key, and show that the same stack number is accepted under two different stores and refused a second time under one.

A stack exists only inside a store and is numbered from one within that store.

stack holds store_id and stack_no together as its primary key, with store_id referencing store.

Number a stack one at Ngerengere, at Vikindu and at Ruvu, then attempt stack one at Ngerengere again.

Model Answer

The primary key of stack is store_id and stack_no together, so stack number one exists once inside each store and cannot exist twice inside any of them, and that same store_id is the foreign key referencing store.

```
 1  CREATE TABLE store (
 2    store_id CHAR(3)     NOT NULL,
 3    village  VARCHAR(20) NOT NULL,
 4    PRIMARY KEY (store_id)
 5  ) ENGINE = InnoDB;
 6
 7  CREATE TABLE stack (
 8    store_id   CHAR(3)      NOT NULL,
 9    stack_no   TINYINT      NOT NULL,
10    sacks      SMALLINT     NOT NULL,
11    weighed_kg DECIMAL(8,2) NOT NULL,
12    PRIMARY KEY (store_id, stack_no),
13    FOREIGN KEY (store_id) REFERENCES store (store_id)
14  ) ENGINE = InnoDB;
15
16  INSERT INTO store (store_id, village) VALUES
17    ('NGE', 'Ngerengere'), ('VIK', 'Vikindu'), ('RUV', 'Ruvu');
18
19  INSERT INTO stack (store_id, stack_no, sacks, weighed_kg) VALUES
20    ('NGE', 1, 40, 3600.00),
21    ('VIK', 1, 22, 1980.00),
22    ('RUV', 1, 18, 1620.00);
23
24  INSERT IGNORE INTO stack (store_id, stack_no, sacks, weighed_kg)
25  VALUES ('NGE', 1, 31, 2790.00);
26
27  SELECT k.store_id, k.stack_no, k.sacks
28  FROM stack k
29  ORDER BY k.store_id, k.stack_no;
```

The server replies

```
+----------+----------+-------+
| store_id | stack_no | sacks |
+----------+----------+-------+
| NGE      |        1 |    40 |
| RUV      |        1 |    18 |
| VIK      |        1 |    22 |
+----------+----------+-------+
```

Program 4.3: *A weak entity keyed by its owner plus its partial key.*

Why the partial key is not enough alone: stack_no counts from one inside each store, so by itself it repeats across the cooperative and identifies nothing. Putting the owner key in front of it makes the pair unique, and the pair is the smallest set of columns that is unique, which is what makes it the key.

Reading the result: Stack number one appears three times, once under each store, and that is the acceptance. It appears once under NGE although it was written twice, and the surviving row still carries the forty sacks of the first write rather than the thirty one of the second, which is the refusal; IGNORE only lets the listing reach the SELECT.

One column doing two jobs: store_id is half of the primary key and the whole of the foreign key. That is what an identifying relationship becomes in a table: no stack can exist without a store, and moving a stack to another store changes what the stack is.

A partial key becomes a real key only in the company of the key it hangs from.

The Master Key: *Map a weak entity to a table keyed by the owner primary key plus the partial key, with that owner key declared as the foreign key as well.*

HOT Question 62

Two intake rows at Ruvu carry the same farmer, the same day and the same weight, and the store took both. Under the old pair of keys only one could have been entered; the number added to tidy the design let the second through.

Assess whether an associative entity should always be given a surrogate key of its own instead of the composite key of the two sides, and reach a judgement.

Model Answer

Keep the composite key of the two sides by default, and take a surrogate only where the same pairing may legitimately occur more than once or where other relations must reference the associative entity; wherever a surrogate is taken, the pairing it replaced has to be declared unique, or the rule that key was enforcing is simply gone.

What the composite key does for nothing: The two foreign keys identify the row and, in the same breath, forbid a second row for the same pairing. No column is added, no value has to be generated, and the columns a later join needs are the key itself, already indexed because it is the primary key.

The case for a surrogate: One short column is easier for another relation to reference than two, and any relation pointing at the associative entity then carries one column instead of two. It also survives a change on either side, since a revised member_id does not have to be chased through a key that never held it.

When the pair is not a key at all: If a member may send maize to the same store twice in a season, the pair member_id and store_id repeats and cannot identify a row. The honest first move is to extend the key with whatever tells the two apart, usually the date, and only after that to ask whether a surrogate would read better.

What a surrogate quietly permits: A generated number makes every row unique, including two rows recording the same pairing on the same day, so the duplicate the composite key refused is now accepted in silence. That is not an argument against surrogates; it is the reason the unique declaration on the pair has to go back in.

The cost of the composite key, stated fairly: It is wider, every referencing relation carries both columns, and every index built on it is larger. Those costs are real and they are small at the size a cooperative works at, becoming an argument only where several other relations must point at the associative rows.

The judgement: Default to the composite key, because it states the rule the design depends on and enforces it at no cost. Move to a surrogate when the pairing may legitimately repeat, or when other relations reference the row, and declare the pairing unique in either case.

The question is never which key looks neater but whether the same pairing is allowed to happen twice.

The Master Key: *Key an associative relation on its two sides unless the pairing may legitimately repeat or other relations must reference it, and where a surrogate is taken, declare the pairing unique anyway.*

HOT Question 63

The weighbridge at Ngerengere belongs to that store and to no other, and Ruvu has none at all. Whether those two facts should be one table or two is the sort of decision that looks like taste and turns out not to be.

Judge whether a one-to-one relationship should be kept as two tables or merged into one, and state the condition that decides it.

Model Answer

Merge the two into one relation where both sides must exist together and are read together, and keep them apart wherever either side is optional, putting the foreign key in the relation whose row always exists and declaring it unique.

The case for merging: One row instead of two means no join to write, no second key to maintain and no way for one half to exist without the other. Where every member has exactly one membership card and every card belongs to one member, card_no and issued_on are simply two more columns of member.

The case for two relations: Where participation is partial, merging fills a block of columns with nulls for every row that has no partner. Ruvu has no weighbridge, so a merged store would carry an empty capacity and an empty calibration date, and a null there cannot be told apart from a figure nobody has entered yet.

The reason that has nothing to do with nulls: The two halves may be governed differently. A relation only the committee may read can be granted separately, and a block of columns that is rarely wanted need not be fetched every time a store is looked up. Merging removes both of those choices for good.

Where the foreign key goes if they stay apart: Put it in the relation that cannot exist without the other, which is weighbridge, and declare it unique so the one-to-one is enforced rather than merely intended. Put store_id nowhere and the pairing is unrecorded; put the weighbridge key in store instead and most rows of store hold a null.

What merging costs when it turns out wrong: Two things that were separate become one row, created together, deleted together and locked together. Undoing that later means splitting a table other programs already read by name, which is far more disturbance than adding a join would have been in the first place.

The judgement: Merge when participation is total on both sides and the two halves are read and controlled together; otherwise keep two relations with a unique foreign key on the side that always exists. The membership card belongs inside member, and the weighbridge stays in its own relation because Ruvu has none.

One to one is the only cardinality whose mapping leaves a genuine choice, and the choice is settled by whether both rows always exist.

The Master Key: *Merge a one-to-one only where both sides are total and are read and controlled together, and otherwise keep two relations with a unique foreign key on the side that always exists.*

Functional dependency, and the anomalies it predicts

Everything in the next three sections rests on one idea, and it is the idea most often defined in a way that decides nothing. This section states a functional dependency as a test on rows, adds the closure and the three axioms that let a designer compute with dependencies rather than guess at them, and then separates full, partial and transitive dependency from one another. It closes on the multivalued dependency, which is a different animal and is regularly confused with an ordinary one, and on the three anomalies a badly placed dependency produces in a database that is actually in use.

BINDER Question 64

Two lines in the record book at Bungu carry the same route number. They carry two different amounts of milk, which nobody minds, and the same collector's name, which is not luck.

Define a functional dependency, and give the test that decides whether one holds.

Model Answer

A functional dependency, written X determines Y, holds in a relation when any two rows that agree on X must also agree on Y, and the test is exactly that: look for two rows agreeing on X and differing on Y, since one such pair settles it.

The two sides, named: X is the determinant, the side that decides, and Y is the dependent, the side that is decided. Either may be one attribute or several. In the round book, route_no is the determinant and collector_name the dependent.

The test speaks of rows, not of names: Take the rows two at a time and compare them on the determinant. Any pair agreeing there and disagreeing on the dependent refutes the dependency, and nothing else need be looked at. Students are often given a constraint describing how one attribute relates to another, which names no rows and so decides nothing.

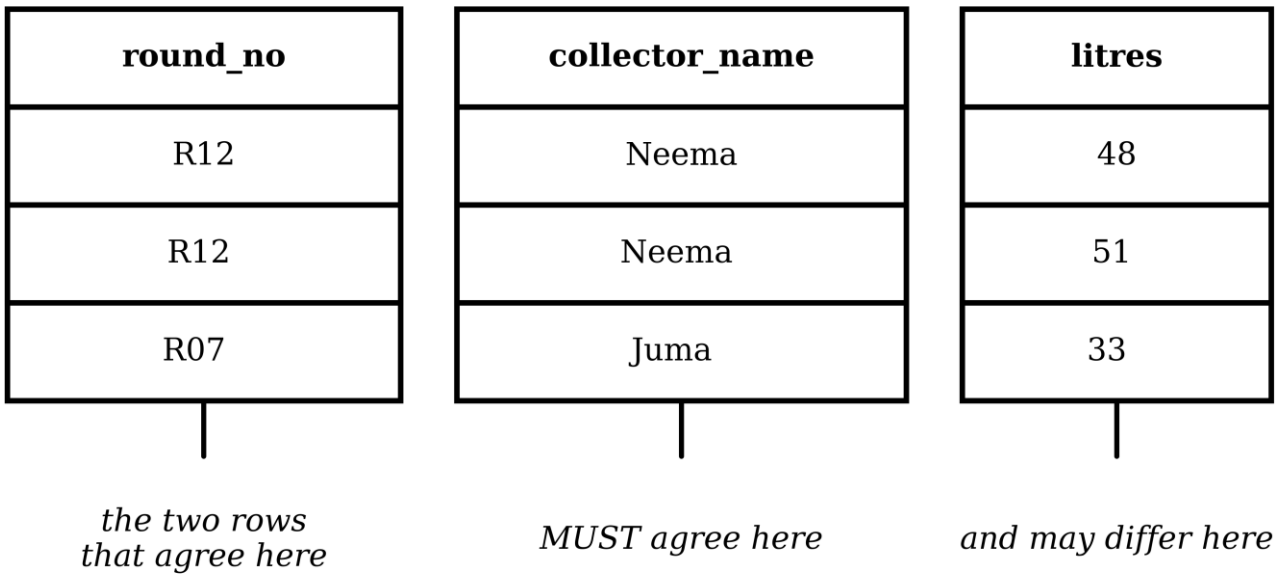


Figure 4.15: *The test for a functional dependency, stated on rows rather than on names.*

Reading the three columns: The two rows carrying R12 agree on the collector, which the dependency requires, and differ on the litres, which it permits. The pair that would break it carries R12 beside a second collector, and there is none.

Rows cannot prove one: The dependency speaks of every row the dairy may ever record, and the book holds three. Rows already written may agree with it for years without establishing it, since tomorrow can bring the pair that breaks it. Disproof takes a pair; proof takes a rule.

Where a dependency comes from: From the meaning of the data. The dairy puts one collector on a route, so route_no determines collector_name; nobody puts one quantity of milk on a route, so route_no does not determine litres.

A dependency is a promise about every row the relation will ever hold: one pair can break it, and no number of rows can make it.

The Master Key: *X determines Y when no two rows may agree on X and differ on Y; a single pair disproves it, and no set of rows proves it.*

BINDER Question 65

Is the pair of columns the clerk at Dumila has chosen enough to identify a row? She cannot settle it by looking, and there are five columns and three rules involved. A short procedure settles it without guessing.

Explain the closure of a set of attributes, state Armstrong's axioms, and say what closure is used for.

Model Answer

The closure of a set of attributes X, written X followed by a plus sign, is every attribute that X determines once the given dependencies have been applied as far as they will go, and Armstrong's axioms are the three rules that generate it: reflexivity, augmentation and transitivity.

How a closure is computed: Begin with the attributes of X themselves. Go through the dependencies, and wherever the whole of a determinant already sits inside the set, add everything it determines. Go round again, and stop when a complete pass adds nothing. What is left is the closure.

Reflexivity: If Y is contained in X then X determines Y. Such a dependency is called trivial: it holds in every relation whatever the rows say, and it is what puts X inside its own closure at the start.

Augmentation: If X determines Y, then X together with any Z determines Y together with that Z. Adding attributes to a determinant never destroys a dependency, which is why a superkey stays a superkey when a column is added to it.

Transitivity: If X determines Y and Y determines Z, then X determines Z. This is the rule that does the real work inside a closure, because it lets a set reach attributes that no single dependency mentions beside it.

What closure is for: It decides superkeys. A set is a superkey exactly when its closure is every attribute of the relation, and a candidate key when that is true of the set and false of every proper subset. It also decides whether a proposed dependency already follows from the ones stated, since X determines Y holds if Y lies inside the closure of X.

Axiom	What it says	One instance on the collection round
Reflexivity	If Y is part of X, then X determines Y	round_no with farm_id determines farm_id
Augmentation	If X determines Y, then X with Z determines Y with Z	round_no determines collector_name, so round_no with farm_id determines collector_name
Transitivity	If X determines Y and Y determines Z, then X determines Z	round_no determines collector_name and collector_name determines collector_phone, so round_no determines collector_phone

Table 4.8: *Armstrong's three axioms, and one instance of each on the collection round.*

Closure turns a question about keys into a short piece of arithmetic on the dependencies, worked on paper and settled the same way twice.

The Master Key: *The closure of X is everything X determines; X is a superkey when that closure is the whole relation, and a candidate key when no proper subset of X reaches as far.*

BINDER Question 66

Milk in one can at Dumila is measured against a round and a farm together. The collector's name needs only the round, and the collector's phone needs only the name. Three shapes of dependence, and two of them are faults.

Distinguish full, partial and transitive dependency, and say which later stage of normalisation removes each.

Model Answer

A dependency is full when every attribute of its determinant is needed and no part of it would do, partial when a non-prime attribute depends on part of a candidate key rather than on the whole of it, and transitive when the key reaches an attribute through a middle determinant that is not itself a candidate key.

Full dependency: In collection, round_no and farm_id together determine litres, and neither alone does: round_no alone is a whole morning of gates, farm_id alone is that one gate on every morning. No part of the determinant can be dropped, so the dependency is full, and a full dependency is not a fault.

Partial dependency: round_no on its own determines collector_name, and round_no is half of the candidate key, so a non-prime attribute hangs off part of a key. This cannot arise at all unless the key is composite, since a key of one attribute has no proper parts. Second normal form removes it, and the next section states that in full.

Transitive dependency: The key determines collector_name, collector_name determines collector_phone, and collector_name is not itself a candidate key, since one collector may run several rounds. So the key reaches the telephone number through a middle attribute rather than directly. Third normal form removes it, and the section after next states that in full.

The clause that is usually dropped: That middle determinant must not itself be a candidate key. A common error is to define a transitive dependency as any chain of two dependencies, but a candidate key determines every attribute of its relation, so with the clause struck out every attribute is reachable by a chain and every relation reads as faulty.

Kind	What sits on the left of the arrow	What it costs
Full	The whole of a candidate key, no part of which can be dropped	Nothing; this is what a key is for
Partial	Part of a composite candidate key	The same fact written once for every row that part appears on
Transitive	A non-key attribute that is not a candidate key	The same fact written once for every row that middle attribute appears on

Table 4.9: *The three shapes of dependence, told apart by what stands on the left of the arrow.*

Naming the shape is the whole of the work, because each shape has one cure and only one.

The Master Key: *Full needs the whole determinant, partial hangs off part of a composite key, and transitive reaches through a middle determinant that is not a candidate key.*

REAL Question 67

On the back of a delivery note at Bungu somebody wrote four rules, each claiming that one column settles another. Six rows in the ledger were enough to knock two of them out, one pair of rows for each.

Decide which of the four proposed dependencies these rows allow and which they refute, and for each one refuted name the two rows that break it.

collection(round_no, farm_id, collector_name, village, litres) holds six rows, given here as round, farm, collector, village and litres.

Row 1: R12, F04, Neema, Bungu, 48.00

Row 2: R12, F09, Neema, Dumila, 51.50

Row 3: R07, F04, Juma, Bungu, 33.00

Row 4: R07, F11, Juma, Berega, 40.00

Row 5: R19, F09, Neema, Dumila, 47.00

Row 6: R19, F11, Neema, Berega, 36.50

A. round_no determines collector_name

B. farm_id determines village

C. round_no determines litres

D. collector_name determines round_no

Model Answer

A and B stand and C and D fall: C is broken by rows 1 and 2, which agree on R12 and carry 48.00 against 51.50, and D by rows 1 and 5, which agree on Neema and carry R12 against R19.

The working, done once per proposal: Gather the rows into groups that agree on the determinant, then read the dependent column inside each group. A group holding two different values refutes the proposal, and the first such pair is the whole of the answer.

A stands: R12 appears on rows 1 and 2 with Neema, R07 on rows 3 and 4 with Juma, and R19 on rows 5 and 6 with Neema. No group disagrees with itself, and behind the rows lies the dairy rule that one collector runs one round.

B stands: F04 appears on rows 1 and 3, both Bungu; F09 on rows 2 and 5, both Dumila; F11 on rows 4 and 6, both Berega. A farm does not move between villages, so a rule stands behind these rows as well.

C falls, on rows 1 and 2: Both carry R12, one carries 48.00 and the other 51.50. That is precisely the pattern the test hunts for, and no further row need be examined. A round calls at several gates and takes a different quantity at each, so the proposal was never going to survive.

D falls, on rows 1 and 5: Both carry Neema, row 1 against R12 and row 5 against R19. One collector may be given more than one round, so a name cannot identify a round. Note that A survives and D does not, although they are the same two attributes with the arrow turned round.

What standing does not amount to: A and B are not contradicted by these rows, and that is all. Put a seventh row on the page giving R12 a second collector and A is gone, which is what it means to say that rows refute a dependency and never establish one.

Proposal	Verdict	The rows that decide it
round_no determines collector_name	Stands	No group disagrees; R12, R07 and R19 each carry one name
farm_id determines village	Stands	No group disagrees; F04, F09 and F11 each carry one village
round_no determines litres	Refuted	Rows 1 and 2: R12 with 48.00 and with 51.50
collector_name determines round_no	Refuted	Rows 1 and 5: Neema with R12 and with R19

Table 4.10: *The four proposals, and the pair of rows that settles each of the two refused.*

Two of the four went down to one pair of rows each, and the two left standing are the ones the dairy's own working rules put there.

The Master Key: *Group the rows by the determinant and read inside each group: one group holding two different values of the dependent refutes the proposal outright.*

REAL Question 68

A note pinned inside the collection shed at Berega lists two sets of columns, both of which identify a row without fail. Only one of them is the smallest set that can, and nothing on the note says which.

Compute the closure of each of the two attribute sets, and use the results to say which is a superkey and which is a candidate key.

collection(round_no, farm_id, collector_name, village, litres)

F1: round_no determines collector_name

F2: farm_id determines village

F3: round_no with farm_id determines litres

Set X is round_no with farm_id.

Set Y is round_no with farm_id and collector_name.

Model Answer

Both closures come to all five attributes, so X and Y are both superkeys, and only X is a candidate key: taking collector_name out of Y leaves a set that still reaches everything, while taking either attribute out of X leaves a set that reaches only two.

Closing X: Start with round_no and farm_id. F1 adds collector_name, F2 adds village, and F3 adds litres, since both of its determinants are present. A further pass adds nothing, so the closure of X is all five attributes.

Closing Y: Start with round_no, farm_id and collector_name. F1 offers collector_name, which is already there; F2 adds village and F3 adds litres. The closure of Y is the same five attributes, which is the first sign that collector_name is not earning its place in the set.

Both are superkeys: A set of attributes is a superkey when its closure is every attribute of the relation. Each closure here is all five, so each set determines the whole row and each identifies a row uniquely.

Minimality is what separates them: A candidate key is a superkey with no superkey inside it. Remove collector_name from Y and what remains is X, whose closure is still everything, so Y contains a smaller superkey and cannot be a candidate key.

Testing X for minimality: Closing round_no alone gives round_no with collector_name, and closing farm_id alone gives farm_id with village. Neither reaches the whole relation, so neither proper subset of X is a superkey, and X is therefore minimal.

The verdict, in the settled words: X is a candidate key and, being the only one here, the primary key. Y is a superkey and nothing more: it identifies a row while carrying an attribute the identification does not need, and choosing it as the key would put collector_name into every relation that refers to a collection.

Set closed	What the closure reaches	Superkey	Candidate key
round_no	round_no, collector_name	No	No
farm_id	farm_id, village	No	No
X: round_no, farm_id	All five attributes	Yes	Yes
Y: round_no, farm_id, collector_name	All five attributes	Yes	No

Table 4.11: *Four closures, and what each one settles about the set that was closed.*

A superkey and a candidate key are indistinguishable until every proper subset has been closed as well, which is the part of the working most often skipped.

The Master Key: *Close the set: if it reaches every attribute it is a superkey, and if no proper subset of it does that too, it is a candidate key.*

REAL Question 69

Taped to the dashboard of the collection lorry at Dumila is a printed sheet setting out three statements about dependency. Exactly one is correct. The other two are the mistakes this topic collects, and both sound reasonable read aloud.

Identify which of the three statements is correct, say precisely what is wrong with each of the other two, and correct them.

A. If two rows agree on the determinant they may still differ on the dependent attribute, provided the difference is rare.

B. A transitive dependency is any chain in which one attribute determines a second and the second determines a third.

C. A multivalued dependency holds when one key value is associated with a set of values that is independent of the other attributes.

Model Answer

Statement C is correct; A is wrong because a dependency admits no exceptions whatever, however rare, and B is wrong because it leaves out the clause saying that the middle determinant must not itself be a candidate key.

Why C is right: It puts the weight on two things at once: one value of the determinant carries a whole set of values, and that set stays the same whatever else the row holds. A collector has a set of villages worked and a set of cans signed for, and neither set is decided by the other.

What is wrong with A: Rareness is not a category this idea has. Two rows agreeing on the determinant and differing on the dependent refute the dependency outright, so a rule true of ninety-nine rows in a hundred is not a dependency at all. Declared as one, it makes the database refuse the hundredth row, which is a true fact about the dairy.

A, corrected: If two rows agree on the determinant they must agree on the dependent, in every row now stored and every row that may ever be stored. Where genuine exceptions exist, the attribute belongs in a relation of its own rather than in a dependency with a hole in it.

What is wrong with B: Take collector(collector_id, phone, collector_name), where the telephone number is unique and so an alternate key. Then collector_id determines phone and phone determines collector_name, which fits B word for word, and yet nothing at all is wrong with the relation.

B, corrected: A transitive dependency is a chain in which the middle determinant is not a candidate key of the relation and the attribute reached is not part of one. Strike that clause and every relation fails, since a candidate key determines every attribute and a chain can be drawn through anything.

What splitting on B would cost: Following B, the collector relation is split into two joined on a unique column, one row facing one row, and every read of a name now needs a join. Nothing was repeated before the split, so nothing was saved by it.

A and B are the two ways this idea is usually softened, one by allowing exceptions to a rule that has none and one by dropping the clause that keeps sound relations whole.

The Master Key: *A dependency admits no exceptions, and a chain is transitive only where the middle determinant is not itself a candidate key.*

REAL Question 70

Nothing on the table was false, and the conclusion drawn from it was. One round number stood against several rows at Bungu, which was read as the round serving many chilling points. It serves one.

Show that the dependency offered here is functional rather than multivalued, and say what a genuine multivalued dependency would look like on the same relation.

round_stop(round_no, farm_id, chilling_point), offered as showing that round_no multidetermines chilling_point.

R12, F04, Bungu

R12, F09, Bungu

R07, F04, Berega

R07, F11, Berega

The claim rests on R12 and R07 each appearing on more than one row.

Model Answer

round_no determines chilling_point functionally, because R12 carries Bungu on both of its rows and R07 carries Berega on both of its, so no round appears with two chilling points; the rows repeat because farm_id varies, not because a round has a set of chilling points.

The test, applied to the offered claim: Group the rows by round_no and read the chilling_point column inside each group. R12 gives Bungu twice and R07 gives Berega twice. No group holds two different values, so the dependency is functional and there is nothing multivalued about it.

What was mistaken for a set: A determinant appearing on several rows is not a determinant with several values. R12 appears twice because the round calls at two farms, and the chilling point is copied onto both rows while remaining one value throughout.

The attribute that is genuinely multidetermined: round_no against farm_id is another matter, since R12 carries F04 on one row and F09 on another. Two different values against one determinant, so round_no does not determine farm_id and a set really is involved here.

What a genuine one would look like: Add run_day and let R12 run on Monday and on Thursday while calling at F04 and F09. Every combination must then appear, because the days are fixed by the round alone and so are the farms, and two facts of two values each force four rows.

Independence is the second half of the test: Those rows are forced only because the days a round runs have nothing to do with the farms it calls at. Were F04 reachable on Mondays only, the two lists would be tied together, the absent combinations would mean something, and no multivalued dependency would be there to find.

The order to ask the questions in: First ask whether the determinant ever appears beside two different values of the attribute. If it never does, the dependency is functional whatever the number of rows suggests. Only when

it does is the second question worth asking, which is whether that set of values holds steady as the other attributes change.

round_no	farm_id	run_day
R12	F04	Monday
R12	F04	Thursday
R12	F09	Monday
R12	F09	Thursday

Table 4.12: *A genuine multivalued dependency, where two independent lists force every combination onto the page.*

The offered table shows one value copied across rows, which is what a functional dependency looks like whenever the key has more than one attribute in it.

The Master Key: *A determinant on many rows is not a determinant with many values: ask first whether it ever appears with two different values, and only then whether the set stays independent.*

BUILD Question 71

Juma ran one round out of Berega and no other. The clerk removed that round from the day's records, and with it went the only place the dairy had ever written down his telephone number.

Create the single-table collection relation, insert one morning of rows, then delete the only round a collector ran and show with a SELECT that the collector's telephone number has gone with it.

collection holds round_no, farm_id, collector_name, collector_phone and litres, keyed by round_no with farm_id.

Insert two rounds: R12 run by Neema and R07 run by Juma, each calling at two farms.

Delete round R07, which is the only round Juma ever ran.

Then list the collectors the table can still name, and count the rows left for Juma.

Model Answer

Deleting R07 removes both of its rows, and because Juma's telephone number was stored only inside those rows, the number leaves the database with them: the first query afterwards can name Neema and nobody else, and the count for Juma comes back zero.

```
 1  CREATE TABLE collection (
 2    round_no        CHAR(3)      NOT NULL,
 3    farm_id         CHAR(3)      NOT NULL,
 4    collector_name  VARCHAR(30)  NOT NULL,
 5    collector_phone CHAR(12)     NOT NULL,
 6    litres          DECIMAL(6,2) NOT NULL,
 7    PRIMARY KEY (round_no, farm_id)
 8  ) ENGINE = InnoDB;
 9
10  INSERT INTO collection
11    (round_no, farm_id, collector_name, collector_phone, litres)
12  VALUES
13    ('R12', 'F04', 'Neema Kiluwa', '0754-118204', 48.00),
14    ('R12', 'F09', 'Neema Kiluwa', '0754-118204', 51.50),
15    ('R07', 'F04', 'Juma Mzee',    '0762-330915', 33.00),
16    ('R07', 'F11', 'Juma Mzee',    '0762-330915', 40.00);
17
18  DELETE FROM collection WHERE round_no = 'R07';
19
20  SELECT DISTINCT c.collector_name, c.collector_phone
21  FROM collection c
22  ORDER BY c.collector_name;
23
24  SELECT COUNT(*) AS rows_naming_juma
25  FROM collection c
26  WHERE c.collector_name = 'Juma Mzee';
```

The server replies

```
+----------------+-----------------+
| collector_name | collector_phone |
+----------------+-----------------+
| Neema Kiluwa   | 0754-118204     |
+----------------+-----------------+
+------------------+
| rows_naming_juma |
+------------------+
|                0 |
+------------------+
```

Program 4.4: *Deleting the last round a collector ran, and the telephone number that goes with it.*

Why the number was in there at all: collector_phone describes a collector, not a can of milk at a farm gate, yet it sits in a relation keyed by round_no with farm_id. A fact about a collector is therefore written once per row of milk and survives only as long as some row of milk survives.

Reading the two results: The table names one collector where it named two a statement earlier, and the count comes back zero. Nothing was refused, no constraint was broken, and the server did exactly what it was asked; the dairy simply lost a telephone number it never meant to store there.

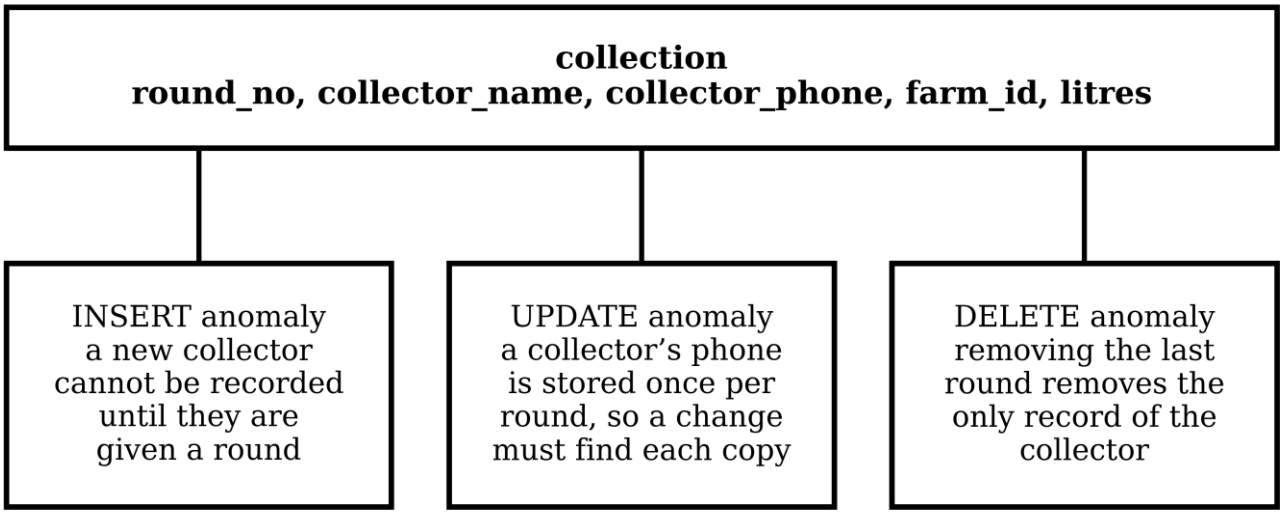


Figure 4.16: *One badly designed relation, and the three separate ways it fails.*

The other two anomalies share the cause: A collector taken on today cannot be recorded until somebody gives them a round, because there is no row to hold them, and changing a number means finding every row that repeats it. Insertion, update and deletion all fail on one fact kept in the wrong relation.

The relation is not broken and the statement is not wrong; the design put a fact about a collector inside a row about milk.

The Master Key: *A fact about one thing stored inside a relation about another disappears when the second one does, and that is the deletion anomaly.*

HOT Question 72

The morning run at Berega now takes three lookups where it used to take one. One relation was split into three because every column seemed to depend on something else, and two of those splits were right.

Assess whether every functional dependency a designer can find should be removed by normalisation, and reach a judgement.

Model Answer

No: normalisation removes only those dependencies whose determinant is not a candidate key of the relation they sit in, and a dependency running out of a candidate key is the one thing that must be kept, because it is what the key is for.

The case for removing every one: Each dependency stores a fact somewhere, and where the fact is not about the row it sits in, it is written more times than it is true. Every such repetition is one of the three anomalies waiting, so removing all of them leaves none behind.

The dependency that must never be removed: A candidate key determines every attribute of its relation, and that is a functional dependency like any other. Remove it and the key goes with it, leaving rows that cannot be told apart. Every relation in a finished design still carries dependencies; the sound ones simply run out of a key.

Trivial dependencies cannot be removed at all: Reflexivity gives round_no with farm_id determines farm_id, and no decomposition touches it, since it says nothing about how anything is stored. Counting these among the faults produces a list nobody can act on and hides the two or three that matter.

The dangerous case is a dependency read off the rows: Rows refute a dependency and never establish one. A designer who reads farm_id determines village off six rows, where the dairy in fact allows a farm to move between villages, has split a relation on a false rule, and the design can no longer record what happens.

A multivalued dependency is not the same machinery: It is not removed by the tests that deal with a partial or a transitive dependency, and treating every dependency alike either misses it entirely or splits a relation that has nothing wrong with it. Deciding the kind comes before deciding the cure.

The condition that decides it: Remove a dependency exactly where its determinant is not a candidate key and the attribute it determines is not part of one; keep every other dependency. That single condition covers both the partial and the transitive case, since neither of them has a candidate key standing on the left.

The judgement: Normalise on dependencies drawn from the meaning of the data rather than from a sample of rows, remove those whose determinant is not a candidate key, keep those that run out of one, and leave the trivial ones alone. A design that has removed every dependency has removed its own keys.

The number of dependencies removed is no measure of a design; the shape of the ones left standing is.

The Master Key: *Remove a dependency only where its determinant is not a candidate key, since the ones running out of a key are the design, not a fault in it.*

Unnormalised form to first and second normal form

An unnormalised relation is not one that has been written carelessly; it is one whose cells hold more than a single value, and the first two normal forms are what put it into a shape a database can work on. This section defines unnormalised form as a state rather than a process, states first normal form and what atomic means, and states second normal form together with the partial dependency it removes. It then takes one tailoring order book from a single wide sheet to a set of keyed relations, and shows the trap that ruins most attempts at the first step: two independent multivalued facts crossed together into rows nobody ever recorded.

BINDER Question 73

The shop at Msowero can tell you what any customer ordered, and cannot tell you how many shirts it made last month without reading every line. Nothing has been recorded wrongly. The trouble is the shape the records are in.

Explain what an unnormalised relation is, and say what normalisation is for.

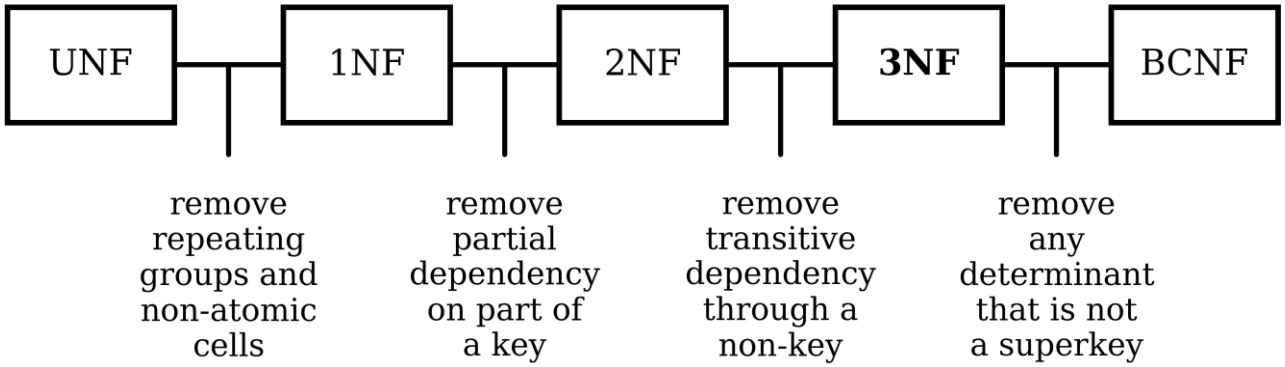
Model Answer

An unnormalised relation is one that holds a repeating group or a cell carrying more than one value, and normalisation is the process that replaces such a relation with a set of relations holding the same facts and none of the insertion, update and deletion anomalies.

Unnormalised form is a state: It names the condition a relation is in, exactly as first normal form and second normal form name conditions. A common error is to define it as a systematic process governed by clear rules; that sentence describes normalisation, and leaves nothing at all to name the state the work starts from.

What puts a relation in that state: Two things do. A repeating group is a set of attributes occurring several times for one row, such as the garments on a single order. A non-atomic cell holds more than one value at once, such as three garment names written into one column with commas between them.

Normalisation is the process: It is a sequence of decompositions, each removing one named kind of dependency and each replacing one relation with two or more that can be joined back to it. It is carried out on paper at design time, and what it produces is a schema rather than data.



3NF is where a candidate is expected to stop; BCNF is where a designer stops

Figure 4.17: *The normalisation ladder, labelled with what each step removes.*

The purpose is the anomalies: Space is not the reason. Holding a garment name once rather than beside every order line does save a little room, and that is a by-product. The purpose is that a fact stored in one place cannot be corrected in one copy and left wrong in another, and that a garment can exist before anybody orders it.

Unnormalised form says where a relation is; normalisation is the work that moves it, and the anomalies are what the work is for.

The Master Key: *Unnormalised form is a state, a relation holding repeating groups or non-atomic cells, and normalisation is the process that removes the anomalies they cause.*

BINDER Question 74

One cell in the Melela order book holds shirt, trousers and kanzu, with commas between them. The column is headed garment, and the shop has quietly stored three facts in a space the design promised would hold one.

State first normal form, explain what atomic means, and say what a relation must have before it can be in it.

Model Answer

A relation is in first normal form when every cell holds exactly one value drawn from the domain of its attribute, no attribute is a repeating group, and the relation has a primary key; atomic means indivisible for the use the database is put to, not indivisible in principle.

The rule in full: Three conditions hold together. Every cell carries a single value, no set of attributes repeats within a row, and one candidate key has been chosen as the primary key. Miss the third and the flattened rows can be duplicated with nothing in the design to forbid it.

Atomic is judged against the use: A fitting date is one value in a shop that never asks about the month by itself. A customer name held as one string is atomic in a shop that only prints it, and not atomic in a shop that orders its work by family name, because the design must then reach inside the value.

What a repeating group looks like: It is a set of attributes recurring for one row, such as a garment code and a quantity appearing three times on one order. It shows itself as columns named garment_1, garment_2 and garment_3, or as one column holding all of them, and the two are the same fault.

Why the key belongs in the rule: Flattening a repeating group copies the outer values onto every new row, so rows come to resemble one another and something must still tell them apart. In the order book that is the order number together with the garment code, and until such a key is named the result is a list rather than a relation.

What is in the cell	Why it fails	What first normal form asks for
shirt, trousers, kanzu in one column	one cell, three values	one row for each garment, one name in each cell
garment_1, garment_2, garment_3 across one row	the same attribute repeated along the row	one row for each garment, with the position dropped
a full name in one column, in a shop that orders its work by family name	the design has to reach inside the value	one column for each part the design uses

Table 4.13: *Three cells that break first normal form, and what the rule asks for instead.*

First normal form is settled by the shape of the rows and the presence of a key, and by nothing else.

The Master Key: *In first normal form every cell holds one value, no attribute repeats within a row, and the relation has a primary key, with atomic judged against the use the data is put to.*

BINDER Question 75

A tailor at Doma renamed one garment code in the order book and found the old name still written beside every line that used it. One half of the key was enough to know the name, and the book had never noticed.

State second normal form and explain what a partial dependency is.

Model Answer

A relation is in second normal form when it is in first normal form and no non-prime attribute depends on part of a candidate key, and a partial dependency is exactly that situation: a non-prime attribute determined by some but not all of a key.

Which way round the rule goes: The thing constrained is the non-prime attribute, meaning one that appears in no candidate key. Students are often taught it backwards, as the primary key being the non-prime attribute, or as the key depending on the attribute. Read the rule as a prohibition on where a non-prime attribute may take its value from.

Full against partial: In the order book, quantity is determined by order_no and garment_code together and by neither alone, so that dependency is full. garment_name is determined by garment_code by itself, and garment_code is one half of the key, so that dependency is partial.

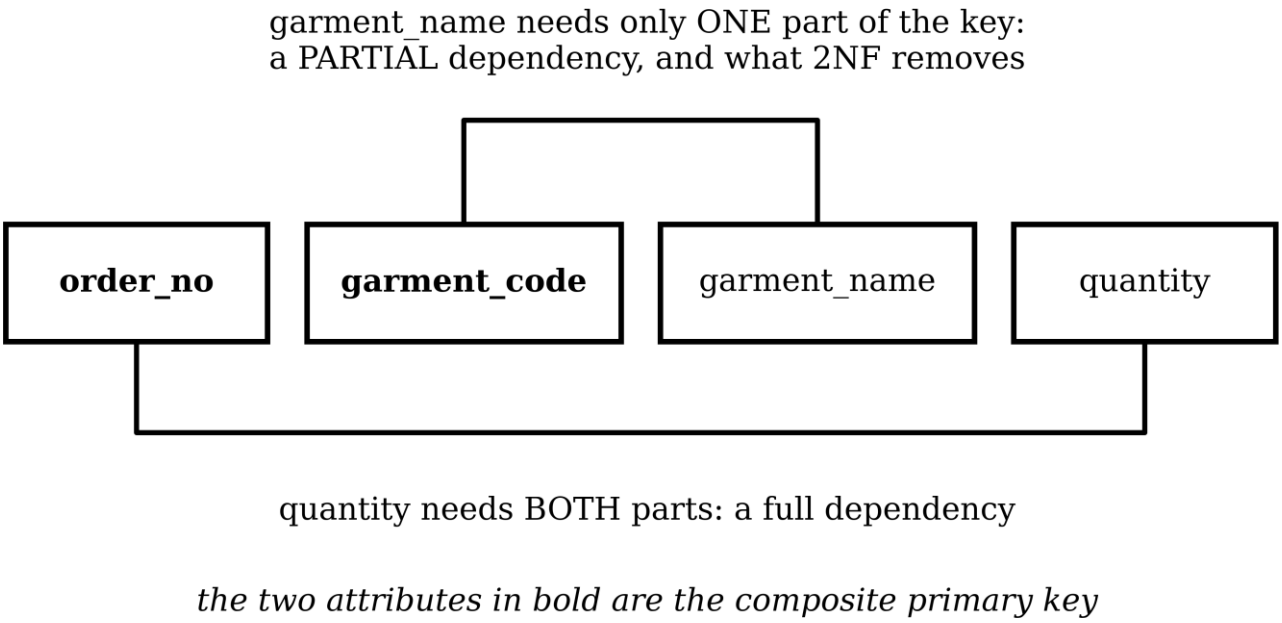


Figure 4.18: *A full dependency and a partial dependency on the same composite key.*

A partial dependency needs a composite key: Part of a key means a proper subset of it. A key of one attribute has no proper subset other than the empty set, so nothing can depend on part of it. A relation in first normal form whose primary key is a single attribute is therefore already in second normal form, and no checking is needed.

What the rule removes: A partially dependent attribute is written once for every row carrying its half of the key, so the same garment name runs down the relation. Moving it, together with the determinant it depends on, into a relation of its own leaves that fact stored exactly once.

Look for a composite key before anything else, because where there is none, second normal form is already satisfied and the work is finished.

The Master Key: *A relation is in second normal form when it is in first normal form and no non-prime attribute depends on part of a candidate key, which can only happen where the key is composite.*

REAL Question 76

Msowero's order book fits on one sheet ruled into four columns, and the last column has to hold every garment on the order. Six garments across three orders live in there, written end to end, and the sheet is not wrong, only unusable.

Take the order book below from unnormalised form to first normal form, showing the rows at each step, and name the key of the result.

order_slip(order_no, customer_name, village, {garment_code, garment_name, quantity})

T0041, Neema Chikawe, Msowero: G7 shirt 3, G9 kanzu 1

T0042, Juma Ligale, Melela: G7 shirt 5

T0043, Asha Mwenda, Doma: G4 dress 2, G7 shirt 1, G9 kanzu 2

The braces mark the group that repeats within one order.

Model Answer

The braced group is a repeating group, which is what makes the sheet unnormalised, and writing one row for each garment on each order, with the order level values copied onto every row, gives six rows in first normal form keyed by order_no together with garment_code.

Why the sheet is unnormalised: The last cell of each line holds a whole list, and the length of that list changes from one order to the next. Nothing on the sheet is untrue, and no query can be written against it, because a column with no fixed number of values in it has nothing an expression can address.

Naming the group that repeats: garment_code, garment_name and quantity travel together, once for each garment on the order. That set of three is the repeating group, and it is the group rather than any single column that has to be lifted out of the sheet.

One row for each member of the group: Take the first order apart into two rows and the third into three, copying order_no, customer_name and village onto each row as you go. Three lines become six rows, every cell holds one value, and no set of attributes repeats along a row.

order_no	customer_name	village	garment_code	garment_name	quantity
T0041	Neema Chikawe	Msowero	G7	shirt	3
T0041	Neema Chikawe	Msowero	G9	kanzu	1
T0042	Juma Ligale	Melela	G7	shirt	5
T0043	Asha Mwenda	Doma	G4	dress	2
T0043	Asha Mwenda	Doma	G7	shirt	1
T0043	Asha Mwenda	Doma	G9	kanzu	2

Table 4.14: *The order book in first normal form, one row for each garment on each order.*

The key of the result: order_no repeats down the relation and garment_code repeats down it, so neither identifies a row. The pair does, and no smaller set does, so order_no together with garment_code is the primary key. The shop never writes one garment twice on an order; it raises the quantity, which is what keeps the pair unique.

Checking the form by eye: First normal form cannot be decided from the dependencies, because atomicity is a property of what stands in a cell. The check is to read the rows and look for a cell holding two values or a set of attributes repeating, which is why the rows above are printed rather than described.

What is still wrong with it: The result is in first normal form and it is not finished. customer_name and village depend on order_no alone, and garment_name depends on garment_code alone, so each of them depends on half of the key. A common error is to stop here and call the design normalised; second normal form is what deals with them.

Flattening a repeating group is the whole of first normal form, and it usually creates the composite key that the next step then has to examine.

The Master Key: *Reach first normal form by writing one row for each member of the repeating group, copying the outer values onto every row, and then naming the key of the result.*

REAL Question 77

An order at Melela is for two garments and has been paid in three instalments. Written out as every combination, it fills six lines, and not one of those six records anything the customer and the tailor ever agreed.

The order book below carries two independent multivalued facts about the same order. Show what goes wrong when it is brought to first normal form by writing out every combination of the two, name the rows that are invented and the rows that are lost, and give the correct treatment.

order_note(order_no, village, {garment_code}, {paid_on})

T0051, Melela: garments G3 and G7; instalments paid on 2026-05-04, 2026-05-19 and 2026-06-02

T0052, Msowero: garment G9, and no instalment recorded yet

Which garment is on an order has nothing to do with which instalment paid for it.

Model Answer

Combining the two groups produces six rows for an order about which only five facts were ever recorded, so every one of the six asserts a pairing of a garment with a payment that the shop never made, while the order carrying a garment and no instalment produces no row at all and disappears from the relation.

Two facts that never meet: The garments on the order and the instalments paid against it are separate lists. Knowing that the order includes a kanzu tells you nothing about which payment covered it, and the shop keeps no such link, because an instalment is paid against the order and not against one garment.

What every combination produces: Two garments and three instalments give six rows. The book recorded two garments and three payments, five facts in all, so six rows now stand where five facts were, and each of the six pairs one garment with one date.

order_no	village	garment_code	paid_on
T0051	Melela	G3	2026-05-04
T0051	Melela	G3	2026-05-19
T0051	Melela	G3	2026-06-02
T0051	Melela	G7	2026-05-04
T0051	Melela	G7	2026-05-19
T0051	Melela	G7	2026-06-02

Table 4.15: *Every combination of two garments and three instalments, and not one row that the order book recorded.*

The rows that are invented: The row holding G3 beside 2026-05-04 states that the payment made on that date was for that garment. Nothing in the order book says so, and the same goes for the other five. All six pairings are the work of the method and not of the shop.

The rows that are lost: T0052 has one garment and no instalment. Every combination of one thing with none is none, so that order leaves no row behind at all, and the fact that a kanzu is being made for that customer has gone out of the relation entirely.

The correct treatment: Two independent multivalued facts need two relations. order_garment holds order_no with garment_code and order_instalment holds order_no with paid_on, each keyed by both of its attributes, while order holds order_no with village. T0051 then keeps two rows and three rows, T0052 keeps its garment, and no pairing is claimed anywhere.

The rule worth carrying away: One relation for each independent multivalued fact. Where both lists are wanted at once they are asked for together rather than stored together, and neither relation on its own claims a pairing that the shop never made.

The combined table looks like first normal form, because every cell holds one value, and it is worse than the sheet it replaced, since it now states things nobody recorded.

The Master Key: *Two independent multivalued attributes need one relation each, because combining them invents every pairing and drops any order that is missing one of the two lists.*

REAL Question 78

Pinned inside the cupboard door at Doma, where the order book is kept, are three statements about normal forms in a careful hand. The apprentice has copied all three into her own notes, and exactly one of them is correct.

Exactly one of the three statements below is true. Name it, then say what is wrong with each of the other two and put it right.

A. A relation in first normal form with a single-attribute primary key may still contain a partial dependency.

B. Unnormalised form is the process of removing repeating groups from a relation.

C. A relation is in second normal form when it is in first normal form and no non-prime attribute depends on only part of a candidate key.

Model Answer

C is the correct statement; A describes something that cannot happen, because a partial dependency needs a proper part of a key and a single attribute has no proper parts, and B hands the definition of a process to a term that names a state.

Why C is right: It carries both halves of the rule: the relation must already be in first normal form, and no non-prime attribute may depend on only part of a candidate key. It says candidate key rather than primary key, which matters where a relation has more than one, and it constrains the non-prime attribute, which is the half most often reversed.

Why A is impossible: Part of a key means a proper subset of it. The only proper subset of a single attribute is the empty set, and a dependency from the empty set would say that the attribute never varies. No relation whose primary key is one attribute can hold a partial dependency, whatever else may be wrong with it.

What A is worth knowing for: It settles a question quickly. A relation in first normal form whose key is a single attribute is in second normal form already, so the only relations worth examining for a partial dependency are those with composite keys.

A corrected: A relation in first normal form with a composite primary key may still contain a partial dependency, and one with a single-attribute primary key cannot.

Why B is wrong: Unnormalised form is a state, not a process. Removing repeating groups is the first step of normalisation and its result is first normal form. B gives the definition of the process to the state, and then has no term left for the condition the relation was in before any work began.

B corrected: Unnormalised form is the state of a relation that holds a repeating group or a cell with more than one value in it, and removing them is the first step of normalisation.

Both of the wrong statements are true sentences attached to the wrong subject, which is why they survive being read aloud.

The Master Key: *Second normal form forbids a non-prime attribute from depending on part of a candidate key, unnormalised form names a state and not a process, and a single-attribute key cannot carry a partial dependency.*

REAL Question 79

Rename one garment at Msowero and the order book fights back. The new name must be typed once for every line that already carried the old one. Any line the tailor misses keeps the old name for good, and the book now disagrees with itself.

Identify the partial dependencies in the relation below, decompose it into second normal form, and name every key in the result.

order_line(order_no, garment_code, garment_name, unit_charge, quantity)

The primary key is order_no together with garment_code.

quantity is how many of that garment the order asks for, and unit_charge is what the shop charges for one of them.

The customer details have already been moved into a relation of their own.

Model Answer

garment_code is one half of the key and it determines garment_name and unit_charge on its own, so both of those dependencies are partial; taking the two attributes out with their determinant gives garment, keyed by garment_code, and leaves order_line keyed by order_no together with garment_code, in which garment_code is now a foreign key.

Confirming the key first: order_no repeats and garment_code repeats, so neither is a key alone. The pair determines every other attribute and no proper part of it does, so the pair is the only candidate key, and garment_name, unit_charge and quantity are all non-prime.

order_no	garment_code	garment_name	unit_charge	quantity
T0041	G7	shirt	12000.00	3
T0041	G9	kanzu	25000.00	1
T0042	G7	shirt	12000.00	5
T0043	G4	dress	30000.00	2
T0043	G7	shirt	12000.00	1
T0043	G9	kanzu	25000.00	2

Table 4.16: *order_line in first normal form, with the garment name and the charge repeated on every line that uses the code.*

The test applied to each dependency: quantity is determined by the whole pair and by no part of it, so it is fully dependent. garment_name and unit_charge are determined by garment_code alone, which is a proper part of the key, and both of them are non-prime, so both dependencies are partial and the relation is not in second normal form.

The decomposition: Move the partially dependent attributes into a relation keyed by the determinant they depend on, and leave everything else where it stands. garment takes garment_code, garment_name and unit_charge, while order_line keeps order_no, garment_code and quantity. Nothing is discarded and no attribute is invented.

garment_code	garment_name	unit_charge
G4	dress	30000.00
G7	shirt	12000.00
G9	kanzu	25000.00

Table 4.17: *garment, keyed by garment_code, holding each name and charge exactly once.*

order_no	garment_code	quantity
T0041	G7	3
T0041	G9	1
T0042	G7	5
T0043	G4	2
T0043	G7	1
T0043	G9	2

Table 4.18: *order_line after the decomposition, keyed by order_no together with garment_code.*

Every key in the result: In garment the primary key is garment_code, and it is the only candidate key there. In order_line the primary key is still order_no together with garment_code, neither part being a key by itself, and garment_code in order_line is a foreign key referencing garment.

Why the result is in second normal form: order_line has one non-prime attribute, quantity, and it depends on the whole of the key. garment has a key of one attribute, so no partial dependency is possible in it at all. Both relations pass, and the name shirt, written three times before, is now written once.

Joining the two back: Matching the three garment rows against the six order_line rows on garment_code returns the six rows the relation started with, and nothing besides. The general test for that belongs to a later section, but no decomposition should be printed without it having been run.

Second normal form is reached by giving each determinant that is part of the key a relation of its own, and leaving the rest of the row where it was.

The Master Key: *Take the attributes that depend on part of the key out together with that determinant, key the new relation on it, and leave it behind in the original as a foreign key.*

BUILD Question 80

The apprentice at Melela asks where the garment names have gone. Only the codes appear in the order lines now, and the words shirt and kanzu are nowhere among them. The answer is one table away, and the join brings them back.

Create the two tables of the second normal form decomposition, with a primary key on each and a foreign key between them, then show with an insert and a query that a garment name is stored once however many order lines ask for it.

garment holds garment_code as its primary key, with the garment name and the charge for one garment.

order_line holds order_no and garment_code together as its primary key, with the quantity.

garment_code in order_line references garment.

Insert six order lines across three orders, three of them asking for the same garment.

Model Answer

The two declarations below carry the second normal form decomposition: garment_code is the whole primary key of garment and one half of the primary key of order_line, where it is also the foreign key, so a garment name is stored once however many order lines ask for it.

```
 1  CREATE TABLE garment (
 2    garment_code CHAR(2)      NOT NULL,
 3    garment_name VARCHAR(20)  NOT NULL,
 4    unit_charge  DECIMAL(8,2) NOT NULL,
 5    PRIMARY KEY (garment_code)
 6  ) ENGINE = InnoDB;
 7
 8  CREATE TABLE order_line (
 9    order_no     CHAR(5)  NOT NULL,
10    garment_code CHAR(2)  NOT NULL,
11    quantity     SMALLINT NOT NULL,
12    PRIMARY KEY (order_no, garment_code),
13    FOREIGN KEY (garment_code) REFERENCES garment (garment_code)
14  ) ENGINE = InnoDB;
15
16  INSERT INTO garment (garment_code, garment_name, unit_charge) VALUES
17    ('G4', 'dress', 30000.00), ('G7', 'shirt', 12000.00),
18    ('G9', 'kanzu', 25000.00);
19
20  INSERT INTO order_line (order_no, garment_code, quantity) VALUES
21    ('T0041', 'G7', 3), ('T0041', 'G9', 1), ('T0042', 'G7', 5),
22    ('T0043', 'G4', 2), ('T0043', 'G7', 1), ('T0043', 'G9', 2);
23
24  SELECT g.garment_code, g.garment_name FROM garment g;
25
26  SELECT l.order_no, g.garment_name, l.quantity
27  FROM order_line l
28  JOIN garment g ON g.garment_code = l.garment_code
29  ORDER BY l.order_no, g.garment_name;
```

The server replies

```
+--------------+--------------+
| garment_code | garment_name |
+--------------+--------------+
| G4           | dress        |
| G7           | shirt        |
| G9           | kanzu        |
+--------------+--------------+
+----------+--------------+----------+
| order_no | garment_name | quantity |
+----------+--------------+----------+
| T0041    | kanzu        |        1 |
| T0041    | shirt        |        3 |
| T0042    | shirt        |        5 |
| T0043    | dress        |        2 |
| T0043    | kanzu        |        2 |
| T0043    | shirt        |        1 |
+----------+--------------+----------+
```

Program 4.5: *The two tables of the decomposition, and a garment name stored once.*

What the two keys say: The single column key of garment says that a code names one garment. The composite key of order_line says that an order asks for a garment on one line and no more, with a quantity to say how many. The foreign key then refuses a line whose code no garment row defines.

Reading the result: garment holds three rows and the word shirt stands in one of them. The second query prints shirt on three lines of the order book, because three order lines borrow that one code, and renaming the garment would now be one update of one row.

The one decision worth naming: unit_charge is DECIMAL because money has to add up exactly, and it sits in garment because it depends on the garment code alone. A charge negotiated separately for each order would depend on the whole key and would belong in order_line instead.

The join puts the name back onto every line that needs it, without the database ever having stored it there.

The Master Key: *Key the garment relation on the code, key the line relation on the order and the code together, declare that code a foreign key, and each garment name is then stored once.*

HOT Question 81

A shop in Doma keeps its whole order inside one column, as a document holding a list of garments, and the database accepts it without complaint. The column has a type the engine understands, so the question is whether anything has been broken.

Judge whether first normal form is a rule about design or a rule about the data type of a column, given that a modern engine can store a document in a single cell.

Model Answer

First normal form is a rule about design and not about the data type of a column, and a document in one cell breaks it exactly when something outside the cell must address a part of what is inside.

The case for the type deciding it: Engines now offer document and array types. MariaDB accepts a JSON column, checks that what goes in is well formed, and can index a value pulled from it through a generated column. Where the engine reaches inside a cell, it is tempting to let the declared type settle atomicity.

Why the type cannot be what decides: A date holds a day, a month and a year, and nobody calls it a breach, because the shop wants the date, never its parts. A customer name in one column is atomic where the shop only prints it, and not atomic where the work is ordered by family name. Same value, same type, two answers.

What a document in a cell costs: No foreign key can point at a garment code buried inside a list, and no unique constraint can be laid on one member of it. Every rule about the parts becomes program code elsewhere, and the anomalies first normal form removes come back too.

Where a document is the right answer: Some values are genuinely opaque. A scanned receipt, the raw reply from a payment service, a block of settings written and read whole: nothing in the design addresses a part of them, and holding each as one value hides nothing.

The condition that decides it: Ask whether anything outside the cell must reach a part of the value: a search condition, a join, a constraint, an index or a count. One yes and the value is not atomic for this design, whatever type the column carries.

The second half of the condition: Ask also whether the number of parts varies from row to row. A list whose length changes, and whose members must be counted or matched, is a repeating group in a new coat, and rows are the only shape the database can work on.

The judgement: First normal form is a rule about design. Keep a document in one cell where nothing addresses its parts, and split it into rows the moment a query, a constraint or a key must reach inside. That the engine will hold it is not permission to put it there.

The column type says what may be stored, the design says what must be reachable, and only the second decides atomicity.

The Master Key: *First normal form is about design rather than data type: a value is atomic when nothing outside the cell must address a part of it, and a document holding a list the database queries is not atomic however the column was declared.*

Third normal form, Boyce-Codd, and where fourth normal form begins

Third normal form is the rung a design is normally expected to reach, and it is also the rule most often written down in a shortened form that says the opposite of what it means. This section states it in full, informally and formally, then goes one rung further to Boyce-Codd normal form, which withdraws the escape that third normal form allows. It closes by naming where fourth normal form begins, so that a relation holding two independent repeating facts can be recognised on sight. Every claim made here about a relation was decided against the relation itself rather than asserted.

BINDER Question 82

A job card in the Kimamba shop carries the customer's town, and so does every other card for that customer. She moves to Nzuguni, and the clerk must now find every card she is written on before the address is right again.

State third normal form informally and formally, and explain what it removes.

Model Answer

A relation is in third normal form when it is in second normal form and no non-prime attribute depends transitively on a candidate key, and formally, when for every non-trivial dependency X to Y, either X is a superkey or every attribute of Y is prime.

The informal statement, negation and all: The words carrying the rule are no and transitively. A non-prime attribute may not be reached through another attribute on the way from the key, and in the job relation the town is reached through customer_id, which is no key.

The formal statement, and the limb that gets dropped: For every non-trivial dependency X to Y, either X is a superkey or every attribute of Y is prime. Both limbs are part of the rule. The second is the one that goes missing, and without it the rule forbids dependencies that third normal form allows.

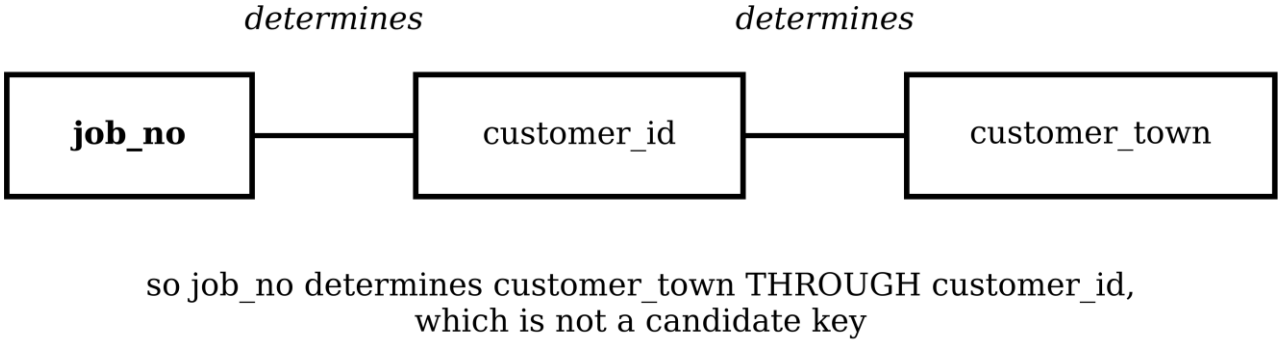


Figure 4.19: *A transitive dependency, and the clause that is always left out of the rule.*

The clause that stops every chain looking transitive: The intermediate determinant must not itself be a candidate key. Strike that clause and every chain qualifies, because a key determines everything, so anything can be described as reached through it. The chain counts only where the middle link is not a key.

What the form removes: A fact about the customer repeated inside every job that customer places. Held once per job, the town must be corrected in every job row, and one row missed leaves two towns for one customer with nothing to say which is right.

The wording to distrust: A common error is a statement that a relation is in third normal form if it is in second normal form and has a transitive dependency of a non-prime attribute. As written, that demands the presence of what the form removes.

The rule is one sentence with a negation inside it, and the negation is the whole of the sentence.

The Master Key: *A relation is in third normal form when every non-trivial dependency has a superkey on its left or nothing but prime attributes on its right.*

BINDER Question 83

Two operators share the guillotine at Nzuguni, and each of them works one machine and one only. The booking sheet is normalised, everybody agrees that it is normalised, and it still writes down which machine an operator works once for every job.

State Boyce-Codd normal form and explain exactly how it differs from third normal form.

Model Answer

A relation is in Boyce-Codd normal form when, for every non-trivial dependency X to Y, X is a superkey, and the whole difference from third normal form is that there is no second limb: the escape allowing a dependency whose right side is prime has been withdrawn.

The rule, with nothing following it: One condition and one only. Every determinant must determine everything, which is what being a superkey means. No question is asked about which attributes are prime, so no relation passes on the strength of what sits on the right of the arrow.

Where the two forms say the same thing: Wherever a relation has a single candidate key, or several that share no attribute, reaching third normal form has already reached Boyce-Codd. The two part company only when two candidate keys overlap, which is why most designs never meet the difference at all.

The relation that separates them: In booking, job_no and machine together determine operator, and operator alone determines machine. The two candidate keys are job_no with machine and job_no with operator, so machine is prime. Third normal form is satisfied by its second limb; Boyce-Codd has no second limb and is not.

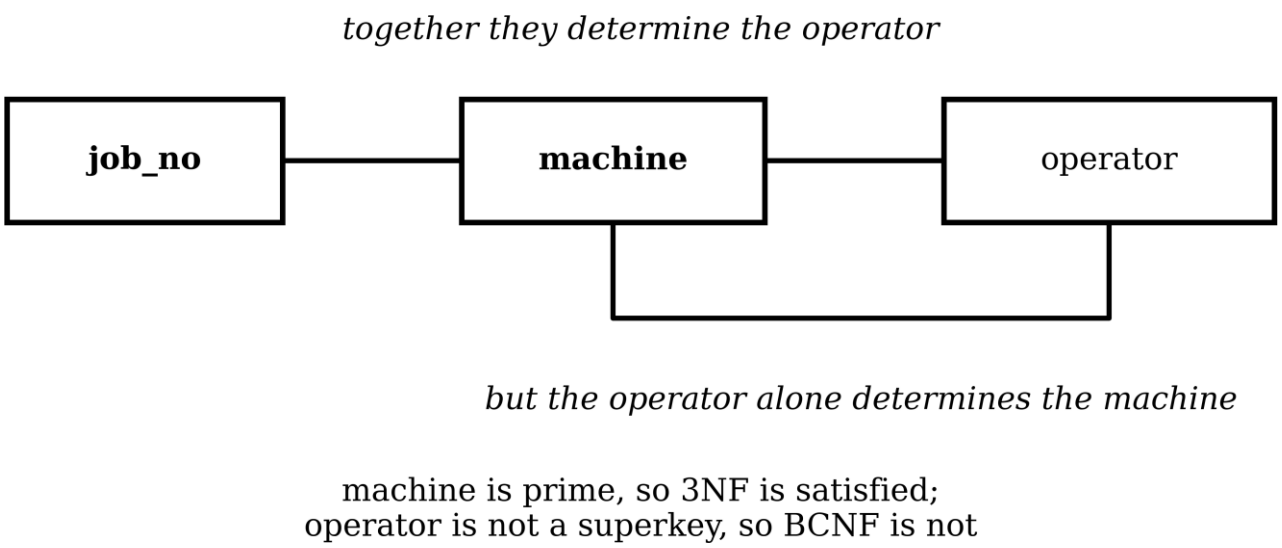


Figure 4.20: *A relation in third normal form that is not in Boyce-Codd normal form.*

What the stronger form removes: The redundancy a determinant that is not a key still carries. Which machine an operator works is one fact, and the booking relation writes it again for every job that operator handles, exactly as an unnormalised design would.

Why it is called the stronger form: Every relation in Boyce-Codd normal form is in third normal form, and not the other way round. Students are often taught nothing past third, so booking is met as finished work that unaccountably repeats itself.

Third normal form asks two questions of a dependency and Boyce-Codd asks one, which is why the shorter rule is the harder one to satisfy.

The Master Key: *Boyce-Codd normal form requires every determinant to be a superkey, dropping the concession third normal form makes for a right side that is wholly prime.*

BINDER Question 84

The finishing register at Ipagala carries a line pairing lamination with the Kimamba run, though no such job was ever done. Nobody entered it, nothing in the register is wrong, and every other pairing on the page is there for the same reason.

Explain what a multivalued dependency is, and say where fourth normal form begins.

Model Answer

A multivalued dependency holds where one value of X carries a whole set of Y values fixed by X alone, and fourth normal form begins where one relation holds two such independent sets about the same key, because storing them together forces every combination of the two onto the page.

How it differs from a functional dependency: A functional dependency pins one value: give it the key and exactly one town comes back. A multivalued dependency pins a set. Job J101 needs trimming and lamination, and asking which finishing it needs returns a list rather than an answer.

The two facts must be independent: The finishing a job needs and the towns it goes to have nothing to do with each other, and that independence is the whole condition. Were one town to accept only laminated work, the two lists would be related and this would not arise.

The rows nobody recorded: Two finishing operations and three delivery towns force six rows, and those six say nothing the two lists did not say apart. Add a third operation and nine rows are needed, three of them recording a pairing that never happened.

Where fourth normal form begins: A relation is in fourth normal form when it is in Boyce-Codd normal form and, for every non-trivial multivalued dependency X to Y, X is a superkey. The remedy is one relation for each independent fact, so the lists are kept apart and never multiplied together.

Why the earlier tests miss it: No functional dependency is broken here, so the relation passes every rule up to Boyce-Codd. The fault is not that an attribute depends on the wrong thing but that two independent lists were put in one relation.

The pairing nobody entered was manufactured by the shape of the relation, which is the sign fourth normal form exists to answer.

The Master Key: *A multivalued dependency gives a key a whole set of values, and fourth normal form begins where one relation holds two such sets that are independent of each other.*

REAL Question 85

The clerk at Kimamba corrects a customer's town on one job sheet, then finds the same town waiting on four more sheets for the same customer. She corrects three of them before the counter gets busy.

Identify the transitive dependency in job_sheet, decompose it into third normal form, and state the primary key and foreign key of each relation in the result.

job_sheet(job_no, customer_id, customer_name, customer_town, quoted), keyed on job_no.

job_no determines customer_id, customer_name, customer_town and quoted.

customer_id determines customer_name and customer_town.

J101 and J102 belong to C004 of Kimamba, J103 to C009 of Nzuguni, J104 to C015 of Ipagala.

Model Answer

The transitive dependency runs job_no to customer_id to customer_name and customer_town, so the decomposition is job(job_no, customer_id, quoted), keyed on job_no with customer_id as a foreign key, together with customer(customer_id, customer_name, customer_town), keyed on customer_id.

Why it is already in second normal form: The key is the single attribute job_no, and a single attribute has no proper part for anything to depend on, so no partial dependency is possible. Second normal form is therefore reached without doing anything, which is exactly why the remaining fault survives unnoticed.

Naming the dependency exactly: customer_id determines customer_name and customer_town. Test it against both limbs: customer_id is not a superkey, since it does not determine job_no or quoted, and neither customer_name nor customer_town is prime, since the only candidate key is job_no. Both limbs fail, so third normal form is broken.

The split itself: Take the determinant and everything it determines into a relation of their own, and leave the determinant behind in what remains as the column the two are joined on. Nothing is invented and nothing is typed twice; both relations are projections of the sheet the shop already keeps.

The keys of the result: job is keyed on job_no and customer is keyed on customer_id. In job, customer_id is a foreign key referencing customer and is not part of the primary key, which is what allows one customer to stand behind many jobs without appearing in the key of either.

What the split bought: The town is held once for each customer rather than once for each job, so the clerk's correction is a single write and cannot be left half done. A decomposition must also be shown to rejoin to the relation it came from, and the next section is where that is tested.

Relation	Primary key	Other columns	Foreign key
job	job_no	quoted	customer_id references customer
customer	customer_id	customer_name, customer_town	none

Table 4.19: *The two relations the decomposition produces, and the key each one carries.*

Five attributes became two relations of three columns each, and the one column standing in both of them is the join.

The Master Key: *A transitive dependency is removed by giving the intermediate determinant a relation of its own and leaving it behind as a foreign key.*

REAL Question 86

Three definitions are pinned inside the lid of the toolbox at Nzuguni, copied out by hand and never questioned since. One of them is right, and the other two are wrong in different ways.

One of the three statements below holds and two do not. Decide which, explain the fault in each of the two, and rewrite them correctly.

A. A relation is in third normal form if it is in second normal form and has a transitive dependency on a non-prime attribute.

B. A relation is in third normal form if it is in second normal form and no non-prime attribute depends transitively on a candidate key.

C. A relation is in third normal form if every attribute depends on the primary key and nothing else.

Model Answer

Statement B is correct; A asserts the presence of exactly what the form removes, and C, which sounds the most confident of the three, forbids two things the form permits and therefore rejects relations that are in third normal form.

Why B is right: It carries the negation, and the negation is the rule. Its formal equivalent is that for every non-trivial dependency X to Y, either X is a superkey or every attribute of Y is prime, and B is that sentence in the language of keys rather than of dependencies.

What A actually says: Read it as written and a relation qualifies because it has the transitive dependency. It is B with the word no struck out, so anybody applying it accepts the undivided job sheet and rejects the decomposition. Correct A by restoring the negation, which makes it B.

Why C is wrong about the key: The rule says superkey, not primary key. A relation may hold two candidate keys, and a dependency whose left side is the alternate key, or any superset of a key, is perfectly lawful. C throws every one of those out for the sole reason that the primary key was not the determinant.

Why C is wrong about prime attributes: The second limb permits a dependency whose right side is entirely prime even when its left side is only part of a key. In booking, operator determines machine, machine is prime, and the relation is in third normal form. C forbids anything depending on anything but the primary key, so it rejects that relation.

What C is nearly a statement of: It is closer to Boyce-Codd normal form than to third, and wrong there too by one word, since Boyce-Codd requires a superkey and not the primary key. Correct C to say that every determinant is a superkey, and it becomes a sound statement of the stronger form.

One of the three is the rule, one is the rule with its negation removed, and one is a different rule stated inaccurately.

The Master Key: *Test a dependency by asking whether its left side is a superkey and, failing that, whether its right side is wholly prime, because either answer makes it lawful.*

REAL Question 87

A second driver takes the Nzuguni run one Friday because the first is away. Two rows now disagree about who serves Nzuguni, and the design that was supposed to prevent this has been normalised as far as most designs ever go.

Show that delivery is in third normal form and not in Boyce-Codd normal form, decompose it, and state what the decomposition costs.

delivery(job_no, delivery_town, driver): one print run may be delivered to several towns.

job_no and delivery_town together determine driver.

driver determines delivery_town, because each driver serves one town and no other.

Rows: J101 to Kimamba by D1, J102 to Kimamba by D1, J102 to Nzuguni by D2, J103 to Ipagala by D3.

Model Answer

delivery is in third normal form because the only dependency whose left side is not a superkey has delivery_town on its right, and delivery_town is prime; it is not in Boyce-Codd normal form because driver is a determinant and not a superkey, and the split into round(driver, delivery_town) and drop(job_no, driver) costs the dependency of driver on job_no and delivery_town, which neither relation can now enforce.

The two candidate keys: job_no with delivery_town determines driver, and job_no with driver determines delivery_town, so each is a candidate key and every attribute of the relation is prime. Overlapping keys of this kind are the only situation in which the two normal forms can disagree.

Why third normal form is satisfied: The dependency in question is driver to delivery_town. The first limb fails, because driver does not determine job_no and so falls short of the whole relation, which a superkey must reach. The second limb holds, because delivery_town is prime, and satisfying either limb is enough.

Why Boyce-Codd is not satisfied: The stronger rule asks only the first question, and driver fails it. What that failure permits is visible in the rows: the fact that D1 serves Kimamba is written twice, once for each job delivered there, and a driver moved to another town must be corrected in every row naming him.

The decomposition: round holds driver and delivery_town, keyed on driver, and drop holds job_no and driver, keyed on both together. driver is the key of round and the shared column, so the two relations restore the original rows exactly, and the fact about D1 is now written once.

The cost, named plainly: The dependency of driver on job_no and delivery_town is not preserved. round can refuse a driver a second town and drop can refuse a repeated pairing, but nothing refuses two drivers for the same job in the same town, because no single relation holds all three columns any more.

What that means at the counter: A rule the schema enforced by itself now needs both relations consulted together on every insertion, which in practice means a program remembers to check it. The gain is one copy of each driver's town; the loss is a constraint that has left the schema.

Boyce-Codd bought the shop one copy of a fact and charged it one rule it can no longer state in a single table.

The Master Key: *A relation with overlapping candidate keys can satisfy third normal form on its second limb and still fail Boyce-Codd, and the split that fixes it may cost a dependency.*

REAL Question 88

One column was added to the finishing register at Ipagala, holding the day a batch ran, and it looked harmless. It brought a second fault into a register that already had one, and the two faults answer to different rules.

Take finishing from second normal form to Boyce-Codd normal form in stages, saying at each stage which rule forced the split.

finishing(job_no, finish_type, machine, batch_no, batch_day), keyed on job_no and finish_type.

job_no and finish_type together determine machine and batch_no.

machine determines finish_type, because each machine does one kind of finishing.

batch_no determines batch_day.

Model Answer

finishing is in second normal form already, is not in third and is not in Boyce-Codd; the first split is forced by batch_no determining batch_day, a non-prime attribute reached through a non-key, and the second by machine determining finish_type, a determinant that is not a superkey, leaving machine, operation and batch.

The keys, and which attributes are prime: job_no with finish_type is a candidate key, and so is job_no with machine, since machine gives finish_type. The prime attributes are therefore job_no, finish_type and machine, and the non-prime ones are batch_no and batch_day.

Why it is in second normal form to begin with: No non-prime attribute depends on part of a key. batch_no depends on the whole of job_no and finish_type, and batch_day depends on batch_no, which is not part of any candidate key at all. There is no partial dependency to remove.

The first split, forced by third normal form: batch_no determines batch_day, batch_no is not a superkey and batch_day is not prime, so both limbs fail. Take batch(batch_no, batch_day) out, keyed on batch_no, and leave batch_no behind. Every dependency of the original still lives inside one of the two parts.

Why the remainder stops there: What is left holds job_no, finish_type, machine and batch_no, and it is in third normal form: machine determines finish_type, and although machine is no superkey, finish_type is prime, so the second limb carries it. The stronger rule is the only one that objects.

The second split, forced by Boyce-Codd: machine is a determinant and not a superkey, so machine(machine, finish_type) comes out keyed on machine, and operation(job_no, machine, batch_no) remains, keyed on job_no with machine. Every determinant in all three relations is now a superkey.

What each rule cost: The third normal form split cost nothing at all. The Boyce-Codd split cost the dependency of machine and batch_no on job_no and finish_type, which no single relation now holds, and that difference between the two stages is not an accident of this example.

Relation	Primary key	Forced by
batch(batch_no, batch_day)	batch_no	Third normal form: batch_day is non-prime and reached through batch_no
machine(machine, finish_type)	machine	Boyce-Codd: machine determines finish_type and is not a superkey
operation(job_no, machine, batch_no)	job_no, machine	What remains once both determinants have been taken out

Table 4.20: *The three relations, and the rule that forced each one out of the register.*

Two splits, two different rules, and only the second of them charged anything for the tidying it did.

The Master Key: *Remove the transitive dependency first and the determinant that is not a superkey second, because the two stages answer to different rules and carry different costs.*

BUILD Question 89

Kimamba to Ipagala, one statement typed at the terminal, and every job the customer has ever placed now shows the new town. Nothing in the job table was touched, and no row of it needed to be.

Create the two tables of the third normal form decomposition, with a primary key on each and a foreign key joining them, and show that moving a customer to another town is now a single write.

customer holds customer_id as its primary key, with a name and a town that may not be null.

job holds job_no as its primary key, the quoted price, and customer_id referencing customer.

Move customer C004 from Kimamba to Ipagala with one statement.

Then join the two tables and show every job with the town its customer is now in.

Model Answer

The town is declared once, in customer, so the single UPDATE below moves C004 to Ipagala by writing one row, and the joined result shows both of that customer's jobs carrying the new town although no row of job was altered.

```
 1  CREATE TABLE customer (
 2    customer_id   CHAR(4)     NOT NULL,
 3    customer_name VARCHAR(30) NOT NULL,
 4    customer_town VARCHAR(20) NOT NULL,
 5    PRIMARY KEY (customer_id)
 6  ) ENGINE = InnoDB;
 7  CREATE TABLE job (
 8    job_no      CHAR(4)       NOT NULL,
 9    customer_id CHAR(4)       NOT NULL,
10    quoted      DECIMAL(10,2) NOT NULL,
11    PRIMARY KEY (job_no),
12    FOREIGN KEY (customer_id) REFERENCES customer (customer_id)
13  ) ENGINE = InnoDB;
14
15  INSERT INTO customer (customer_id, customer_name, customer_town)
16  VALUES ('C004', 'Neema Stationers', 'Kimamba'),
17         ('C009', 'Baraka Academy', 'Nzuguni');
18  INSERT INTO job (job_no, customer_id, quoted)
19  VALUES ('J101', 'C004', 120000.00),
20         ('J102', 'C004', 84000.00),
21         ('J103', 'C009', 150000.00);
22
23  UPDATE customer
24  SET customer_town = 'Ipagala'
25  WHERE customer_id = 'C004';
26  SELECT j.job_no, c.customer_name, c.customer_town
27  FROM job j
28  JOIN customer c ON j.customer_id = c.customer_id
29  ORDER BY j.job_no;
```

The server replies

```
+--------+------------------+---------------+
| job_no | customer_name    | customer_town |
+--------+------------------+---------------+
| J101   | Neema Stationers | Ipagala       |
| J102   | Neema Stationers | Ipagala       |
| J103   | Baraka Academy   | Nzuguni       |
+--------+------------------+---------------+
```

Program 4.6: *The third normal form decomposition, and a town corrected in one place.*

How the schema meets the specification: Each table declares its own primary key, and customer_id in job is a foreign key rather than part of that key, so many jobs may name one customer. customer_town appears in customer and nowhere else, which is what removing the transitive dependency amounts to in a schema.

Reading the result: One UPDATE touching one row moved both of that customer's jobs to Ipagala, and C009 is untouched at Nzuguni. There is no second copy of the town to fall out of step with the first, so the update anomaly is not merely avoided here, it is unavailable.

The decisions worth naming: quoted is DECIMAL because money must add up exactly, and customer_id is CHAR because a code with a leading zero would lose it as an integer. The foreign key also keeps a job from naming a customer the shop has no record of.

The join puts the town back beside every job at the moment it is asked for, which is the only moment it was ever needed there.

The Master Key: *Store a fact about the customer in the customer table and fetch it back with a join, and one write corrects it everywhere it appears.*

HOT Question 90

One table at Nzuguni is still open on the screen at closing time. Two designers cannot agree: one wants its last determinant taken out and the rule moved into the program that inserts rows, the other wants it left exactly as it stands.

Judge whether a designer should normalise to Boyce-Codd normal form or stop at third, and state the condition that decides it.

Model Answer

Go to Boyce-Codd normal form wherever the split loses no dependency, and stop at third normal form wherever it would, unless the repetition the remaining determinant carries is actually producing rows that disagree, in which case take Boyce-Codd and put the lost rule into the program that writes.

What third normal form leaves behind: A determinant that is not a superkey may remain, provided everything it determines is prime. That determinant repeats its fact once for every row naming it, which is the same redundancy normalisation was begun to remove, surviving inside a design everybody agrees is normalised.

The case for going on: Boyce-Codd asks one question of a dependency and third normal form asks two, so the stronger form needs no judgement about which attributes are prime. It removes the last determinant that is not a key, and with it the last redundancy that functional dependencies can describe.

The case for stopping: Third normal form can always be reached without losing a dependency. Boyce-Codd sometimes cannot, and where it cannot, a rule the schema enforced by itself becomes a rule some program must remember. A constraint that depends on being remembered is a constraint that will eventually be forgotten.

How often the question even arises: Only where two candidate keys overlap. A relation with one candidate key is in Boyce-Codd the moment it is in third normal form, so for most of a schema the two forms are the same demand and the argument has nothing to be about.

The condition that decides it: Attempt the split and ask whether every dependency of the original still lies wholly inside one of the parts. If it does, the stronger form is free and should be taken. If one dependency is left spanning two relations, the choice becomes a trade and has to be argued.

How to argue that trade: Weigh how often the repeated fact changes against how often the lost rule would be broken. A driver who seldom changes town is a small redundancy; a clerk entering two drivers for one town is a wrong row that nothing in the schema will catch.

The judgement: Take Boyce-Codd where it is free and stop at third normal form where it is not, keeping the constraint inside the schema by preference, because a schema enforces its rules against every program that ever connects and an application enforces them only against itself.

The verdict is not that one form is better but that a rule the database can hold is worth more than one copy of a fact.

The Master Key: *Normalise to Boyce-Codd where the split costs no dependency, and otherwise stop at third normal form unless the repetition is already producing rows that contradict each other.*

Testing a decomposition: lossless join, dependency preservation, denormalisation

A decomposition is not finished when the parts have been written down; it is finished when the parts have been joined back and the result compared with what they came from. This section gives that test in the form that settles any split into two in a few seconds, and then gives the second question a decomposition must answer, which is whether every rule the original could enforce can still be enforced by one part on its own. It closes with denormalisation, which puts redundancy back deliberately, and with the one condition that justifies doing so. The emphasis falls where it does because a split that cannot be rejoined does not usually lose rows, it invents them, and rows that were never true are far harder to notice than rows that have gone missing.

BINDER Question 91

Three rows went into the split at Mtumba and five came back out. Nobody deleted anything and nobody typed anything in, and two of the five say that a member saves under a treasurer she has never handed a shilling to.

Explain what makes a decomposition lossless, state the test that decides it, and say what a decomposition that fails the test actually does to the rows.

Model Answer

A decomposition is lossless when joining the parts back gives exactly the rows the original held, none missing and none added, and for a split into two parts the test is that the attributes common to both parts must form a superkey of at least one of them.

What the join has to give back: The original relation itself and nothing beside it. Both halves of that sentence carry weight, and the second half is the one that gets dropped, because two tables joining to something larger than they came from are not a decomposition of anything.

The test, in one line: Take the attributes the two parts share and ask whether they are a superkey of either part. Splitting membership into member with group_name, and group_name with treasurer, passes it: group_name determines treasurer, so the shared column is the whole key of the second part.

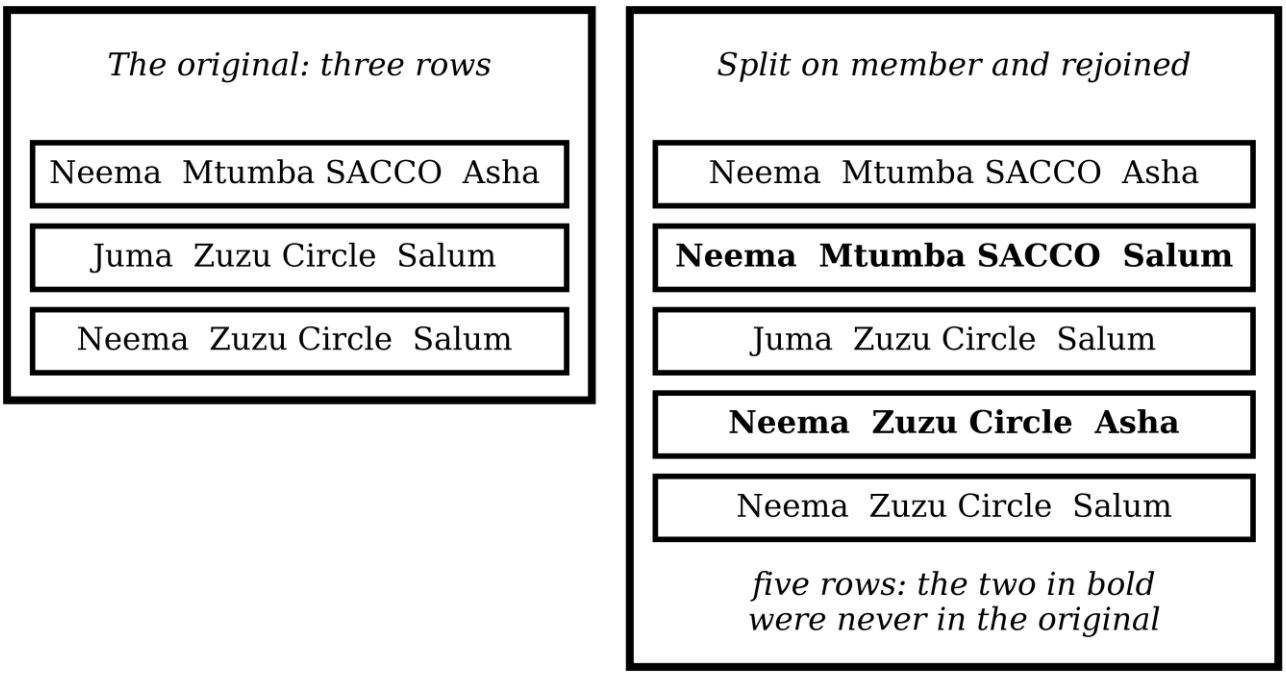


Figure 4.21: *A decomposition that invents two rows that were never in the original.*

What a failed split really does: It invents rows. Split that same register on member instead and each part holds three rows, but Neema stands in two rows of each, so joining on member pairs both of her groups with both treasurers and five rows come back.

The failure nobody looks for: The word says loss, so a reader looks for a row that has gone missing, and none has. Every row of the register is still there, with two strangers standing among them that look exactly like data and are not.

Why the shared column must be a key: Unless it identifies one row inside one part, a single value of it stands beside several rows in both parts and the join multiplies them together. group_name identifies one treasurer; member identifies neither a group nor a treasurer.

A decomposition earns the word lossless only once the parts have been joined and the rows compared.

The Master Key: *A binary decomposition is lossless when the attributes shared by the two parts are a superkey of at least one of them, and a split that fails invents rows rather than losing them.*

BINDER Question 92

Two collectors turned up for the Nala Group at the same slot on the same morning, and both tables took the work without complaint. Neither table is wrong on its own, and the rule that would once have refused the second one is written nowhere.

Explain what it means for a decomposition to preserve dependencies, and show that a decomposition can be lossless and still cost something.

Model Answer

A decomposition preserves dependencies when every functional dependency the original carried can still be checked inside one part on its own, and a split that rejoins perfectly can fail this, because a dependency whose attributes end up in different parts is a rule neither table can enforce alone.

What preservation asks: That the left side and the right side of a dependency sit together inside one part. Where they do, the server can refuse a row that breaks the rule without consulting the other table. Where they do not, the rule survives as a fact about the data and stops being a constraint.

A split that rejoins and still costs: roster holds slot_no, group_name and collector: a slot and a group fix which collector is on duty, and each collector serves one group only. Splitting it into collection, holding slot_no and collector, and assignment, holding collector and group_name, is lossless, because collector is the key of assignment.

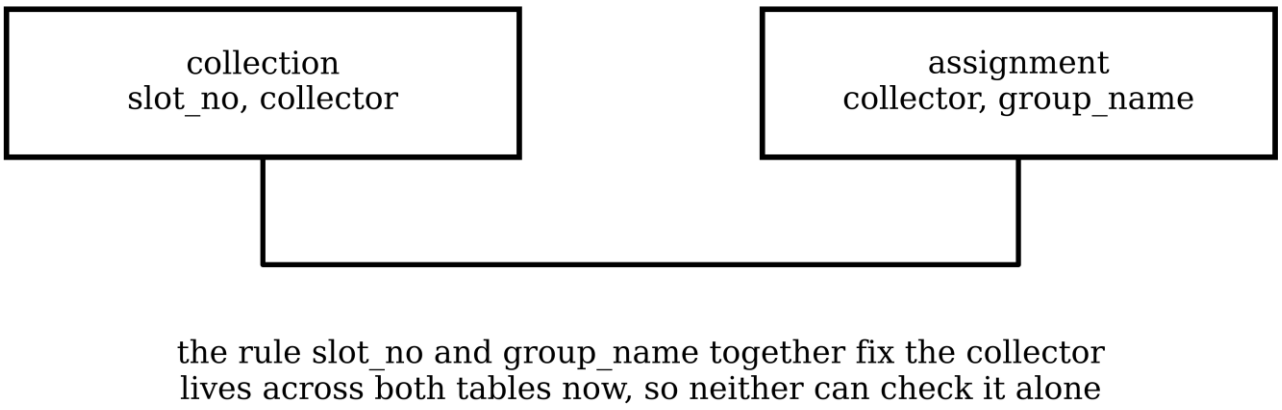


Figure 4.22: *A dependency that survives the split but can no longer be checked by either table alone.*

The rule with nowhere left to live: slot_no and group_name together determine collector, and after the split no part holds all three columns. Nothing refuses a second collector for the Nala Group in that slot, and noticing that one arrived means joining both tables on every insertion.

Who keeps the rule afterwards: A program does, which means a person remembers. A constraint inside the schema is enforced against every program that ever connects, and a constraint moved into one program holds only while that program is the only thing writing.

Two questions, not one: Every relation can be split losslessly, and every relation can reach third normal form with both properties intact. What cannot always be had together is Boyce-Codd normal form and every dependency.

A decomposition is judged twice, once on the rows it gives back and once on the rules it leaves behind.

The Master Key: *A decomposition preserves dependencies when each one can be checked inside a single part, and one that cannot is enforceable only by joining the parts on every write.*

REAL Question 93

Rows nobody ever entered appear in the joined result at Zuzu. Two ways of cutting the register in half were drawn on the same sheet; both look tidy, both keep every column, and only one of them can be put back together.

Decide which of the two proposed decompositions is lossless by applying the common-attribute test, confirm the verdict by joining the parts back and counting the rows, and name any row the join invents.

membership(member, group_name, treasurer) records who saves in which group and which treasurer receives that group’s money.

group_name determines treasurer, because a group has one treasurer.

The register holds Neema in Mtumba SACCO under Asha, Juma in Zuzu Circle under Salum, and Neema in Zuzu Circle under Salum.

Split one: member_group(member, group_name) with group_officer(group_name, treasurer).

Split two: member_group(member, group_name) with member_officer(member, treasurer).

Model Answer

Split one is lossless and split two is not: the column shared by split one is group_name, which is a superkey of group_officer, while the column shared by split two is member, which is a superkey of neither part, and joining split two returns five rows where the register holds three.

Applying the test to split one: The shared column is group_name. In group_officer it determines treasurer and so determines every attribute of that part, which is what makes it a superkey there. One part is enough, and the test is satisfied before a single row is looked at.

Applying the test to split two: The shared column is member. In member_group it fails, because Neema does not fix a group; in member_officer it fails again, because Neema does not fix a treasurer. Neither part is keyed on the shared column, so the split is rejected.

Joining split one back: member_group holds three rows and group_officer holds two, since projection removes the duplicate treasurer. Joining them on group_name gives three rows, one for each row of the register and identical to it, so nothing was lost and nothing added.

Joining split two back: member_group holds three rows and member_officer holds three. Neema appears in two rows of each, and the join pairs each with each, so she draws four rows and Juma draws one. Five rows come back from a register of three.

Naming the invented rows: Neema with Mtumba SACCO under Salum, and Neema with Zuzu Circle under Asha. Neither was ever true: Asha receives for Mtumba SACCO and Salum receives for Zuzu Circle, and the split has crossed them over because it kept the member beside each fact instead of the group.

What a search for missing rows would report: Nothing at all. All three rows of the register sit among the five, so a reader checking for loss finds none and passes a split that cannot be trusted. Counting is what exposes it and comparing row by row is what proves it.

Proposed split	Shared column	A superkey of a part?	Rows after rejoining
member_group with group_officer	group_name	Yes, of group_officer	Three, the register exactly
member_group with member_officer	member	No, of neither part	Five, two of them invented

Table 4.21: *The two proposed splits, the column each shares, and what the join gives back.*

One split gives back the register and the other gives back the register plus two pairings nobody ever made.

The Master Key: *Apply the common-attribute test first, because it settles the question without data, and then join the parts and count the rows to see the invention for yourself.*

REAL Question 94

Somebody copied the first sentence onto a card and taped it above the counter at Nala, where it has been quoted ever since. It is the one of the three that lets a bad split through without a word of complaint.

Two of the three statements below are false. Point to the true one, then name the error in each false one and give a corrected version.

A. A decomposition is lossless when no row of the original is missing after the parts are joined back.

B. A decomposition is lossless when the attributes shared by the two parts are a superkey of at least one of them.

C. A decomposition that preserves every dependency is always lossless.

Model Answer

Statement B is correct; A states only half of the requirement and passes a decomposition that invents rows, and C is false because preservation and losslessness are independent properties, so a split can keep every dependency inside one part and still fail to rejoin.

Why B is right: It is the test itself, and it decides the question from the schema alone, with no rows needed. The shared attributes must determine every attribute of one part, which is what makes each row of that part meet exactly one partner in the join.

What A leaves out: The rows the join adds. A splits the requirement in half and keeps the half that is easy to check, so the register split on member passes it: all three original rows are present among the five, and the two that were never true are not looked for.

Why that half is the dangerous half: A missing row announces itself, because somebody asks where it went. An invented row is a sentence the database now states about the group, in the same shape as every true row beside it, and nobody asks where it came from.

Correcting A: Say that no row of the original is missing and no row appears that the original did not hold. Written that way it becomes a sound check on the data, though still a slower one than B, which answers from the attributes before any rows exist.

Why C is false: Split the register on treasurer instead, into group_name with treasurer and member with treasurer. The dependency of treasurer on group_name sits whole inside the first part and is preserved, yet the shared column is a key of neither part, and once one treasurer receives for two groups the join invents rows.

Correcting C: The two properties are tested separately and neither implies the other. A decomposition may rejoin exactly and lose a rule, as the collectors did, or keep every rule and fail to rejoin, as splitting on treasurer does, so a design is checked for both.

One of the three is the test, one is the easier half of the test, and one is a claim that two separate questions are the same question.

The Master Key: *Losslessness and dependency preservation are independent, and a check that looks only for missing rows will pass a decomposition that invents them.*

REAL Question 95

The two printed tables at Mtumba add up to seventy-seven thousand shillings, exactly what the ledger says. Put them back together and one group's savings have moved into another group, and the total is still right to the shilling.

Show that the two printed tables cannot be rejoined to the ledger they came from, identify the single value that is inconsistent between them, and correct it.

ledger(receipt_no, member, group_code, group_name, amount), keyed on receipt_no, with group_code determining group_name.

The ledger holds R41 Neema G1 Mtumba SACCO 20000, R42 Juma G2 Zuzu Circle 15000, R43 Halima G1 Mtumba SACCO 12000, and R44 Rehema G3 Nala Group 30000.

entry(receipt_no, member, group_code, amount) is printed with all four receipts.

group_register(group_code, group_name) is printed as G1 Mtumba SACCO, G2 Zuzu Circle, G3 Zuzu Circle.

Model Answer

They cannot be rejoined because G3 has been given two meanings across the pair: in entry it is the code on Rehema's receipt, which the ledger places in the Nala Group, and in group_register it is named Zuzu Circle, so the join returns a row for R44 that the ledger never held and drops the one it did; the value to correct is the group name printed against G3.

The split itself is sound: The shared column is group_code and it is the primary key of group_register, so the decomposition passes the common-attribute test and is lossless as a design. Nothing is wrong with the reasoning above these two tables. What is wrong is one value inside them.

What the join actually returns: Four rows, exactly as many as the ledger holds. Three of them match the ledger. The fourth reads R44, Rehema, G3, Zuzu Circle and thirty thousand shillings, and no such row was ever entered, while the row naming the Nala Group has disappeared from the result.

Why counting the rows misses it: Four went in and four came out, so every count agrees and every total agrees. One row was lost and one invented in the same movement, which is why the parts have to be compared with the original row by row and not merely tallied against it.

Finding the inconsistent value: Work back from the code. G3 appears in entry only on Rehema's receipt, and the ledger puts that receipt in the Nala Group, so G3 means the Nala Group. group_register says Zuzu Circle, which already holds G2, so one name now carries two codes and one code carries two meanings.

What it does to the money: Zuzu Circle appears to have taken in forty-five thousand shillings rather than fifteen thousand, and the Nala Group nothing at all, while the ledger total is unchanged at seventy-seven thousand. The books balance and two groups have been told the wrong figure.

The correction, and the one that would not work: Write Nala Group against G3 in group_register, and the pair rejoins to the ledger exactly. Changing Rehema's code to G2 in entry instead removes the contradiction and still gives the wrong answer, because the row would then say Zuzu Circle, which is not where her thirty thousand went.

The two tables disagree about one word, and one word is all a decomposition needs to stop being a decomposition of anything.

The Master Key: *Join the tables as printed and compare them with the original, because a decomposition can be correct in design and still be wrong in the data that reaches the page.*

BUILD Question 96

Two tables built from one register, joined back, and three numbers on one line: that is what the Zuzu committee asked for instead of a paragraph. The row count will not do it, because a count can agree while the rows do not.

Write the statements that build the lossless decomposition of membership and prove it, by joining the two tables and comparing the result with the original by row as well as by count.

membership(member, group_name, treasurer) already holds the three rows drawn in Figure 4.21.

member_group holds member and group_name together as its primary key.

group_officer holds group_name as its primary key, with the treasurer that group pays.

Fill both from membership, join them, and print the original count, the rejoined count, and the number of rejoined rows the register does not hold.

Model Answer

Both tables below are filled by projection from membership itself, so no value is typed twice, and the join gives back three rows against the three the register holds with none invented, which is the proof the specification asked for.

```
 1  CREATE TABLE member_group (
 2    member     VARCHAR(20) NOT NULL,
 3    group_name VARCHAR(20) NOT NULL,
 4    PRIMARY KEY (member, group_name)
 5  ) ENGINE = InnoDB;
 6
 7  CREATE TABLE group_officer (
 8    group_name VARCHAR(20) NOT NULL,
 9    treasurer  VARCHAR(20) NOT NULL,
10    PRIMARY KEY (group_name)
11  ) ENGINE = InnoDB;
12
13  INSERT INTO member_group (member, group_name)
14  SELECT DISTINCT m.member, m.group_name FROM membership m;
15  INSERT INTO group_officer (group_name, treasurer)
16  SELECT DISTINCT m.group_name, m.treasurer FROM membership m;
17
18  SELECT (SELECT COUNT(*) FROM membership) AS original_rows,
19         COUNT(*)                          AS rejoined_rows,
20         COUNT(*) - COUNT(m.member)        AS invented_rows
21  FROM member_group g
22  JOIN group_officer o ON g.group_name = o.group_name
23  LEFT JOIN membership m
24    ON m.member     = g.member
25   AND m.group_name = g.group_name
26   AND m.treasurer  = o.treasurer;
```

The server replies

```
+---------------+---------------+---------------+
| original_rows | rejoined_rows | invented_rows |
+---------------+---------------+---------------+
|             3 |             3 |             0 |
+---------------+---------------+---------------+
```

Program 4.7: *A lossless decomposition, joined back and counted against the original.*

How the schema meets the specification: group_name is the primary key of group_officer, and that single declaration is what makes the shared column a superkey of one part and the decomposition lossless. member_group is keyed on both its columns, because a member may save in more than one group.

Reading the three numbers: Three rows in, three rows back, and none of them a stranger. The third number is the one that matters, because equal counts alone would not show that the rows are the same rows, and the outer join is what looks each rejoined row up in the register.

The decision worth naming: The parts are filled with SELECT DISTINCT, which is what makes each one a projection rather than a copy. Two rows of the register carry Salum and group_officer keeps one of them, so the treasurer of a group is now written once.

The proof is three numbers wide, and a decomposition that cannot produce it has not been shown to be a decomposition.

The Master Key: *Fill the parts from the original by projection, join them back, and count both the rows returned and the rows the original never held.*

BUILD Question 97

Nobody added a row and nobody removed one. The two tables at Nala were filled from the register itself, joined back the same afternoon, and the server counted five where the register has three.

Write the statements that build the bad decomposition of membership, fill both tables from the register, join them, and print the rows the join gives back together with the count.

membership(member, group_name, treasurer) again holds the three rows of the register.

member_group holds member and group_name; member_officer holds member and treasurer.

Both are filled from membership, so no value can have been mistyped.

Join them on the column they share and show what comes back.

Model Answer

Splitting on member puts the shared column in a part where it is a key of neither, so the join below returns five rows against the three the register holds, and two of the five are pairings that were never made.

```
 1  CREATE TABLE member_group (
 2    member     VARCHAR(20) NOT NULL,
 3    group_name VARCHAR(20) NOT NULL,
 4    PRIMARY KEY (member, group_name)
 5  ) ENGINE = InnoDB;
 6
 7  CREATE TABLE member_officer (
 8    member    VARCHAR(20) NOT NULL,
 9    treasurer VARCHAR(20) NOT NULL,
10    PRIMARY KEY (member, treasurer)
11  ) ENGINE = InnoDB;
12
13  INSERT INTO member_group (member, group_name)
14  SELECT DISTINCT m.member, m.group_name FROM membership m;
15  INSERT INTO member_officer (member, treasurer)
16  SELECT DISTINCT m.member, m.treasurer FROM membership m;
17
18  SELECT g.member, g.group_name, o.treasurer
19  FROM member_group g
20  JOIN member_officer o ON g.member = o.member
21  ORDER BY g.member, g.group_name, o.treasurer;
22
23  SELECT (SELECT COUNT(*) FROM membership) AS original_rows,
24         COUNT(*)                          AS rejoined_rows
25  FROM member_group g
26  JOIN member_officer o ON g.member = o.member;
```

The server replies

```
+--------+--------------+-----------+
| member | group_name   | treasurer |
+--------+--------------+-----------+
| Juma   | Zuzu Circle  | Salum     |
| Neema  | Mtumba SACCO | Asha      |
| Neema  | Mtumba SACCO | Salum     |
| Neema  | Zuzu Circle  | Asha      |
| Neema  | Zuzu Circle  | Salum     |
+--------+--------------+-----------+
+---------------+---------------+
| original_rows | rejoined_rows |
+---------------+---------------+
|             3 |             5 |
+---------------+---------------+
```

Program 4.8: *A bad decomposition, joined back, returning more rows than it started with.*

Where the split goes wrong: member is the shared column and it is the whole key of neither part, since a member saves in more than one group and pays more than one treasurer. Each of her rows on the left therefore meets each of her rows on the right.

Reading the result: The third and fourth rows of the join were never in the register: Neema with Mtumba SACCO under Salum, and Neema with Zuzu Circle under Asha. The three true rows are all still present, which is exactly why a reader looking for something missing declares the split safe.

The decision worth naming: Both tables were filled from the register by projection rather than typed, so no keying error can be blamed for the two extra rows. They were manufactured by the choice of shared column, and the count is the shortest proof of it.

Five rows came out of three, and every one of them was produced by statements that did exactly what they were told.

The Master Key: *A decomposition on the wrong shared column does not lose rows, it manufactures them, and joining the parts and counting is what brings them into view.*

HOT Question 98

The monthly statement at Mtumba takes eleven seconds to come back, and somebody has already proposed copying the group name into the contributions table to be rid of the join. Nobody has yet asked what those eleven seconds were measured on.

Judge when denormalisation is the right decision, and state the condition that decides it.

Model Answer

Denormalise only when a measurement, and not an impression, shows a named report too slow and shows the join to be the reason; then add the smallest redundancy that answers it and write the thing that keeps the copies in step in the same change, because the schema will no longer guarantee that they agree.

What the word actually means: Putting a fact back in two places deliberately, in a design that was normalised first, to spare a query a join it has been shown to need sparing. A design that was never normalised is not denormalised; it is simply undesigned, and it carries the same redundancy by accident.

The case for it: A statement joining four tables may be read fifty times a day and written once a month. Reading is what the group does with its records, so a copy that costs one column and one write a month can be a good bargain if the reading is genuinely slow.

Why a feeling about joins is not enough: A join on a key of a few thousand rows is not slow, and the usual reason a report crawls is a missing index or a query fetching columns nobody reads. Both are cheaper to correct than a duplicated column and neither costs anything in correctness.

The measurement that decides it: Time the statement as it stands, then time the same statement with the join removed by hand and the value supplied directly. If the difference does not matter to somebody waiting at a counter, there is nothing to buy, and the argument is over before it starts.

What is given up, named plainly: The redundancy has to be kept in step by something. Held once, the group name could not disagree with itself, and the schema said so. Held twice, the schema can no longer say it, so something else must, and that something is now the application.

Where that duty lands: On a trigger, which stays with the schema and fires for every connection, or on the one program that owns every write, which holds only until somebody types an UPDATE at the terminal. The first is a constraint moved; the second is a constraint remembered.

The judgement: Normalise and stop there, and denormalise only where a measured report is too slow, the measurement names the join as the cost, and the redundancy needed is one column that seldom changes. Add the column and the trigger together, or the copies will part company inside a season.

Denormalisation buys reading speed with a promise the database can no longer keep on its own.

The Master Key: *Denormalise on a measurement rather than an impression, keep the redundancy to one column that rarely changes, and write what will hold the copies together before the column exists.*

HOT Question 99

Put the rule where the machine enforces it, or put the fact in one place only. The committee at Nala has to choose between those two, because the split drawn on the table in front of them cannot deliver both.

Assess whether a designer should ever accept a decomposition that loses a dependency, and reach a judgement.

Model Answer

Yes, but only where the redundancy being removed is already producing rows that contradict each other and the lost rule can be given to something that cannot be bypassed; otherwise keep the decomposition that preserves every dependency, and never accept one that is not lossless, because that is not a trade at all.

The two costs are not the same kind: A lost dependency leaves the data correct and the rules thinner. A lost join leaves the data wrong, because the parts return rows the original never held. The first is a decision a designer may take; the second is a fault, and nothing is bought with it.

The case for accepting the loss: The dependency that will not fit inside one part is usually the one keeping a fact in two places. The collectors example writes which group a collector serves once for every slot she works, and each of those copies is a row that can drift out of step with the others.

The case for refusing it: A rule inside the schema is enforced against every program that ever connects, including the one written next year by somebody who never read the design. A rule enforced by joining two tables on each insertion is enforced only where somebody remembered to write the join.

The choice usually is not forced: A lossless decomposition preserving every dependency is always available at third normal form. The question arises one rung higher, where a determinant that is not a key must come out, so it touches a few relations rather than a whole schema.

What has to be in place first: Somewhere for the rule to go that cannot be walked round: a trigger, or a unique constraint on a table that holds the pairing, or one service through which every write passes. A dependency handed to a program that must remember it is a dependency abandoned.

How to weigh the two: Ask how often the repeated fact changes and how often the lost rule would be broken. A collector who keeps her group for years is a small redundancy; two collectors entered for one slot is a wrong row nothing in the schema will now refuse.

The judgement: Accept a lost dependency where the repetition is already producing contradictions and the rule can be lodged somewhere that fires for every writer; otherwise stop at the decomposition that keeps every rule. Losing a dependency is a price; losing the join is a defect, and the two are never weighed together.

A designer may trade a rule for a single copy of a fact, and may never trade away the rows themselves.

The Master Key: *Accept a lost dependency only when the redundancy is causing contradictions and the rule can be enforced somewhere unavoidable, and never accept a decomposition that cannot be rejoined.*

Creating and altering tables: types, and the constraints that hold a schema together

A table is not a container for whatever anybody types; it is a set of promises about what may be stored, and CREATE TABLE is where those promises are written down. This section declares them: the primary key that stops a second row claiming an identity, the foreign key that refuses a crate belonging to no farm, the NOT NULL, UNIQUE and CHECK declarations that move a rule out of the programs and into the schema, the DEFAULT that fills a gap, and the data types that decide what a column can hold at all. It then alters a table that is already carrying data, and shows what the same declarations are worth under a storage engine that reads them and throws them away.

BINDER Question 100

Two crates at Msalato carry the same number stencilled on the side. The clerk who booked in the second one was not careless; nothing in the system he types into has ever stopped him, and nothing in it ever will.

Explain what a constraint is, and why a schema that declares its constraints is different in kind from one that does not.

Model Answer

A constraint is a rule about the data written into the definition of the table itself and enforced by the server on every insert and every update, so a schema that declares its constraints can hold only what its design allows, while a schema that declares none holds whatever the last program to write to it happened to send.

What the word names: PRIMARY KEY, NOT NULL, UNIQUE, CHECK, FOREIGN KEY and DEFAULT are parts of a table definition, not code that runs beside it. The server tests each one as the row is written, and a write that breaks a constraint does not happen at all.

each line refuses

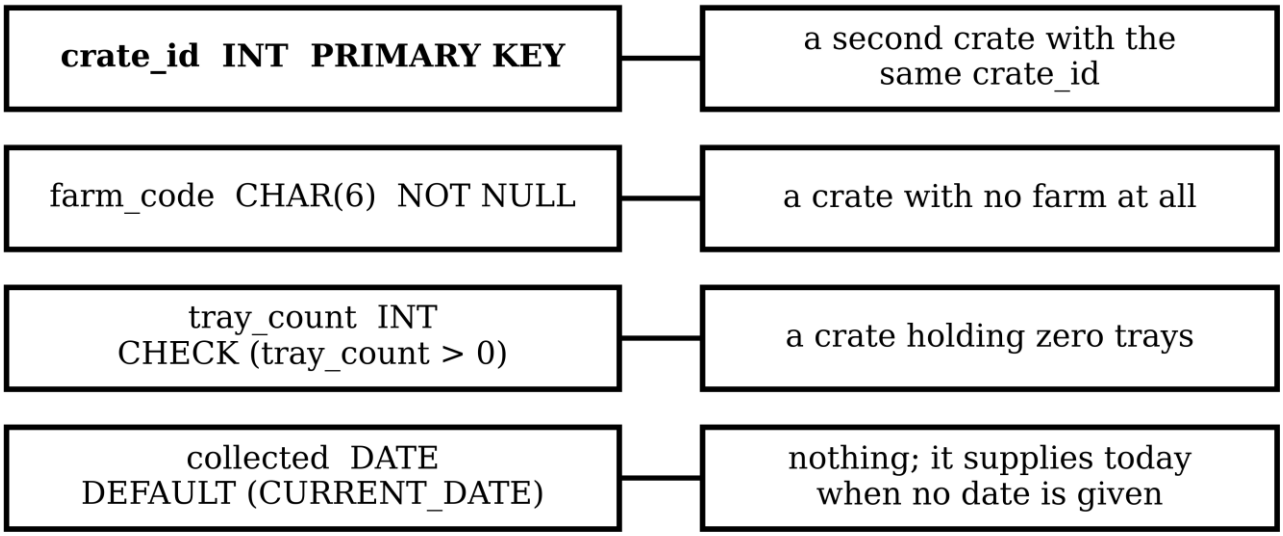


Figure 4.23: *Four constraints on one table, and the row each of them refuses.*

The one place that cannot be bypassed: Every route to the data meets the schema: the page written in PHP, the clerk typing at a client, the import script somebody wrote in an afternoon. A rule kept in one program holds until the second program, and there is always a second program.

The declaration that refuses nothing: DEFAULT supplies a value where none was given instead of rejecting the row, and it belongs to the same idea. What a row contains is settled by the table rather than by whoever wrote the form that filled it in.

A difference in kind: Without constraints a table can hold rows its own design calls impossible: two crates with one identity, a crate belonging to no farm, a count of minus four trays. Declaring them makes those rows unwritable rather than merely unwelcome, which is not a difference of tidiness.

Data outlives the programs that wrote it, and a rule kept in the schema outlives them with it.

The Master Key: *A constraint is a rule declared in the table and enforced by the server on every write, so it holds for every program and every route, which is what no rule kept in application code can promise.*

BINDER Question 101

The farm codes at Bihawana all begin with a zero, and on the day they were typed into a column made for numbers the zeros stopped appearing. Nothing was reported to anybody. The column had done exactly what a number column does.

Set out the data types a schema needs, saying what each one is for, and state why money takes DECIMAL and why a code with leading zeros takes CHAR.

Model Answer

Choose the type that says what the value is: INT and BIGINT for whole numbers that are counted, DECIMAL for money and for any measured quantity, CHAR and VARCHAR for text and for codes, DATE, TIME, DATETIME and TIMESTAMP for the four different things people call a date, and BOOLEAN for a flag with two states.

Whole numbers, and how big: INT holds any whole number from -2147483648 to 2147483647, which is more than a supplier will ever count in trays or crates. BIGINT reaches 9223372036854775807 and earns its extra space only where a count genuinely passes two thousand million.

Type	What it holds	Use it for
INT	whole numbers, -2147483648 to 2147483647	counts, and identifiers the supplier issues itself
BIGINT	whole numbers as far as 9223372036854775807	a count that really does pass two thousand million
DECIMAL(m,n)	m digits exactly, n of them after the point	money, weights, any measured quantity
CHAR(n)	exactly n characters, every time	a code of fixed width: a farm code, a crate label
VARCHAR(n)	up to n characters, stored as given	a shop name, a village, anything of unpredictable length
DATE	a day, as 2026-08-04	the day a crate was collected
TIME	a time of day, or a length of time	the hour a lorry left, hours a driver worked
DATETIME	a day and a time, as 2026-08-04 06:30:00	the moment a collection was made
TIMESTAMP	a day and a time held against UTC	when a row itself was last changed
BOOLEAN	0 or 1, kept as TINYINT(1)	a two-way flag: paid or not paid

Table 4.22: *The types this supplier's schema needs, and what each one is for.*

Money and measured quantities: DECIMAL(m,n) keeps m digits with n of them after the point, exactly as written, so DECIMAL(8,2) leaves six digits before the point and two after. FLOAT keeps a binary approximation instead: a price goes in, a slightly different number comes back, and two amounts that should be equal are not.

A code is text, not a number: A farm code, a telephone number and a crate label are never added, averaged or subtracted, so they are text. An integer column strips a leading zero as it stores the value, and the farm code 004512 becomes 4512 with nothing reported to anyone.

The four types for time: DATE is a day. TIME is a time of day or a length of time. DATETIME is a day and a time together. TIMESTAMP is a day and a time held against UTC and shown in the session time zone, and it is the one that can set itself when a row changes. Students are often taught, wrongly, that DATE and TIME are one type between them.

The type is the first constraint on a column, and it is chosen once and lived with for years.

The Master Key: *Match the type to what the value is: DECIMAL for money because FLOAT is an approximation, CHAR for a code because INT strips its leading zeros, and one of four distinct types wherever a date or a time is meant.*

REAL Question 102

Nobody has read the statement that made the order table since the afternoon it was typed at Isimani. The server accepted every word of it without a murmur, and five separate faults went into the supplier's schema unopposed.

The statement below created the supplier's order table and the server raised no objection to any part of it. Name each fault, say exactly what the server does about it, and give the corrected statement.

```
CREATE TABLE egg_order (
order_no INT,
farm_code INT,
trays INT,
price FLOAT,
placed DATE
```

);

Model Answer

The server does nothing about any of it, and that is the whole trouble: every one of the five faults is legal SQL, so the table is created exactly as written and each fault turns up later as data nobody can trust rather than as a message somebody sees.

Nothing identifies a row: With no PRIMARY KEY the same order can be entered twice and both rows stay, and no other table can ever point at this one, because a foreign key must reference a key. MariaDB adds none of its own. Declare order_no INT NOT NULL AUTO_INCREMENT and PRIMARY KEY (order_no).

A code typed as a number: farm_code is given 004512 and INT stores four thousand five hundred and twelve. Both leading zeros are gone at the moment of the insert, with no warning, since nothing was lost as far as arithmetic goes. Declare it CHAR(6), and make it a FOREIGN KEY referencing farm.

Money typed as FLOAT: A price of 12345.67 goes into the FLOAT column and the server hands back 12345.7, while the total of that one row reads 12345.669921875 and a test of price against 12345.67 comes out false. Declare DECIMAL(8,2), which stores the figure typed and no other.

A day where a moment is meant: Collections are made twice a day, and DATE keeps only the day. The server truncates the time, notes the truncation and stores the row, so the morning collection and the afternoon one become the same value and cannot be told apart afterwards. Declare DATETIME.

Nothing is required: Every column will take NULL, so a row with no farm, no count and no price is a legal row and the totals quietly disagree with the sheets. Declare NOT NULL on each column that must have a value, which here is all of them.

The fault	What the server does	The correction
No PRIMARY KEY	creates the table; accepts duplicate rows for ever	order_no INT NOT NULL AUTO_INCREMENT, PRIMARY KEY (order_no)
farm_code INT	stores 004512 as 4512, silently	farm_code CHAR(6) NOT NULL, and a FOREIGN KEY to farm
price FLOAT	stores an approximation; 12345.67 comes back as 12345.7	price DECIMAL(8,2) NOT NULL
placed DATE	truncates the time part and keeps the row	placed DATETIME NOT NULL
No NOT NULL	accepts a row with nothing in it	NOT NULL on every column that must have a value

Table 4.23: *Each fault, what the server does about it, and what the column should have said.*

Not one of these is a syntax error, which is why the statement has survived so long, and why the damage is in the data rather than on the screen.

The Master Key: *A statement the server accepts is not a statement that is right: a missing key, a code in an integer, money in a FLOAT and a moment in a DATE are all legal, and all of them are paid for later.*

BUILD Question 103

Zero trays, and the crate was still booked in. It happened twice last month at Msalato, both times towards the end of a long day, and both times the store keeper found it a week later while counting back through the sheets.

Create the crate table for the poultry supplier with a primary key, a unique crate label, a farm code and a count that may not be null, and a check that a crate carries at least one tray, then show the server refusing a crate of no trays.

crate_id identifies the crate and is issued by the server itself.

label is stencilled on the crate, and no two crates carry the same one.

farm_code and tray_count must both have a value.

A crate carries at least one tray.

collected is today unless a date is given.

Then attempt a crate of no trays.

Model Answer

The four constraints below do four different jobs inside one statement: PRIMARY KEY makes crate_id the identity, UNIQUE refuses a second crate stencilled with a label already in use, NOT NULL requires a farm and a count, and CHECK refuses a crate of no trays, which is what the last insert runs into.

```
 1  CREATE TABLE crate (
 2    crate_id   INT     NOT NULL AUTO_INCREMENT,
 3    label      CHAR(8) NOT NULL,
 4    farm_code  CHAR(6) NOT NULL,
 5    tray_count INT     NOT NULL,
 6    collected  DATE    NOT NULL DEFAULT (CURRENT_DATE),
 7    PRIMARY KEY (crate_id),
 8    CONSTRAINT label_once UNIQUE (label),
 9    CONSTRAINT trays_at_least_one CHECK (tray_count > 0)
10  ) ENGINE = InnoDB;
11
12  INSERT INTO crate (label, farm_code, tray_count) VALUES
13    ('MSA-0141', '004512', 30),
14    ('MSA-0142', '004512', 24);
15
16  INSERT INTO crate (label, farm_code, tray_count) VALUES
17    ('BIH-0007', '004513', 0);
```

The server refuses it

```
ERROR 4025 (23000) at line 16: CONSTRAINT `trays_at_least_one` failed for
`poultry_supply`.`crate`
```

Program 4.9: *Four constraints in one table, and the check refusing a crate of no trays.*

Which declaration refuses which row: PRIMARY KEY refuses a second crate_id, UNIQUE a second MSA-0141, NOT NULL a crate with no farm code, and CHECK a count of zero or less. Four rules, one statement, and not one of them written into a program.

Reading the refusal: The server names the constraint that failed and the table it belongs to, and stores nothing of the row: the two crates written before it are untouched, and the count that was refused is not sitting anywhere waiting to be found.

Two decisions worth naming: farm_code is CHAR(6) because a farm code is never arithmetic and an integer would strip its leading zeros. collected takes DEFAULT (CURRENT_DATE), so a crate booked in without a date is dated by the server rather than left empty.

The rule that a crate carries trays now lives in the table, where the tired clerk at the end of a long day cannot get past it.

The Master Key: *Declare the identity, the uniqueness, the required columns and the sensible range in the CREATE TABLE, and the row that breaks any of them is refused before it exists.*

BUILD Question 104

A crate reaches the depot carrying a farm code nobody recognises. Somewhere between Bihawana and the sorting shed a digit changed, and the row it produced would point at a farm that has never existed.

Create the farm table and the crate table joined by a foreign key, and show the server refusing a crate whose farm code matches no farm.

farm is keyed by farm_code, a six-character code.

crate carries the farm_code of the farm it came from, and it must have one.

A crate may name only a farm that exists.

Book in a crate from farm 004512, then one from farm 009999.

Model Answer

The FOREIGN KEY in crate names farm_code as a reference to farm, so InnoDB checks the parent before it writes any child row, and the crate from 009999 is refused with ERROR 1452 because no farm carries that code.

```
 1  CREATE TABLE farm (
 2    farm_code CHAR(6)     NOT NULL,
 3    village   VARCHAR(20) NOT NULL,
 4    PRIMARY KEY (farm_code)
 5  ) ENGINE = InnoDB;
 6
 7  CREATE TABLE crate (
 8    crate_id   INT     NOT NULL AUTO_INCREMENT,
 9    farm_code  CHAR(6) NOT NULL,
10    tray_count INT     NOT NULL,
11    PRIMARY KEY (crate_id),
12    CONSTRAINT crate_from_farm
13      FOREIGN KEY (farm_code) REFERENCES farm (farm_code)
14  ) ENGINE = InnoDB;
15
16  INSERT INTO farm (farm_code, village) VALUES
17    ('004512', 'Msalato'), ('004513', 'Bihawana');
18  INSERT INTO crate (farm_code, tray_count) VALUES ('004512', 30);
19  INSERT INTO crate (farm_code, tray_count) VALUES ('009999', 24);
```

The server refuses it

```
ERROR 1452 (23000) at line 19: Cannot add or update a child row: a foreign key constraint
fails (`poultry_supply`.`crate`, CONSTRAINT `crate_from_farm` FOREIGN KEY (`farm_code`)
REFERENCES `farm` (`farm_code`))
```

Program 4.10: *A foreign key under InnoDB, refusing a crate whose farm does not exist.*

What the declaration says: The column named in FOREIGN KEY must hold a value that already appears in the referenced key of the parent, and the referenced column has to be a key, which is why farm_code is the primary key of farm rather than an ordinary column.

Reading the refusal: The server names the database, the child table, the constraint and both columns, and the crate is not written. The two farms and the first crate are already stored and stay stored; only the offending statement is undone.

Where the check runs: This is referential integrity enforced by the server on every insert and every update of crate, at the cost of a look at farm each time. No program was asked to check first, and no program can decline to.

A crate from a farm that does not exist is now a statement the server will not carry out, rather than a row somebody has to find.

The Master Key: *A FOREIGN KEY declares that a child value must already exist as a key value in the parent, and under InnoDB an orphan is refused with ERROR 1452.*

BUILD Question 105

An older machine in the office at Isimani took the crate whose farm was missing. The supplier's own server, given the identical two statements not a character apart, refused it outright. Both were told to enforce the same foreign key.

Create the same two tables under the MyISAM engine, run the identical orphan insert, and show with a SELECT what the server did with it.

The tables, the foreign key and the two inserts are exactly those of the previous question.

Only the storage engine is different.

Then list the crates, and ask the server which engine each table is using.

Model Answer

MyISAM parses the FOREIGN KEY clause, keeps it as an index and throws the rule itself away, so the identical insert is accepted in silence and the second crate sits in the table naming a farm that does not exist.

```
 1
 2  CREATE TABLE farm (
 3    farm_code CHAR(6)     NOT NULL,
 4    village   VARCHAR(20) NOT NULL,
 5    PRIMARY KEY (farm_code)
 6  ) ENGINE = MyISAM;
 7
 8  CREATE TABLE crate (
 9    crate_id   INT     NOT NULL AUTO_INCREMENT,
10    farm_code  CHAR(6) NOT NULL,
11    tray_count INT     NOT NULL,
12    PRIMARY KEY (crate_id),
13    CONSTRAINT crate_from_farm
14      FOREIGN KEY (farm_code) REFERENCES farm (farm_code)
15  ) ENGINE = MyISAM;
16
17  INSERT INTO farm (farm_code, village) VALUES
18    ('004512', 'Msalato'), ('004513', 'Bihawana');
19  INSERT INTO crate (farm_code, tray_count) VALUES ('004512', 30);
20  INSERT INTO crate (farm_code, tray_count) VALUES ('009999', 24);
21
22  SELECT c.crate_id, c.farm_code, c.tray_count
23  FROM crate c
24  ORDER BY c.crate_id;
25  SELECT t.table_name, t.engine
26  FROM information_schema.tables t
27  WHERE t.table_schema = 'poultry_supply'
28  ORDER BY t.table_name;
```

The server replies

```
+----------+-----------+------------+
| crate_id | farm_code | tray_count |
+----------+-----------+------------+
|        1 | 004512    |         30 |
|        2 | 009999    |         24 |
+----------+-----------+------------+
+------------+--------+
| table_name | engine |
+------------+--------+
| crate      | MyISAM |
| farm       | MyISAM |
+------------+--------+
```

Program 4.11: *The same foreign key under MyISAM, which accepts the orphan without a word.*

What the second result proves: The engine is asked for by name and both tables answer MyISAM, so nothing here turns on a typing error. The same declaration that produced ERROR 1452 a page ago has produced a stored orphan, and the only thing changed is the engine.

Accepted, not reported: There is no error, no warning and no note. Ask MyISAM afterwards what it made of the clause and it shows an ordinary index on farm_code and no constraint at all: the rule was read, dropped, and never mentioned again.

The rule this settles: Referential integrity is a property of the engine, not of the SQL. Declare InnoDB on every table that has a foreign key or is referenced by one, whether by the ENGINE clause or by the session default, and check the engine on any database inherited from somebody else.

The frightening part is the silence: a schema can carry every constraint on paper and enforce not one of them.

The Master Key: *MyISAM keeps a foreign key as an index and discards the rule, so the orphan it accepts in silence is the reason InnoDB is the only engine to build referential integrity on.*

BUILD Question 106

Shop names started arriving longer than the column that held them, and the supplier now wants a credit limit kept against each customer as well. Nobody is going to type the schema out again, and nobody has to.

Add a column to an existing customer table, widen the type of another, add a constraint to it, and show the structure the server ends up with.

Add credit_limit, money, which is zero until somebody sets it.

Widen shop_name from 30 characters to 60.

One telephone number belongs to one customer and no other.

Then show the structure of the altered table.

Model Answer

ALTER TABLE changes a table that is already carrying rows: ADD COLUMN gives every existing customer the default of 0.00 immediately, MODIFY COLUMN restates shop_name as VARCHAR(60), ADD CONSTRAINT puts a UNIQUE on phone, and DESCRIBE shows all three changes in place.

```
 1  CREATE TABLE customer (
 2    customer_id INT         NOT NULL AUTO_INCREMENT,
 3    shop_name   VARCHAR(30) NOT NULL,
 4    phone       CHAR(10)    NOT NULL,
 5    PRIMARY KEY (customer_id)
 6  ) ENGINE = InnoDB;
 7
 8  INSERT INTO customer (shop_name, phone) VALUES
 9    ('Duka la Mayai Bihawana', '0713004512');
10
11  ALTER TABLE customer
12    ADD COLUMN credit_limit DECIMAL(8,2) NOT NULL DEFAULT 0.00;
13
14  ALTER TABLE customer
15    MODIFY COLUMN shop_name VARCHAR(60) NOT NULL;
16
17  ALTER TABLE customer
18    ADD CONSTRAINT phone_once UNIQUE (phone);
19
20  DESCRIBE customer;
```

The server replies

```
+--------------+--------------+------+-----+---------+----------------+
| Field        | Type         | Null | Key | Default | Extra          |
+--------------+--------------+------+-----+---------+----------------+
| customer_id  | int(11)      | NO   | PRI | NULL    | auto_increment |
| shop_name    | varchar(60)  | NO   |     | NULL    |                |
| phone        | char(10)     | NO   | UNI | NULL    |                |
| credit_limit | decimal(8,2) | NO   |     | 0.00    |                |
+--------------+--------------+------+-----+---------+----------------+
```

Program 4.12: *Three alterations to a table that already holds a row, and the structure they leave.*

Reading the structure: The shop already stored is not asked about any of it: shop_name is now varchar(60), phone is marked UNI for the constraint just added, and credit_limit stands at 0.00 on that row, because a NOT NULL column added to a table with rows in it needs a DEFAULT for the rows already there.

The trap in MODIFY: MODIFY COLUMN replaces the whole definition rather than editing part of it, so NOT NULL has to be written again. Leave it out and the column silently becomes nullable, and a rule the table had this morning is gone by lunchtime.

Taking things away again: DROP COLUMN removes a column and everything stored in it, and DROP TABLE removes the definition as well. The server refuses an alteration that would break a constraint: dropping the index behind a foreign key gives ERROR 1553.

A schema is altered far more often than it is created, and every alteration is a chance to lose a rule nobody notices is missing.

The Master Key: *ALTER TABLE adds, changes and constrains a table in place, and because MODIFY restates the whole column definition, every attribute that must survive has to be written into it again.*

BUILD Question 107

A delivery to Isimani was cancelled before the lorry left, and the note was removed from the system that afternoon. Its lines of crates were still there the next morning, counted into a total that nobody could explain.

Declare a foreign key with ON DELETE and ON UPDATE actions for the lines of a delivery note, delete a cancelled note, renumber another, and show with a SELECT what became of the lines.

delivery is keyed by delivery_id; delivery_line is keyed by delivery_id and line_no together.

A line has no meaning apart from the note it belongs to.

Enter note D00311 with two lines and note D00312 with one.

Delete D00311, renumber D00312 as D00399, then list the lines.

Model Answer

ON DELETE CASCADE and ON UPDATE CASCADE are the actions this relationship calls for, because a line of a delivery note has no existence away from the note: deleting D00311 takes its two lines with it, and renumbering D00312 carries its line across to D00399.

```
 1  CREATE TABLE delivery (
 2    delivery_id CHAR(6)     NOT NULL,
 3    village     VARCHAR(20) NOT NULL,
 4    PRIMARY KEY (delivery_id)
 5  ) ENGINE = InnoDB;
 6
 7  CREATE TABLE delivery_line (
 8    delivery_id CHAR(6)  NOT NULL,
 9    line_no     TINYINT  NOT NULL,
10    trays       SMALLINT NOT NULL,
11    PRIMARY KEY (delivery_id, line_no),
12    CONSTRAINT line_from_delivery
13      FOREIGN KEY (delivery_id) REFERENCES delivery (delivery_id)
14      ON DELETE CASCADE
15      ON UPDATE CASCADE
16  ) ENGINE = InnoDB;
17
18  INSERT INTO delivery (delivery_id, village) VALUES
19    ('D00311', 'Isimani'), ('D00312', 'Msalato');
20  INSERT INTO delivery_line (delivery_id, line_no, trays) VALUES
21    ('D00311', 1, 60), ('D00311', 2, 45), ('D00312', 1, 80);
22
23  DELETE FROM delivery WHERE delivery_id = 'D00311';
24  UPDATE delivery SET delivery_id = 'D00399'
25  WHERE delivery_id = 'D00312';
26
27  SELECT l.delivery_id, l.line_no, l.trays
28  FROM delivery_line l
29  ORDER BY l.delivery_id, l.line_no;
```

The server replies

```
+-------------+---------+-------+
| delivery_id | line_no | trays |
+-------------+---------+-------+
| D00399      |       1 |    80 |
+-------------+---------+-------+
```

Program 4.13: *Cascading actions firing on a deleted note and on a renumbered one.*

What the result shows: Of the three lines written, one remains. The two belonging to the cancelled note went with it, and the survivor is no longer filed under D00312 but under D00399, which is the update action firing. Neither statement mentioned delivery_line.

Why cascade is right here: A line of a delivery note is part of that note and describes nothing on its own, so leaving it behind creates a total belonging to no delivery. Where the child does stand alone, a crate outliving the round it went out on, RESTRICT is the honest action and the parent cannot be deleted while children remain.

The other choices: SET NULL empties the child column instead, which needs a nullable column and is for a child that survives its parent. NO ACTION and RESTRICT both refuse the delete. Declaring nothing gives RESTRICT, so a foreign key never silently orphans anything under InnoDB.

The action is declared once, in the table, and fires on every delete and every renumbering thereafter, including the ones nobody remembers to think about.

The Master Key: *Choose the referential action from the domain: CASCADE where the child is part of the parent, RESTRICT where it stands on its own, and SET NULL only where the child may outlive the parent.*

HOT Question 108

Two programmers at Bihawana keep one rule in two places, one of them in the schema and the other in the code that writes to it, and neither will give his copy up. The rule they are arguing over has never once been broken.

Judge whether referential integrity should be enforced by the database or by the application, and state the condition that decides it.

Model Answer

Enforce it in the database and treat whatever the application checks as a courtesy to the user rather than as the guarantee; the condition is whether every write to the data, for as long as the data lasts, can be guaranteed to pass through code you control, and outside a single program that nothing else ever touches it cannot.

What the application does better: It can say what went wrong in words a shopkeeper understands, refuse the entry before anything is sent, and enforce rules no CHECK can express, such as a crate being refused after the round has closed unless a supervisor has signed for it. None of that is in dispute.

The routes nobody counted: Data is written by more than the program it was designed for: an import of last season's crates, a report tool with an edit screen, a clerk at the client, and whatever replaces the application in four years. The rule has to be written into each of them, and one that forgets is enough.

The check the database cannot avoid making: InnoDB reads the parent on every insert and update of the child and locks the row it read. That is real work, and it is the honest argument for dropping the constraint where writes are the measured bottleneck, or where the parent has been put on a different server from the child.

Where each message belongs: ERROR 1452 is not a sentence to show a customer. The application catches it and prints something a person can act on, which is the division of labour rather than a duplication of it: the server decides whether the row is allowed, the program decides what the user is told.

What silence costs when it is discovered: Orphans do not announce themselves. Crates naming farms nobody has heard of accumulate for a season, and the foreign key added afterwards is refused until every one of them has been found and dealt with by hand, which is a week of somebody's life.

The judgement: Declare the foreign key in the database, on InnoDB, and keep the application check for the message it can give. Give up the database copy only where one program owns the data, nothing else can reach it, and the cost has been measured rather than assumed, and write down what is now unenforced.

The database is the last thing standing when the programs have been rewritten, so the rules that matter most are the ones kept where it can see them.

The Master Key: *Enforce referential integrity in the database unless a single program provably owns every write for the whole life of the data, and let the application check the same rule only so that it can explain itself.*

Inserting, updating and deleting rows

Everything a database holds arrived through an INSERT, and everything it has stopped holding left through a DELETE, a TRUNCATE or a DROP. This section writes rows in, changes rows that are already stored, and takes rows away again, and it spends most of its care on the clause that decides which rows a change touches. A WHERE clause that is nearly right is the most expensive fault in this chapter, because the server carries it out without complaint and reports success either way, so the habit taught here is to run the SELECT before the UPDATE. The three statements that remove things are separated here too, because they are not three ways of doing one job.

BINDER Question 109

The reading sheets from the Migoli kiosk arrive in a folder each Monday, and by Tuesday every line on them is a row in the database. The statement that puts them there can be written two ways, and only one survives a new column.

Explain the two forms of INSERT, say why naming the columns is the safer habit, and state what the server puts in a column that is left out.

Model Answer

INSERT has two forms: the first names the columns it is filling and gives a value for each of them, and the second omits that list and must then supply a value for every column of the table in the order the table declares them, which is why naming the columns is the safer habit.

Naming the columns: The list in brackets after the table name says which column each value belongs to, so the statement no longer depends on the order the columns were declared in. A reader can see what is being filled and what is left to the server.

Omitting the list: With no list the server matches values to columns by position, so every column must be given a value, in the declared order. Add a column later and the statement draws ERROR 1136, column count does not match value count; reorder two columns of the same type and each value lands in the wrong place without a word.

Many rows in one statement: One INSERT can carry several rows, each in its own brackets, separated by commas after a single VALUES keyword. The server does the work once instead of once per row, and if any row is refused none of them is stored.

What arrives in a column left out: An AUTO_INCREMENT column takes the next number from the counter the table keeps, a column with a DEFAULT takes that default, and a column that allows NULL and has no default takes NULL. A NOT NULL column with no default is filled in for nobody: the row is refused with ERROR 1364.

Naming the columns costs one line and buys a statement that still means what it said after somebody has altered the table.

The Master Key: *Name the columns in every INSERT, because the form that omits the list depends on the number and the order of the columns, and a change to either turns a working statement into an error or into a row of values in the wrong columns.*

BINDER Question 110

Only one of the three statements that remove things is about rows. A clerk at Ifunda who wanted last month's sales gone typed the shortest of them, and by the afternoon the table itself was no longer in the database.

Compare DELETE, TRUNCATE and DROP: what each one removes, what each does to the AUTO_INCREMENT counter, and whether each can be undone.

Model Answer

DELETE removes the rows its WHERE clause chooses and leaves everything else standing, TRUNCATE removes every row and sets the AUTO_INCREMENT counter back to 1, and DROP removes the table itself with its columns, its keys and its constraints, so only the first of the three is a statement about rows.

What each one removes: DELETE removes the rows the WHERE clause picks out, and every row if there is no WHERE clause. TRUNCATE removes every row and takes no WHERE clause at all. DROP removes the definition as well, so a SELECT afterwards answers ERROR 1146, the table does not exist.

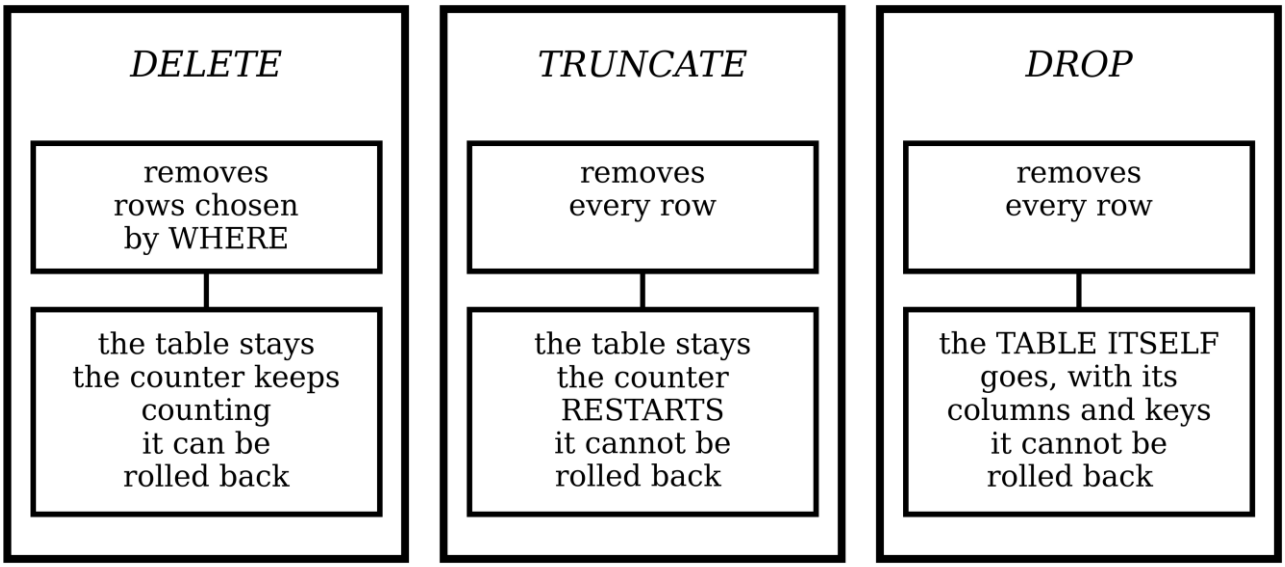


Figure 4.24: *What DELETE, TRUNCATE and DROP each remove, and what each leaves behind.*

The counter underneath: An AUTO_INCREMENT column counts up from a figure the table keeps for itself. DELETE never touches it, so a table emptied that way carries on numbering where it stopped. TRUNCATE sets it back to 1, and DROP takes it away with everything else.

Whether it can be undone: DELETE is the only one of the three a transaction can take back; TRUNCATE and DROP commit as they run and no rollback reaches them. Under the usual setting a change takes effect the moment the statement finishes unless a transaction was started first.

Which one the job wants: Reach for DELETE with a WHERE clause to remove some rows, TRUNCATE to empty a table you mean to keep filling, and DROP only when the table itself is finished with. TRUNCATE is refused on a table another table points at, with ERROR 1701.

Two of the three cannot be taken back, which is reason enough to read the statement once more before it runs.

The Master Key: *DELETE removes chosen rows, keeps the counter and can be rolled back; TRUNCATE removes every row, sets the counter back to 1 and cannot; DROP removes the table itself, so these are three different statements and not three ways of removing data.*

REAL Question 111

Three statements are written on a card taped above the counter at Kising'a, where the attendant can see them while she types. Two of them are wrong, and the one that is right is the one that sounds most alarming.

Exactly one of the three statements below is correct. Say which, name the fault in each of the other two, state exactly what the server does, and give the correction.

A. Backticks may be used around a value in an INSERT when the value contains digits only.

B. An UPDATE with no WHERE clause changes every row in the table.

C. TRUNCATE and DELETE without a WHERE clause do exactly the same thing.

Model Answer

B is the correct one, and it is correct exactly as written: an UPDATE with no WHERE clause changes every row in the table, and the server carries it out and reports success. A is wrong because backticks quote identifiers and never values, and C is wrong twice over, on the counter and on whether the statement can be undone.

Why A is wrong: Backticks are how MySQL and MariaDB quote an identifier, the name of a table or a column. Put them round 1234 in a VALUES list and the server goes looking for a column of that name, finds none and answers ERROR 1054, unknown column 1234 in VALUES. A number needs no marks around it at all, and text needs the ordinary single quote mark.

Why B is right, and what follows from it: The WHERE clause is optional, and its absence means every row. The server does not ask whether that was intended; it changes them all and reports the number changed as though each one were meant. That is why the SELECT with the same WHERE clause comes first, while the rows can still be counted and read.

Why C is wrong on the counter: DELETE with no WHERE clause empties the table and leaves the AUTO_INCREMENT counter exactly where it stopped, so the next row carries on from there. TRUNCATE sets the counter back to 1. Empty a table of 3 rows each way and insert one, and the ids that come back are 4 and 1.

Why C is wrong on undoing: DELETE is a statement about rows and a transaction can take it back. TRUNCATE commits as it runs, and nothing takes it back afterwards. A statement that cannot be undone and a statement that can are not the same thing, whatever they leave in the table.

Written on the card	What the server actually does	The correction
A. Backticks round a value of digits	reads 1234 as a column name; ERROR 1054, unknown column 1234 in VALUES	write the number plain, with no marks round it
B. No WHERE clause changes every row	changes every row and reports success	correct as it stands; run the SELECT first
C. TRUNCATE is the same as DELETE	TRUNCATE also resets the counter and cannot be undone	they differ on the counter and on undoing

Table 4.24: *The three written statements, what the server does about each, and what each should say.*

The alarming statement is the true one, and the two that read like sensible shortcuts are the pair that cost data.

The Master Key: *Backticks name columns and never values, an UPDATE with no WHERE clause changes every row and says so cheerfully, and TRUNCATE differs from DELETE on the counter and on whether anything can be undone.*

BUILD Question 112

Nobody typed a number into the id column at Migoli, and nobody typed a tariff either, yet both are filled in on every row stored last week. The server put them there, and it will say what it put in if it is asked.

Create the reading table for the kiosk meters, insert rows both by naming the columns and by omitting the list, and show with a SELECT what the AUTO_INCREMENT and DEFAULT columns received.

reading_id identifies the reading and is issued by the server.

Every reading names its kiosk, the day it was taken and the volume in cubic metres.

The tariff is 50.00 a cubic metre unless another figure is given.

Insert one reading, then two more in a single statement, naming the columns each time.

Then insert a fourth without naming any columns at all.

Model Answer

The first three inserts name the columns they fill, so reading_id and tariff are left to the server and come back as 1, 2 and 3 at the default tariff of 50.00; the fourth names no columns, which obliges it to supply all 5 values in the declared order, the id and the tariff among them.

```
 1  CREATE TABLE reading (
 2    reading_id INT          NOT NULL AUTO_INCREMENT,
 3    kiosk_id   INT          NOT NULL,
 4    taken_on   DATE         NOT NULL,
 5    volume_m3  DECIMAL(8,3) NOT NULL,
 6    tariff     DECIMAL(6,2) NOT NULL DEFAULT 50.00,
 7    PRIMARY KEY (reading_id)
 8  ) ENGINE = InnoDB;
 9
10  INSERT INTO reading (kiosk_id, taken_on, volume_m3)
11  VALUES (1, '2026-08-03', 12.500);
12
13  INSERT INTO reading (kiosk_id, taken_on, volume_m3) VALUES
14    (2, '2026-08-03',  9.250),
15    (3, '2026-08-03', 15.750);
16
17  INSERT INTO reading
18  VALUES (4, 2, '2026-08-04', 10.000, 60.00);
19
20  SELECT reading_id, kiosk_id, taken_on, volume_m3, tariff
21  FROM reading
22  ORDER BY reading_id;
```

The server replies

```
+------------+----------+------------+-----------+--------+
| reading_id | kiosk_id | taken_on   | volume_m3 | tariff |
+------------+----------+------------+-----------+--------+
|          1 |        1 | 2026-08-03 |    12.500 |  50.00 |
|          2 |        2 | 2026-08-03 |     9.250 |  50.00 |
|          3 |        3 | 2026-08-03 |    15.750 |  50.00 |
|          4 |        2 | 2026-08-04 |    10.000 |  60.00 |
+------------+----------+------------+-----------+--------+
```

Program 4.14: *Both forms of INSERT, and what the counter and the default put in the columns nobody filled.*

What the server supplied: Three statements mention neither reading_id nor tariff, and all three rows have both. The counter issued 1, 2 and 3, and the DEFAULT clause wrote 50.00 on each of them.

What the fourth form had to know: Omitting the list obliged it to know that the table has 5 columns and the order they stand in, and to issue the id itself. The counter would have done that unasked, which is one job fewer to get wrong.

The decision worth naming: volume_m3 is DECIMAL(8,3) rather than FLOAT, because a metered volume is a measured quantity and has to come back exactly as it went in. The tariff is money, so it is DECIMAL as well.

Leaving a column out is a decision, not an omission, and the table settles what happens next.

The Master Key: *Name the columns and the server fills the rest from the counter and the defaults it was given; omit the list and every column becomes your responsibility, in the order the table happens to declare them.*

BUILD Question 113

Two backticks turned a meter number into a column name. The statement looks ordinary, the value is plainly a number, and the server answers with a complaint about a column that nobody has ever declared.

Insert a reading whose meter number is written in backticks, show exactly what the server does with it, and give the statement that works.

Model Answer

Backticks quote identifiers in MySQL and MariaDB and never values, so the server reads 1234 as the name of a column, finds no column of that name in reading and refuses the whole statement with ERROR 1054; written plain, with no marks round it at all, the same number is stored as the key it was meant to be.

```
1  INSERT INTO reading (reading_id, kiosk_id, taken_on, volume_m3)
2  VALUES (`1234`, 2, '2026-08-05', 8.250);
```

The server refuses it

```
ERROR 1054 (42S22) at line 1: Unknown column '1234' in 'VALUES'
```

Program 4.15: *A meter number in backticks, which the server reads as a column name.*

```
1  INSERT INTO reading (reading_id, kiosk_id, taken_on, volume_m3)
2  VALUES (1234, 2, '2026-08-05', 8.250);
3
4  SELECT reading_id, kiosk_id, taken_on, volume_m3, tariff
5  FROM reading
6  WHERE reading_id = 1234;
```

The server replies

```
+------------+----------+------------+-----------+--------+
| reading_id | kiosk_id | taken_on   | volume_m3 | tariff |
+------------+----------+------------+-----------+--------+
|       1234 |        2 | 2026-08-05 |     8.250 |  50.00 |
+------------+----------+------------+-----------+--------+
```

Program 4.16: *The same statement with the number written plain, and the row it stores.*

What the server was asked for: Everything inside a VALUES list is a value except an identifier, and an identifier there can only be a column of the same row. The message names what it hunted for and where it hunted, and no part of the row was stored.

The three kinds of marks: A backtick protects a name that needs protecting, one carrying a space or a reserved word. A single quote mark encloses text and dates. A number takes neither, and the tariff and the volume in the second listing take none either.

Why the fault is easy to make: Editors and printed notes often set every identifier in backticks, and a hand that has typed them round a table name all afternoon types them round 1234 as well. The result is not a wrong number stored, which is the mercy in it.

The statement is refused whole, so the correction costs one retyping and nothing else.

The Master Key: *Backticks quote identifiers and never values, so a number wrapped in them becomes a column name and draws ERROR 1054, unknown column, rather than the row that was intended.*

BUILD Question 114

The tariff at the Ifunda kiosk went up on Monday and the readings already entered still carry the old figure. The clerk about to type the UPDATE can find out which rows it will touch while they can still be read.

Raise the tariff on every reading from the Ifunda kiosk, running the SELECT that tests the WHERE clause before the UPDATE that uses it, and show how many rows changed.

The reading table already holds 5 readings from 3 kiosks.

Ifunda is kiosk 2.

Its tariff rises from 50.00 to 65.00 a cubic metre.

Run the SELECT that tests the WHERE clause first, then the UPDATE, then read the count.

Model Answer

The SELECT carries exactly the WHERE clause the UPDATE will carry, and it returns readings 2 and 4, which are the rows the change is about to touch; the UPDATE then reports 2 rows changed, the same 2 the SELECT displayed, and the other 3 readings are left alone.

```
 1  SELECT reading_id, taken_on, volume_m3, tariff
 2  FROM reading
 3  WHERE kiosk_id = 2;
 4
 5  UPDATE reading
 6  SET tariff = 65.00
 7  WHERE kiosk_id = 2;
 8
 9  SELECT ROW_COUNT() AS rows_changed;
10
11  SELECT reading_id, taken_on, volume_m3, tariff
12  FROM reading
13  WHERE kiosk_id = 2;
```

The server replies

```
+------------+------------+-----------+--------+
| reading_id | taken_on   | volume_m3 | tariff |
+------------+------------+-----------+--------+
|          2 | 2026-08-03 |     9.250 |  50.00 |
|          4 | 2026-08-04 |    10.000 |  50.00 |
+------------+------------+-----------+--------+
+--------------+
| rows_changed |
+--------------+
|            2 |
+--------------+
+------------+------------+-----------+--------+
| reading_id | taken_on   | volume_m3 | tariff |
+------------+------------+-----------+--------+
|          2 | 2026-08-03 |     9.250 |  65.00 |
|          4 | 2026-08-04 |    10.000 |  65.00 |
+------------+------------+-----------+--------+
```

Program 4.17: *The SELECT that tests a WHERE clause, and the UPDATE that then uses it.*

Why the SELECT goes first: An UPDATE reports success whether its WHERE clause was right or wrong, and the rows it changed look no different from the rows it should have changed. The SELECT shows them while they can still be checked against the sheets.

Reading the count: ROW_COUNT returns how many rows the statement just changed, 2 here, which is the number the SELECT displayed. A count larger than expected is the warning that arrives soon enough to matter.

SET and WHERE do different jobs: SET says what changes, WHERE says which rows. A comma left after the last assignment, before WHERE, is a syntax error: the server answers ERROR 1064 and points at the text beginning with WHERE, because it was still expecting another column to set.

The SELECT costs a second and turns an UPDATE from something you hope is right into something you have already seen.

The Master Key: *Run the SELECT with the same WHERE clause first and count the rows it returns, because an UPDATE reports success either way and the rows it changed by mistake look exactly like the rows it was meant to change.*

BUILD Question 115

The DELETE ran, the server said nothing, and the cancelled card was still in the table the next morning. Nothing had gone wrong that anyone could see, and nothing had been removed either.

A card was entered with a space at the end of its number and the DELETE meant to remove it was typed with a space at the front. Show what the server does, find the fault and correct the statement.

Card MIG-041 was entered with a space at the end of the number.

The DELETE was typed with a space at the front of it.

Show how many rows each DELETE removed, and how wide the stored value really is.

Then remove the card properly.

Model Answer

The first DELETE succeeds and removes nothing: ROW_COUNT reports 0, because a space at the front is part of the value being compared and no card number begins with one, while the trailing space the insert stored is ignored by the comparison, so the corrected statement finds the row and removes it.

```
 1  INSERT INTO card (card_no, village) VALUES
 2    ('MIG-041 ', 'Migoli'),
 3    ('IFU-018', 'Ifunda');
 4
 5  DELETE FROM card
 6  WHERE card_no = ' MIG-041';
 7  SELECT ROW_COUNT() AS rows_deleted;
 8
 9  SELECT card_no, CHAR_LENGTH(card_no) AS width
10  FROM card
11  ORDER BY card_no;
12
13  DELETE FROM card
14  WHERE card_no = 'MIG-041';
15  SELECT ROW_COUNT() AS rows_deleted;
16
17  SELECT card_no, CHAR_LENGTH(card_no) AS width
18  FROM card
19  ORDER BY card_no;
```

The server replies

```
+--------------+
| rows_deleted |
+--------------+
|            0 |
+--------------+
+----------+-------+
| card_no  | width |
+----------+-------+
| IFU-018  |     7 |
| MIG-041  |     8 |
+----------+-------+
+--------------+
| rows_deleted |
+--------------+
|            1 |
+--------------+
+---------+-------+
| card_no | width |
+---------+-------+
| IFU-018 |     7 |
+---------+-------+
```

Program 4.18: *A DELETE that succeeds and removes nothing, the width that explains it, and the correction.*

Success is not the same as an effect: The server did what it was told: it looked for rows matching that value and removed all of them, and there were none. Nothing is reported, because nothing went wrong, and ROW_COUNT of 0 is the only sign there is.

Which space bites: Under the usual collation the comparison ignores spaces at the end, so the stored width of 8 where 7 was expected does no harm by itself. A space at the front is an ordinary character and the comparison fails on it, which is the whole fault.

Finding it before it hides: CHAR_LENGTH shows a value wider than it looks, and a SELECT carrying the WHERE clause the DELETE was going to carry would have returned no rows at all, which is the same warning one statement earlier.

A DELETE that removes nothing is the quietest fault in this section, and the row count is the only thing that tells on it.

The Master Key: *A statement that succeeds has not necessarily done anything, so read the row count: a space at the front of a value makes the comparison fail while a space at the end is ignored, and a SELECT run first shows both.*

BUILD Question 116

Empty the table twice, in two different ways, and the next sale is not numbered the same. The rows are gone either way, but the counter underneath them remembers one emptying and forgets the other.

Show the difference DELETE and TRUNCATE make to the AUTO_INCREMENT counter: empty the sales table each way, insert one sale after each, and print the id it received.

Three sales are recorded, and are numbered by the server.

Empty the table with DELETE, insert one sale, and show its id.

Empty it again with TRUNCATE, insert the same sale, and show its id.

The two ids are the answer.

Model Answer

Three sales go in and are numbered 1, 2 and 3; DELETE removes all three and leaves the counter where it stopped, so the next sale is numbered 4; TRUNCATE empties the table and sets the counter back, so the sale inserted after it is numbered 1 again, which is a number that has already been on a receipt.

```
 1  CREATE TABLE sale (
 2    sale_id INT          NOT NULL AUTO_INCREMENT,
 3    card_no VARCHAR(10)  NOT NULL,
 4    litres  DECIMAL(7,2) NOT NULL,
 5    PRIMARY KEY (sale_id)
 6  ) ENGINE = InnoDB;
 7
 8  INSERT INTO sale (card_no, litres) VALUES
 9    ('MIG-041', 20.00),
10    ('IFU-018', 15.00),
11    ('MIG-042', 40.00);
12
13  DELETE FROM sale;
14  INSERT INTO sale (card_no, litres) VALUES ('MIG-041', 20.00);
15  SELECT sale_id AS id_after_delete, card_no
16  FROM sale;
17
18  TRUNCATE TABLE sale;
19  INSERT INTO sale (card_no, litres) VALUES ('MIG-041', 20.00);
20  SELECT sale_id AS id_after_truncate, card_no
21  FROM sale;
```

The server replies

```
+-----------------+---------+
| id_after_delete | card_no |
+-----------------+---------+
|               4 | MIG-041 |
+-----------------+---------+
+------------------+---------+
| id_after_truncate | card_no |
+------------------+---------+
|                 1 | MIG-041 |
+------------------+---------+
```

Program 4.19: *The same sale inserted after a DELETE and after a TRUNCATE, and the two ids it received.*

What the two ids prove: Same table, same card, same insert, and two different numbers: 4 after the DELETE and 1 after the TRUNCATE. The rows removed were identical, so the difference belongs to the statements and not to the data.

Why a repeated id costs something: Receipts, complaints and stock sheets are filed under the sale id. After a TRUNCATE the next sale takes a number that is already written on somebody else's slip, and two different sales answer to it.

The rest of the difference: TRUNCATE takes no WHERE clause and cannot be undone, and it is refused outright on a table that another table points at. DELETE accepts a WHERE clause, which is why it is the statement to reach for while a table is in use.

The rows look equally gone in both tables, and the next row inserted is where the difference finally shows.

The Master Key: *DELETE empties a table and leaves the counter counting, while TRUNCATE empties it and starts the numbering again at 1, so the choice between them is a choice about the identities the table will hand out next.*

HOT Question 117

Every tariff in the Migoli table now reads the same figure, down to the last cell. The line that did it was typed late one night, it was legal, it ran in under a second, and it did exactly what it said.

Judge whether a database should allow an UPDATE or a DELETE with no WHERE clause at all, and state the condition that decides it.

Model Answer

It should allow them, because no server can tell an emptying that was intended from one that was not, and the guard belongs where the difference is visible: on the client a person types into, in safe update mode, with the SELECT run first; the condition is whether a person is typing the statement by hand against live data.

The case for refusing them: Most of the damage here comes from a WHERE clause that was meant to be there and was not: a mistyped statement, a line broken in the wrong place, a paste that lost its last line. A server refusing the bare form would stop the commonest destructive accident there is.

What refusing would cost: Clearing a table before a fresh import, wiping a day of provisional readings, resetting a copy kept for practice: all of them want every row gone and none is a mistake. A rule against the bare form drives people to write a condition that matches everything, which reads as though it had been thought about.

Where the guard actually belongs: The client can refuse what the language allows. In safe update mode an UPDATE or a DELETE with no WHERE clause on a key column, and no LIMIT, is refused with ERROR 1175, and the statement itself is not changed at all. It is a setting on the session, chosen by the person at the keyboard.

The rights the account holds: A connection that only records sales has no business holding the right to delete anything, and a connection that only reads reports has no business holding either. Nothing typed through them can empty a table, whatever the WHERE clause says or fails to say.

The habit no setting can replace: Write the statement as a SELECT, run it, read how many rows come back, and only then change the first word to UPDATE or DELETE. It costs seconds, it works on every server and every client, and it catches the mistyped condition as well as the missing one.

The judgement: Keep the bare statement legal and guard the seat, not the language. Where a person is typing against live data, safe update mode goes on and the SELECT comes first; where the statement lives in a script somebody else has read and that runs the same way every time, it needs neither.

The line that changed every tariff was not an illegal statement, and treating it as one would only teach the office a longer way of typing it.

The Master Key: *Leave UPDATE and DELETE with no WHERE clause legal, because a server cannot tell an intended emptying from an accident, and put the guard on the seat instead: safe update mode wherever a person types against live data, and a SELECT with the same WHERE clause before every change.*

Querying one table: SELECT, filtering, NULL and ordering

A SELECT statement is the only way a stored row ever reaches a person, and most of the practical work in this chapter is made of it. This section takes one table and asks it questions: which columns to show, which rows to keep, in what order to show them, and how many of them to stop at. It spends most of its care on the thing in a

column that is not a value at all, because a row holding NULL falls out of a filter and out of the opposite filter as well, and the two results together no longer add back to the table they came from.

BINDER Question 118

A sheet from the Nduli landing beach carries every basket weighed since dawn, and the buyer wants the sites and nothing else. The statement that fetches them is written in one order and carried out in another.

Explain what a SELECT statement does, name its clauses in the order they must be written, and explain what DISTINCT adds to it.

Model Answer

SELECT names the columns to show and FROM names the table they come from, and any clause that follows is written in one fixed order, WHERE, GROUP BY, HAVING, ORDER BY and LIMIT; DISTINCT removes repeated rows, and it has to be written, because SQL keeps the duplicates that relational projection drops by itself.

The two clauses nothing works without: SELECT lists the columns wanted and FROM names the table to take them from; every other clause is optional. The asterisk asks for all of them, which is quick at the keyboard and a poor habit in a stored statement, because the result changes shape the day a column is added.

The written order is fixed: A clause may be left out but the clauses may not be reordered. Put ORDER BY in front of WHERE and the server answers ERROR 1064 and quotes the text it stopped at. WHERE follows FROM, and LIMIT is always last.

Written order is not working order: The server reads FROM first and the SELECT list late, which is why a name invented in that list can be used by ORDER BY, which runs after it, and not by WHERE, which runs before it.

Projection in SQL keeps the duplicates: Relational projection returns a set, so repeated rows disappear of their own accord. SQL returns one row for every row that qualified, so the 6 landings in this section give back 6 site names where projection would give 3.

What DISTINCT is applied to: DISTINCT is written once, straight after SELECT, and it compares whole rows of the result, every selected column together. Add one more column to the list and rows that had collapsed into one come apart again.

SELECT and FROM are the statement, and everything after them is a decision about what comes back.

The Master Key: *Write the clauses in the order SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY and LIMIT, and remember that SQL keeps duplicate rows unless DISTINCT is written, because a SELECT list is not relational projection.*

BINDER Question 119

The Kibena boat came in before the buyer did, so the tally has a weight and no price. Nothing was typed into that box, and what stands in it is not a value at all, nor is it zero.

Explain what NULL means in a column, and how three-valued logic decides which rows a WHERE clause keeps.

Model Answer

NULL means no value is recorded, so it is not zero, not an empty string and not equal to itself; every comparison written with it comes back unknown rather than true or false, and a WHERE clause keeps only rows whose condition is true, which is why IS NULL exists.

What NULL is not: Zero is a value and it says the fish went for nothing. An empty string is a value too, text of no characters. NULL says the price is not recorded here, a statement about the tally and not about the money.

Three values, not two: A condition is true, false or unknown. On the server, 5 > NULL is NULL, 5 = NULL is NULL and NULL = NULL is NULL as well, since nothing is known about either side; NULL IS NULL answers, and it answers 1.

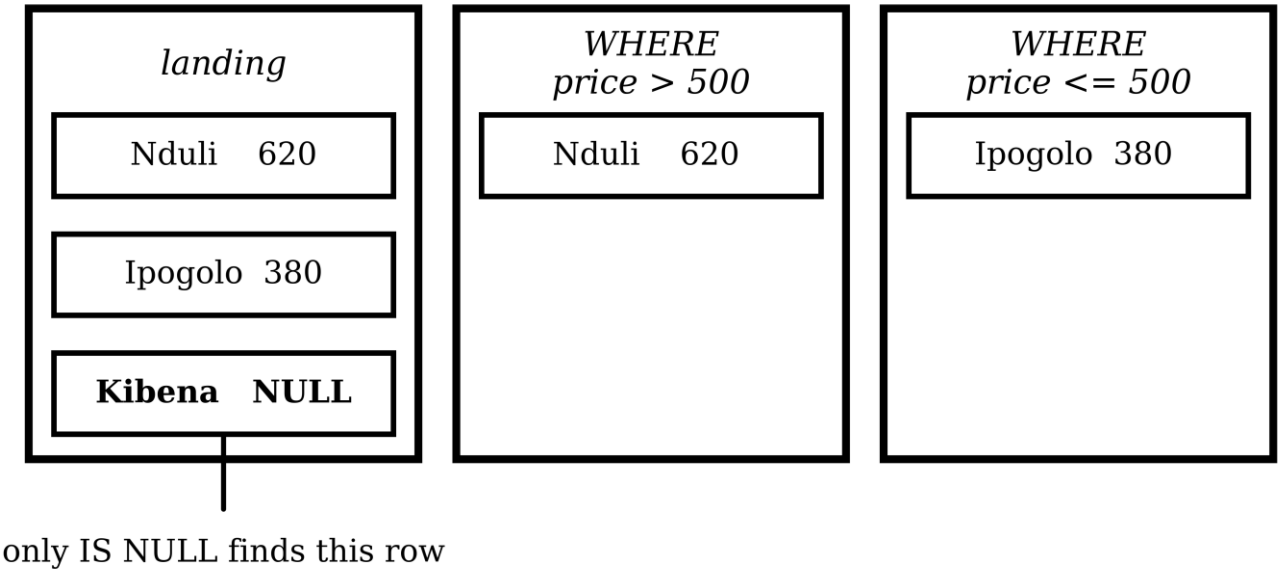


Figure 4.25: *A row that falls out of a filter and out of its opposite.*

The row that falls out of both: Figure 4.25 puts one day of the tally through price > 500 and then through price <= 500. The Nduli basket answers the first and the Ipogolo basket the second, while the Kibena row answers neither, so the two results hold 2 rows where the day held 3.

What AND and OR make of unknown: FALSE AND NULL is 0, because one false part settles a conjunction whatever the other part turns out to be. TRUE OR NULL is 1 for the mirror reason. TRUE AND NULL stays NULL, since the unknown part is all that is left to decide it.

Why IS NULL had to be invented: Equality asks whether two values are the same, and here there is no value to compare, so the language adds an operator asking about presence instead. IS NULL and IS NOT NULL answer definitely about what is absent.

A filter keeps what is true, and unknown is not true, which is the whole surprise.

The Master Key: *NULL is the absence of a value, so every comparison with it is unknown and a WHERE clause drops that row from a filter and from its opposite alike, leaving IS NULL as the only way to find it.*

REAL Question 120

Three lines are typed on a page pinned inside the ice store at Ipogolo, where the tally clerk can read them while she works. Two of them are wrong, and the one that survives sounds like a complaint about the server.

One of the three statements below is correct and the other two are not. Name the correct one, say exactly what the server does with each of the other two, and give the correction for each.

A. WHERE price = NULL finds every row whose price is unknown.

B. A row whose price is NULL is returned by neither WHERE price > 500 nor WHERE price <= 500.

C. SELECT DISTINCT and SELECT return the same rows whenever the table has a primary key.

Model Answer

B is the correct one, exactly as written: a row whose price is NULL is returned by neither of those two filters, so the pair of results holds 5 rows between them where the table holds 6. A is wrong because a comparison with NULL is unknown and never true, and C is wrong because DISTINCT acts on the columns the SELECT list asks for and not on the whole row.

Why A is wrong: It asks whether an unrecorded price equals an unrecorded price, and the answer is unknown, not true, so no row is kept. The statement runs, reports no error and returns nothing at all, which is the quietest

wrong answer in this section. The correction is WHERE price IS NULL, an operator about presence rather than equality.

Why B is right, and what it costs: The 6 landings split into 3 priced above 500 and 2 priced at 500 or below, and the sixth is in neither result, because the condition it was tested against was unknown both times. No adjustment of the figure 500 will recover it, and nothing has gone wrong that an error message could report.

Why C is wrong: A primary key makes whole rows of the table distinct, and DISTINCT is not applied to whole rows of the table. It is applied to the result, meaning the columns the SELECT list asked for, and a list that leaves the key out can certainly hold duplicates.

The counterexample in one table: The landing table has landing_id as its primary key, and SELECT site returns 6 rows while SELECT DISTINCT site returns 3. The two forms agree only when the selected columns are themselves unique, which is what a key guarantees for the row and not for a projection of it.

Typed on the page	What the server actually does	The correction
A. price = NULL finds unknown prices	the comparison is unknown, so no row is kept, and no error is raised either	write WHERE price IS NULL
B. the NULL row is in neither result	correct; 3 rows above and 2 at or below, out of the 6 stored	correct as it stands, and IS NULL finds the sixth
C. a key makes DISTINCT idle	DISTINCT acts on the selected columns; SELECT site gives 6 rows and DISTINCT gives 3	true only when the selected columns include a key

Table 4.25: *The three lines on the page, what the server does with each, and what each should say.*

Two of the lines describe a server that would be easier to teach, and the third describes the one on the desk.

The Master Key: *A comparison with NULL is unknown and never true, so a filter and its opposite both discard the row and only IS NULL finds it, while DISTINCT works on the columns selected and not on the row, so a primary key does not make it idle.*

BUILD Question 121

The tally book at Ipogolo gives one line for every basket, and the same three landing sites come round again and again down the page. A clerk reading it aloud says Nduli over and over. One word makes the server say each name once.

Create the landing table for the tally, insert the week of landings with one price still unknown, and show what a bare SELECT of the site column returns beside what SELECT DISTINCT returns.

Every landing carries its site, its species, the day it was weighed and the weight in kilograms.

The price a kilogram is settled with the buyer, and one landing has not been sold yet.

Six landings arrive, from three sites, over two days.

Show the whole table, then the site column alone, then the distinct sites.

Model Answer

price is the one column left open to NULL, because a basket can be weighed before it is sold, and the other 5 columns are NOT NULL because a landing with no site or no weight records nothing; the 6 rows go in, and the site column then gives 6 rows to a bare SELECT and 3 to SELECT DISTINCT.

```
 1  CREATE TABLE landing (
 2    landing_id INT          NOT NULL AUTO_INCREMENT,
 3    site       VARCHAR(10)  NOT NULL,
 4    species    VARCHAR(10)  NOT NULL,
 5    landed_on  DATE         NOT NULL,
 6    weight_kg  DECIMAL(7,2) NOT NULL,
 7    price      DECIMAL(7,2)     NULL,
 8    PRIMARY KEY (landing_id)
 9  ) ENGINE = InnoDB;
10
11  INSERT INTO landing (site, species, landed_on, weight_kg, price)
12  VALUES
13    ('Nduli',   'sato',    '2026-08-03',  42.50, 620.00),
14    ('Ipogolo', 'dagaa',   '2026-08-03', 118.00, 380.00),
15    ('Kibena',  'sangara', '2026-08-03',  76.25, NULL),
16    ('Nduli',   'dagaa',   '2026-08-04',  60.00, 540.00),
17    ('Ipogolo', 'sato',    '2026-08-04',  55.75, 600.00),
18    ('Kibena',  'sangara', '2026-08-04', 128.50, 410.00);
19
20  SELECT landing_id, site, species, landed_on, weight_kg, price
21  FROM landing
22  ORDER BY landing_id;
23
24  SELECT site FROM landing;
25
26  SELECT DISTINCT site FROM landing;
```

The server replies

```
+------------+---------+---------+------------+-----------+--------+
| landing_id | site    | species | landed_on  | weight_kg | price  |
+------------+---------+---------+------------+-----------+--------+
|          1 | Nduli   | sato    | 2026-08-03 |     42.50 | 620.00 |
|          2 | Ipogolo | dagaa   | 2026-08-03 |    118.00 | 380.00 |
|          3 | Kibena  | sangara | 2026-08-03 |     76.25 |   NULL |
|          4 | Nduli   | dagaa   | 2026-08-04 |     60.00 | 540.00 |
|          5 | Ipogolo | sato    | 2026-08-04 |     55.75 | 600.00 |
|          6 | Kibena  | sangara | 2026-08-04 |    128.50 | 410.00 |
+------------+---------+---------+------------+-----------+--------+
+---------+
| site    |
+---------+
| Nduli   |
| Ipogolo |
| Kibena  |
| Nduli   |
| Ipogolo |
| Kibena  |
+---------+
+---------+
| site    |
+---------+
| Nduli   |
| Ipogolo |
| Kibena  |
+---------+
```

Program 4.20: *The landing tally, and the same column read with DISTINCT and without it.*

The column that may be empty: A price of 0.00 would be a price, and a false one. NULL says the figure is not recorded, and price is declared without NOT NULL for that reason alone, while site, species, landed_on and weight_kg all carry it.

What the bare SELECT gave back: It gave 6 rows, one for each landing, with all 3 site names appearing twice over. Relational projection would have returned the set of 3 names; SQL returns one row per row, and DISTINCT is what closes the gap between them.

The decision worth naming: weight_kg and price are DECIMAL, because a weight somebody is paid for and a price somebody is charged must come back exactly as they went in. site and species are VARCHAR, since no arithmetic is done on either.

This is the table every other listing in this section reads from.

The Master Key: *Declare NOT NULL on every column that must hold a value and leave NULL to the one recording what is not yet known, and remember that a bare SELECT returns a row for every row, so DISTINCT is what gives the set of names back.*

BUILD Question 122

Type the comparison into the client at Nduli and the server answers NULL where a zero was expected. Nothing has broken, and that same answer is what quietly removes a row from a filter and from the opposite filter as well.

Run the comparisons that involve NULL and print what each returns, then show the tally filtered by IS NULL and by IS NOT NULL.

Model Answer

Three of the 4 comparisons come back NULL, since 5 > NULL, 5 = NULL and NULL = NULL are each unknown and only NULL IS NULL answers, with 1; the third result writes the two filter conditions into the SELECT list, and the Kibena row answers NULL under price > 500 and NULL again under price <= 500.

```
 1  SELECT 5 > NULL AS five_gt_null, 5 = NULL AS five_eq_null,
 2         NULL = NULL AS null_eq_null, NULL IS NULL AS null_is_null;
 3
 4  SELECT FALSE AND NULL AS false_and_null,
 5         TRUE OR NULL AS true_or_null,
 6         TRUE AND NULL AS true_and_null;
 7
 8  SELECT site, price,
 9         price > 500 AS above_500,
10         price <= 500 AS not_above_500
11  FROM landing
12  WHERE landed_on = '2026-08-03'
13  ORDER BY landing_id;
14
15  SELECT site, price
16  FROM landing
17  WHERE price IS NULL;
18
19  SELECT site, price
20  FROM landing
21  WHERE price IS NOT NULL;
```

The server replies

```
+--------------+--------------+--------------+--------------+
| five_gt_null | five_eq_null | null_eq_null | null_is_null |
+--------------+--------------+--------------+--------------+
|         NULL |         NULL |         NULL |            1 |
+--------------+--------------+--------------+--------------+
+----------------+--------------+---------------+
| false_and_null | true_or_null | true_and_null |
+----------------+--------------+---------------+
|              0 |            1 |          NULL |
+----------------+--------------+---------------+
+---------+--------+-----------+---------------+
| site    | price  | above_500 | not_above_500 |
+---------+--------+-----------+---------------+
| Nduli   | 620.00 |         1 |             0 |
| Ipogolo | 380.00 |         0 |             1 |
| Kibena  |   NULL |      NULL |          NULL |
+---------+--------+-----------+---------------+
+--------+-------+
| site   | price |
+--------+-------+
| Kibena |  NULL |
+--------+-------+
+---------+--------+
| site    | price  |
+---------+--------+
| Nduli   | 620.00 |
| Ipogolo | 380.00 |
| Nduli   | 540.00 |
| Ipogolo | 600.00 |
| Kibena  | 410.00 |
+---------+--------+
```

Program 4.21: *The comparisons that return neither true nor false, and the two tests that do.*

What AND and OR make of unknown: FALSE AND NULL is 0, because one false part decides a conjunction on its own, and TRUE OR NULL is 1 for the same reason on the other side. TRUE AND NULL is NULL, since the unknown part is all that is left to settle it.

Reading the two test columns: above_500 and not_above_500 hold the very truth values a WHERE clause would act on. The Nduli row answers 1 and 0, the Ipogolo row answers 0 and 1, and the Kibena row answers NULL twice, so a filter keeps it neither time.

The pair of tests that does add up: IS NULL returns the 1 unpriced landing and IS NOT NULL returns the other 5, and 1 and 5 make the 6 the table holds. Absence is the one thing about that row which can be tested and answered definitely.

The blank in the result is not a missing number; it is the server declining a question that has no answer.

The Master Key: *Every comparison with NULL returns NULL rather than true or false, so a WHERE clause built out of comparisons discards the unpriced row twice over, and only IS NULL and IS NOT NULL cut the table cleanly in two.*

BUILD Question 123

A buyer sends one message to the office at Ipogolo: he wants sato and sangara, from Nduli or Kibena, in baskets no lighter than forty kilograms and no heavier than a hundred. Three different operators answer him in one clause.

Write the query that answers the buyer using LIKE, IN and BETWEEN in one WHERE clause, and say what each of the three does that the other two cannot.

The species name begins with sa, which takes in the sato and the sangara.

The site is Nduli or Kibena.

The weight runs from 40 up to 100 kilograms, both ends counted.

Model Answer

One WHERE clause carries the three tests joined by AND, and the tally answers with 2 of its 6 landings: LIKE matches a pattern inside a value, IN tests membership of a written list, and BETWEEN tests a range with both ends included.

```
1  SELECT landing_id, site, species, weight_kg
2  FROM landing
3  WHERE species LIKE 'sa%'
4    AND site IN ('Nduli', 'Kibena')
5    AND weight_kg BETWEEN 40.00 AND 100.00
6  ORDER BY landing_id;
```

The server replies

```
+------------+--------+---------+-----------+
| landing_id | site   | species | weight_kg |
+------------+--------+---------+-----------+
|          1 | Nduli  | sato    |     42.50 |
|          3 | Kibena | sangara |     76.25 |
+------------+--------+---------+-----------+
```

Program 4.22: *One WHERE clause carrying a pattern, a list and a range.*

What LIKE does alone: LIKE compares a value against a pattern, where the per cent sign stands for any run of characters and the underscore for exactly one. It is the only one of the three that looks inside a value, and it alone turns away the Nduli dagaa, which passes the other two tests.

What IN does alone: IN is a written list, true when the column equals any member of it, so it says in one line what a chain of OR would say in three. The Ipogolo sato matches the pattern and sits inside the weight range, and IN alone removes it.

What BETWEEN does alone: BETWEEN is a range test and both ends belong to it, so the clause here means at or above 40.00 and at or below 100.00. The smaller bound is written first, or it matches nothing, and the upper bound alone removes the Kibena basket of 128.50 kilograms.

Three tests of three different shapes, and the rows surviving all of them are the rows the buyer asked for.

The Master Key: *LIKE matches a pattern inside a value, IN tests membership of a list and BETWEEN tests a range including both of its ends, so a question mixing all three shapes needs all three operators in one WHERE clause.*

BUILD Question 124

The list the Ipogolo clerk prints for the buyer should read site by site, heaviest basket first, and stop at four lines. Run it without the clause that says so and the order is whatever the server found convenient.

Print the tally by site with the heaviest basket first inside each site, and stop the result after the first four rows.

Sites in alphabetical order.

Inside one site, the heaviest basket first.

Only the first four rows are wanted.

Model Answer

ORDER BY site ASC, weight_kg DESC sorts on two columns in two directions, the names rising and the weights falling inside each name, and LIMIT 4 keeps the first 4 rows of that sorted result, the two Ipogolo landings and the two Kibena ones; without ORDER BY a result set has no defined order at all.

```
1  SELECT site, species, weight_kg
2  FROM landing
3  ORDER BY site ASC, weight_kg DESC
4  LIMIT 4;
```

The server replies

```
+---------+---------+-----------+
| site    | species | weight_kg |
+---------+---------+-----------+
| Ipogolo | dagaa   |    118.00 |
| Ipogolo | sato    |     55.75 |
| Kibena  | sangara |    128.50 |
| Kibena  | sangara |     76.25 |
+---------+---------+-----------+
```

Program 4.23: *Two sort columns in two directions, and a limit applied after the sorting.*

How two sort columns work together: The second column is consulted only where the first one ties, so the weights fall inside a site and never across sites. ASC is the default and is written out here so that the two directions can be seen beside each other.

Where an order comes from: A table holds a set of rows and none of them is the first. Any order a result seems to have without ORDER BY came from how the server happened to read the rows, and it changes when rows are updated or an index is added.

What LIMIT counts: LIMIT is applied after the sorting, so it takes the first 4 rows of the ordered result rather than 4 rows chosen anyhow. The 2 Nduli landings fall off the end, which is a decision the ORDER BY made and not the LIMIT.

An order on the page is a guarantee only where a clause asked for it.

The Master Key: *ORDER BY is the only thing that gives a result set an order at all, it takes a direction for each column and consults the second only where the first ties, and LIMIT counts its rows after the sorting.*

BUILD Question 125

The unpriced landing from Kibena does not vanish when the list is sorted by price; it goes to one end or the other. Which end is not a thing to reason out from first principles, so the server is asked, once each way.

Sort the tally by price ascending and then descending, say where the row holding no price lands each time, and show how to place it deliberately.

Model Answer

Ascending, MariaDB puts the NULL first, above the lowest recorded price of 380.00; descending it puts the NULL last, below the highest of 620.00, so an absent value sorts here as though it were smaller than every number in the column, and standard SQL does not settle where it belongs.

```
 1  SELECT site, price
 2  FROM landing
 3  ORDER BY price ASC;
 4
 5  SELECT site, price
 6  FROM landing
 7  ORDER BY price DESC;
 8
 9  SELECT site, price
10  FROM landing
11  ORDER BY price IS NULL, price;
```

The server replies

```
+---------+--------+
| site    | price  |
+---------+--------+
| Kibena  |   NULL |
| Ipogolo | 380.00 |
| Kibena  | 410.00 |
| Nduli   | 540.00 |
| Ipogolo | 600.00 |
| Nduli   | 620.00 |
+---------+--------+
+---------+--------+
| site    | price  |
+---------+--------+
| Nduli   | 620.00 |
| Ipogolo | 600.00 |
| Nduli   | 540.00 |
| Kibena  | 410.00 |
| Ipogolo | 380.00 |
| Kibena  |   NULL |
+---------+--------+
+---------+--------+
| site    | price  |
+---------+--------+
| Ipogolo | 380.00 |
| Kibena  | 410.00 |
| Nduli   | 540.00 |
| Ipogolo | 600.00 |
| Nduli   | 620.00 |
| Kibena  |   NULL |
+---------+--------+
```

Program 4.24: *The same column sorted both ways, and the unpriced landing placed on purpose.*

What the first two results show: The same 6 rows come back both times and the unpriced Kibena landing changes ends, at the top of the first result and at the foot of the second. Only the direction of the sort changed between them.

Not a rule of the language: Standard SQL leaves the placement of an absent value to the implementation, so another server is free to put it at the opposite end, and a query moved between two servers can change which rows a LIMIT keeps without anybody having edited it.

Placing them on purpose: The third statement sorts on price IS NULL first, which is 0 for a recorded price and 1 for an absent one, so every priced landing comes before the unpriced one while the prices themselves still rise.

The answer came from the server rather than from reasoning, which is the only place it could have come from.

The Master Key: *MariaDB sorts NULL before every value on an ascending ORDER BY and after every value on a descending one, and because the standard does not fix that, a sort on a column that may be empty deserves a run rather than an assumption.*

HOT Question 126

Somebody decided that an unknown price would be stored as zero, so that nothing in the column is ever missing and every query stays simple. The first list of cheap landings that went out had a Kibena basket at the top of it.

Judge whether a column such as price should be allowed to hold NULL at all, or whether the designer should forbid it and record the absence with a marker value instead, and state the condition that decides it.

Model Answer

Allow NULL where the fact may genuinely be unknown when the row is written, and declare NOT NULL wherever a value must exist for the row to mean anything; the condition that decides it is whether a marker can be found outside the values the column measures, because a marker inside them, 0.00 in a price column, is a real price to every comparison, sort and filter.

The case for forbidding it: A column that can never be empty is simpler in every direction. Three-valued logic never arrives, a filter and its opposite add back to the table, and nobody writing a report has to know which columns hide an absence. That simplicity is real, and it is what makes a marker tempting.

What a marker inside the domain costs: A price of 0.00 is a price. It sorts below 380.00, it passes any filter asking for cheap fish, and nothing in the table says it is not real. The lie then has to be remembered in every statement anybody writes against that column, for as long as the column exists.

When a marker is honest: Sometimes a value outside the measured range exists. A column of settlement states can carry a state meaning not yet settled, a genuine member of that list. A price has no spare number, since every number in the range is one somebody could have paid.

What NULL buys in exchange: NULL sits outside the values a column measures, and the server enforces what follows: it equals nothing, it satisfies no comparison, and IS NULL is the only way to reach it. The cost is that whoever writes the query has to think about absence, which is the cost being paid deliberately.

Where NOT NULL is the right answer: site, species, landed_on and weight_kg carry NOT NULL, because a landing with no weight is not a landing at all. The server then refuses any row that leaves one of them out, with ERROR 1364, and those columns can be filtered and sorted with none of this arising.

The judgement: Allow the column to hold NULL, and reach for a marker only where the domain has room for one honestly outside what the column measures. Where it has not, a NOT NULL column with a marker inside the domain does not remove the unknown; it disguises it, and the reports come out wrong quietly.

The zero in the price column did not make the unknown go away; it made it look like a bargain.

The Master Key: *Let a column hold NULL wherever the value may genuinely be unknown when the row is written, because a marker taken from inside the column's own range compares, sorts and filters as though it were real.*

Aggregation and grouping: the WHERE and HAVING rule

One row at a time is not how anybody reads a rent roll. The office wants how many stalls there are, what the month should bring in, which market pays most and which stall pays least, and each of those is a single number standing in for many rows. This section takes the five functions that produce such a number, the clause that makes one number for each group instead of one for the whole table, and the clause that then filters the groups; and it spends its last care on what an aggregate does with a column holding no value, because the answer is the opposite of what NULL does everywhere else.

BINDER Question 127

The roll on the desk at Igominyi runs to eight lines, and the officer wants two numbers from it: how many stalls, and what the month should bring in. Five functions turn many rows into one figure, and one counts differently from the rest.

Explain what an aggregate function does, name the ones SQL provides, and say what each of them returns when the column holds no value.

Model Answer

An aggregate function reads many rows and gives back one value: COUNT(*) returns how many rows there are, COUNT(rent) how many hold a rent, COUNT(DISTINCT stall_type) how many kinds of trade appear, and SUM, AVG, MIN and MAX give the total, the mean, the smallest and the largest of the values they find; every one except COUNT(*) skips NULL.

What an aggregate gives back: One value for a whole set of rows, so a statement carrying an aggregate and nothing else answers with one line however many rows it read. The 8 stalls come back as one number.

Counting rows against counting values: COUNT(*) counts rows and asks nothing of any column, so a row counts even where every column in it is empty. COUNT(rent) counts the rows whose rent holds a value, 7 of the 8 here, and the two part company the moment one NULL appears.

Counting how many different ones: COUNT(DISTINCT stall_type) throws away repeats before counting, so 8 stalls give 3, one for produce, one for fish and one for cloth. DISTINCT may be written inside SUM and AVG as well, though it is rarely what a total means.

The four that summarise the values themselves: SUM adds them, AVG divides that total by how many it added, MIN gives the smallest and MAX the largest. SUM and AVG need numbers; MIN and MAX take text and dates too, where smallest means earliest in the collating order.

The rule running through all of them: Every aggregate except COUNT(*) skips a NULL rather than passing it on, the reverse of what NULL does in an ordinary expression, where anything touching it comes back unknown. An aggregate quietly narrows itself to the rows holding a value.

One statement, one line back, and the empty box played no part in it.

The Master Key: *An aggregate turns many rows into one value, COUNT(*) counts rows while COUNT of a column counts the values in it, and every aggregate except COUNT(*) skips NULL rather than being spoiled by it.*

BINDER Question 128

One line for the market is not what the office wants; it wants one for each. GROUP BY is the clause that says so, and the order the server carries the clauses out in decides which of the two filters may name a total.

Explain what GROUP BY does, what HAVING filters, and give the rule that decides whether a condition belongs in WHERE or in HAVING.

Model Answer

GROUP BY collapses the rows into one row for each distinct value of the column named, and HAVING then filters those groups; the difference is not what they may mention but when they run, since the server evaluates FROM, then WHERE, then GROUP BY, then HAVING, then SELECT, then ORDER BY, so WHERE acts before any group exists and HAVING after they are formed.

What grouping actually makes: One row for each distinct value in the grouping column, so the 3 markets give 3 rows whatever the number of stalls behind them. Name two columns and the groups form on the pair.

The order the server works in: FROM first, then WHERE, then GROUP BY, then HAVING, then SELECT, then ORDER BY, which is 6 stages. The two filters stand on either side of the stage forming the groups, and everything else follows from that.

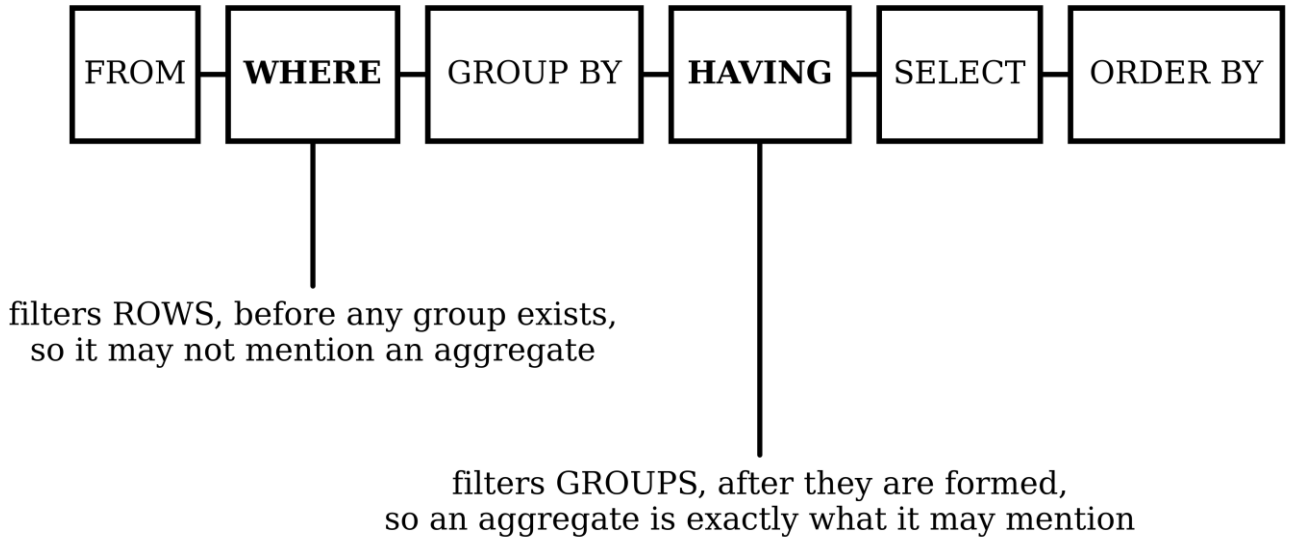


Figure 4.26: *The order the server evaluates a query in, and where each filter acts.*

Why WHERE cannot name a total: Figure 4.26 puts WHERE two stages ahead of the grouping, so when it runs no group exists and no total has been worked out. There is nothing for SUM to have added, and the server answers ERROR 1111.

Why a total belongs in HAVING: HAVING is evaluated once the groups are formed, so an aggregate is exactly what it exists to compare. Arriving late costs it something: it cannot cut the reading of the table down as an early filter can.

Accounting for every column in the list: A column in the SELECT list must be named in GROUP BY or wrapped in an aggregate, since a group is many rows and an ungrouped column may hold many values. MariaDB otherwise prints a value from one row of the group, chosen by nobody; under ONLY_FULL_GROUP_BY it refuses, with ERROR 1055.

Both clauses take a condition, and only one stands where a total exists.

The Master Key: *WHERE runs before the groups exist and HAVING after they are formed, so a condition naming an aggregate belongs in HAVING and one naming a plain column belongs in WHERE, where it can still cut the work down.*

REAL Question 129

Written on the inside cover of the ledger at Iyunga are three lines about counting, read by the rent clerk every morning. One is sound. The other two are the kind of belief that survives because nothing in the roll ever contradicts it.

Decide which of the three lines is correct, say what the server actually does with each of the other two, and write the correction each of them needs.

A. HAVING may be used in place of WHERE whenever the query has no GROUP BY clause.

B. AVG(rent) always gives the same answer as SUM(rent) divided by COUNT(*).

C. COUNT(*) and COUNT(rent) return different numbers whenever any rent is NULL.

Model Answer

C is the correct line: COUNT(*) counts rows while COUNT(rent) counts recorded rents, so a single NULL anywhere in the column parts them, and in this roll they answer 8 and 7. A is wrong about when the two clauses run, and B is wrong exactly where a rent is missing, which is the case C is about.

The fault in A: MariaDB does accept HAVING with no GROUP BY, which is why the belief survives, but the clauses are not interchangeable. WHERE filters rows before any grouping and HAVING filters what the grouping stage produced, and standard SQL treats a query with HAVING and no GROUP BY as one single group, where a bare column in the condition is not valid at all.

What A costs where it is accepted: A condition moved out of WHERE into HAVING is evaluated after the whole table has been read and grouped, so an index on that column cannot be used to skip rows. The rows returned are the same and the work done is not, and the difference grows with every stall added to the roll.

The fault in B: AVG divides by the number of values it actually added, not by the number of rows it looked at. The total here is 101500.00 over 7 recorded rents, so AVG returns 14500.00, while the same total over the 8 rows gives 12687.50. The two forms agree only while no rent is NULL.

The identity that does hold: AVG(rent) equals SUM(rent) divided by COUNT(rent), because both sides then skip the same rows. Written with COUNT(*) underneath, the expression answers a different question without saying so, namely what the total comes to per stall when a rent nobody has agreed is treated as zero.

Why C stands: COUNT(*) counts rows and never looks inside one; COUNT(rent) counts the rows where rent holds a value. One NULL is enough to separate them, and this roll holds 1, which is why the two counts differ by 1 rather than by more.

Written on the card	What the server actually does	The correction
A. HAVING will stand in for WHERE	it runs, because MariaDB allows it, but after the grouping stage and with no index available	put a condition on a plain column in WHERE
B. AVG is SUM over COUNT(*)	both run and disagree, 14500.00 against 12687.50, and neither reports anything wrong	AVG(rent) is SUM(rent) over COUNT(rent)
C. the two counts differ when a rent is NULL	correct; COUNT(*) gives 8 and COUNT(rent) gives 7 on this roll	correct as it stands, and the gap is the number of NULLs

Table 4.26: *The three lines on the card, what the server does with each, and what each should say.*

The card can stay above the counter once two of its three lines have been rewritten.

The Master Key: *COUNT(*) counts rows and COUNT of a column counts values, so one NULL parts them; AVG divides by COUNT of the column and never by COUNT(*); and HAVING is not a late spelling of WHERE, because it runs after the grouping and no index can serve it there.*

BUILD Question 130

A stall at Iyunga changed hands last week and the new rent has not been agreed, so the box for it is empty. Ask the roll how many stalls it holds and the answer depends on which word you count with.

Create the rent roll, put a stall with no agreed rent into it, and show COUNT(*), COUNT(rent) and COUNT(DISTINCT stall_type) side by side in one result.

Every stall carries the market it stands in, what it sells, the trader holding it and the monthly rent.

One stall has changed hands and its rent is not yet agreed.

Eight stalls, three markets, three kinds of trade.

Show the whole roll, then the three counts on one line.

Model Answer

rent is the only column left open to NULL, because a stall can stand in the roll before its rent is settled, and the three counts then answer 8, 7 and 3: COUNT(*) counts rows, COUNT(rent) counts the rows where a rent is recorded, and COUNT(DISTINCT stall_type) counts the kinds of trade rather than the stalls.

```
 1  CREATE TABLE stall (
 2    stall_id   INT          NOT NULL AUTO_INCREMENT,
 3    market     VARCHAR(10)  NOT NULL,
 4    stall_type VARCHAR(10)  NOT NULL,
 5    trader     VARCHAR(12)  NOT NULL,
 6    rent       DECIMAL(8,2)     NULL,
 7    PRIMARY KEY (stall_id)
 8  ) ENGINE = InnoDB;
 9
10  INSERT INTO stall (market, stall_type, trader, rent)
11  VALUES
12    ('Igominyi', 'produce', 'Neema',  12000.00),
13    ('Igominyi', 'produce', 'Juma',   12000.00),
14    ('Igominyi', 'fish',    'Asha',   18000.00),
15    ('Iyunga',   'produce', 'Baraka',  9000.00),
16    ('Iyunga',   'cloth',   'Halima', 25000.00),
17    ('Iyunga',   'cloth',   'Salum',      NULL),
18    ('Isongole', 'fish',    'Zawadi', 15000.00),
19    ('Isongole', 'produce', 'Rehema', 10500.00);
20
21  SELECT stall_id, market, stall_type, trader, rent
22  FROM stall
23  ORDER BY stall_id;
24
25  SELECT COUNT(*) AS stall_rows,
26         COUNT(rent) AS rent_rows,
27         COUNT(DISTINCT stall_type) AS stall_types
28  FROM stall;
```

The server replies

```
+----------+----------+------------+--------+----------+
| stall_id | market   | stall_type | trader | rent     |
+----------+----------+------------+--------+----------+
|        1 | Igominyi | produce    | Neema  | 12000.00 |
|        2 | Igominyi | produce    | Juma   | 12000.00 |
|        3 | Igominyi | fish       | Asha   | 18000.00 |
|        4 | Iyunga   | produce    | Baraka |  9000.00 |
|        5 | Iyunga   | cloth      | Halima | 25000.00 |
|        6 | Iyunga   | cloth      | Salum  |     NULL |
|        7 | Isongole | fish       | Zawadi | 15000.00 |
|        8 | Isongole | produce    | Rehema | 10500.00 |
+----------+----------+------------+--------+----------+
+------------+-----------+-------------+
| stall_rows | rent_rows | stall_types |
+------------+-----------+-------------+
|          8 |         7 |           3 |
+------------+-----------+-------------+
```

Program 4.25: *The rent roll, and three counts of the same eight rows.*

Reading the three counts together: The 8 and the 7 differ by the 1 stall whose rent is not agreed, and the 3 is smaller again because it counts kinds of trade and not stalls. Three questions of the same table, three different things being counted, one line of answer.

Where the asterisk changes the meaning: COUNT(*) is a count of rows and looks at no column at all, so it is the only one of the three a NULL cannot move. Write COUNT(rent) in its place and the question has quietly become how many rents are recorded.

The decision behind the column types: rent is DECIMAL because it is money and must come back exactly as it went in, and it is the one column declared without NOT NULL. market, stall_type and trader are VARCHAR, since nothing arithmetic is ever done to any of them.

One result line, three counts, and the gap between the first two is the empty box.

The Master Key: *COUNT(*) counts rows, COUNT of a column counts the values recorded in it and COUNT of DISTINCT values counts the different ones, which on this roll are 8, 7 and 3.*

BUILD Question 131

The average rent chalked on the board at Isongole is smaller than the figure AVG printed a moment later, from the same roll. The board figure came from dividing the total by the number of stalls, and neither statement had an error in it.

Show AVG(rent), SUM(rent) divided by COUNT(*) and SUM(rent) divided by COUNT(rent) in one result, and say which two agree and why.

Model Answer

AVG(rent) and SUM(rent) over COUNT(rent) agree, both printing 14500.000000, while SUM(rent) over COUNT(*) prints 12687.500000, because AVG divides by the number of values it added and not by the number of rows it read; the identity students are often taught is wrong by exactly the rows whose rent is NULL.

```
1  SELECT SUM(rent) AS rent_total,
2         COUNT(*) AS stall_rows,
3         COUNT(rent) AS rent_rows
4  FROM stall;
5
6  SELECT AVG(rent) AS avg_rent,
7         SUM(rent) / COUNT(*) AS over_all_rows,
8         SUM(rent) / COUNT(rent) AS over_rent_rows
9  FROM stall;
```

The server replies

```
+------------+------------+-----------+
| rent_total | stall_rows | rent_rows |
+------------+------------+-----------+
|  101500.00 |          8 |         7 |
+------------+------------+-----------+
+--------------+---------------+----------------+
| avg_rent     | over_all_rows | over_rent_rows |
+--------------+---------------+----------------+
| 14500.000000 |  12687.500000 |   14500.000000 |
+--------------+---------------+----------------+
```

Program 4.26: *The same average worked out three ways, two of which agree.*

What each of the three expressions asked: The first two asked for the mean of the 7 recorded rents, and the extra decimal places are how MariaDB prints a division, not a statement about the money. The third asked what the total comes to across all 8 stalls, which is a different question and should not be written on a board as the average rent.

Where the difference of 1812.50 comes from: Nothing has gone missing. Dividing 101500.00 by 8 instead of by 7 spreads one unchanged total over one more stall, and the 1812.50 between the two figures is what treating an unagreed rent as though it were zero costs the answer.

The identity in the form that survives NULL: AVG of a column is SUM of that column over COUNT of that column, never over COUNT(*). The two forms coincide only while every row holds a value, so a query right today changes its answer the first day a stall stands with no rent agreed.

Both figures are sound arithmetic, and only one answers the question that was asked.

The Master Key: *AVG divides by COUNT of the column and not by COUNT(*), so the familiar identity is wrong by exactly the rows holding NULL and agrees only while there are none.*

BUILD Question 132

The market supervisor asks a question the roll cannot answer one row at a time: what does each of the three markets bring in, and how many stalls stand in it? One statement answers for all three at once.

Answer the supervisor with one GROUP BY statement giving, for each market, the number of stalls, the number of rents recorded, the monthly total and the smallest and largest rent.

Model Answer

GROUP BY market forms one group for each of the 3 market names and every aggregate in the SELECT list is then worked out inside its own group, giving Igominyi 42000.00, Iyunga 34000.00 and Isongole 25500.00; market itself may be printed beside them only because it is the column grouped by, so it holds one value throughout a group.

```
1  SELECT market,
2         COUNT(*) AS stalls,
3         COUNT(rent) AS rents,
4         SUM(rent) AS monthly_total,
5         MIN(rent) AS lowest,
6         MAX(rent) AS highest
7  FROM stall
8  GROUP BY market
9  ORDER BY market;
```

The server replies

```
+----------+--------+-------+---------------+----------+----------+
| market   | stalls | rents | monthly_total | lowest   | highest  |
+----------+--------+-------+---------------+----------+----------+
| Igominyi |      3 |     3 |      42000.00 | 12000.00 | 18000.00 |
| Isongole |      2 |     2 |      25500.00 | 10500.00 | 15000.00 |
| Iyunga   |      3 |     2 |      34000.00 |  9000.00 | 25000.00 |
+----------+--------+-------+---------------+----------+----------+
```

Program 4.27: *One line for each market, and five aggregates worked out inside it.*

Why the grouped column may be printed: Every row in a group carries the same market, so there is one value to print and no choice to make. A column neither grouped by nor inside an aggregate has no single value to offer, and the server must pick one from a row of its choosing or refuse the statement.

The group where the two counts part: Iyunga shows 3 stalls and 2 rents, because SUM, MIN and MAX all passed over the stall with no rent agreed. Its 34000.00 is the total of what is recorded, not of what the market will collect, and only the 2 beside the 3 says so.

What the last clause is doing: GROUP BY promises one row for each group and nothing about the order they arrive, so ORDER BY market is written to fix it. Grouping and sorting are separate operations, and sorting has to be asked for.

Three markets in, three lines out, and every figure on a line belongs to that market.

The Master Key: *GROUP BY forms one group for each distinct value of the grouping column and every aggregate is then computed within a group, which is why the grouped column, and only that one, may be printed beside the aggregates.*

BUILD Question 133

Two reports leave the office at Igominyi on the same morning, one filtered before the totals were worked out and one after, and their totals disagree. Neither is wrong. Then somebody asks for a total inside a WHERE clause.

Filter the same grouped query twice, once with WHERE and once with HAVING, then put the total into the WHERE clause and record what the server says.

The first statement keeps only stalls whose rent is at least 12000.00, and totals what is left.

The second counts every stall, then keeps only markets whose total is at least 30000.00.

The third puts the total into the WHERE clause on purpose.

Model Answer

The WHERE version throws rows away before grouping, so Iyunga comes back with 1 stall and 25000.00 in place of 3 and 34000.00, while the HAVING version leaves every stall counted and drops Isongole instead, whose 25500.00 falls short of the 30000.00 asked for; the third statement is refused with ERROR 1111, because WHERE is evaluated before any group exists and there is no total there to compare.

```
 1  SELECT market, COUNT(*) AS stalls, SUM(rent) AS collected
 2  FROM stall
 3  WHERE rent >= 12000.00
 4  GROUP BY market
 5  ORDER BY market;
 6
 7  SELECT market, COUNT(*) AS stalls, SUM(rent) AS collected
 8  FROM stall
 9  GROUP BY market
10  HAVING SUM(rent) >= 30000.00
11  ORDER BY market;
12
13  SELECT market, SUM(rent) AS collected
14  FROM stall
15  WHERE SUM(rent) >= 30000.00
16  GROUP BY market;
```

The server replies, then refuses

```
+----------+--------+-----------+
| market   | stalls | collected |
+----------+--------+-----------+
| Igominyi |      3 |  42000.00 |
| Isongole |      1 |  15000.00 |
| Iyunga   |      1 |  25000.00 |
+----------+--------+-----------+
+----------+--------+-----------+
| market   | stalls | collected |
+----------+--------+-----------+
| Igominyi |      3 |  42000.00 |
| Iyunga   |      3 |  34000.00 |
+----------+--------+-----------+
ERROR 1111 (HY000) at line 13: Invalid use of group function
```

Program 4.28: *The same query filtered before grouping, after grouping, and once too early.*

What the row filter did to the totals: Isongole falls from 2 stalls to 1 and from 25500.00 to 15000.00, because WHERE removed a row before there was any total to remove it from. The unagreed rent goes as well, since NULL is not at least 12000.00 and is not below it either.

What the group filter left alone: Under HAVING every stall is still counted, so the totals printed are the whole ones and entire markets are the thing dropped. Igominyi at 42000.00 and Iyunga at 34000.00 stay, and Isongole leaves with its 25500.00 intact and unprinted.

What the third statement was refused for: ERROR 1111 reports an invalid use of a group function, and the line it names is the line that statement begins on. Nothing is wrong with the arithmetic; the clause is simply standing at a stage of the query where no group, and so no total, has been made yet.

Two filters, two honest answers, and everything between them is a question of when.

The Master Key: *WHERE filters rows before the groups are formed and so changes the totals themselves, HAVING filters formed groups and leaves their totals as they are, and an aggregate written into WHERE is refused with ERROR 1111.*

BUILD Question 134

Nobody expects a query matching no rows to answer at all. Ask this roll for rents above one hundred thousand and one line still comes back, carrying a zero in one column and something that is not a number in the next.

Run the aggregates over a filter that matches no rows at all, add COALESCE to the total, then group the same filter, and describe what came back each time.

Model Answer

COUNT(*) and COUNT(rent) both return 0 while SUM, AVG and MIN return NULL, because a count of nothing is a number and a total of nothing is not; COALESCE puts 0.00 where a report needs a figure, and the same filter with GROUP BY printed nothing at all.

```
 1  SELECT COUNT(*) AS stall_rows,
 2         COUNT(rent) AS rent_rows,
 3         SUM(rent) AS rent_total,
 4         AVG(rent) AS avg_rent,
 5         MIN(rent) AS lowest
 6  FROM stall
 7  WHERE rent > 100000.00;
 8
 9  SELECT COALESCE(SUM(rent), 0.00) AS rent_total
10  FROM stall
11  WHERE rent > 100000.00;
12
13  SELECT market, SUM(rent) AS rent_total
14  FROM stall
15  WHERE rent > 100000.00
16  GROUP BY market;
```

The server replies

```
+------------+-----------+------------+----------+--------+
| stall_rows | rent_rows | rent_total | avg_rent | lowest |
+------------+-----------+------------+----------+--------+
|          0 |         0 |       NULL |     NULL |   NULL |
+------------+-----------+------------+----------+--------+
+------------+
| rent_total |
+------------+
|       0.00 |
+------------+
```

Program 4.29: *What the aggregates answer when the filter matches nothing.*

What an empty filter came back with: One line, not none. An aggregate with no GROUP BY returns exactly one row however little matched, and this one gave 0 for both counts and NULL for the total, the mean and the smallest.

Why a total of nothing is not zero: SUM adds the values handed to it and was handed none, so there is no total to report and the server says so with NULL. Zero would claim rents were collected and came to nothing, a different statement and a false one.

The result that never printed: The third statement grouped the same empty filter and printed nothing whatever, because groups are built out of rows and there were none.

Giving a report something to print: COALESCE returns the first of its arguments that is not NULL, so the second statement printed 0.00. That is a decision about the report and not about the roll, honest only where nothing collected means zero.

A count of nothing and a total of nothing are different answers, and the server gives each.

The Master Key: *COUNT over no rows returns 0 while SUM, AVG, MIN and MAX return NULL, and grouping over no rows returns no row at all, so a report needing a figure must supply one with COALESCE.*

HOT Question 135

Every time the counter screen at Igominyi is opened, the month's total is worked out again from all eight stalls. A clerk wants it kept in a column instead, written down once and corrected whenever a rent changes.

Judge whether a total that is asked for constantly should be computed by an aggregate every time or stored in a column and kept up to date, and state the condition that decides it.

Model Answer

Compute it with an aggregate every time, and store it in a column only where the total is read far more often than the rows behind it change and the aggregate has been measured too slow on the row count the table will really hold; the condition is a measurement and not a feeling, because a stored total is a second copy of a fact that the server will not keep true for you.

The case for working it out each time: An aggregate cannot disagree with the rows it read, being derived from them at the moment of asking. One place holds the truth, every INSERT, UPDATE and DELETE is complete on its own, and nothing has to remember to adjust a figure kept elsewhere.

What the aggregate genuinely costs: SUM over a market means reading every stall row of that market. Over 8 rows it costs nothing worth naming; over a table grown to hundreds of thousands and read on every screen it is the slowest thing in the office, and an index on the grouping column reduces that reading without removing it.

The case for the stored column: A stored total is fetched in one row lookup whatever stands behind it, and its upkeep is paid by the writers, who are few, instead of the readers, who are many. Where a screen is opened all day against a table that hardly changes, that trade is a real one.

What the stored column costs back: Every statement changing a rent must change the total inside the same transaction, or the two quietly disagree. A second application, or one correction typed by hand at the client, leaves a figure that is wrong and looks exactly as trustworthy as a right one.

What to measure before choosing: Time the aggregate against the row count the roll will carry in three years and under the readers it will really have. If it is fast enough there the argument is finished and nothing is stored, whatever the query looks like written out.

Where the judgement lands: Compute it. Store the total only where measurement on real volumes shows the aggregate too slow, the reads outnumber the writes heavily, and every write able to change it goes through one path updating the stored figure in the same transaction; any two of those three are not enough.

A stored total is quicker to read and easier to be wrong about, and only a measurement says whether that trade is worth making.

The Master Key: *Work a needed total out with an aggregate every time, and store it only where measurement on the real row count shows the aggregate too slow and every write that could change it updates the stored figure inside the same transaction.*

Joins: inner, outer, cross and self

A join reads two tables in one statement and pairs their rows on a condition you state. This section names the kinds, INNER, LEFT OUTER, RIGHT OUTER and CROSS, adds the self join, which pairs a table with a second copy of itself, and says for each of them which rows survive. The work is done on a mobile money agent's ledger holding 3 agents and 3 float top-ups, small enough for every row of every result to be counted by eye. Two things take most of the care here: a join pairs rows rather than filtering them, so it can return more rows than either table holds, and an outer join puts NULLs in the result that were never stored in either table.

BINDER Question 136

Two books lie open on the counter at Kiwira: one names the agents, the other records every float paid in. The answer to a single question is spread across both, and the way they are brought together decides which rows survive.

Explain what a join is, classify the kinds of join, and say for each kind which rows survive.

Model Answer

A join is a SELECT that reads two tables in one statement and pairs each row of the first with the rows of the second that satisfy a condition you state in the ON clause, and the four kinds differ only in which unpaired rows they keep afterwards.

A join pairs rows, it does not filter them: The server takes each row on the left and keeps it once for every row on the right that satisfies the condition, so a row matching twice appears twice. Agent A1 has 2 top-ups and comes back on 2 lines, and a result can therefore be longer than either table it was built from.

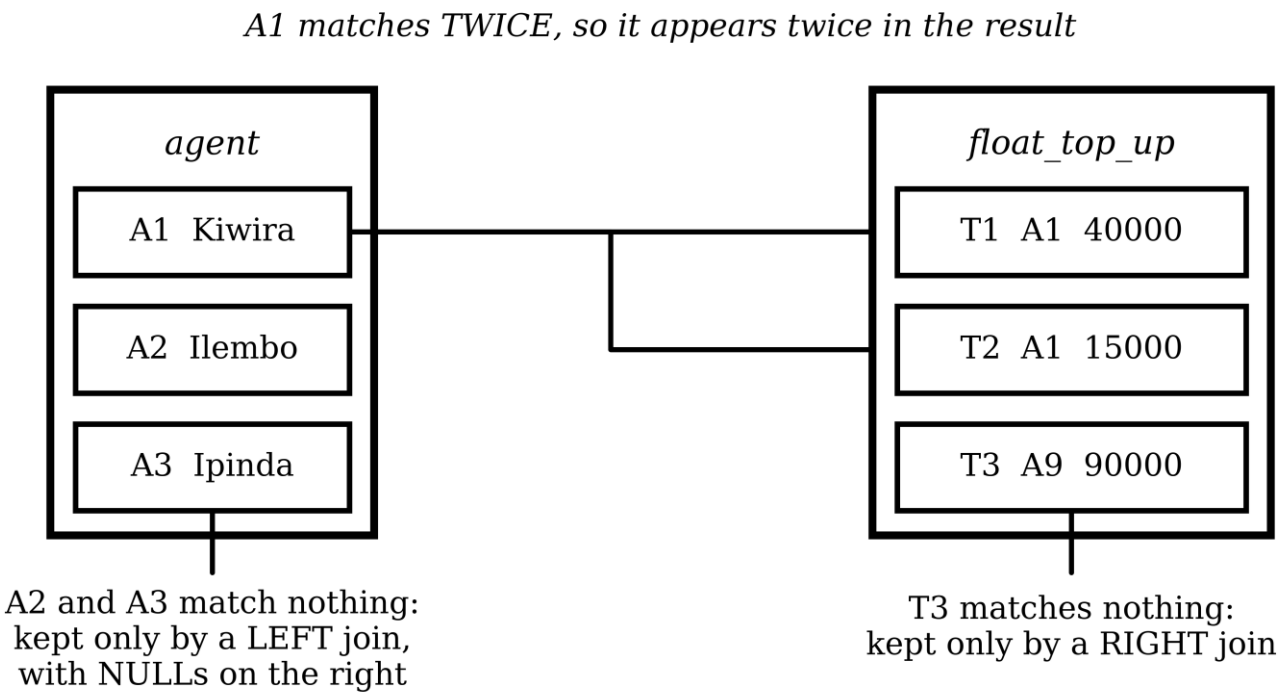


Figure 4.27: *Which rows survive each kind of join, and the row that matches twice.*

INNER JOIN: Only the pairs survive. An agent with no top-up is not in the result, and a top-up whose agent code stands in no agent row is not there either, so both tables lose their unmatched rows.

LEFT OUTER JOIN and RIGHT OUTER JOIN: LEFT keeps every row of the table written before the JOIN word, matched or not, and fills the columns taken from the other table with NULL. RIGHT keeps every row of the table written after it. The word OUTER may be left out and usually is.

CROSS JOIN: No condition is stated, so every row on the left is paired with every row on the right: 3 agents and 3 top-ups give 9 rows. The same result arrives by accident whenever an ordinary join is written with its ON clause missing, and the server does not complain.

Kind of join	Which rows survive	What it returns from this ledger
INNER JOIN	only the pairs that satisfy the ON condition	2 rows, with A1 on both of them
LEFT JOIN	every row of the left table, matched or not	4 rows: the 2 pairs, then A2 and A3 with NULLs
RIGHT JOIN	every row of the right table, matched or not	3 rows: the 2 pairs, then T3 with NULLs
CROSS JOIN	every row on the left with every row on the right	9 rows, longer than either table

Table 4.27: *The four kinds of join, what each keeps, and what each returns from 3 agents and 3 top-ups.*

The kind of join chosen is a decision about the unmatched rows, and about nothing else.

The Master Key: *A join pairs rows and does not filter them, so it can return more rows than either table holds; INNER keeps the pairs alone, LEFT and RIGHT keep the unmatched rows of the side they name, and CROSS keeps every combination there is.*

BINDER Question 137

The server refused the statement outright at Ilembo, and nothing was wrong with the question it asked. The table name simply appeared twice in one line, and the server could not tell which of the two copies each column belonged to.

Explain table aliases and the self join, and say why a table joined to itself needs two aliases and cannot work without them.

Model Answer

A table alias is a short name given to a table for the length of one statement, written after the table name with AS, and a self join is a join of a table to a second copy of itself, which cannot be written without two aliases because the server has no other way of telling the two copies apart.

Writing an alias: AS a after the table name makes a the name of that table everywhere else in the statement, and every column is then written a.agent_code or t.amount. The keyword AS is optional here and in standard SQL, and the alias lasts no longer than the statement.

Why every column is qualified: Both tables in this ledger hold a column called agent_code. Ask for it unqualified and the server answers ERROR 1052, the column is ambiguous, because it cannot choose for you. Qualifying also tells a reader which table each value came from.

What a self join is for: One table can answer a question about pairs of its own rows: two top-ups paid in by one agent, or two agents working one village. The table is named twice in the FROM clause, and the ON condition compares one copy with the other.

Why two aliases are not optional: Named twice under its own name, the table has one name for two copies, and the server refuses the statement outright with ERROR 1066, not unique table or alias. With t1 and t2 declared, t1.agent_code and t2.agent_code name different rows of the same table.

Two copies of one table need two names before a single column reference can mean anything.

The Master Key: *An alias renames a table for one statement and every column is then qualified with it; a self join names the same table twice, so it needs two aliases, and without them the server refuses the statement with ERROR 1066.*

REAL Question 138

The counter notebook at Ipinda is open at a page of three statements about joins, two of them false. The one that sounds least likely is the one that holds, and the friendliest of the three quietly destroys a result.

One of the three statements below is right. Say which, name the fault in each of the other two, say what the server does with each, and correct them.

A. An inner join always returns at least as many rows as the smaller of the two tables.

B. Adding WHERE t.amount > 0 to a LEFT JOIN leaves the unmatched rows in the result.

C. A LEFT JOIN returns every row of the left table, with NULLs where the right table has no match.

Model Answer

C is the correct one, and it is correct exactly as written: a LEFT JOIN returns every row of the left table, with NULLs where the right table has no match. A is wrong in both directions at once, and B describes the commonest way there is of destroying an outer join without noticing.

Why A is wrong at the bottom: An inner join can return no rows at all. Join the 3 agents to the 3 top-ups on an agent code that no top-up carries and the result is empty, which is fewer rows than the smaller table holds and not at least as many.

Why A is wrong at the top: It can also return more rows than either table. A1 matches T1 and T2, so it comes back twice, and one further top-up for A1 would make the result longer than either table. A join multiplies as readily as it drops.

Why B is wrong: The unmatched agents leave the join carrying NULL in every column taken from the right table. NULL is not a value, so t.amount > 0 is neither true nor false but unknown, and WHERE keeps only what is true. Those rows go.

What B leaves behind: What survives is exactly the set of rows an INNER JOIN would have returned. The word LEFT is still in the statement and is doing nothing, which is why the fault is so hard to see on the page.

The correction: Move the condition into the ON clause. ON decides which pairs are formed, and the outer join then adds the unmatched left rows back with their NULLs, so the agents with no top-up stay in the result where they were wanted.

Written on the page	What the server actually does	The correction
A. At least as many rows as the smaller table	returns one row per matching pair: no rows at all if none match, and more rows than either table if one row matches twice	an inner join returns a row for every pair, and nothing for a row that pairs with nothing
B. WHERE on the right-hand column keeps the unmatched rows	the manufactured NULLs make the comparison unknown, so the unmatched rows are dropped and an inner join is left	move the condition into the ON clause
C. Every left row, with NULLs where nothing matched	returns exactly that, 4 rows here from 3 agents	correct as it stands

Table 4.28: *The three written statements, what the server does about each, and what each should say.*

Two of the three read as common sense, and the correct one is the one that sounds like an inconvenience.

The Master Key: *A LEFT JOIN keeps every left row and manufactures the NULLs beside it; an inner join can return anything from no rows to more rows than either table; and a condition on the right-hand table belongs in ON, because in WHERE it removes the very rows the outer join was written to keep.*

BUILD Question 139

Three agents, three top-ups, and one result with nine rows in it. Nobody typed the extra ones and nothing has gone wrong: the server was asked for every combination, and every combination is what it returned.

Create the two tables of the ledger, join them with an INNER JOIN so that each top-up appears beside the agent who paid it in, and then pair every agent with every top-up on purpose.

The agent table holds 3 agents: A1 at Kiwira, A2 at Ilembo and A3 at Ipinda.

The float_top_up table holds 3 top-ups: T1 and T2 paid in by A1, and T3 recorded against agent A9, who is not an agent.

Show the inner join of the two tables, ordered by top-up.

Then write the cross product deliberately, and count the rows it returns.

Model Answer

The INNER JOIN returns 2 rows from two tables of 3 rows each, and A1 stands on both of them, once beside T1 and once beside T2, because a join keeps a row once for every row it matches; the CROSS JOIN states no condition and returns 9 rows, longer than either table.

```
 1  CREATE TABLE agent (
 2    agent_code CHAR(2)     NOT NULL,
 3    village    VARCHAR(20) NOT NULL,
 4    PRIMARY KEY (agent_code)
 5  ) ENGINE = InnoDB;
 6
 7  CREATE TABLE float_top_up (
 8    top_up_id  CHAR(2)       NOT NULL,
 9    agent_code CHAR(2)       NOT NULL,
10    amount     DECIMAL(12,2) NOT NULL,
11    PRIMARY KEY (top_up_id)
12  ) ENGINE = InnoDB;
13
14  INSERT INTO agent (agent_code, village) VALUES
15    ('A1', 'Kiwira'), ('A2', 'Ilembo'), ('A3', 'Ipinda');
16
17  INSERT INTO float_top_up (top_up_id, agent_code, amount) VALUES
18    ('T1', 'A1', 40000.00),
19    ('T2', 'A1', 15000.00),
20    ('T3', 'A9', 90000.00);
21
22  SELECT a.agent_code, a.village, t.top_up_id, t.amount
23  FROM agent AS a
24  INNER JOIN float_top_up AS t
25          ON t.agent_code = a.agent_code
26  ORDER BY t.top_up_id;
```

The server replies

```
+------------+---------+-----------+----------+
| agent_code | village | top_up_id | amount   |
+------------+---------+-----------+----------+
| A1         | Kiwira  | T1        | 40000.00 |
| A1         | Kiwira  | T2        | 15000.00 |
+------------+---------+-----------+----------+
```

Program 4.30: *The two tables of the ledger, and the inner join that puts A1 on two rows.*

```
1  SELECT a.agent_code, a.village, t.top_up_id, t.amount
2  FROM agent AS a
3  CROSS JOIN float_top_up AS t
4  ORDER BY a.agent_code, t.top_up_id;
```

The server replies

```
+------------+---------+-----------+----------+
| agent_code | village | top_up_id | amount   |
+------------+---------+-----------+----------+
| A1         | Kiwira  | T1        | 40000.00 |
| A1         | Kiwira  | T2        | 15000.00 |
| A1         | Kiwira  | T3        | 90000.00 |
| A2         | Ilembo  | T1        | 40000.00 |
| A2         | Ilembo  | T2        | 15000.00 |
| A2         | Ilembo  | T3        | 90000.00 |
| A3         | Ipinda  | T1        | 40000.00 |
| A3         | Ipinda  | T2        | 15000.00 |
| A3         | Ipinda  | T3        | 90000.00 |
+------------+---------+-----------+----------+
```

Program 4.31: *The cross product, written deliberately as a CROSS JOIN.*

Why A1 appears twice: The condition is satisfied by 2 rows of float_top_up, so the agent row is paired with each of them in turn. Nothing is duplicated in either table.

What the inner join dropped: A2 and A3 have no top-up and are not in the result. T3 carries agent code A9, which stands in no agent row, so it is missing too.

Every combination, asked for on purpose: CROSS JOIN pairs each of the 3 agents with each of the 3 top-ups. Written as CROSS it is a decision; the same 9 rows reached by leaving ON off an ordinary join are an accident.

The decision worth naming: float_top_up carries no foreign key, because the ledger has to hold T3, whose agent is missing, for the outer joins to have anything to show.

One result is shorter than the tables it came from and the other is longer, and both are ordinary.

The Master Key: *A join pairs rows, so a row that matches twice appears twice, and a CROSS JOIN that states no condition returns every combination there is: 3 agents and 3 top-ups give 9 rows.*

BUILD Question 140

The filter mentioned an amount, and two agents who have never received a shilling of float dropped off the list with it. Neither of them has a row in the table the filter names, which is exactly why they went.

Show the LEFT JOIN that keeps the agents with no top-up, show what a WHERE condition on the right-hand column does to them, and give the corrected form with the condition in the ON clause.

Keep every agent, with the top-ups that belong to each.

Then ask only for top-ups above 20000.00, with the condition in the WHERE clause.

Then ask the same thing again with the condition in the ON clause.

Print all three results, and count the rows of each.

Model Answer

The LEFT JOIN returns 4 rows, more than either table holds: A1 twice, then A2 and A3 once each with NULL in both columns taken from the right table; adding WHERE t.amount > 20000.00 cuts it to 1 row, and moving that same condition into the ON clause gives 3 rows with the unmatched agents still standing.

```
 1  SELECT a.agent_code, a.village, t.top_up_id, t.amount
 2  FROM agent AS a
 3  LEFT JOIN float_top_up AS t
 4         ON t.agent_code = a.agent_code
 5  ORDER BY a.agent_code, t.top_up_id;
 6
 7  SELECT a.agent_code, a.village, t.top_up_id, t.amount
 8  FROM agent AS a
 9  LEFT JOIN float_top_up AS t
10         ON t.agent_code = a.agent_code
11  WHERE t.amount > 20000.00
12  ORDER BY a.agent_code;
13
14  SELECT a.agent_code, a.village, t.top_up_id, t.amount
15  FROM agent AS a
16  LEFT JOIN float_top_up AS t
17         ON t.agent_code = a.agent_code
18        AND t.amount > 20000.00
19  ORDER BY a.agent_code;
```

The server replies

```
+------------+---------+-----------+----------+
| agent_code | village | top_up_id | amount   |
+------------+---------+-----------+----------+
| A1         | Kiwira  | T1        | 40000.00 |
| A1         | Kiwira  | T2        | 15000.00 |
| A2         | Ilembo  | NULL      |     NULL |
| A3         | Ipinda  | NULL      |     NULL |
+------------+---------+-----------+----------+
+------------+---------+-----------+----------+
| agent_code | village | top_up_id | amount   |
+------------+---------+-----------+----------+
| A1         | Kiwira  | T1        | 40000.00 |
+------------+---------+-----------+----------+
+------------+---------+-----------+----------+
| agent_code | village | top_up_id | amount   |
+------------+---------+-----------+----------+
| A1         | Kiwira  | T1        | 40000.00 |
| A2         | Ilembo  | NULL      |     NULL |
| A3         | Ipinda  | NULL      |     NULL |
+------------+---------+-----------+----------+
```

Program 4.32: *A LEFT JOIN, the same join filtered in WHERE, and the correction in ON.*

Where the NULLs came from: Neither table holds a NULL. The server manufactures one in every column of float_top_up on the rows that matched nothing, because a result is rectangular and those columns have to hold something.

What the WHERE clause did to them: A comparison with NULL is unknown and never true, so WHERE removed A2 and A3 the moment it mentioned t.amount. What is left is the inner join, with the word LEFT still standing in the statement.

Why ON behaves differently: ON is applied while the pairs are being formed, before the outer join puts the unmatched left rows back. It decides which pairs form, and an agent that forms none is kept anyway, with NULLs.

One clause moved one line changes 1 row into 3, and only the first and the third results are outer joins.

The Master Key: *An outer join manufactures the NULLs on its unmatched side, so a WHERE condition on a right-hand column turns a LEFT JOIN back into an inner join; put such a condition in the ON clause, where it decides pairing and not survival.*

BUILD Question 141

One top-up in the book at Ilembo carries an agent number that nobody recognises, and an ordinary join loses it without saying so. Turning the join round is enough to make it visible, and there are two ways of turning it round.

Demonstrate a RIGHT JOIN that keeps the top-up whose agent is missing, obtain the same result with a LEFT JOIN and the tables written the other way round, and say which form to prefer.

Keep every top-up, with the agent who paid it in where there is one.

Write it first as a RIGHT JOIN, with agent named first.

Then write it as a LEFT JOIN, with float_top_up named first.

The two results must agree row for row.

Model Answer

The RIGHT JOIN keeps every row of float_top_up and returns 3 rows, T3 among them with NULL in agent_code and village because agent A9 stands in no agent row; naming the two tables the other way round and joining with LEFT returns exactly the same 3 rows, and that is the form to prefer.

```
 1  SELECT a.agent_code, a.village, t.top_up_id, t.amount
 2  FROM agent AS a
 3  RIGHT JOIN float_top_up AS t
 4          ON t.agent_code = a.agent_code
 5  ORDER BY t.top_up_id;
 6
 7  SELECT a.agent_code, a.village, t.top_up_id, t.amount
 8  FROM float_top_up AS t
 9  LEFT JOIN agent AS a
10         ON a.agent_code = t.agent_code
11  ORDER BY t.top_up_id;
```

The server replies

```
+------------+---------+-----------+----------+
| agent_code | village | top_up_id | amount   |
+------------+---------+-----------+----------+
| A1         | Kiwira  | T1        | 40000.00 |
| A1         | Kiwira  | T2        | 15000.00 |
| NULL       | NULL    | T3        | 90000.00 |
+------------+---------+-----------+----------+
+------------+---------+-----------+----------+
| agent_code | village | top_up_id | amount   |
+------------+---------+-----------+----------+
| A1         | Kiwira  | T1        | 40000.00 |
| A1         | Kiwira  | T2        | 15000.00 |
| NULL       | NULL    | T3        | 90000.00 |
+------------+---------+-----------+----------+
```

Program 4.33: *A RIGHT JOIN, and the LEFT JOIN that returns the same rows.*

What the right join kept: T3 is the row an inner join loses. RIGHT keeps it and fills the two columns that came from agent with NULL, which is the sign on the page that this top-up belongs to nobody on the list.

The same answer, written the other way: Naming float_top_up first and joining agent to it with LEFT gives row for row the same result. Every RIGHT JOIN can be rewritten this way, and the ON condition is untouched by the change.

Which form to prefer: Write LEFT. The table whose rows are all kept is then the first one named, so the statement is read in the order it is written, and a chain of joins mixing LEFT with RIGHT has to be read in both directions at once.

RIGHT JOIN is correct, standard and rarely worth writing.

The Master Key: *A RIGHT JOIN keeps every row of the table on its right and can always be rewritten as a LEFT JOIN with the tables swapped, so prefer LEFT and let the table whose rows are all kept be the one named first.*

BUILD Question 142

Some servers can keep the unmatched rows of both tables in one statement. This one cannot, and it says so before any table is read, which leaves the work to two joins and the word that stitches their results together.

Show what this server does with FULL OUTER JOIN, then produce the same result as the UNION of a LEFT JOIN and a RIGHT JOIN.

Ask for every agent and every top-up in one statement, matched where they match.

Write it first as a FULL OUTER JOIN and print what the server says.

Then write it as a LEFT JOIN and a RIGHT JOIN joined by UNION.

Order the combined result by agent and then by top-up.

Model Answer

MariaDB and MySQL have no FULL OUTER JOIN, so the statement is refused with ERROR 1064 before any table is read; the result it would have produced is assembled from a LEFT JOIN and a RIGHT JOIN combined with UNION, and comes to 5 rows.

```
1  SELECT a.agent_code, a.village, t.top_up_id, t.amount
2  FROM agent AS a
3  FULL OUTER JOIN float_top_up AS t
4               ON t.agent_code = a.agent_code;
```

The server refuses it

```
ERROR 1064 (42000) at line 1: You have an error in your SQL syntax; check the manual that
corresponds to your MariaDB server version for the right syntax to use near 'FULL OUTER JOIN
float_top_up AS t
             ON t.agent_code = a.agent_code' at line 3
```

Program 4.34: *FULL OUTER JOIN, which this server does not implement.*

```
 1  SELECT a.agent_code, a.village, t.top_up_id, t.amount
 2  FROM agent AS a
 3  LEFT JOIN float_top_up AS t
 4         ON t.agent_code = a.agent_code
 5  UNION
 6  SELECT a.agent_code, a.village, t.top_up_id, t.amount
 7  FROM agent AS a
 8  RIGHT JOIN float_top_up AS t
 9          ON t.agent_code = a.agent_code
10  ORDER BY agent_code, top_up_id;
```

The server replies

```
+------------+---------+-----------+----------+
| agent_code | village | top_up_id | amount   |
+------------+---------+-----------+----------+
| NULL       | NULL    | T3        | 90000.00 |
| A1         | Kiwira  | T1        | 40000.00 |
| A1         | Kiwira  | T2        | 15000.00 |
| A2         | Ilembo  | NULL      |     NULL |
| A3         | Ipinda  | NULL      |     NULL |
+------------+---------+-----------+----------+
```

Program 4.35: *The union of a LEFT JOIN and a RIGHT JOIN, which does the same work.*

What the server objected to: The complaint is a syntax error and it names the text it stopped at, so nothing ran. FULL OUTER JOIN is standard SQL; this server has never implemented it, and the fault lies with the server and not with the writing.

How the union rebuilds it: The LEFT JOIN supplies 4 rows and the RIGHT JOIN 3, and 2 of them are the same pairs twice over. UNION removes the duplicates, so the result holds 5 rows. UNION ALL would print those 2 pairs twice.

Ordering the combined result: ORDER BY at the end applies to the whole union rather than to either half, so it names the result columns and cannot qualify them with an alias, which is the one place in this section where a bare column name is right.

Two joins and one word do the work that the missing kind of join would have done alone.

The Master Key: *MySQL and MariaDB refuse FULL OUTER JOIN as a syntax error, and the same result is built as the UNION of a LEFT JOIN and a RIGHT JOIN, which keeps every matched row once and every unmatched row from both sides.*

BUILD Question 143

The float book shows two payments in from the same agent, and the supervisor wants every such pair listed on one line. Both rows belong to the same table, so the table has to be read twice in a single statement.

Write a self join that lists every pair of top-ups paid in by one agent, with the village, using two aliases on the one table.

Each line of the answer carries one pair of top-ups belonging to one agent.

A top-up must not be paired with itself, and no pair may be listed twice.

Show the village as well, which means joining the agent table once more.

Order the answer by the first top-up of each pair.

Model Answer

The self join names float_top_up twice, as t1 and t2, and pairs each top-up with a later top-up of the same agent, which answers the question directly: 1 pair exists, T1 of 40000.00 with T2 of 15000.00, both paid in by A1 of Kiwira.

```
 1  SELECT a.village, t1.top_up_id AS first_id,
 2         t1.amount AS first_amount, t2.top_up_id AS second_id,
 3         t2.amount AS second_amount
 4  FROM float_top_up AS t1
 5  INNER JOIN float_top_up AS t2
 6          ON t2.agent_code = t1.agent_code
 7         AND t2.top_up_id > t1.top_up_id
 8  INNER JOIN agent AS a
 9          ON a.agent_code = t1.agent_code
10  ORDER BY t1.top_up_id;
```

The server replies

```
+---------+----------+--------------+-----------+---------------+
| village | first_id | first_amount | second_id | second_amount |
+---------+----------+--------------+-----------+---------------+
| Kiwira  | T1       |     40000.00 | T2        |      15000.00 |
+---------+----------+--------------+-----------+---------------+
```

Program 4.36: *A self join listing every pair of top-ups paid in by one agent.*

Why the aliases are the whole trick: t1 and t2 are one table read twice, and every column has to say which reading it belongs to. Take the aliases away and the server answers ERROR 1066, not unique table or alias, before it looks at anything else.

What the inequality is doing: Without t2.top_up_id > t1.top_up_id every row would pair with itself, and every genuine pair would appear twice, once in each order. The inequality keeps one line per pair.

Three table references, two tables: agent is joined once more to supply the village, so the statement names 3 tables and reads only 2 of them. A multi-table join is built exactly so, one JOIN and one ON for each table added.

One table, read twice under two names, answers a question about pairs of its own rows.

The Master Key: *A self join reads one table twice under two aliases and compares the copies in its ON clause, where an inequality keeps each pair once instead of listing it twice and pairing every row with itself.*

HOT Question 144

Move one condition out of WHERE and into ON, and a result of one row becomes a result of three. The clerk at Kiwira who moved it back again lost two agents from her list and never saw them go.

Judge whether a condition belongs in the ON clause or the WHERE clause, and state the rule that decides it.

Model Answer

A condition belongs in the ON clause when it says which rows of the two tables belong together, and in the WHERE clause when it says which rows of the finished result are wanted; the rule that decides it is that on an inner join the two clauses return the same rows, while on an outer join any condition mentioning the optional side must sit in ON or it cancels the outer join.

What each clause acts on: ON is read while the pairs are being formed, and an outer join then adds back the rows that formed no pair, with NULLs. WHERE is read afterwards, on the finished result, and it cannot tell a manufactured NULL from any other.

On an inner join it makes no difference: An inner join keeps nothing but pairs, so a condition that rejects a pair rejects the row wherever it was written. The two forms return the same rows, and the choice is then about reading: the joining condition in ON, the filter in WHERE.

On an outer join it decides the answer: A condition on the optional side, written in WHERE, removes every manufactured row, because a comparison with NULL is unknown. The outer join is undone and an inner join is left, announced by no message at all.

The preserved side behaves differently: A condition on the kept table, written in ON, does not remove its rows at all. It only stops them pairing, so they return with NULLs beside them. Kiwira in the ON clause still leaves Ilembo and Ipinda in the result, which surprises everybody once.

The one condition on the optional side that belongs in WHERE: IS NULL. After a LEFT JOIN, WHERE t.top_up_id IS NULL keeps only the rows that matched nothing, which is how the agents with no float top-up are listed. It reads the manufactured NULL on purpose, and nothing else can.

The judgement: Put the joining condition in ON always, and put any other condition in ON as well whenever it mentions the optional side of an outer join. Leave in WHERE what is about the finished result, and IS NULL where the unmatched rows are the answer.

Which clause a condition sits in stops being a matter of taste the moment one of the joins is outer.

The Master Key: *ON decides which rows pair and WHERE decides which rows of the finished result survive, so on an inner join the two are interchangeable, and on an outer join a condition about the optional side must sit in ON or it silently turns the outer join into an inner one.*

Subqueries, views and stored routines

A subquery is a complete SELECT written inside another statement, and what decides its behaviour is not how it looks but how often it is run. This section separates a plain subquery, worked out once, from a correlated one, worked out again for every row of the outer query; it sets IN against EXISTS and shows what one NULL does to NOT IN; it combines two results with UNION and with UNION ALL; and it ends with three things a database stores besides rows, which are the view, the stored procedure and the trigger. The work is done on a guest house register at Ivuna holding 5 rooms and 4 bookings, small enough for every row of every result to be counted by eye.

BINDER Question 145

The rate card taped above the desk at Ivuna lists five rooms. One question, asked twice in almost the same words, sends the server to that card once in the first form and five times in the second.

Explain what a subquery is, classify subqueries by what they return and by whether they are correlated, and say how often the inner query is evaluated in each case.

Model Answer

A subquery is a complete SELECT written inside brackets within another statement, and it is classified two ways at once: by what it hands back, one value, one row or a set of rows, and by whether it names a column of the query outside it, which is the one thing that decides whether it is worked out once or once for every row the outer query examines.

Where a subquery may stand: It may stand in the WHERE clause, the HAVING clause, the SELECT list and the FROM clause, where it becomes a derived table and must be given a name of its own. The brackets mark where the inner statement ends.

Scalar, row and table subqueries: A scalar subquery returns one value, one row of one column, and may stand wherever a value may stand. A row subquery returns one row of several columns and is compared with a bracketed list. A table subquery returns a set of rows and is read by IN, EXISTS, ANY or ALL.

A plain subquery is evaluated once: Nothing inside the brackets refers to the query outside them, so its answer cannot change while the outer query runs. The server works the average out once and compares all 5 rooms against that same figure of 28200.00.

A correlated subquery is evaluated once for every outer row: It names a column of the outer query, here r.wing, so its answer moves as the outer query moves from room to room: 5 rooms means 5 evaluations. The alias on the outer table is what makes that reference possible, so it is never optional here.

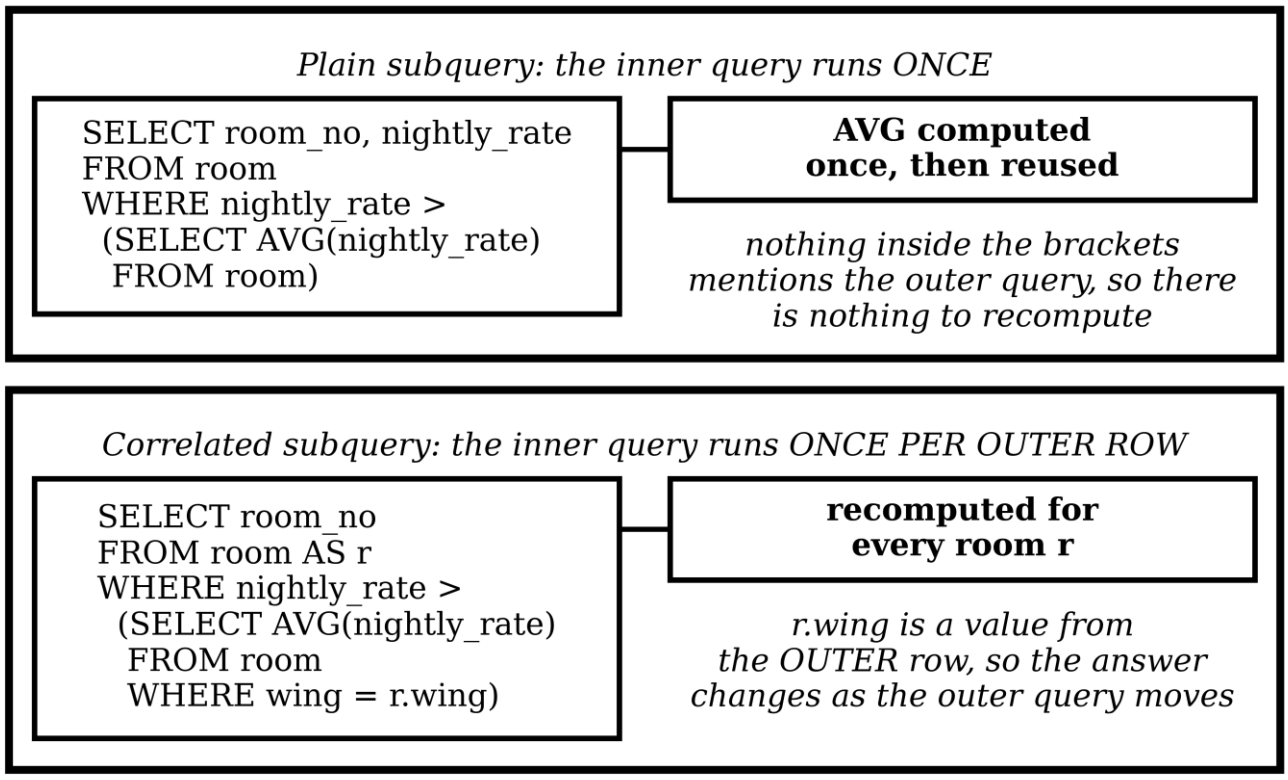


Figure 4.28: *The same comparison written twice, and the one word that turns a single evaluation into one for every room.*

Classified by	Kind	What it means
what it returns	scalar	one row of one column, so it stands where a value stands and may be compared with a plain equals sign
what it returns	row	one row of several columns, compared with a bracketed list of the same length
what it returns	table	a set of rows, read by IN, EXISTS, ANY or ALL rather than by comparison with a value
what it mentions	plain	nothing inside the brackets comes from outside them, so the server evaluates it once
what it mentions	correlated	a column of the outer query appears inside the brackets, so the server evaluates it once for every outer row

Table 4.29: *The two ways a subquery is classified, and what each kind means when it is read.*

The whole reading of a subquery turns on one thing, which is whether anything inside the brackets came from outside them.

The Master Key: *A subquery is a SELECT written inside another statement, classified by what it returns and by whether it names a column from outside itself; a plain one is evaluated once, and a correlated one once for every row of the outer query.*

BINDER Question 146

By evening at Msangano the occupancy list and the register disagreed, and the guest standing at the desk was in one of them and not the other. The list had been copied into a second table so that reading it would be quicker.

Explain what CREATE VIEW stores, say what happens to a view when the table under it changes, and state which views can be written through and which cannot.

Model Answer

A view stores a query and no rows whatever: CREATE VIEW occupied_tonight AS SELECT saves that SELECT under a name, and every read of the view runs it again over the tables underneath, so the rows it shows are worked out when somebody reads them and are never a copy.

What is actually stored: The server keeps the text of the SELECT and the names of its columns, and nothing else. No space is taken for rows, which is why creating a view over a huge table is instant.

Why a change underneath shows at once: A booking inserted into booking appears at the very next read of occupied_tonight, with no statement run against the view. Nothing was copied, so there is nothing to bring up to date and no rebuild to forget.

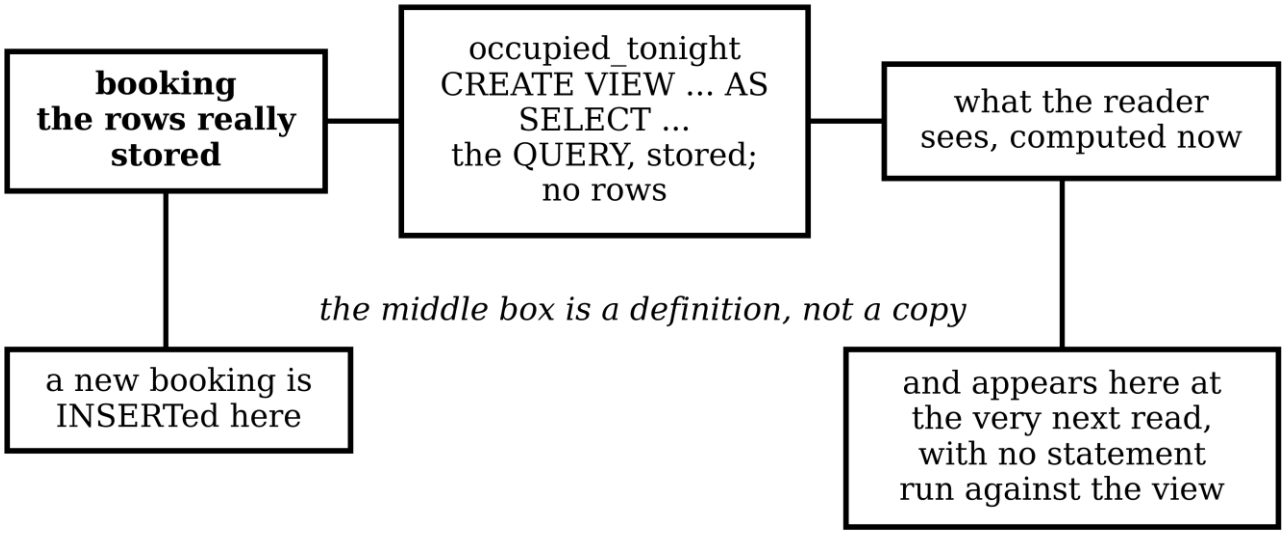


Figure 4.29: *What a view holds, and where the rows come from when somebody reads it.*

What a view is for: It gives a name to a query worth repeating, so one careful statement is written once and read by name afterwards. It also shows a reader some columns of a table and not others, having only the columns its SELECT names.

The views you can write through: A view over one table, with no aggregate, no DISTINCT, no GROUP BY and no UNION, is updatable, because each of its rows stands for one row underneath: an UPDATE or DELETE through it reaches the base table, and an INSERT does too if the view names every column that table requires.

Why an aggregate makes a view read-only: A row of a view that averages by wing stands for 3 rooms at once, so an instruction to change it names no single row the server could alter. It refuses with ERROR 1288, naming the view.

A view is a stored question, answered afresh every time somebody asks it.

The Master Key: *CREATE VIEW stores the query and not the rows, so a view is recomputed at every read and never falls out of step with its base tables; it can be written through only when each of its rows stands for one row of one table.*

REAL Question 147

At the foot of the printed room list at Kamsamba somebody has added three lines about subqueries, and the clerk typed all three. Two of them cost her rooms she could see with her own eyes, standing empty on the board behind her.

Exactly one of the three lines below is correct. Say which, name the fault in each of the other two, state exactly what the server does, and give the correction.

A. A view keeps its own copy of the rows, so a booking added to the table underneath is not in the view until the view is rebuilt.

B. NOT IN with a subquery returns the rows that match nothing the subquery returned, whatever that subquery happens to contain.

C. A subquery compared with an equals sign must return at most one row, or the statement fails.

Model Answer

C is the correct one, and it is correct as written: a subquery compared with an equals sign must return at most one row. A is wrong about the one thing a view is, and B is wrong only when the subquery returns a NULL, which is when it is wrong in the most expensive way available: the statement comes back with no rows at all, and with no error and no warning to say why.

Why A is wrong: A view stores the query, not the rows. The rows are worked out when the view is read, so a booking inserted into booking is in occupied_tonight the next time anybody looks, and the two can never disagree.

What the server does about A: There is no rebuilding statement to run, because there is nothing to rebuild. A reader who waits for one waits for ever, and a reader who builds a real copy of the rows in a second table has taken on the job of keeping it true.

Why B is wrong: NOT IN expands into a chain of comparisons joined by AND. Compare a room with a NULL and the answer is unknown, so the chain can only be false or unknown and never true, and WHERE keeps only what is true.

What the server does about B: It returns the empty set. Booking B004 has no room allocated yet, so a NOT IN over booking room numbers returns 0 rows when 2 rooms have never been let, and the server reports no error and raises no warning.

The correction for B: Exclude the NULLs inside the subquery with IS NOT NULL, or ask the question the other way with NOT EXISTS, which tests whether any row comes back rather than comparing values and is therefore untouched by the NULL.

What happens when C is broken: The garden wing holds 2 rooms, so a subquery asked for its rate returns 2 rows and the server stops with ERROR 1242. Nothing is compared and nothing is returned, which at least is a fault a reader can see.

```
1  SELECT room_no, nightly_rate
2  FROM room
3  WHERE nightly_rate = (SELECT nightly_rate
4                        FROM room
5                        WHERE wing = 'garden');
```

The server refuses it

```
ERROR 1242 (21000) at line 1: Subquery returns more than 1 row
```

Program 4.37: *A subquery compared with an equals sign, returning more than one row.*

Written in the message	What the server actually does	The correction
A. A view keeps its own copy of the rows	stores the query and runs it again at every read, so a new booking is in the view immediately	read the view; there is nothing to rebuild
B. NOT IN returns the rows that match nothing	one NULL in the subquery makes the condition unknown for every row, so it returns 0 rows and says nothing	exclude the NULLs with IS NOT NULL, or write NOT EXISTS
C. A subquery compared with an equals sign returns at most one row	runs, and stops with ERROR 1242 the moment a second row comes back	correct as it stands

Table 4.30: *The three lines from the message, what the server does about each, and what each should have said.*

The one that reads like a warning is the true one, and the friendly one in the middle empties a result without a word.

The Master Key: *A view stores the query and not the rows; a subquery compared with an equals sign must return at most one row or the server raises ERROR 1242; and NOT IN over a subquery holding one NULL returns nothing at all, silently.*

BUILD Question 148

Twenty eight thousand two hundred shillings is the average room rate at Ivuna, and no room is priced at it. Two rooms stand above that figure, and a different pair stands above the average of its own wing.

Create the room table of the register, list the rooms priced above the average for the whole guest house, and then list the rooms priced above the average for their own wing.

The room table holds 5 rooms: R01 and R02 in the garden wing, R03, R04 and R05 in the annexe.

List every room whose nightly rate is above the average rate of all 5 rooms.

Then list every room whose nightly rate is above the average rate of its own wing.

Both answers must come from a subquery, not from an average worked out by hand.

Model Answer

The plain subquery works out one average for the whole house, 28200.00, and returns R01 and R02; the correlated subquery works out an average for each wing as the outer query reaches it, and returns R02 and R05, so R01 leaves and R05 joins.

```
 1  CREATE TABLE room (
 2    room_no      CHAR(3)      NOT NULL,
 3    wing         VARCHAR(10)  NOT NULL,
 4    nightly_rate DECIMAL(8,2) NOT NULL,
 5    PRIMARY KEY (room_no)
 6  ) ENGINE = InnoDB;
 7
 8  INSERT INTO room (room_no, wing, nightly_rate) VALUES
 9    ('R01', 'garden', 30000.00),
10    ('R02', 'garden', 45000.00),
11    ('R03', 'annexe', 18000.00),
12    ('R04', 'annexe', 22000.00),
13    ('R05', 'annexe', 26000.00);
14
15  SELECT room_no, wing, nightly_rate
16  FROM room
17  WHERE nightly_rate > (SELECT AVG(nightly_rate) FROM room)
18  ORDER BY room_no;
19
20  SELECT room_no, wing, nightly_rate
21  FROM room AS r
22  WHERE nightly_rate > (SELECT AVG(nightly_rate)
23                        FROM room
24                        WHERE wing = r.wing)
25  ORDER BY room_no;
```

The server replies

```
+---------+--------+--------------+
| room_no | wing   | nightly_rate |
+---------+--------+--------------+
| R01     | garden |     30000.00 |
| R02     | garden |     45000.00 |
+---------+--------+--------------+
+---------+--------+--------------+
| room_no | wing   | nightly_rate |
+---------+--------+--------------+
| R02     | garden |     45000.00 |
| R05     | annexe |     26000.00 |
+---------+--------+--------------+
```

Program 4.38: *The room table, an average taken once, and the same average taken wing by wing.*

What makes the second one correlated: The words wing = r.wing put a column of the outer query inside the brackets, and the alias r on the outer FROM clause is the only reason the inner query may name a value from the row being examined.

How often each inner query runs: The first runs once: one average, then 5 comparisons against it. The second runs 5 times, once for each room the outer query reaches, and hands back 37500.00 for a garden room and 22000.00 for an annexe room.

Why R01 falls out and R05 comes in: R01 at 30000.00 is above the house average of 28200.00 but below its own wing average of 37500.00, and R05 at 26000.00 is below the house average and above the annexe average of 22000.00.

The decision worth naming: The rate is DECIMAL rather than a binary floating type, so a comparison with an average cannot drift by a fraction of a shilling.

Two statements asking almost the same thing return different rooms, and each is right for its own question.

The Master Key: *A plain subquery gives every outer row the same answer, while a correlated one names a column of the outer query and is worked out again for each row, which is why one measures a room against the house and the other against its own wing.*

BUILD Question 149

The register at Ivuna shows one booking taken by telephone with no room allocated to it yet. That single empty cell is enough to make a perfectly sensible question about empty rooms come back with nothing in it at all.

Create the booking table, list the rooms that appear in it with IN, count the rooms that do not with NOT IN, and then obtain the true answer with NOT EXISTS and with a corrected NOT IN.

The booking table holds 4 bookings, one of which has no room allocated yet.

List the rooms that appear in the register at all.

Then count the rooms that appear nowhere in it.

Then answer that same question twice more, correctly.

Model Answer

IN returns the 3 rooms named in the register; NOT IN over the same subquery returns 0 rooms although 2 have never been let, because booking B004 carries NULL in room_no; NOT EXISTS returns R04 and R05, and NOT IN does too once the NULLs go.

```
 1  CREATE TABLE booking (
 2    booking_ref CHAR(4)     NOT NULL,
 3    room_no     CHAR(3)         NULL,
 4    guest_name  VARCHAR(20) NOT NULL,
 5    nights      INT         NOT NULL,
 6    status      VARCHAR(10) NOT NULL,
 7    PRIMARY KEY (booking_ref),
 8    FOREIGN KEY (room_no) REFERENCES room (room_no)
 9  ) ENGINE = InnoDB;
10
11  INSERT INTO booking
12    (booking_ref, room_no, guest_name, nights, status)
13  VALUES
14    ('B001', 'R01', 'Neema',  2, 'in house'),
15    ('B002', 'R03', 'Juma',   1, 'departed'),
16    ('B003', 'R02', 'Asha',   3, 'in house'),
17    ('B004', NULL,  'Baraka', 2, 'booked');
18
19  SELECT room_no, wing
20  FROM room
21  WHERE room_no IN (SELECT room_no FROM booking)
22  ORDER BY room_no;
23
24  SELECT COUNT(*) AS rooms_never_let
25  FROM room
26  WHERE room_no NOT IN (SELECT room_no FROM booking);
```

The server replies

```
+---------+--------+
| room_no | wing   |
+---------+--------+
| R01     | garden |
| R02     | garden |
| R03     | annexe |
+---------+--------+
+-----------------+
| rooms_never_let |
+-----------------+
|               0 |
+-----------------+
```

Program 4.39: *The booking table, the rooms that appear in it, and the count of the rooms that do not.*

```
 1  SELECT room_no, wing
 2  FROM room AS r
 3  WHERE NOT EXISTS (SELECT 1
 4                    FROM booking AS b
 5                    WHERE b.room_no = r.room_no)
 6  ORDER BY room_no;
 7
 8  SELECT room_no, wing
 9  FROM room
10  WHERE room_no NOT IN (SELECT room_no
11                        FROM booking
12                        WHERE room_no IS NOT NULL)
13  ORDER BY room_no;
```

The server replies

```
+---------+--------+
| room_no | wing   |
+---------+--------+
| R04     | annexe |
| R05     | annexe |
+---------+--------+
+---------+--------+
| room_no | wing   |
+---------+--------+
| R04     | annexe |
| R05     | annexe |
+---------+--------+
```

Program 4.40: *The same question answered with NOT EXISTS, and with NOT IN once the NULLs are excluded.*

What NOT IN really means: It is a chain of not-equals joined by AND. R04 is not R01, not R02 and not R03, all true, and then is not NULL, which is unknown, so the whole chain is unknown and the row goes. No room survives.

Why IN survives the same NULL: IN is a chain of equals joined by OR, and one true is enough. R01 matches R01 whatever else the subquery holds, so the NULL fails to help instead of poisoning the answer.

What EXISTS asks instead: EXISTS asks whether the inner query returns any row at all, and never looks at the values. For R04 the correlated inner query finds no booking and returns nothing, so NOT EXISTS is true and R04 is kept.

The decision worth naming: room_no is nullable on purpose, because a booking exists before a room is chosen for it. The NULL is data, not a fault, and any statement reading that column must survive it.

Nothing in the first answer looks wrong, which is precisely what makes it costly.

The Master Key: *NOT IN over a subquery holding even one NULL returns no rows at all, silently, while IN is unharmed by it; write NOT EXISTS, or exclude the NULLs, whenever the column may be empty.*

BUILD Question 150

Six lines came back for four rooms, and the housekeeper at Msangano had to read the list twice before she saw that two of them were repeats. One word in the statement decides whether the repeats are there at all.

Combine the two lists with UNION ALL and then with UNION, count the rows each returns, and show what the server does when the two SELECT statements do not have the same number of columns.

One list of room numbers for the housekeeper, from two sources.

The first part: every room priced at 25000.00 or more, which is made up daily whether it is let or not.

The second part: every room named in the booking register.

Print the combined list with the repeats and then without them, and count both.

Model Answer

UNION ALL returns 6 rows, putting the second result after the first and changing nothing, and UNION returns 4, having removed the 2 rows already there; the order they come back in, R05 before R03, is a by-product of that removal and is promised by nothing.

```
 1  SELECT room_no
 2  FROM room
 3  WHERE nightly_rate >= 25000.00
 4  UNION ALL
 5  SELECT room_no
 6  FROM booking
 7  WHERE room_no IS NOT NULL;
 8
 9  SELECT room_no
10  FROM room
11  WHERE nightly_rate >= 25000.00
12  UNION
13  SELECT room_no
14  FROM booking
15  WHERE room_no IS NOT NULL;
```

The server replies

```
+---------+
| room_no |
+---------+
| R01     |
| R02     |
| R05     |
| R01     |
| R02     |
| R03     |
+---------+
+---------+
| room_no |
+---------+
| R01     |
| R02     |
| R05     |
| R03     |
+---------+
```

Program 4.41: *The same two lists combined with UNION ALL and then with UNION.*

```
1  SELECT room_no, nightly_rate
2  FROM room
3  WHERE nightly_rate >= 25000.00
4  UNION
5  SELECT room_no
6  FROM booking
7  WHERE room_no IS NOT NULL;
```

The server refuses it

```
ERROR 1222 (21000) at line 1: The used SELECT statements have a different number of columns
```

Program 4.42: *The same union with one extra column on the first SELECT.*

What each word does: UNION ALL joins the two results end to end and keeps every line. UNION compares whole rows across the result and keeps one of each, so the 2 rooms in both lists come back once.

The rule both forms share: Both SELECT statements must return the same number of columns, or the server stops with ERROR 1222 before reading a table; paired columns must also hold comparable types, and the result takes its column names from the first SELECT.

Order is not part of the bargain: Students are often taught, wrongly, that UNION sorts what it returns. This server did not: R05 came back before R03. Removing the repeats disturbs the order without promising a new one, and only ORDER BY settles it.

Which of the two to write: Write UNION ALL when repeats cannot arise or are wanted, since nothing has to be compared; write UNION when the same fact really has been reached twice and should count once.

One word removes two lines here, and would remove two real guests sharing a name.

The Master Key: *UNION removes the duplicate rows of the combined result and UNION ALL keeps them, both need the same number of columns with comparable types, and neither promises an order, so write ORDER BY when the order matters.*

BUILD Question 151

The board behind the desk at Kamsamba is rubbed out and written again every time a guest arrives. The list of rooms occupied tonight is not rubbed out by anybody, and it is right every single time it is read.

Create the occupied_tonight view, read it, insert a booking into the table underneath, read the view again without touching it, update a guest name through it, and then show what the server does to a view built on an average.

occupied_tonight carries the reference, room and guest of every booking marked in house.

Read the view, then insert booking B005 for Halima in room R04, marked in house.

Read the view again, with no statement of any kind run against the view itself.

Then correct a guest name through the view, and try the same thing on a view of an average.

Model Answer

The view returns 2 rows, and after one INSERT into booking it returns 3, with nothing at all run against the view in between, because reading it runs its stored SELECT again; an UPDATE through it reaches booking, while the second view groups an average, so each of its rows stands for several rooms and the server refuses to update it with ERROR 1288.

```
 1  CREATE VIEW occupied_tonight AS
 2  SELECT booking_ref, room_no, guest_name
 3  FROM booking
 4  WHERE status = 'in house';
 5
 6  SELECT booking_ref, room_no, guest_name
 7  FROM occupied_tonight;
 8
 9  INSERT INTO booking
10    (booking_ref, room_no, guest_name, nights, status)
11  VALUES ('B005', 'R04', 'Halima', 1, 'in house');
12
13  SELECT booking_ref, room_no, guest_name
14  FROM occupied_tonight;
15
16  UPDATE occupied_tonight
17  SET guest_name = 'Neema Mwaka'
18  WHERE booking_ref = 'B001';
19
20  SELECT booking_ref, guest_name
21  FROM booking
22  WHERE booking_ref = 'B001';
```

The server replies

```
+-------------+---------+------------+
| booking_ref | room_no | guest_name |
+-------------+---------+------------+
| B001        | R01     | Neema      |
| B003        | R02     | Asha       |
+-------------+---------+------------+
+-------------+---------+------------+
| booking_ref | room_no | guest_name |
+-------------+---------+------------+
| B001        | R01     | Neema      |
| B003        | R02     | Asha       |
| B005        | R04     | Halima     |
+-------------+---------+------------+
+-------------+-------------+
| booking_ref | guest_name  |
+-------------+-------------+
| B001        | Neema Mwaka |
+-------------+-------------+
```

Program 4.43: *A view created, read, read again after an insert underneath it, and written through.*

```
 1  CREATE VIEW wing_average AS
 2  SELECT wing, AVG(nightly_rate) AS average_rate
 3  FROM room
 4  GROUP BY wing;
 5
 6  SELECT wing, average_rate
 7  FROM wing_average;
 8
 9  UPDATE wing_average
10  SET average_rate = 30000.00
11  WHERE wing = 'annexe';
```

The server replies, and then refuses

```
+--------+--------------+
| wing   | average_rate |
+--------+--------------+
| annexe | 22000.000000 |
| garden | 37500.000000 |
+--------+--------------+
ERROR 1288 (HY000) at line 9: The target table wing_average of the UPDATE is not updatable
```

Program 4.44: *A view over an average, and the update it refuses.*

Nothing was rebuilt between the two reads: The only statement run between them was an INSERT into booking. The view gained a row because the SELECT it stores was run a second time, over a table that now holds one more row that satisfies its WHERE clause.

Why the first view can be written through: It reads one table, carries its primary key and aggregates nothing, so each view row stands for exactly one booking row. The UPDATE therefore has a row to change, and the change is visible in booking itself.

Why the second cannot: The annexe line of wing_average stands for 3 rooms at once. There is no single rate to set to 30000.00 and no rule for splitting the change between them, so the server names the view and refuses rather than guessing.

One view answers a question and one view answers a question about a group, and only the first can be written back through.

The Master Key: *A view is the stored query and never a copy, so it follows its base tables at every read; it accepts writes only when each of its rows stands for one base row, which an aggregate destroys.*

BUILD Question 152

The arrivals log at Ivuna filled itself. By closing time it held one line for every guest who had checked in, and no one had typed a statement against it all day. The code that wrote those lines lives inside the database.

Show what the client does to a stored routine typed without DELIMITER, write the routine properly and call it, then create a trigger and show that it fires without being called.

A procedure rooms_in_wing takes a wing name and lists the rooms in that wing.

Type it first without DELIMITER, and print what comes back.

Then create it properly and call it for the annexe.

Then create an AFTER INSERT trigger on booking that writes to arrival_log, and insert one booking.

Model Answer

Without DELIMITER the client cuts the routine at the first semicolon in its body and sends a fragment, which the server refuses with ERROR 1064 against line 6, a line with nothing wrong in it; with DELIMITER the routine arrives whole, the call for the annexe returns its 3 rooms, and the trigger writes 1 log row for an INSERT that never mentions it.

```
1  CREATE PROCEDURE rooms_in_wing(IN want_wing VARCHAR(10))
2  BEGIN
3    SELECT room_no, nightly_rate
4    FROM room
5    WHERE wing = want_wing
6    ORDER BY room_no;
7  END;
```

The server refuses it

```
ERROR 1064 (42000) at line 1: You have an error in your SQL syntax; check the manual that
corresponds to your MariaDB server version for the right syntax to use near '' at line 6
```

Program 4.45: *The procedure typed without DELIMITER, and cut in half by the client.*

```
 1  DELIMITER //
 2
 3  CREATE PROCEDURE rooms_in_wing(IN want_wing VARCHAR(10))
 4  BEGIN
 5    SELECT room_no, nightly_rate
 6    FROM room
 7    WHERE wing = want_wing
 8    ORDER BY room_no;
 9  END //
10
11  DELIMITER ;
12
13  CALL rooms_in_wing('annexe');
```

The server replies

```
+---------+-------------+
| room_no | nightly_rate |
+---------+-------------+
| R03     |     18000.00 |
| R04     |     22000.00 |
| R05     |     26000.00 |
+---------+-------------+
```

Program 4.46: *The same procedure with the delimiter changed, and the call that runs it.*

```
 1  CREATE TABLE arrival_log (
 2    log_id      INT         NOT NULL AUTO_INCREMENT,
 3    booking_ref CHAR(4)     NOT NULL,
 4    guest_name  VARCHAR(20) NOT NULL,
 5    PRIMARY KEY (log_id)
 6  ) ENGINE = InnoDB;
 7
 8  DELIMITER //
 9
10  CREATE TRIGGER log_arrival
11  AFTER INSERT ON booking
12  FOR EACH ROW
13  BEGIN
14    INSERT INTO arrival_log (booking_ref, guest_name)
15    VALUES (NEW.booking_ref, NEW.guest_name);
16  END //
17
18  DELIMITER ;
19
20  INSERT INTO booking
21    (booking_ref, room_no, guest_name, nights, status)
22  VALUES ('B005', 'R04', 'Halima', 1, 'in house');
23
24  SELECT log_id, booking_ref, guest_name
25  FROM arrival_log;
```

The server replies

```
+--------+-------------+------------+
| log_id | booking_ref | guest_name |
+--------+-------------+------------+
|      1 | B005        | Halima     |
+--------+-------------+------------+
```

Program 4.47: *A trigger on booking, and the log row it writes without being asked.*

What DELIMITER changes, and where: DELIMITER is an instruction to the client, not to the server. It stops the client cutting the statement at every semicolon, so the ones inside the body travel with it.

A procedure runs when it is called: CREATE PROCEDURE stores the code under a name and does nothing else; nothing happens until CALL names it. Parameters are declared IN, OUT or INOUT, saying which way each value travels.

A trigger is never called: It names an event instead, AFTER INSERT ON booking, and the server runs it for every row inserted by any client, whether the writer knows of it or not. NEW reads the row that caused it.

The decision worth naming: The trigger fires after the insert, because a log line for a booking the server then rejected would record an arrival that never happened.

One of them runs because somebody asked for it, and the other because something happened.

The Master Key: *A stored procedure and a trigger are both code held in the database, but a procedure waits to be CALLed while a trigger fires on the event it names, and DELIMITER stops the client cutting either at the first semicolon in its body.*

HOT Question 153

One booking at Msangano was saved with no room against it. The program that writes bookings would have refused it, but the clerk who typed it went straight into the client that evening, and nothing else was watching.

A guest house wants one rule kept without exception: every booking marked in house must name a room. Judge whether such a rule belongs inside the database or inside the program that writes the bookings, and give the condition that decides it.

Model Answer

Put the rule inside the database whenever it must hold of the data itself whatever happens to be writing, and put it in the program only when it belongs to the working practice of the day rather than to the data; the condition that decides it is whether the rule would still have to hold if a second program were written next year, and for this rule it would, so it belongs in the database.

What the database side buys: Code held in the server runs for every writer there is: the office program, a clerk at the client, a script loading last month bookings. The rule cannot be forgotten by a program written after it, because it was never in that program.

What the database side costs: A trigger is invisible at the point of use. A row appears in a table nobody wrote to and the statement that caused it gives no sign, so a reader has to know to look for triggers before the behaviour makes sense.

What the program side buys: It is where a reader looks first, it sits beside the rest of the work it belongs to, and it can answer the guest at the desk in words rather than with a number and a table name.

What the program side costs: It holds only inside the programs that carry it. A second program, a data load or one clerk at the client walks straight past it, and the database then holds rows breaking a rule everybody believes is kept.

A trigger is not the first thing to reach for: A declared constraint is checked by the server for every writer, as a trigger is, but it is a declaration rather than a routine, and it names itself in the refusal. Reach for a trigger only where a constraint cannot express the rule.

Where a stored procedure sits in this: A procedure enforces nothing, because it runs only when somebody calls it, so it can be walked past exactly as application code can. Its value is keeping one careful piece of work in one place.

The judgement: Declare the rule as a constraint if it can be declared, use a trigger where it cannot, and keep in the program only what changes with the business rather than with the data. A rule that would survive the program being rewritten does not belong inside it.

The question is not which side is safer in general, but which one every future writer must pass through.

The Master Key: *A rule that must hold of the data whatever writes it belongs in the database, declared as a constraint where possible and enforced by a trigger where not, because a rule kept only in one program is kept only while that program is the only writer.*

Transactions and ACID

A transaction is a group of statements the server carries out as one indivisible piece of work, and this section says where its boundary is drawn, what the four ACID properties promise, and how START TRANSACTION, COMMIT, ROLLBACK and SAVEPOINT draw it. The work is done on a seed supplier's despatch book holding a store table of 3 seed lots and a despatch table of 2 lines, small enough to count every bag by eye. One

thing takes most of the care here. Under the setting MariaDB installs, every statement is already a committed transaction on its own, so a ROLLBACK typed after an UPDATE undoes nothing whatever, and students are often taught, wrongly, that it undoes the change.

BINDER Question 154

Twenty bags leave the store at Buhongwa and are counted onto the lorry. For a moment the books show them nowhere: gone from the shelf, not yet on the load sheet, and anybody reading the store column then would find it twenty bags short.

Explain what a transaction is, say where its boundary is drawn, and name the statements that draw it.

Model Answer

A transaction is a group of statements the server carries out as one indivisible piece of work, so that either every statement in it takes effect or none of them does, and its boundary is drawn by START TRANSACTION in front of the group and by COMMIT or ROLLBACK behind it.

The unit of work is the group: One despatch takes two statements, one to reduce the store column and one to write the despatch line, and neither is worth anything on its own. The transaction is what tells the server that the two belong together and must stand or fall together.

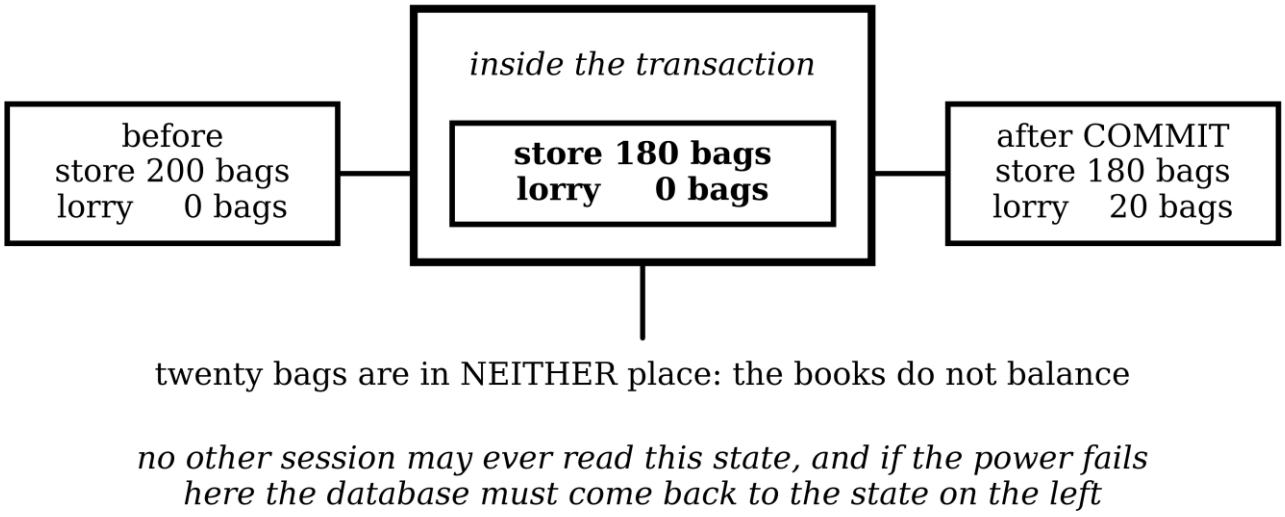


Figure 4.30: *The transaction boundary, and the state in the middle that no other session may read.*

Where the boundary is drawn: START TRANSACTION opens the transaction, and every statement after it belongs to the same unit until the unit is closed. COMMIT closes it and makes every change in it permanent. ROLLBACK closes it and discards every change made since the start, as though the group had never run.

The state in the middle: Between the two statements the twenty bags are in neither place and the books do not balance. That state is real and exists inside the server, and the transaction is what keeps it from being read by any other session and from surviving a power cut.

One open transaction to a session: A session holds one open transaction at a time, so a second START TRANSACTION commits the first before opening the next. Transactions do not nest, and a savepoint is what stands in for nesting.

A transaction settles a question no single statement can settle, which is what belongs with what.

The Master Key: *A transaction is a group of statements carried out as one indivisible unit, opened by START TRANSACTION and closed either by COMMIT, which makes every change in it permanent, or by ROLLBACK, which discards every change made since the start.*

BINDER Question 155

Four words are chalked on the wall above the desk at Nyegezi, one letter each, and the storekeeper can recite them all. Asked which of the four promises that a committed despatch survives a power cut, he names the wrong one.

State the four ACID properties in order and say for each one exactly what it refuses to allow.

Model Answer

ACID is atomicity, consistency, isolation and durability, in that order, and each of the four refuses a different thing: atomicity refuses a transaction that takes effect in part, consistency refuses one that would leave a declared rule broken, isolation refuses one session the sight of another's unfinished work, and durability refuses to lose a transaction once it has been committed.

Atomicity: All of it or none of it. What atomicity refuses is the half-finished transaction: twenty bags gone from the store with no despatch line written, left standing in the books for the next reader to find.

Consistency: Every rule the schema declares holds before the transaction and holds after it. Primary keys, NOT NULL, foreign keys and CHECK constraints are all in force at the boundary, so a transaction that would leave one of them broken is refused rather than committed.

Isolation: What one transaction has not yet committed is not visible to another. The storekeeper reading the store row while a despatch is half written is shown the figure as it stood before that despatch began, and never the state in between.

Durability: Once COMMIT has returned, the change survives. A power cut a second later cannot lose it, because the server writes the change to durable storage before it reports the commit and not afterwards.

Property	What it refuses	What that refusal means in the despatch book
Atomicity	a transaction that takes effect in part	20 bags gone from the store with no despatch line to account for them
Consistency	a transaction that would leave a declared rule broken	a despatch line carrying a seed code that stands in no store row
Isolation	one session the sight of another's uncommitted work	the counter reading 180 bags before the despatch has been committed
Durability	the loss of a transaction that has been committed	the Igoma despatch missing from the book after the power returns

Table 4.31: *The four properties, what each one refuses, and what that refusal means in this despatch book.*

The four are not four ways of saying one thing; each refuses something the other three would allow.

The Master Key: *ACID is atomicity, consistency, isolation and durability: all of it or none of it, every declared rule still holding either side of the boundary, no sight of another session's unfinished work, and no losing what has been committed.*

REAL Question 156

The row read 180 bags before the rollback at Igoma and it reads 180 bags after it. An UPDATE went in, the mistake was seen at once, ROLLBACK was typed, and the server accepted that rollback without a word of complaint.

Explain what autocommit does, show what a ROLLBACK typed after an UPDATE really undoes under it, and give the statement that makes the rollback work.

Model Answer

Autocommit is on by default in MariaDB, and it wraps every single statement in a transaction of its own and commits it the moment it succeeds, so the UPDATE was committed before ROLLBACK was ever typed and the rollback had nothing left to undo: the store still reads 180 bags. Put START TRANSACTION in front of the UPDATE and the same ROLLBACK restores the 200.

```
1  SELECT bags_held FROM store WHERE seed_code = 'MZ01';
2
3  UPDATE store
4  SET bags_held = bags_held - 20
5  WHERE seed_code = 'MZ01';
6
7  ROLLBACK;
8
9  SELECT bags_held FROM store WHERE seed_code = 'MZ01';
```

The server replies

```
+-----------+
| bags_held |
+-----------+
|       200 |
+-----------+
+-----------+
| bags_held |
+-----------+
|       180 |
+-----------+
```

Program 4.48: *A ROLLBACK typed under autocommit, which undoes nothing at all.*

```
 1  SELECT bags_held FROM store WHERE seed_code = 'MZ01';
 2
 3  START TRANSACTION;
 4
 5  UPDATE store
 6  SET bags_held = bags_held - 20
 7  WHERE seed_code = 'MZ01';
 8
 9  ROLLBACK;
10
11  SELECT bags_held FROM store WHERE seed_code = 'MZ01';
```

The server replies

```
+-----------+
| bags_held |
+-----------+
|       200 |
+-----------+
+-----------+
| bags_held |
+-----------+
|       200 |
+-----------+
```

Program 4.49: *The same statements with START TRANSACTION in front, where the rollback does undo them.*

What autocommit does: With autocommit on, the server opens a transaction for each statement, runs it, and commits it as soon as it succeeds. Nothing is left open, so nothing is left to undo. This is the setting a school laboratory installs and the one a student meets at the client.

Why the rollback found nothing: ROLLBACK undoes the transaction that is open in the session, and under autocommit there is none open by the time it is typed. The server accepts the statement, reports no error and changes nothing. The silence is the trap: nothing on the screen says the rollback was empty.

The statement that fixes it: START TRANSACTION opens a transaction explicitly and suspends autocommit until that transaction ends. Every statement after it belongs to the transaction, so ROLLBACK now has work to do, which is why the second listing comes back at 200 bags.

The other way in: SET autocommit = 0 turns the setting off for the whole session, and the first statement that touches data then opens a transaction by itself with no START TRANSACTION needed. It stays open until COMMIT or ROLLBACK, and the next statement opens another.

Reading the two results: The listings differ by one line and by 20 bags. A common error is to write a rollback example with no START TRANSACTION in it at all and report that the change was undone; run it, and the store row shows that it was not.

A rollback that reports no error has not necessarily undone anything.

The Master Key: *Autocommit is on by default and commits every statement on its own, so a ROLLBACK typed after an UPDATE undoes nothing; open the transaction first with START TRANSACTION, or turn autocommit off, and only then has the rollback anything to work on.*

REAL Question 157

Inside the despatch folder at Buhongwa is a sheet nobody has replaced since the store opened, with three statements about ACID one under another. Two are wrong, and the more dangerous of the two is wrong in a way that says something perfectly true.

Exactly one of the three statements below is correct. Say which, name the fault in each of the other two, say what the property really promises, and give the correction.

A. Consistency means that one session cannot see what another session has changed but not yet committed.

B. Durability means the server holds a committed transaction in memory until the session disconnects.

C. Atomicity means a transaction either takes effect in full or leaves no trace at all.

Model Answer

C is the correct one and it is correct exactly as written: atomicity means a transaction either takes effect in full or leaves no trace at all. A states something perfectly true and files it under the wrong letter, because keeping one session from another's uncommitted work is isolation, and B has durability backwards, since memory is precisely what a power cut empties.

Why A is wrong: What A describes is isolation, word for word, and consistency is not about other sessions at all. A true sentence filed under the wrong letter is the commonest error there is in this topic, and it is harder to catch than an outright falsehood because nothing in it reads wrongly.

What consistency really refuses: Consistency is about the rules the schema declares: primary keys, NOT NULL, foreign keys and CHECK constraints. They hold before the transaction and they hold after it, so a transaction that would leave a despatch line pointing at a seed code no store row carries is refused.

Why B is wrong: B describes the one arrangement durability exists to rule out. A change held in memory is lost when the power goes, so a server behaving as B says would lose committed work at every failure, which is the opposite of the promise.

What durability really promises: Once COMMIT has returned to the client the change is on durable storage and survives a restart. The server writes its record of the change before it reports the commit, not afterwards, and that ordering is what makes the promise keepable.

Why C is right as it stands: It carries both halves. In full, or no trace at all: the second half is the one usually dropped, and it is the half that rules out a transaction leaving twenty bags in neither the store nor the despatch book.

Written on the card	What the property really promises	The correction
A. Consistency keeps one session from seeing another's uncommitted changes	every rule the schema declares holds either side of the transaction boundary	the sentence is true of isolation, so file it under isolation
B. Durability holds a committed transaction in memory until the session disconnects	a committed change is on durable storage before COMMIT returns to the client	memory is what a power cut empties, so say durable storage, and say before
C. Atomicity is in full or no trace at all	exactly that, and both halves of it are needed	correct as it stands

Table 4.32: *The three written statements, what each property really promises, and what each should say.*

Two of the three sentences are true about transactions and false about the letter they are written under.

The Master Key: *Consistency is the declared rules holding either side of the boundary, isolation is what another session may see, and durability is surviving a failure after COMMIT, so a true fact filed under the wrong letter is still a wrong answer.*

BUILD Question 158

The load sheet for Igoma says twenty bags of maize. Recording it takes two statements, one to take the bags out of the store and one to write the despatch line, and the books balance only at the moment both of them have happened.

Create the two tables of the despatch book, then move twenty bags of maize to Igoma inside one transaction and commit it.

The store table holds 3 seed lots: MZ01 maize 200 bags, BN02 beans 60 bags, SG03 sorghum 45 bags.

The despatch table holds 2 lines already: D1 to Buhongwa and D2 to Nyegezi.

Take 20 bags of maize out of the store and write despatch line D3 to Igoma, as one unit of work.

Print the store row and the whole despatch book after the commit.

Model Answer

The transaction holds two statements, an UPDATE that takes 20 bags out of the store and an INSERT that writes despatch line D3 to Igoma, and COMMIT makes both of them permanent at the same instant: the store row comes back at 180 bags and the despatch book comes back with 3 lines.

```
 1  CREATE TABLE store (
 2    seed_code CHAR(4)     NOT NULL,
 3    seed_name VARCHAR(16) NOT NULL,
 4    bags_held INT         NOT NULL,
 5    PRIMARY KEY (seed_code)
 6  ) ENGINE = InnoDB;
 7
 8  CREATE TABLE despatch (
 9    despatch_no CHAR(4)     NOT NULL,
10    seed_code   CHAR(4)     NOT NULL,
11    village     VARCHAR(16) NOT NULL,
12    bags_sent   INT         NOT NULL,
13    PRIMARY KEY (despatch_no),
14    FOREIGN KEY (seed_code) REFERENCES store (seed_code)
15  ) ENGINE = InnoDB;
16
17  INSERT INTO store (seed_code, seed_name, bags_held) VALUES
18    ('MZ01', 'maize',   200),
19    ('BN02', 'beans',    60),
20    ('SG03', 'sorghum',  45);
21
22  INSERT INTO despatch (despatch_no, seed_code, village, bags_sent)
23  VALUES ('D1', 'MZ01', 'Buhongwa', 15),
24         ('D2', 'BN02', 'Nyegezi',  10);
25
26  SELECT seed_code, seed_name, bags_held FROM store;
```

The server replies

```
+-----------+-----------+-----------+
| seed_code | seed_name | bags_held |
+-----------+-----------+-----------+
| BN02      | beans     |        60 |
| MZ01      | maize     |       200 |
| SG03      | sorghum   |        45 |
+-----------+-----------+-----------+
```

Program 4.50: *The two tables of the despatch book, with the rows the section works from.*

```
 1  SELECT bags_held FROM store WHERE seed_code = 'MZ01';
 2
 3  START TRANSACTION;
 4
 5  UPDATE store
 6  SET bags_held = bags_held - 20
 7  WHERE seed_code = 'MZ01';
 8
 9  INSERT INTO despatch (despatch_no, seed_code, village, bags_sent)
10  VALUES ('D3', 'MZ01', 'Igoma', 20);
11
12  COMMIT;
13
14  SELECT bags_held FROM store WHERE seed_code = 'MZ01';
15  SELECT despatch_no, village, bags_sent FROM despatch;
```

The server replies

```
+-----------+
| bags_held |
+-----------+
|       200 |
+-----------+
+-----------+
| bags_held |
+-----------+
|       180 |
+-----------+
+-------------+----------+-----------+
| despatch_no | village  | bags_sent |
+-------------+----------+-----------+
| D1          | Buhongwa |        15 |
| D2          | Nyegezi  |        10 |
| D3          | Igoma    |        20 |
+-------------+----------+-----------+
```

Program 4.51: *Twenty bags moved to Igoma inside one transaction, and committed.*

Two statements, one boundary: Neither statement is right on its own. START TRANSACTION puts both in the same unit and COMMIT makes them permanent together, so no reader ever meets a store column that has been reduced with no despatch line to account for the bags.

Where the bags went: 200 bags less 20 leaves 180 in the store, and the 20 appear on line D3 to Igoma. The two figures are the same 20 bags counted in two places, which is what the transaction exists to keep in step.

The design decision worth naming: The despatch table declares its seed code a foreign key into store, so the server refuses a despatch line for a seed lot that is not held. That is consistency working at the boundary, and declaring it costs nothing.

Two statements, one boundary, and the books balance on both sides of it.

The Master Key: *Statements that must all happen or none happen go inside one START TRANSACTION and one COMMIT, which makes every change in the group permanent at the same instant.*

BUILD Question 159

The lorry for Nyegezi broke an axle before it left the yard. The store column had already been reduced and the despatch line already written, and both of those changes now had to go back exactly the way they came.

Show the same two statements inside a transaction that is rolled back, and prove that both tables are left exactly as they were.

Start from the same 3 seed lots and the same 2 despatch lines.

Take 20 bags of maize out of the store and write despatch line D3 to Nyegezi.

Read the store row while the transaction is still open, then roll it back.

Print the store row and the whole despatch book afterwards.

Model Answer

The store row reads 180 bags while the transaction is open and 200 bags again after the ROLLBACK, and the despatch book comes back with its original 2 lines, so both statements were undone by the one rollback and neither table keeps any trace of them.

```
 1  START TRANSACTION;
 2
 3  UPDATE store
 4  SET bags_held = bags_held - 20
 5  WHERE seed_code = 'MZ01';
 6
 7  INSERT INTO despatch (despatch_no, seed_code, village, bags_sent)
 8  VALUES ('D3', 'MZ01', 'Nyegezi', 20);
 9
10  SELECT bags_held FROM store WHERE seed_code = 'MZ01';
11
12  ROLLBACK;
13
14  SELECT bags_held FROM store WHERE seed_code = 'MZ01';
15  SELECT despatch_no, village, bags_sent FROM despatch;
```

The server replies

```
+-----------+
| bags_held |
+-----------+
|       180 |
+-----------+
+-----------+
| bags_held |
+-----------+
|       200 |
+-----------+
+-------------+----------+-----------+
| despatch_no | village  | bags_sent |
+-------------+----------+-----------+
| D1          | Buhongwa |        15 |
| D2          | Nyegezi  |        10 |
+-------------+----------+-----------+
```

Program 4.52: *The same transaction rolled back, leaving both tables as they were.*

What the reading inside the transaction shows: The session that made the change sees it: 180 bags, on the row it has just updated. That reading is honest and it is private, because the change belongs to a transaction that has not been committed.

What the rollback undid: Both statements, not the last one. ROLLBACK returns the session to the state at START TRANSACTION, so the UPDATE and the INSERT go together, and the despatch book is back to 2 lines with no D3 in it.

Nothing has to be undone by hand: A clerk correcting this without a transaction would have to add 20 bags back and delete line D3, and would have to remember both. The rollback is one statement and it cannot forget half the work.

The safest way to undo a group of changes is to have never committed them.

The Master Key: *ROLLBACK returns the session to the state it was in at START TRANSACTION, undoing every statement in the group at once, so an abandoned piece of work leaves no trace in any table it touched.*

BUILD Question 160

Nine hundred bags of beans on one despatch line, when the whole store holds sixty. The storekeeper at Igoma saw it the moment he pressed the key, and the twenty bags already recorded above it were perfectly good.

Mark the good work with a savepoint, undo only the wrong line, and show that the transaction is still open and still has to be committed.

Inside one transaction, write despatch line D3 to Igoma and take its 20 bags out of the store.

Mark that point with a savepoint named loaded.

Then write a wrong line, D4 for 900 bags of beans, and undo only that line.

Print the despatch book before the commit, and the book and the store row after it.

Model Answer

SAVEPOINT loaded puts a named mark inside the open transaction, ROLLBACK TO loaded undoes only what was done after that mark, so the wrong line D4 goes and line D3 with its store adjustment stays, and the transaction is still open afterwards: the despatch book holds 3 lines before COMMIT and the same 3 lines after it.

```
 1  START TRANSACTION;
 2
 3  INSERT INTO despatch (despatch_no, seed_code, village, bags_sent)
 4  VALUES ('D3', 'MZ01', 'Igoma', 20);
 5
 6  UPDATE store
 7  SET bags_held = bags_held - 20
 8  WHERE seed_code = 'MZ01';
 9
10  SAVEPOINT loaded;
11
12  INSERT INTO despatch (despatch_no, seed_code, village, bags_sent)
13  VALUES ('D4', 'BN02', 'Igoma', 900);
14
15  ROLLBACK TO loaded;
16
17  SELECT despatch_no, village, bags_sent FROM despatch;
18
19  COMMIT;
20
21  SELECT despatch_no, village, bags_sent FROM despatch;
22  SELECT bags_held FROM store WHERE seed_code = 'MZ01';
```

The server replies

```
+-------------+----------+-----------+
| despatch_no | village  | bags_sent |
+-------------+----------+-----------+
| D1          | Buhongwa |        15 |
| D2          | Nyegezi  |        10 |
| D3          | Igoma    |        20 |
+-------------+----------+-----------+
+-------------+----------+-----------+
| despatch_no | village  | bags_sent |
+-------------+----------+-----------+
| D1          | Buhongwa |        15 |
| D2          | Nyegezi  |        10 |
| D3          | Igoma    |        20 |
+-------------+----------+-----------+
+-----------+
| bags_held |
+-----------+
|       180 |
+-----------+
```

Program 4.53: *A savepoint, and one wrong line undone without ending the transaction.*

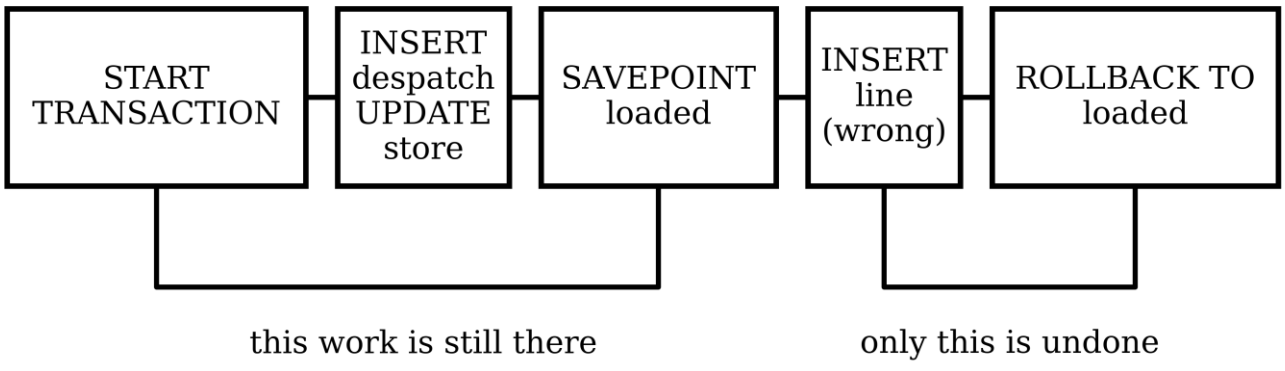


Figure 4.31: *What ROLLBACK TO undoes, what it leaves standing, and why a commit is still owed.*

What a savepoint is: A named mark inside an open transaction. It commits nothing and it ends nothing; it only gives the session a point it can be returned to, and one transaction may carry several marks under different names.

What ROLLBACK TO does and does not do: It undoes the work done since the named mark and leaves the transaction open. The despatch line for Igoma and the store adjustment beside it are untouched, and the wrong line for 900 bags of beans is gone.

The commit is still owed: Nothing is permanent until COMMIT is typed, which is why the despatch book is printed twice here. Walk away after the rollback to the mark and the whole transaction is lost, good work and all.

A savepoint is a way of retreating a few steps rather than the whole way back.

The Master Key: *SAVEPOINT names a mark inside an open transaction and ROLLBACK TO returns to that mark, undoing only what followed it and leaving the transaction open, so COMMIT or ROLLBACK is still owed afterwards.*

BUILD Question 161

Two people are at the books at once. One is halfway through moving twenty bags out of the store at Buhongwa; the other, at the counter, asks that very row how many bags are left, and is told two hundred.

Run two sessions at the same time and show that a change one of them has not yet committed cannot be seen by the other.

Session A turns autocommit off, so its statements open a transaction of their own.

Session A takes 20 bags out of the store and writes despatch line D3 to Igoma, leaving the transaction open.

Session B leaves autocommit on and reads the store row.

Session A commits, and session B reads the same row again.

Model Answer

Session B is told 200 bags while session A is holding an uncommitted change to that very row, and is told 180 bags by the same SELECT once session A has committed, which is atomicity and isolation shown rather than asserted: the state in which the bags have left the store and reached nowhere is real inside session A and invisible outside it.

The two sessions, run together

```
A> SET autocommit = 0;
B> SET autocommit = 1;
A> UPDATE store SET bags_held = bags_held - 20 WHERE seed_code = 'MZ01';
   1 row(s) affected
A> INSERT INTO despatch VALUES ('D3', 'MZ01', 'Igoma', 20);
   1 row(s) affected
B> SELECT bags_held FROM store WHERE seed_code = 'MZ01';
   +-----------+
   | bags_held |
   +-----------+
   |       200 |
   +-----------+
A> COMMIT;
B> SELECT bags_held FROM store WHERE seed_code = 'MZ01';
   +-----------+
   | bags_held |
   +-----------+
   |       180 |
   +-----------+
```

Program 4.54: *Two sessions at one row, either side of a COMMIT.*

Why session A needed no START TRANSACTION: Its first statement turned autocommit off, so the UPDATE that follows opens a transaction by itself and holds it open. Session B left autocommit on, which is why each of its SELECTs is a finished transaction of its own.

What session B was shown the first time: 200 bags, the figure as it stood before session A began. Session A had already reduced the row and written the despatch line, and none of that had been committed, so none of it was there to be seen.

What changed at the COMMIT: Not the row itself, which session A had already changed when its UPDATE ran. What the COMMIT changed is who may see that change, and the same SELECT from session B answers 180 bags immediately afterwards.

One connection can only claim this; two connections show it.

The Master Key: *A change made inside an open transaction is real to the session that made it and invisible to every other session until COMMIT, so the moment when the books do not balance is never a moment anybody else can read.*

HOT Question 162

The rollback at Nyegezi cleared away nothing at all. The working table had been built inside the transaction on purpose, exactly as the yard has always done it, so that one rollback would take the table and every row with it.

Judge whether a CREATE TABLE or an ALTER TABLE may be placed inside a transaction, and state the rule that decides where a schema change belongs.

Model Answer

It may be typed there and the server will accept it, but it must not be relied on, because CREATE TABLE and ALTER TABLE commit the open transaction implicitly before they run, so every data change made since START TRANSACTION becomes permanent at that moment and the ROLLBACK below finds nothing to undo: keep every schema change outside the transaction boundary, and put it before rather than after.

```
 1  START TRANSACTION;
 2
 3  UPDATE store
 4  SET bags_held = bags_held - 20
 5  WHERE seed_code = 'MZ01';
 6
 7  CREATE TABLE despatch_note (
 8    note_no CHAR(4)     NOT NULL,
 9    village VARCHAR(16) NOT NULL,
10    PRIMARY KEY (note_no)
11  ) ENGINE = InnoDB;
12
13  ROLLBACK;
14
15  SELECT bags_held FROM store WHERE seed_code = 'MZ01';
16  SHOW TABLES;
```

The server replies

```
+-----------+
| bags_held |
+-----------+
|       180 |
+-----------+
+-------------------------+
| Tables_in_seed_despatch |
+-------------------------+
| despatch                |
| despatch_note           |
| store                   |
+-------------------------+
```

Program 4.55: *A CREATE TABLE inside a transaction, and the rollback that finds nothing to undo.*

What an implicit commit is: A statement that ends the open transaction without anybody typing COMMIT. In MySQL and MariaDB the data definition statements do this: CREATE TABLE, ALTER TABLE, DROP TABLE, TRUNCATE and CREATE INDEX all commit whatever is open before they begin.

What the listing shows: The store row reads 180 bags after the rollback, so the UPDATE was committed by the CREATE TABLE that followed it, and the new table is still listed. Two things were made permanent by a statement that announced neither of them.

The case for writing them together: It is a real case. A single script that builds a working table, fills it, corrects the live rows and tidies up can be handed to somebody else, and it is tempting to wrap the whole of it in one boundary so that one rollback cleans up after a failure.

Why that case fails here: The boundary the script appears to draw is not the boundary the server keeps. Each schema statement quietly ends the transaction, so a script of that shape holds not one unit of work but several, divided at points the writer never chose and cannot see.

What it costs when it goes wrong: A failure halfway leaves the earlier data changes committed and the later ones not, with no statement that will take them back. Recovery is then by hand, by somebody who must first work out where the server drew its lines.

The order that is safe: Make the schema change first and on its own, check that it is there, and only then open the transaction that moves data. The transaction is then made entirely of statements that a rollback can genuinely undo.

The judgement: Keep every schema change outside the boundary and ahead of it. The single exception is a script whose whole work is the schema change, with no data change to protect: there is then no transaction worth having, and the implicit commit takes nothing away.

The rollback that undoes nothing is not broken; the boundary simply closed earlier than the writer thought.

The Master Key: *CREATE TABLE and ALTER TABLE commit the open transaction implicitly before they run, so a rollback after one of them undoes nothing; make every schema change before the transaction begins, and keep the boundary around data changes alone.*

Concurrency, locking and isolation levels

Two mechanics at one parts tray can do four different things to each other, and this section names all four, shows every one of them happening on a real server, and says which of them the isolation level in force actually refuses. The work is done on a bicycle repair shop at Nyakato, whose tray holds 3 kinds of part and whose job sheet holds 3 jobs, small enough to count by eye. One thing takes most of the care here. MariaDB runs at REPEATABLE READ unless somebody sets another level, not at READ COMMITTED, and no level below SERIALIZABLE prevents a lost update at all, which is why the cure for one is a lock rather than a setting.

BINDER Question 163

The tray on the bench at Nyakato held forty spokes at eight in the morning. Two mechanics booked parts out of it before nine, one taking ten and one taking five, and neither of them miscounted. The tray then held thirty-five.

Name the four faults that can appear when two transactions run at the same time, and define each one by what the two transactions do to each other.

Model Answer

The four are the lost update, the dirty read, the non-repeatable read and the phantom read, and each is defined by what two transactions do to each other: a lost update is a committed change destroyed by a write worked out from a value read before it, a dirty read is a read of work another transaction has not committed, a non-repeatable read is one row giving two answers inside one transaction, and a phantom read is one set of rows changing size inside one transaction.

Lost update: Both mechanics read 40, both subtract from the 40 they read, and the second write destroys the first. Nothing is refused and no message appears, because both writes were legal; the count is simply wrong.

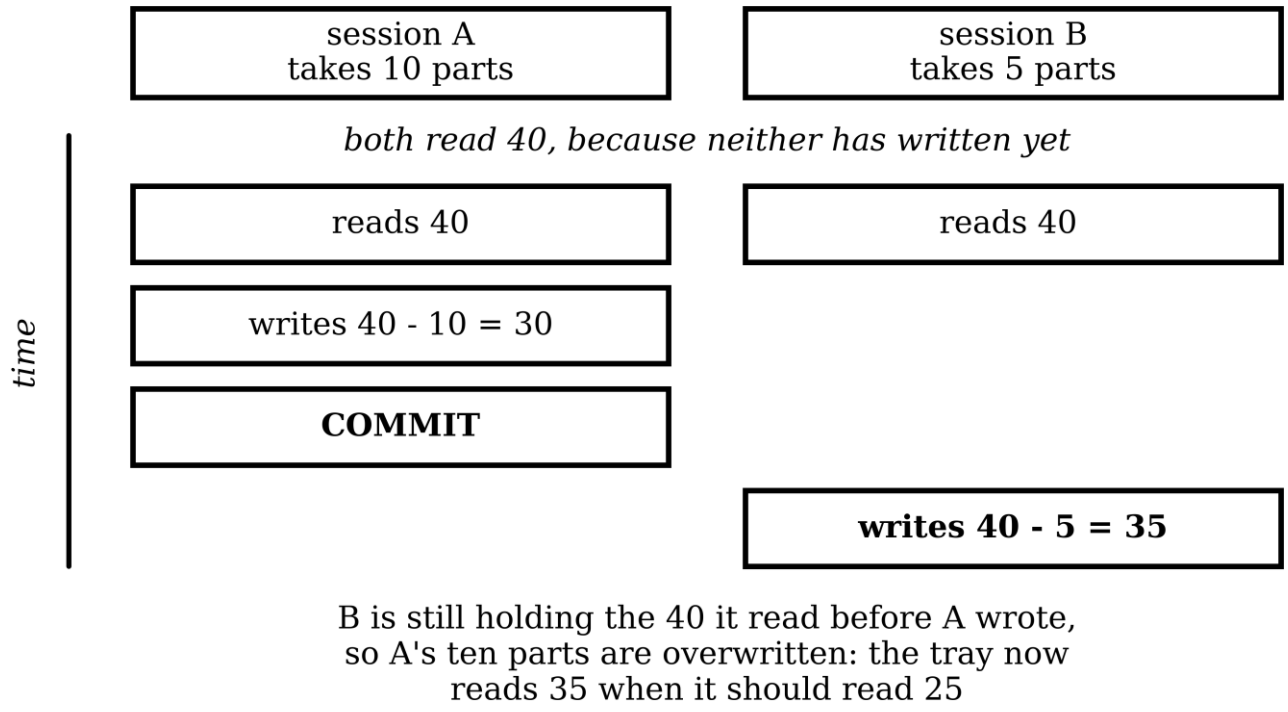


Figure 4.32: *Two sessions that both read 40, and the second write that destroys the first.*

Dirty read: One transaction reads a row another has written and not committed. The danger is not that the figure is old but that it may never be true at all, because the transaction that wrote it can still roll back.

Non-repeatable read: One transaction reads the same row twice and is given two different answers, because another transaction changed that row and committed between the readings. The row was there both times; what moved is its value.

Phantom read: One transaction asks the same question twice and the second answer matches a different number of rows. What changed is the set the condition selects, not a value in a row already in it.

Telling the last two apart: A non-repeatable read is a row you already had, changed under you. A phantom is a row you did not have, arriving or leaving. Most of the confusion in this topic lives in that distinction.

Fault	What the two sessions do	What goes wrong
Lost update	both read the tray, then both write a figure worked out from what they read	the second write destroys the first, and the parts it took are unaccounted for
Dirty read	one changes a row and does not commit, the other reads that row	the reader holds a figure that may never become true, and here never did
Non-repeatable read	one reads a row, the other changes it and commits, the first reads it again	one row gives two answers inside one transaction
Phantom read	one counts the rows matching a condition, the other inserts a row that matches it and commits	the same question matches a different number of rows inside one transaction

Table 4.33: *The four faults, what the two sessions have to do to produce each one, and what the shop is left with.*

The four are different events, and naming them correctly is most of the work.

The Master Key: *A lost update is a write destroyed by a write built on an earlier reading, a dirty read is a read of uncommitted work, a non-repeatable read is one row giving two answers inside one transaction, and a phantom read is one set of rows changing size.*

BINDER Question 164

Not one setting on the machine in the corner at Bujora has been changed since the day it was installed. One of those untouched settings decides which faults can still reach the parts tray, and it is stricter than the answer most people give.

State the four isolation levels in order, say which of the faults each one still permits, and name the level MariaDB uses when nobody sets one.

Model Answer

The four levels are READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ and SERIALIZABLE, from weakest to strongest, and MariaDB uses REPEATABLE READ unless somebody sets another: READ UNCOMMITTED permits all three read faults, READ COMMITTED refuses the dirty read, REPEATABLE READ refuses the non-repeatable read as well and under InnoDB refuses the phantom too, and SERIALIZABLE refuses everything by making a plain read take a lock.

The two weakest levels: READ UNCOMMITTED lets a transaction read rows another has written and not committed, which is the dirty read itself and buys almost nothing. READ COMMITTED refuses that outright but gives every statement a fresh view, so both the non-repeatable read and the phantom remain.

REPEATABLE READ, the level MariaDB installs: One view of the data is fixed when the transaction first reads and held until it ends, so a query repeated inside the transaction answers as it did the first time. Students are often taught, wrongly, that MariaDB starts at READ COMMITTED. The statement that changes the level is SET SESSION TRANSACTION ISOLATION LEVEL, typed before the transaction opens.

SERIALIZABLE, and what it costs: The strongest level makes a plain SELECT take a shared lock and hold it to the end of the transaction, so a reader blocks a writer and a writer blocks a reader. No anomaly survives it, and what arrives instead is waiting, and deadlocks built from statements that only read.

The fault the four levels are not defined against: All three faults in the table are faults of reading. A lost update is a fault of writing, and no level below SERIALIZABLE prevents one, so it is cured with a lock rather than with a setting.

Isolation level	Dirty read	Non-repeatable read	Phantom read
READ UNCOMMITTED	permitted	permitted	permitted
READ COMMITTED	refused	permitted	permitted
REPEATABLE READ, the default	refused	refused	refused by InnoDB, permitted by the standard
SERIALIZABLE	refused	refused	refused

Table 4.34: *What each level permits, as this server behaves, with the one place where InnoDB is stricter than the standard requires.*

A level does not decide what a transaction may do, only what another transaction may see of it.

The Master Key: *The four levels run READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ and SERIALIZABLE from weakest to strongest, MariaDB uses REPEATABLE READ when nobody sets one, and each level upwards refuses one more fault of reading at the price of more waiting.*

REAL Question 165

Somebody wrote three statements about concurrency on the back of a job card at Igombe, and every mechanic has read them for a month. Two of the three describe something that really happens and file it under the wrong name.

Exactly one of the three statements below is correct. Say which, name the fault each of the other two really describes, and give the correction.

A. A dirty read is when the same row gives two different answers inside one transaction.

B. A phantom read is when a transaction reads a row that another transaction has changed but not committed.

C. A lost update is when a committed change is overwritten by a write worked out from a value read before that change.

Model Answer

C is correct as it stands. A describes a non-repeatable read and files it under the dirty read, and B describes a dirty read and files it under the phantom, so both wrong sentences are true of something and both fail at the name in front: the transcript below is the fault A really describes, one row answering 40 and then 30 inside one transaction that was never left.

The two sessions, run together

```
A> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
B> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
A> SET autocommit = 0;
B> SET autocommit = 0;
A> START TRANSACTION;
A> SELECT in_tray FROM tray WHERE part_code = 'SP12';
   +---------+
   | in_tray |
   +---------+
   |      40 |
   +---------+
B> START TRANSACTION;
B> UPDATE tray SET in_tray = in_tray - 10 WHERE part_code = 'SP12';
   1 row(s) affected
B> COMMIT;
A> SELECT in_tray FROM tray WHERE part_code = 'SP12';
   +---------+
   | in_tray |
   +---------+
   |      30 |
   +---------+
A> COMMIT;
```

Program 4.56: *One row, two answers, inside a transaction session A never left.*

What A really describes: One row, two answers, inside one transaction, is the non-repeatable read. Session A reads 40, session B takes 10 spokes and commits, and session A reads that same row again and is told 30, having issued no COMMIT of its own in between.

What a dirty read really is: A read of a row version that has not been committed. What decides it is not whether the value later changed but whether the transaction that wrote it had finished, and only READ UNCOMMITTED permits it.

What B really describes: A row changed and not yet committed is the dirty read exactly, so B has swapped two of the four names. Nothing in B mentions a set of rows at all, and a phantom is never about one row.

What a phantom really is: A set of rows changing size inside one transaction, which is why it is counted rather than read: the same condition matching two rows, and then three, with no row that session A already held having moved.

Why C is right as it stands: It carries both halves, the change that was committed and the value read before that change. Drop the second half and nothing is described, because a write that overwrites a committed change with a fresh reading of it is an ordinary update.

Why a wrong name is the dangerous error: A sentence describing a real fault under the wrong name is harder to catch than an untruth, because every word of the description reads correctly. Name the fault first, then read the description against the name.

Written on the page	What it really describes	The correction
A. A dirty read is one row giving two different answers inside one transaction	the non-repeatable read	a dirty read is a read of work that has not been committed
B. A phantom read is reading a row changed but not committed	the dirty read	a phantom read is one set of rows changing size inside one transaction
C. A lost update is a committed change overwritten by a write worked out from an earlier reading	exactly that, and both halves of it are needed	correct as it stands

Table 4.35: *The three typed sentences, the fault each one really describes, and what each should say.*

Two of the three sentences are true about concurrency and false about the fault they are written under.

The Master Key: *A dirty read is uncommitted work read, a non-repeatable read is one row answering twice inside one transaction, a phantom read is one set of rows changing size, and a true description filed under the wrong name is still a wrong answer.*

REAL Question 166

Two jobs started in the same minute at Nyakato, one on a wheel and one on a set of brakes. Each mechanic had booked out one part and each then needed the part the other was holding. Nobody typed anything for the next second.

Explain what a deadlock is, say exactly what the server does about it and what it does to the work already done, and say what the application should do next.

Model Answer

A deadlock is two transactions each holding a row the other is waiting for, and InnoDB does not wait it out: it finds the cycle at once, chooses one of the two as its victim, rolls that whole transaction back and returns ERROR 1213 to it, while the other is released at that instant and commits normally. What the application should do with the victim is run its work again, because a deadlock is an event to expect rather than a fault to prevent.

The two sessions, run together

```
A> SET autocommit = 0;
B> SET autocommit = 0;
A> START TRANSACTION;
B> START TRANSACTION;
A> UPDATE tray SET in_tray = in_tray - 10 WHERE part_code = 'SP12';
   1 row(s) affected
B> UPDATE tray SET in_tray = in_tray - 2 WHERE part_code = 'BC08';
   1 row(s) affected
A> UPDATE tray SET in_tray = in_tray - 1 WHERE part_code = 'BC08';
   ... waiting: the server reports this session in a lock wait
B> UPDATE tray SET in_tray = in_tray - 5 WHERE part_code = 'SP12';
   ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
   ... A resumes here
   1 row(s) affected
A> COMMIT;
B> COMMIT;
A> SELECT part_code, in_tray FROM tray ORDER BY part_code;
   +-----------+---------+
   | part_code | in_tray |
   +-----------+---------+
   | BC08      |      11 |
   | IT05      |      25 |
   | SP12      |      30 |
   +-----------+---------+
A> COMMIT;
```

Program 4.57: *Two transactions that need each other's rows, and the one the server refuses.*

What makes the cycle: Session A held the spoke row and asked for the brake cable row; session B held the brake cable row and asked for the spokes. Each wait can end only when the other ends, so no amount of waiting will resolve it.

What the server did about it: It found the cycle immediately and returned ERROR 1213 to session B alone. Session A was released at that instant, its UPDATE reported one row changed, and it went on to commit as though nothing had happened.

What the victim lost: The whole transaction, not the statement that failed. Session B had already taken 2 brake cables, and after the rollback that row reads 11, which is the 12 it began with less the 1 session A took.

Which of the two is chosen: The server chooses, normally taking the transaction that has done the less work, and the choice is not the application's to make. Both sessions must therefore be written to survive being the victim.

A deadlock is not a timeout: A wait that simply runs out is a different event with a different number: the server reports ERROR 1205 when a lock wait exceeds its limit. A deadlock is refused at once, because waiting longer could not possibly help.

What to do about them: Run the victim's transaction again from its start. Deadlocks grow rarer when every piece of code takes rows in the same order and transactions are kept short, but rarer is the honest goal here, not never.

The session that is refused is not the one that did anything wrong; it is the one the server could most cheaply undo.

The Master Key: *A deadlock is a cycle of waits, and InnoDB breaks it at once by rolling one whole transaction back and returning ERROR 1213 to it, so the correct response in code is to run the refused transaction again rather than to try to prevent every deadlock.*

BUILD Question 167

Fifteen spokes left the tray at Nyakato in one morning, ten for one wheel and five for another, and both jobs were written up correctly. The tray was counted at closing time and held thirty-five, which is ten more than the job sheet allows.

Create the parts tray and the job sheet, then run two sessions that each read the tray before either of them writes to it, and show what the tray holds afterwards.

The tray holds 3 kinds of part: SP12 spokes 40, BC08 brake cables 12, IT05 inner tubes 25.

The job sheet holds 3 jobs: J1 for Neema done, J2 for Juma waiting, J3 for Asha waiting.

Session A takes 10 spokes and session B takes 5, each reading the tray first and then writing the figure it worked out.

Print the tray row after both sessions have committed.

Model Answer

Both sessions read 40 before either of them wrote, so both subtract from 40: session A writes 30 and commits, session B then writes 35, and the tray comes back at 35 when 15 spokes have left it and it should read 25. Nothing was refused and no message appeared, because session B wrote a perfectly legal statement carrying a number that had stopped being true.

```
 1  CREATE TABLE tray (
 2    part_code CHAR(4)     NOT NULL,
 3    part_name VARCHAR(16) NOT NULL,
 4    in_tray   INT         NOT NULL,
 5    PRIMARY KEY (part_code)
 6  ) ENGINE = InnoDB;
 7
 8  CREATE TABLE job (
 9    job_no     CHAR(4)     NOT NULL,
10    owner_name VARCHAR(16) NOT NULL,
11    part_code  CHAR(4)     NOT NULL,
12    job_state  VARCHAR(8)  NOT NULL,
13    PRIMARY KEY (job_no),
14    FOREIGN KEY (part_code) REFERENCES tray (part_code)
15  ) ENGINE = InnoDB;
16
17  INSERT INTO tray (part_code, part_name, in_tray) VALUES
18    ('SP12', 'spoke',       40),
19    ('BC08', 'brake cable', 12),
20    ('IT05', 'inner tube',  25);
21
22  INSERT INTO job (job_no, owner_name, part_code, job_state) VALUES
23    ('J1', 'Neema', 'SP12', 'done'),
24    ('J2', 'Juma',  'BC08', 'waiting'),
25    ('J3', 'Asha',  'IT05', 'waiting');
26
27  SELECT part_code, part_name, in_tray FROM tray;
28  SELECT job_no, owner_name, part_code, job_state FROM job;
```

The server replies

```
+-----------+-------------+---------+
| part_code | part_name   | in_tray |
+-----------+-------------+---------+
| BC08      | brake cable |      12 |
| IT05      | inner tube  |      25 |
| SP12      | spoke       |      40 |
+-----------+-------------+---------+
+--------+------------+-----------+-----------+
| job_no | owner_name | part_code | job_state |
+--------+------------+-----------+-----------+
| J1     | Neema      | SP12      | done      |
| J2     | Juma       | BC08      | waiting   |
| J3     | Asha       | IT05      | waiting   |
+--------+------------+-----------+-----------+
```

Program 4.58: *The parts tray and the job sheet, with the rows this section works from.*

The two sessions, run together

```
A> SET autocommit = 0;
B> SET autocommit = 0;
A> START TRANSACTION;
B> START TRANSACTION;
A> SELECT in_tray FROM tray WHERE part_code = 'SP12';
   +---------+
   | in_tray |
   +---------+
   |      40 |
   +---------+
B> SELECT in_tray FROM tray WHERE part_code = 'SP12';
   +---------+
   | in_tray |
   +---------+
   |      40 |
   +---------+
A> UPDATE tray SET in_tray = 40 - 10 WHERE part_code = 'SP12';
   1 row(s) affected
A> COMMIT;
B> UPDATE tray SET in_tray = 40 - 5 WHERE part_code = 'SP12';
   1 row(s) affected
B> COMMIT;
A> SELECT in_tray FROM tray WHERE part_code = 'SP12';
   +---------+
   | in_tray |
   +---------+
   |      35 |
   +---------+
A> COMMIT;
```

Program 4.59: *Ten spokes and five spokes, both worked out from the same reading of the tray.*

Where the ten spokes went: Nowhere. They are on the job sheet and not in the tray, and the tray now claims ten it cannot produce. The fault is named for what was lost, which is session A's write and not the parts.

Why the server allowed it: Session A had committed and released the row before session B wrote, so there was nothing left to wait for and nothing to refuse. What was stale was the number inside session B's statement, and no server can see that.

The design decision worth naming: The count is a column, so any statement that changes it should read it in the same statement. Session B typed a number it had read a moment earlier, and that is the whole of the fault.

Two legal statements, no error message, and ten spokes that the tray says it holds.

The Master Key: *A lost update needs no failure and no error: two sessions read the same row, each writes a figure worked out from that reading, and the second write silently destroys the first.*

BUILD Question 168

The figure on the screen at Bujora said thirty, and it was copied onto the job sheet in ink. The session that had written that thirty gave up its work a moment later, and the tray had never held thirty at all.

Set both sessions to READ UNCOMMITTED and show one of them reading work the other has not committed and never will, then run the same steps again at READ COMMITTED.

Session A takes 10 spokes inside an open transaction and does not commit.

Session B reads the tray, session A rolls back, and session B reads the tray again.

Run the whole demonstration at READ UNCOMMITTED, then run the same steps at READ COMMITTED.

Model Answer

At READ UNCOMMITTED session B is shown 30, a figure session A had written and not committed, and after session A rolls back the same SELECT answers 40: the 30 was true at no moment whatever, and session B has already read it. The same steps at READ COMMITTED answer 40 both times, because that level shows a session no version of a row that has not been committed.

The two sessions, run together

```
A> SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
B> SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
A> SET autocommit = 0;
B> SET autocommit = 0;
A> START TRANSACTION;
A> UPDATE tray SET in_tray = in_tray - 10 WHERE part_code = 'SP12';
   1 row(s) affected
B> START TRANSACTION;
B> SELECT in_tray FROM tray WHERE part_code = 'SP12';
   +---------+
   | in_tray |
   +---------+
   |      30 |
   +---------+
A> ROLLBACK;
B> SELECT in_tray FROM tray WHERE part_code = 'SP12';
   +---------+
   | in_tray |
   +---------+
   |      40 |
   +---------+
B> COMMIT;
```

Program 4.60: *A figure read at READ UNCOMMITTED that the rollback then takes away.*

The two sessions, run together

```
A> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
B> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
A> SET autocommit = 0;
B> SET autocommit = 0;
A> START TRANSACTION;
A> UPDATE tray SET in_tray = in_tray - 10 WHERE part_code = 'SP12';
   1 row(s) affected
B> START TRANSACTION;
B> SELECT in_tray FROM tray WHERE part_code = 'SP12';
   +---------+
   | in_tray |
   +---------+
   |      40 |
   +---------+
A> ROLLBACK;
B> SELECT in_tray FROM tray WHERE part_code = 'SP12';
   +---------+
   | in_tray |
   +---------+
   |      40 |
   +---------+
B> COMMIT;
```

Program 4.61: *The same steps one level up, where session B is shown nothing uncommitted.*

What the level changed: Nothing whatever about session A, which ran the same two statements and abandoned them either way. What a level decides is only what the other session is allowed to see while the first one is still undecided.

Why a dirty read is worse than an old figure: An old figure was true once. The 30 was true at no moment at all, because the transaction that produced it ended in a rollback, so a job sheet carrying it records something that never happened.

What the stricter level still permits: A fresh view for every statement, which is exactly what a non-repeatable read needs. Refusing the dirty read is the whole of what READ COMMITTED buys, and it is bought back by one statement typed before the transaction opens.

One level decides whether a figure that never existed can be read and written down.

The Master Key: *A dirty read is a read of work that has not been committed, only READ UNCOMMITTED permits it, and a rollback afterwards turns whatever was read into a record of something that never happened.*

BUILD Question 169

The waiting jobs at Igombe were counted twice inside one transaction, a minute apart, and the two counts came back two and three. Nothing on that screen had been touched between them.

Count the waiting jobs twice inside one transaction while another session inserts a waiting job and commits between the two counts. Run it at READ COMMITTED, then at REPEATABLE READ, and say which of the two MariaDB uses when nobody sets a level.

Session A counts the waiting jobs twice, inside one transaction it never leaves.

Between the two counts, session B inserts job J4 for Baraka and commits it.

Ask for the count in a form that prints a number, so the change can be seen.

Run it at READ COMMITTED, then run the same steps at REPEATABLE READ.

Model Answer

At READ COMMITTED session A counts 2 waiting jobs and then 3 without ever leaving its transaction, and the row session B committed between the two counts is a phantom: what changed is the number of rows the condition matches, not a value in any row session A already had. At REPEATABLE READ, which is the level MariaDB uses when nobody sets one, both counts answer 2.

The two sessions, run together

```
A> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
B> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
A> SET autocommit = 0;
B> SET autocommit = 0;
A> START TRANSACTION;
A> SELECT COUNT(*) AS jobs_waiting FROM job WHERE job_state = 'waiting';
   +--------------+
   | jobs_waiting |
   +--------------+
   |            2 |
   +--------------+
B> START TRANSACTION;
B> INSERT INTO job VALUES ('J4', 'Baraka', 'SP12', 'waiting');
   1 row(s) affected
B> COMMIT;
A> SELECT COUNT(*) AS jobs_waiting FROM job WHERE job_state = 'waiting';
   +--------------+
   | jobs_waiting |
   +--------------+
   |            3 |
   +--------------+
A> COMMIT;
```

Program 4.62: *A row that arrives inside somebody else's transaction, counted at READ COMMITTED.*

The two sessions, run together

```
A> SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
B> SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
A> SET autocommit = 0;
B> SET autocommit = 0;
A> START TRANSACTION;
A> SELECT COUNT(*) AS jobs_waiting FROM job WHERE job_state = 'waiting';
   +--------------+
   | jobs_waiting |
   +--------------+
   |            2 |
   +--------------+
B> START TRANSACTION;
B> INSERT INTO job VALUES ('J4', 'Baraka', 'SP12', 'waiting');
   1 row(s) affected
B> COMMIT;
A> SELECT COUNT(*) AS jobs_waiting FROM job WHERE job_state = 'waiting';
   +--------------+
   | jobs_waiting |
   +--------------+
   |            2 |
   +--------------+
A> COMMIT;
```

Program 4.63: *The same steps at the level MariaDB installs, where the count does not move.*

Why the question is asked as a count: A result of no rows prints nothing at all at the client, so a question answered by nothing looks like a question nobody asked. COUNT(*) prints a number, and a number moving from 2 to 3 is something a reader can see.

What the default level is doing: REPEATABLE READ fixes one view of the data at session A's first read and holds it until session A ends. Session B's row is committed and perfectly real, and session A will see it in its next transaction, but not inside this one.

What this level does not cover: The phantom is refused here and the lost update is not. Two sessions can still read one row under this level and write over each other, which is what the last question in this section turns on.

A phantom is counted rather than seen, because what moved is the set and not the row.

The Master Key: *A phantom read is one set of rows changing size inside one transaction, READ COMMITTED permits it, and REPEATABLE READ, which is what MariaDB installs, holds one view of the data from the transaction's first read to its end.*

BUILD Question 170

Session B's UPDATE at Nyakato did not come back. No message appeared and no prompt returned, and the mechanic at that screen was one reach away from the power switch, which is the single thing that would have made it worse.

Run the same two mechanics again, with session B asking for the row while session A is still holding it and with session B's UPDATE naming the column instead of a number, and show what happens to session B.

Session A takes 10 spokes inside an open transaction and does not commit yet.

Session B asks to take 5 while session A is still holding that row.

Session B subtracts from the column, in_tray - 5, rather than from a number.

Print the tray row at the end, and print the setting that bounds a lock wait.

Model Answer

Session B waits, and it waits for exactly as long as session A holds the row: the server reports session B in a lock wait, session A's COMMIT is what releases it, and session B's UPDATE then subtracts from the row session A left behind, so the tray comes back at 25 with nothing lost. Session B was never refused and never timed out, because a lock is a wait and not an error.

The two sessions, run together

```
A> SET autocommit = 0;
B> SET autocommit = 0;
A> START TRANSACTION;
A> UPDATE tray SET in_tray = in_tray - 10 WHERE part_code = 'SP12';
   1 row(s) affected
B> START TRANSACTION;
B> UPDATE tray SET in_tray = in_tray - 5 WHERE part_code = 'SP12';
   ... waiting: the server reports this session in a lock wait
A> COMMIT;
   ... B resumes here
   1 row(s) affected
B> COMMIT;
A> SELECT in_tray FROM tray WHERE part_code = 'SP12';
   +---------+
   | in_tray |
   +---------+
   |      25 |
   +---------+
A> COMMIT;
```

Program 4.64: *A second writer held at the row until the first writer commits.*

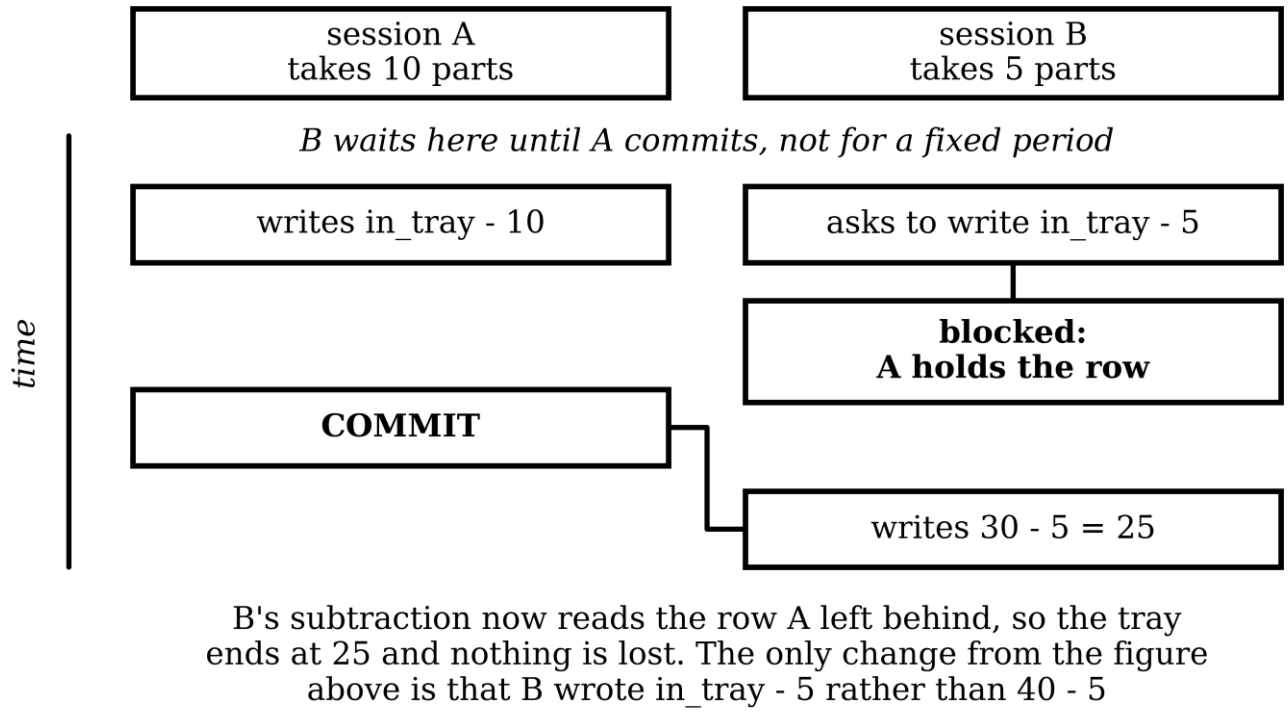


Figure 4.33: *The same two sessions with the column named in session B's UPDATE, and the commit that releases the wait.*

```
1  SELECT @@innodb_lock_wait_timeout;
```

The server replies

```
+----------------------------+
| @@innodb_lock_wait_timeout |
+----------------------------+
|                         50 |
+----------------------------+
```

Program 4.65: *How long a lock wait may run on this server before the server gives up on it.*

What session A was holding: An exclusive lock on that one row, taken by its UPDATE and kept until its transaction ended. That is why session A's COMMIT is the thing that releases session B, and nothing else could: a row lock is held until COMMIT or ROLLBACK, never for a period of time.

The two changes from the run that lost ten spokes: Session B asked while session A still held the row, which is why it waited at all, and its UPDATE names the column, which is why what it wrote was worked out from the row session A left behind: 25 rather than 35.

What bounds the wait: Had session A never committed, session B would not have waited for ever. A lock wait ends with ERROR 1205 when innodb_lock_wait_timeout expires, and this server allows it 50 seconds.

A blocked writer has not failed; it is standing in a queue of one.

The Master Key: *A row lock is held until the transaction that took it commits or rolls back, so a second writer waits rather than failing, and it resumes the instant the holder commits.*

HOT Question 171

The workshop at Igombe raised both its clients to the strongest level, and the tray has been right ever since. What the mechanics complain about now is a message that ends a job outright and sends the work back to its start.

The shop wants a lost update to be impossible. Judge whether to raise both sessions to SERIALIZABLE or to keep the default level and lock the rows the work is about to change, and state the rule that decides it.

Model Answer

Keep REPEATABLE READ and lock the row the work is about to change, either by writing the arithmetic into the UPDATE itself or by reading the row with SELECT ... FOR UPDATE, and raise a transaction to SERIALIZABLE only where its correctness depends on rows it merely read: the two demonstrations below show SERIALIZABLE saving the tray by killing a transaction outright with ERROR 1213, and FOR UPDATE saving the same tray with no refusal at all.

The two sessions, run together

```
A> SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
B> SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
A> SET autocommit = 0;
B> SET autocommit = 0;
A> START TRANSACTION;
B> START TRANSACTION;
A> SELECT in_tray FROM tray WHERE part_code = 'SP12';
   +---------+
   | in_tray |
   +---------+
   |      40 |
   +---------+
B> SELECT in_tray FROM tray WHERE part_code = 'SP12';
   +---------+
   | in_tray |
   +---------+
   |      40 |
   +---------+
A> UPDATE tray SET in_tray = 40 - 10 WHERE part_code = 'SP12';
   ... waiting: the server reports this session in a lock wait
B> UPDATE tray SET in_tray = 40 - 5 WHERE part_code = 'SP12';
   ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
   ... A resumes here
   1 row(s) affected
A> COMMIT;
B> COMMIT;
A> SELECT in_tray FROM tray WHERE part_code = 'SP12';
   +---------+
   | in_tray |
   +---------+
   |      30 |
   +---------+
A> COMMIT;
```

Program 4.66: *The two mechanics again, with both sessions raised to SERIALIZABLE.*

The two sessions, run together

```
A> SET autocommit = 0;
B> SET autocommit = 0;
A> START TRANSACTION;
B> START TRANSACTION;
A> SELECT in_tray FROM tray WHERE part_code = 'SP12' FOR UPDATE;
   +---------+
   | in_tray |
   +---------+
   |      40 |
   +---------+
B> SELECT in_tray FROM tray WHERE part_code = 'SP12' FOR UPDATE;
   ... waiting: the server reports this session in a lock wait
A> UPDATE tray SET in_tray = 40 - 10 WHERE part_code = 'SP12';
   1 row(s) affected
A> COMMIT;
   ... B resumes here
   +---------+
   | in_tray |
   +---------+
   |      30 |
   +---------+
B> UPDATE tray SET in_tray = 30 - 5 WHERE part_code = 'SP12';
   1 row(s) affected
B> COMMIT;
A> SELECT in_tray FROM tray WHERE part_code = 'SP12';
   +---------+
   | in_tray |
   +---------+
   |      25 |
   +---------+
A> COMMIT;
```

Program 4.67: *The default level again, with the row locked by the read that precedes the write.*

What SERIALIZABLE does: It makes a plain SELECT take a shared lock and hold it to the end of the transaction, so two sessions cannot both be holding 40 and then both write. The anomaly becomes impossible rather than unlikely.

What it did here: Both reads were granted, session A's UPDATE then waited on session B's shared lock, session B's UPDATE closed the cycle, and the server refused session B with ERROR 1213. The tray is right, at 30, and session B's five spokes are recorded nowhere.

The price, stated plainly: Every plain SELECT now takes a lock, so readers block writers and writers block readers, and deadlocks appear between statements that only ever read. Work does not go wrong quietly; it is refused loudly and has to be run again.

The cheaper protection: SELECT ... FOR UPDATE locks one row, the row the work is about to change, and leaves every other reader alone. Session B's read waits, comes back with 30 rather than 40, and its arithmetic is then done on a row nobody else can move.

The cheapest protection of all: Write the arithmetic into the statement. An UPDATE that says in_tray - 5 reads and writes the row in one step, under a lock the server takes for itself, and it costs nothing whatever.

What decides between them: The scope of what must not move. Where the rows that must not move are the rows the transaction is going to change, lock those rows and stay at the default. Where the answer depends on rows the transaction only read, or on rows that do not exist yet, no row lock covers it and the level is the only thing that does.

The judgement: Keep the default, lock what you are about to change, and write every piece of work to run again on ERROR 1213. Reserve SERIALIZABLE for the short and rare transaction that reads widely and must decide from what it read, and never set it as the shop's general level.

The strongest level is not the safest choice; it moves the cost from a wrong figure to a refused transaction.

The Master Key: *No level below SERIALIZABLE prevents a lost update, so lock the row the work is about to change, either with the arithmetic inside the UPDATE or with SELECT ... FOR UPDATE, and keep SERIALIZABLE for the transaction whose answer depends on rows it only read.*

Indexing, storage and query optimisation

An index is a second structure the server keeps beside a table so that a lookup can go to a row instead of reading every row in turn. This section builds one on a borehole maintenance log of 6 services in 3 villages, small enough to count by eye, and reads the plan the server prints for each statement with EXPLAIN. The cost is stated in the same breath as the saving throughout, because an index is not a setting on a column: it is a structure that occupies disk and is maintained on every INSERT, UPDATE and DELETE. The 3 ways an index that already exists still goes unused are worth as much to a reader as the rule for building one.

BINDER Question 172

A pump fails at Didia, and the fitter wants the last service recorded for that village. The log holds five thousand rows, and the server reads all five thousand of them to find two. Nothing is broken; nothing has been built either.

Explain how the rows of a table are stored and reached, say what an index is and what one entry in it holds, and state what an index costs.

Model Answer

An index is a separate structure, kept sorted, holding one entry for every row of the table: the value of the indexed column and the address of that row, so the server descends a few nodes instead of reading the table from first row to last, and it pays for that in disk and in work on every write.

How the rows sit on disk: A table is stored as rows inside pages, in no order a query can depend on. With nothing else to consult, the server reads every page and tests every row. That is a full scan, and it costs the whole table however few rows match.

What one entry holds: An index on village holds the village values in sorted order and, beside each one, the address of the row it came from. It is not a second copy of the table: a column the index does not name is not in it, and the server fetches the row itself to read it.

Why the descent is short: The entries are held in a B-tree, whose nodes each carry many keys and whose leaves all sit at one depth, so its height rises far more slowly than its row count. One node is read at each level, and the last gives the address of the row, as Figure 4.34 shows.

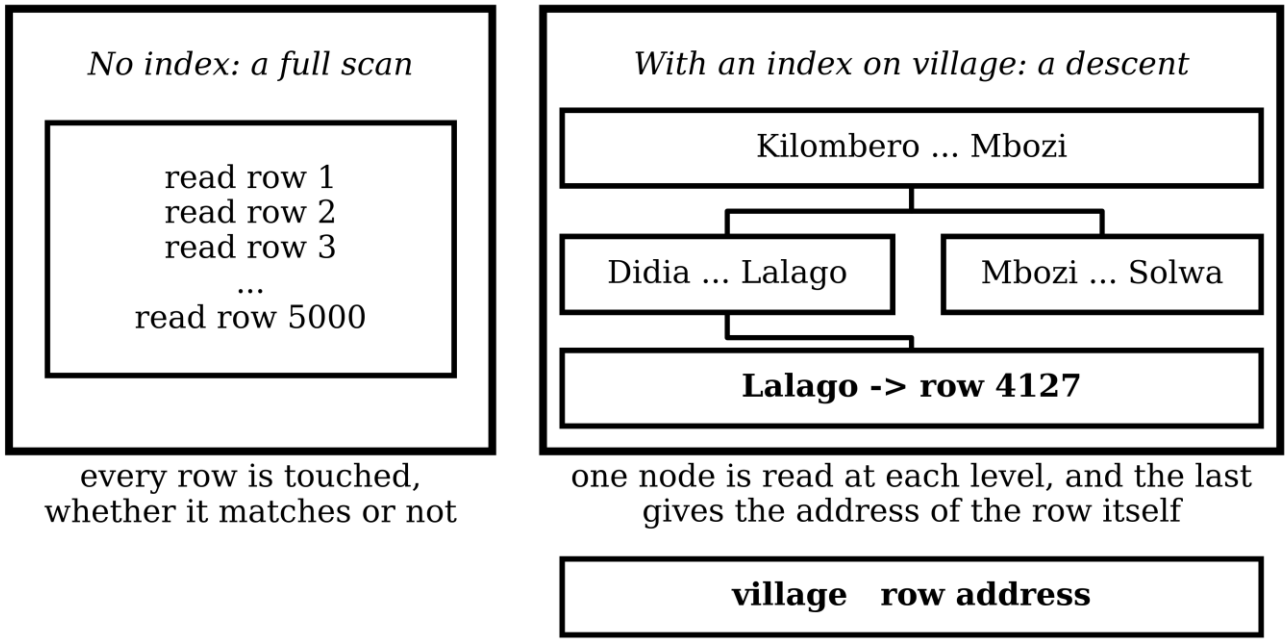


Figure 4.34: *A full scan against a descent of an index on village, and what the index itself holds.*

What it costs to keep: The entries occupy disk of their own, and every INSERT, every DELETE and every UPDATE of an indexed column is applied to the index as well as to the table. Students are often taught that an index makes a database faster; it makes some reads faster and every write slower.

Everything an index saves on a read it charges on a write.

The Master Key: *An index is a separate sorted structure holding the indexed value and the address of its row, so a lookup descends a few nodes instead of scanning the table, and it is paid for in disk space and in work on every INSERT, UPDATE and DELETE.*

BINDER Question 173

Village by village, then date by date, the list inside the workshop door at Solwa runs down the page. Give it a village and the answer comes at once. Give it a date alone and it cannot help, though every date is there.

Write the statement that creates an index, explain what a composite index is, and state the rule that decides which conditions it can serve.

Model Answer

CREATE INDEX names the index, names the table and lists its columns in the order they are sorted in, and a composite index over more than one column can be entered only at its leading column: an index on village and serviced_on serves a condition on village, or on village with serviced_on, and cannot serve one on serviced_on alone.

Writing the statement: CREATE INDEX idx_village_date ON service (village, serviced_on); builds one index over two columns. The name is yours to choose, and one carrying the columns saves reading the catalogue later. Name a single column and the statement builds an index on that column alone.

The leading-column rule: The entries are sorted by village first and by serviced_on only within one village, exactly as the card in the workshop is ordered. So the index is read from its leading column inwards: give village and it applies, give the date alone and there is nowhere to begin the descent. Figure 4.35 puts the three conditions beside the verdict on each.

A function on the column has the same effect: Wrap a column in a function and the index goes with it. YEAR(serviced_on) is not what the index holds, so the sorted order of the dates is no use and the value must be computed for every row.

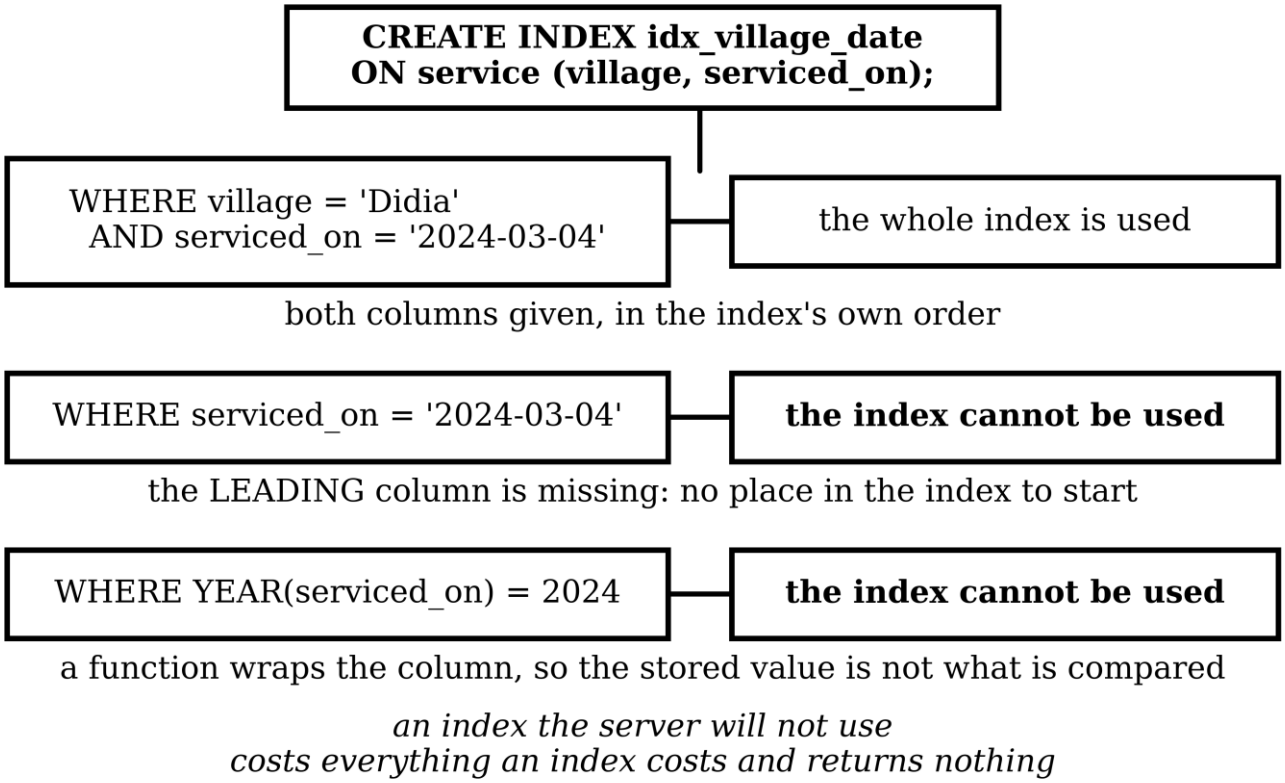


Figure 4.35: *One composite index and three conditions, with the server's verdict on each.*

When the index answers the whole query: If every column a statement names is in the index, the row is never fetched, because the index already holds the answer. Such an index is said to cover the query, and the plan says so in its Extra column.

An index is entered at its first column, and everything else follows from the order the columns were written in.

The Master Key: *CREATE INDEX names the index, the table and its columns in sorted order, and a composite index is usable only from its leading column inwards, so an index on village and serviced_on serves a condition on village but never one on serviced_on alone.*

REAL Question 174

Three claims about indexes are pinned inside the toolbox lid at Lalago, and the storeman has worked to all three for a year. Two of them cost him something on every write, and the least generous one is the one that holds.

Exactly one of the three claims below is correct. Say which, name the fault in each of the other two, state exactly what the server does about it, and give the correction.

A. Every column that a WHERE clause names should be given an index of its own.

B. An index on village and serviced_on together also serves a query naming serviced_on alone.

C. A column declared PRIMARY KEY is already indexed, so a CREATE INDEX on that column builds a structure the server will not choose.

Model Answer

C is the correct one, and it warns of an error worth avoiding: the primary key declaration has already built an index, so a second one on the same column is accepted, listed in the catalogue and never chosen. A is wrong because every index is paid for on every write, and B is wrong because a composite index cannot be entered at its second column.

Why A is wrong about writes: Every INSERT, every DELETE and every UPDATE of an indexed column maintains each index on the table as well as the table itself. Index every column a report mentions and every write maintains as many structures as the table has columns.

Why A is wrong about reads as well: An index earns its keep when the value asked for picks out few rows. This log holds 3 villages in 6 services, so a lookup by village still reaches a third of it, and the fewer rows a value picks out the more the index is worth.

What the server does about A: Nothing at all. Every CREATE INDEX is accepted and no message ever says that an index is useless. The price is paid quietly in slower writes and in disk, and only the plan the server prints shows whether an index is used.

Why B is wrong: The entries are sorted by village first, so a condition naming serviced_on alone has nowhere to start. The plan for it prints possible_keys NULL and key NULL, with type ALL and rows 6, which is the whole log read row by row.

The correction for B: Either name the leading column as well, which lets the one index serve the query, or build a second index on serviced_on alone and accept its cost. Reversing the two columns escapes nothing; it only changes which of the two queries is served.

Why C is correct: A primary key is an index, and InnoDB stores the table itself in that key order. Create another on the same column and the catalogue lists it, the plan names it among the keys considered, and the key chosen stays PRIMARY.

Pinned inside the lid	What the server actually does	The correction
A. Index every column a WHERE clause names	accepts every CREATE INDEX without comment, then maintains all of them on every INSERT, UPDATE and DELETE	index the columns whose conditions pick out few rows, and only where the plan shows a scan
B. An index on village and serviced_on serves serviced_on alone	prints possible_keys NULL, key NULL and type ALL, and reads all 6 rows	name the leading column as well, or build a second index on serviced_on
C. A PRIMARY KEY column is already indexed	correct: PRIMARY stands in the catalogue although nobody created it, and a second index on that column is never chosen	correct as it stands

Table 4.36: *The three claims, what the server really does about each, and what each should say.*

Two of the three cost something on every write and return nothing for it, and the third is a warning rather than an instruction.

The Master Key: *Every index is maintained on every write, so a column is worth indexing only where its conditions pick out few rows of a table big enough to matter, a composite index is entered at its leading column alone, and a primary key is already an index, so indexing it again buys nothing.*

BUILD Question 175

Two rows come back at Didia, the same two both times, out of the same question asked twice. The plan the server prints above them is not the same at all: one line of it has fallen from six to two.

Build the log, read the plan the server prints for a lookup by village with no index in place, then create the index and read the plan again.

The log holds 6 services: 2 at Didia, 2 at Solwa and 2 at Lalago.

Ask for the services at Didia, and print the plan before the query.

Create an index on village and ask for the same plan again.

Print the rows the query itself returns, both times.

Model Answer

With no index the plan says type ALL, key NULL and rows 6, which is the whole log read row by row; with idx_village in place the same statement is planned as type ref on key idx_village with rows 2, and it returns the same two rows in the same order.

```
 1  CREATE TABLE service (
 2    service_no   INT         NOT NULL,
 3    village      VARCHAR(20) NOT NULL,
 4    serviced_on  DATE        NOT NULL,
 5    litres       INT         NOT NULL,
 6    PRIMARY KEY (service_no)
 7  ) ENGINE = InnoDB;
 8
 9  INSERT INTO service (service_no, village, serviced_on, litres) VALUES
10    (1, 'Didia',  '2024-03-04',  900),
11    (2, 'Solwa',  '2024-05-19',  450),
12    (3, 'Lalago', '2024-07-02', 1200),
13    (4, 'Didia',  '2025-01-11',  700),
14    (5, 'Solwa',  '2025-02-08',  500),
15    (6, 'Lalago', '2025-04-23', 1100);
16
17  EXPLAIN SELECT service_no, serviced_on, litres
18  FROM service
19  WHERE village = 'Didia'\G
20
21  SELECT service_no, serviced_on, litres
22  FROM service
23  WHERE village = 'Didia'
24  ORDER BY service_no;
```

The server replies

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: service
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 6
        Extra: Using where
+------------+-------------+--------+
| service_no | serviced_on | litres |
+------------+-------------+--------+
|          1 | 2024-03-04  |    900 |
|          4 | 2025-01-11  |    700 |
+------------+-------------+--------+
```

Program 4.68: *The borehole log, and the plan for a lookup by village with no index in place.*

```
 1  CREATE INDEX idx_village ON service (village);
 2
 3  EXPLAIN SELECT service_no, serviced_on, litres
 4  FROM service
 5  WHERE village = 'Didia'\G
 6
 7  SELECT service_no, serviced_on, litres
 8  FROM service
 9  WHERE village = 'Didia'
10  ORDER BY service_no;
```

The server replies

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: service
         type: ref
possible_keys: idx_village
          key: idx_village
      key_len: 82
          ref: const
         rows: 2
        Extra: Using index condition
+------------+-------------+--------+
| service_no | serviced_on | litres |
+------------+-------------+--------+
|          1 | 2024-03-04  |    900 |
|          4 | 2025-01-11  |    700 |
+------------+-------------+--------+
```

Program 4.69: *The same statement once an index on village exists.*

Reading the plan: EXPLAIN does not run the statement; it prints the plan the optimiser has chosen for it. The two entries to read first are key, which names the index chosen or NULL where none was, and rows, the number of rows the server expects to examine.

Why the plan is printed one column to a line: A plan has 10 columns and prints wider than this page, so each EXPLAIN here is ended with \G instead of a semicolon, which is the client instruction for the vertical form. The statement is unchanged; only the shape of the output is.

What the index changed, and what it did not: The rows returned are identical and so is their order. What changed is the work: 2 rows expected instead of 6. On a log of five thousand services that is the difference between a page and the whole book.

Column of the plan	What it reports	What to look for
type	how the rows are reached: ALL is a full scan, ref an equality lookup in an index, range a bounded stretch of one, const a single row found by a unique key	ALL is the warning
possible_keys	the indexes the optimiser considered for this statement	NULL means no index applies at all
key	the index it actually chose	NULL means every row will be read
key_len	how many bytes of the chosen index the condition could use	on a composite index it says how far in the condition reached
rows	how many rows the server expects to examine	compare it with the size of the table
Extra	what happens besides reading rows: Using where, Using index condition, Using index	Using index means the table is never read

Table 4.37: *The columns of a plan worth reading, and what each one is warning about.*

The plan is where an index is proved, because the rows a query returns never show whether one was used.

The Master Key: *EXPLAIN prints the plan without running the statement, and the two entries worth reading first are key, which names the index the server chose or NULL where it chose none, and rows, the number of rows it expects to examine.*

BUILD Question 176

One index, two columns, three questions. Two of the three are answered from the index and the third reads the whole log, and the difference lies not in how hard the questions are but in which column each one begins with.

Create one composite index over village and serviced_on, and use EXPLAIN to show which of three conditions it serves whole, which it serves in part, and which it cannot serve at all.

Index village and serviced_on together, in that order.

Plan a condition that gives both columns.

Plan a condition that gives village alone.

Plan a condition that gives serviced_on alone.

Model Answer

The condition naming both columns uses the whole index, key_len 85; the condition naming village alone uses the part that belongs to village, key_len 82; and the condition naming serviced_on alone cannot use the index at all, so the plan falls back to type ALL with key NULL and rows 6.

```
 1  CREATE INDEX idx_village_date ON service (village, serviced_on);
 2
 3  EXPLAIN SELECT litres
 4  FROM service
 5  WHERE village = 'Didia'
 6    AND serviced_on = '2024-03-04'\G
 7
 8  EXPLAIN SELECT litres
 9  FROM service
10  WHERE village = 'Didia'\G
11
12  EXPLAIN SELECT litres
13  FROM service
14  WHERE serviced_on = '2024-03-04'\G
15
16  SELECT COUNT(*) AS services_in_the_log FROM service;
```

The server replies

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: service
         type: ref
possible_keys: idx_village_date
          key: idx_village_date
      key_len: 85
          ref: const,const
         rows: 1
        Extra: Using index condition
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: service
         type: ref
possible_keys: idx_village_date
          key: idx_village_date
      key_len: 82
          ref: const
         rows: 2
        Extra: Using index condition
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: service
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 6
        Extra: Using where
+--------------------+
| services_in_the_log |
+--------------------+
|                  6 |
+--------------------+
```

Program 4.70: *One composite index, and the plans for three conditions on it.*

What key_len is telling you: key_len is the number of bytes of the index the condition could use. village is a VARCHAR(20) in a four-byte character set, with two bytes of length, so 82 in all, and a DATE adds 3. A key_len of 85 means both columns were used and 82 means one was.

Why the third plan has no index: Nothing in this index begins with a date. The dates are sorted only inside one village, so with no village given there is nowhere to start, and the server has no candidate at all: possible_keys is NULL as well as key.

The design decision: Either the condition names the leading column, or a second index on serviced_on is built and paid for on every write. An index on serviced_on and village would answer the third question and lose the second, so the order follows the queries that run most.

One composite index serves the conditions that begin where it begins, and no others.

The Master Key: *A composite index is read from its leading column inwards, so key_len says how far into it a condition reached, and a condition naming only the second column leaves the server with no candidate index and a full scan.*

BUILD Question 177

An index built at Solwa that same morning went unused by lunchtime. The question was a plain one, every service done during 2024, and the way it was written wrapped the date column in YEAR. The server did not look at the index once.

Show with EXPLAIN that a function wrapped round an indexed column loses the index, and rewrite the condition so that the same rows come back through it.

Index serviced_on.

Plan the condition that asks for the year 2024 through the YEAR function.

Plan the condition that asks for the same dates as a range.

Count the services the range finds.

Model Answer

YEAR(serviced_on) = 2024 is planned as type ALL with possible_keys NULL and key NULL, so the index is not even a candidate; written instead as a range from 2024-01-01 up to but not including 2025-01-01, the same question is planned as type range on key idx_serviced_on, and it finds the same 3 services.

```
 1  CREATE INDEX idx_serviced_on ON service (serviced_on);
 2
 3  EXPLAIN SELECT litres
 4  FROM service
 5  WHERE YEAR(serviced_on) = 2024\G
 6
 7  EXPLAIN SELECT litres
 8  FROM service
 9  WHERE serviced_on >= '2024-01-01'
10    AND serviced_on <  '2025-01-01'\G
11
12  SELECT COUNT(*) AS services_in_2024
13  FROM service
14  WHERE serviced_on >= '2024-01-01'
15    AND serviced_on <  '2025-01-01';
```

The server replies

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: service
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 6
        Extra: Using where
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: service
         type: range
possible_keys: idx_serviced_on
          key: idx_serviced_on
      key_len: 3
          ref: NULL
         rows: 3
        Extra: Using index condition
+------------------+
| services_in_2024 |
+------------------+
|                3 |
+------------------+
```

Program 4.71: *The same question asked through a function and asked as a range.*

What the server is comparing: An index holds the stored values in sorted order. YEAR(serviced_on) is not a stored value but one computed after a row is read, so every row must be read to compute it.

Why the range form keeps the index: A range is a pair of bounds on the stored value itself. The server descends to the first date on or after the lower bound and walks forward along the sorted entries until it passes the upper one, which is the access the plan prints as range.

Why the upper bound is written this way: The bound is less than 2025-01-01 rather than less than or equal to 2024-12-31, so a value carrying a time of day cannot slip past it. The habit costs nothing and survives a later change of the column to DATETIME.

A condition that compares a stored value can use an index, and a condition that compares something computed from it cannot.

The Master Key: *Wrapping an indexed column in a function removes the index from the plan altogether, because what is compared is no longer what is stored, and the cure is to rewrite the condition as a range over the column itself.*

BUILD Question 178

The service table has four columns and the index on it names two. Ask only for those two at Lalago and the server never opens the table: the answer is already in the index, and the plan says so in one word.

Show a query answered from the index alone, then the same query asking for one column more, which cannot be.

Index village and serviced_on together.

Ask for serviced_on for the services at Solwa, and plan it.

Ask for litres for the same services, and plan that too.

Print the rows the covered query returns.

Model Answer

The first plan says Extra Using where; Using index, which means the query was answered from idx_village_date without the table being opened; the second asks for litres, which the index does not hold, so each matching row has to be fetched and Extra says Using index condition instead.

```
 1  CREATE INDEX idx_village_date ON service (village, serviced_on);
 2
 3  EXPLAIN SELECT serviced_on
 4  FROM service
 5  WHERE village = 'Solwa'\G
 6
 7  EXPLAIN SELECT litres
 8  FROM service
 9  WHERE village = 'Solwa'\G
10
11  SELECT serviced_on
12  FROM service
13  WHERE village = 'Solwa'
14  ORDER BY serviced_on;
```

The server replies

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: service
         type: ref
possible_keys: idx_village_date
          key: idx_village_date
      key_len: 82
          ref: const
         rows: 2
        Extra: Using where; Using index
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: service
         type: ref
possible_keys: idx_village_date
          key: idx_village_date
      key_len: 82
          ref: const
         rows: 2
        Extra: Using index condition
+-------------+
| serviced_on |
+-------------+
| 2024-05-19  |
| 2025-02-08  |
+-------------+
```

Program 4.72: *A query answered from the index alone, and the same query asking for one column more.*

What makes an index covering: An index covers a query when every column that query names is in the index. Nothing is declared: the same index covers one statement and not the next, and it is the list of columns asked for, not only the condition, that decides.

Where the server admits it: Extra is the entry that says so. Using index means the table was never touched. Using index condition means the condition was tested inside the index first and only the surviving rows were fetched, which is cheaper than a scan and dearer than a covered read.

The decision worth naming: Adding a column to an index so that it covers a heavy report is a real technique and it is not free: the entries grow, so the index takes more disk and more work on every write. Cover the report that runs hourly, not the one that runs in December.

A covering index answers the question without the table, and it is the columns asked for, not the columns tested, that decide it.

The Master Key: *An index covers a query when it holds every column the query names, so the server answers from the index alone and prints Using index in the Extra column of the plan.*

BUILD Question 179

Six rows of borehole services occupy one page of disk at Didia. The two indexes beside them occupy two pages, and nobody typed a statement to create one of the three structures now standing on the table.

List every index the table really carries, show which one the server chooses for a lookup by service number, count what the indexes occupy, and remove the one that is doing nothing.

Declare service_no as the primary key of the log.

Create an index on service_no and an index on village.

List every index the table carries, from the catalogue the server keeps.

Plan a lookup by service_no, print what the indexes occupy, and drop the one never chosen.

Model Answer

The catalogue lists 3 indexes where only 2 were created, because PRIMARY was built by the primary key declaration itself; the plan for a lookup by service_no lists idx_service_no among the keys considered and chooses PRIMARY, and the two secondary indexes occupy 32768 bytes beside 16384 bytes of rows.

```
 1  CREATE TABLE service (
 2    service_no   INT         NOT NULL,
 3    village      VARCHAR(20) NOT NULL,
 4    serviced_on  DATE        NOT NULL,
 5    litres       INT         NOT NULL,
 6    PRIMARY KEY (service_no)
 7  ) ENGINE = InnoDB;
 8
 9  INSERT INTO service (service_no, village, serviced_on, litres) VALUES
10    (1, 'Didia',  '2024-03-04',  900),
11    (2, 'Solwa',  '2024-05-19',  450),
12    (3, 'Lalago', '2024-07-02', 1200),
13    (4, 'Didia',  '2025-01-11',  700),
14    (5, 'Solwa',  '2025-02-08',  500),
15    (6, 'Lalago', '2025-04-23', 1100);
16
17  CREATE INDEX idx_service_no ON service (service_no);
18  CREATE INDEX idx_village    ON service (village);
19
20  SELECT index_name, seq_in_index, column_name
21  FROM information_schema.statistics
22  WHERE table_schema = DATABASE()
23    AND table_name = 'service'
24  ORDER BY index_name, seq_in_index;
```

The server replies

```
+---------------+-------------+-------------+
| index_name    | seq_in_index | column_name |
+---------------+-------------+-------------+
| idx_service_no |            1 | service_no  |
| idx_village    |            1 | village     |
| PRIMARY        |            1 | service_no  |
+---------------+-------------+-------------+
```

Program 4.73: *Two indexes created by hand, and the three the table turns out to carry.*

```
 1  EXPLAIN SELECT village
 2  FROM service
 3  WHERE service_no = 4\G
 4
 5  SELECT data_length, index_length
 6  FROM information_schema.tables
 7  WHERE table_schema = DATABASE()
 8    AND table_name = 'service';
 9
10  DROP INDEX idx_service_no ON service;
11
12  SELECT COUNT(*) AS indexes_left
13  FROM information_schema.statistics
14  WHERE table_schema = DATABASE()
15    AND table_name = 'service';
```

The server replies

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: service
         type: const
possible_keys: PRIMARY,idx_service_no
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 1
        Extra:
+-------------+--------------+
| data_length | index_length |
+-------------+--------------+
|       16384 |        32768 |
+-------------+--------------+
+--------------+
| indexes_left |
+--------------+
|            2 |
+--------------+
```

Program 4.74: *The index the server chooses, what the indexes occupy, and the one that is dropped.*

The index nobody created: PRIMARY KEY builds an index, and InnoDB stores the table itself in that key order. A CREATE INDEX on the same column is accepted, kept and maintained, and the plan shows it considered and passed over in favour of PRIMARY.

What the indexes occupy: index_length is 32768 bytes against a data_length of 16384: six rows of services, and twice as much space again for the structures beside them. A page is the smallest unit InnoDB allocates, so each of the two indexes has taken one.

Removing what is never chosen: DROP INDEX takes the redundant structure away and leaves 2 entries in the catalogue, PRIMARY and idx_village. An index the plan never names is the cheapest thing in a database to remove and the most expensive to keep.

A table carries indexes nobody asked for and can carry indexes nobody uses, and only the catalogue and the plan will say which is which.

The Master Key: *A PRIMARY KEY declaration has already built an index, so indexing that column again adds a structure the server stores, maintains and never chooses, and only DROP INDEX stops the paying.*

HOT Question 180

The district office wants every column of the service log indexed before the new reports arrive at Lalago. The reports will certainly run faster. The fitters entering services all day will not, and somebody has to decide between them.

Judge whether adding indexes makes a database faster, and state the rule that decides whether a particular column should be indexed.

Model Answer

Adding indexes does not make a database faster; it makes the reads they serve faster and every write slower, so the judgement is taken one column at a time: index a column whose conditions pick out few rows of a table large enough for a scan to hurt, and leave every other column alone.

The case for the index: A read served by an index descends a few nodes and touches only the rows it wants. A read with no index touches every row, and that cost rises with the table while the cost of the descent barely moves. On a log of five thousand services this is the whole of the difference.

The case against: Every index is a structure the server stores and maintains. An INSERT writes the row and an entry in each index, a DELETE removes both, and an UPDATE of an indexed column moves the entry. That cost rises with the number of indexes and never falls.

The index that is never used at all: Worse than a costly index is one the plan never names: a leading column not given, a column wrapped in a function, a table small enough that the server reads it whole. It is maintained on every write and returns nothing on any read.

How selective the column is decides it: What an index is worth depends on how few rows the value picks out. A service number picks out 1 row of 6; a village picks out 2 of 6. Where a column holds two or three values, a lookup still reaches much of the table and there is little left to save.

Read the plan, do not time the clock: Timing a query on a log of 6 rows proves nothing, because the whole table is one page and the server may read it either way. Read the plan instead, on data of the size the table will really hold, and look at type, key and rows.

The judgement: Build the index where the plan shows type ALL on a table whose rows run into thousands, the column is named in a WHERE clause or a join condition of a query that runs often, and its values are many and well spread. Otherwise leave it, and drop any index the plan never names.

The column in question	What decides it	The judgement
named in the WHERE clause of a report that runs often, on a table of thousands of rows	the plan shows type ALL, and the value asked for picks out a handful of rows	build it
on a table of a few hundred rows	the whole table is a page or two, and the server reads it rather than descend anything	leave it
holding two or three distinct values	a lookup still reaches a large share of the rows, so little is saved	leave it, unless the value asked for is a rare one
written far more often than it is read	every INSERT, UPDATE and DELETE maintains the index, and few reads are served by it	leave it
already the leading column of an index that exists	that index serves the condition already	leave it, and add a second index only for a different leading column

Table 4.38: *Five columns a supervisor might ask to have indexed, and the judgement on each.*

More indexes is not a faster database; the right indexes are, and the plan is the only thing that says which those are.

The Master Key: *An index makes the reads it serves faster and every write slower, so build one where the plan shows a scan of a large table on a selective column that a frequent query names, and leave, or drop, every index the plan never names.*

Access control, SQL injection and password storage

This section is about what a database does when somebody outside the clinic reaches it. A vaccination register of 3 children and 2 nurses is small enough to count by eye, and it carries the 3 defences that matter: an account holding only the privileges its own work needs, a statement that keeps typed text out of its own syntax, and a password column that holds no password. The attack is run here before it is defeated, because a warning nobody has watched carried out is a warning nobody acts on.

BINDER Question 181

A password written on the inside of a drawer at Mwanhuzi was found a year after the register went in. The account it named held every privilege on every database on the server, granted in one line on the day of installation.

Explain what a privilege is, write the statements that give one and take one back, and state the principle of least privilege.

Model Answer

A privilege is a named permission to carry out one kind of operation on one named object, GRANT gives one and REVOKE takes one back, and least privilege means that an account holds exactly the privileges its own work needs and no others, so a clerk who records vaccinations is given INSERT and SELECT on the dose table and never ALL PRIVILEGES on every database on the server.

The four things a GRANT names: The privileges, the object they apply to, the user, and the host that user may connect from. Name a database and a table and the privilege reaches that one table; name a star and a star and it reaches everything the server holds, the accounts included.

An account is a user and a host together: The clerk connecting from the machine in the clinic and the clerk connecting from anywhere else are two accounts, not one. Students are often taught that a privilege belongs to a user; it belongs to a user at a host, so a grant to one leaves the other holding nothing.

Taking one back: REVOKE mirrors GRANT and reads FROM where GRANT reads TO, and it removes privileges only. The account remains, still able to connect, until DROP USER removes it. Revoking every privilege an account holds is not the same as closing the account.

Why the principle is worth the trouble: Every privilege an account holds is held by anything reaching the server through it: a mistyped statement, a stolen password, an injected condition. An account that cannot read the nurse table cannot leak it, whatever statement arrives.

A privilege granted because granting it was quicker is the one that will be used against the register.

The Master Key: *GRANT names a privilege, an object, a user and a host, REVOKE takes the privilege back without closing the account, and least privilege means an account holds only what its own work needs, so a clerk gets INSERT and SELECT on one table rather than ALL PRIVILEGES on everything.*

BINDER Question 182

Type a child's card number into the box on the register page at Nyalikungu and one child comes back. Type an apostrophe, the word OR and a comparison that is always true, and the whole register comes back instead.

Explain what SQL injection is, say exactly what makes it possible, and name the one change that ends it.

Model Answer

SQL injection is what happens when text a user typed is joined onto a statement and the server then reads part of that text as syntax instead of as a value; it is possible because the statement and the value arrive at the server as one string, and it ends when the statement is sent first and the value is sent after it, as a parameter the server can only ever treat as data.

A value crossing into the grammar: The fault is not the apostrophe. An apostrophe is merely how a value is made to end early; a number pasted in with no quotes around it needs none at all. Injection is a value becoming part of the statement, and every form of it is that.

What the server is actually given: One string, which it parses from end to end. It cannot tell which characters the developer wrote and which the visitor typed, because by the time the string arrives there is no difference left to see. Figure 4.36 puts the two arrangements side by side.

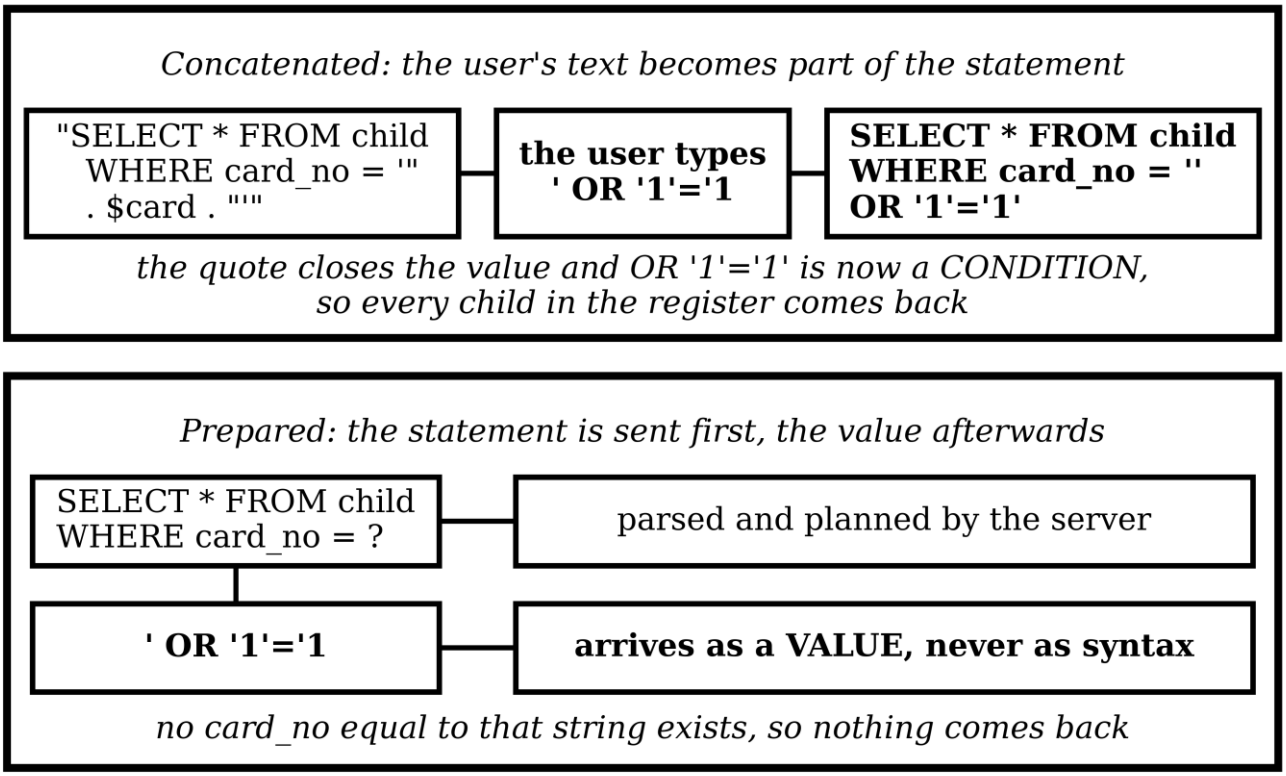


Figure 4.36: *A statement built by joining text, and the same lookup with the value sent separately.*

Escaping is a patch: Doubling the apostrophe keeps that value inside its own quotes, and it does work when it is done. It has to be done to every value, every time, in the character set the server is really using, which is a promise about code nobody has written.

Parameterisation is the fix: The statement travels to the server with a placeholder where the value belongs, and is parsed and planned while the value does not yet exist. The value follows separately, so no character in it can alter a plan that was made before it arrived.

Escaping tries to keep a value harmless inside the syntax; parameterisation keeps it out altogether.

The Master Key: *SQL injection is typed text becoming part of a statement rather than a value in it, so escaping only patches the join, and the repair is to send the statement first with a placeholder and the value afterwards, because a value arriving after parsing can never be read as syntax.*

REAL Question 183

Nothing was stolen and nothing was changed, but by the end of the morning at Nyalikungu an outsider knew what the database was called and which tables it did not have. Every word of that came from the register page itself.

Read the message the server returned below. Say what an outsider learns from each part of it, name the rule the page breaks by printing it, and say what the page should show instead.

Model Answer

The message names the database, vaccination_register, and says that staff is not a table in it, so an outsider who typed one probe into a card number box has been handed the name of the schema and can strike one guess off a list without reading a single row; the rule the page breaks is that a diagnostic message is written for whoever can repair the fault, and that person is never the visitor.

```
1  -- typed: ' UNION SELECT 1, 1, 1 FROM staff WHERE 'x'='x
2  SELECT card_no, child_name, village
3  FROM child
4  WHERE card_no = '' UNION SELECT 1, 1, 1 FROM staff WHERE 'x'='x';
```

The server refuses it

```
ERROR 1146 (42S02) at line 2: Table 'vaccination_register.staff' doesn't exist
```

Program 4.75: *A probe typed into the card number box, and what the server returned.*

Where the message came from: The typed value ended the card number early and added a query of its own naming a table called staff. The statement reached the server, failed to resolve that name, and the server said so. The visitor never saw the statement; the reply described it for him.

The first fact handed over: The name of the schema. It appears because the server qualifies any table it cannot find with the database it looked in, so vaccination_register is now known to somebody who was never told it, and every later probe can name it deliberately.

The second fact handed over: That there is no table called staff. A refusal reads like a defence, but it is an answer, and the next guess costs the outsider nothing. A guess that draws some different complaint has found a table that does exist, which is how a schema is mapped from the outside.

What the page should print: One neutral sentence saying the lookup could not be completed, and a reference the clinic can quote. The words the server chose belong in a log that the operator reads, where they are worth as much as ever and cost nothing.

What hiding the message does not do: It does not stop the injection. The probe still ran, and a value that reaches the server as syntax can do damage that no message needs to report. Silencing the page hides the map; only a parameter stops the statement being altered at all.

What the message says	What an outsider now knows	What the page should have shown
the error number and its state	the server is MySQL or MariaDB, so the dialect and the metadata tables are known	nothing about the server at all
the database name that qualifies the missing table	the schema is called vaccination_register, and can be named in the next probe	no schema name, in any form
that this table does not exist	one guess is disproved free, and a guess that fails differently has found a real table	the same neutral reply for every failure, so no guess is answered

Table 4.39: *One refusal read part by part, and what each part of it gives away.*

A message that helps whoever is reading it will help an outsider exactly as willingly as it helps the clinic.

The Master Key: *A server message names the schema, the object it could not resolve and the shape of the statement, so a page that prints it lets an outsider map the database one failed guess at a time; the message belongs in a log the operator reads, and the visitor gets one neutral reply for every failure.*

BUILD Question 184

A nurse at Kisorya typed a card number with a stray apostrophe in it, and the page answered with every child on the register. She had not attacked anything. She had ended a value early, and something else had begun where it stopped.

Build the register, run the statement the page produces for an ordinary card number, then run the statement it produces when the typed value ends the card number early, and count what each returns.

The register holds 3 children, one at Mwanhuzi, one at Nyalikungu and one at Kisorya.

Look up one card number, and count the rows that come back.

Run the statement the same box produces when the typed value adds a condition that is always true, and count those rows too.

Then run the statement a typed UNION produces, and show what it reaches.

Model Answer

The ordinary lookup returns the 1 child whose card number was asked for; the same statement, built after the typed value has ended that value early and added a condition true of every row, returns all 3 of the 3 children; and a typed UNION brings back the 2 rows of the nurse table under the headings of the child table.

```
 1  -- typed: VR002
 2  SELECT card_no, child_name, village
 3  FROM child
 4  WHERE card_no = 'VR002';
 5
 6  -- typed: ' OR '1'='1
 7  SELECT card_no, child_name, village
 8  FROM child
 9  WHERE card_no = '' OR '1'='1';
10
11  SELECT COUNT(*) AS rows_returned
12  FROM child
13  WHERE card_no = '' OR '1'='1';
14
15  SELECT COUNT(*) AS children_in_the_register
16  FROM child;
```

The server replies

```
+---------+-------------+------------+
| card_no | child_name  | village    |
+---------+-------------+------------+
| VR002   | Baraka Said | Nyalikungu |
+---------+-------------+------------+
+---------+-------------+------------+
| card_no | child_name  | village    |
+---------+-------------+------------+
| VR001   | Amina Juma  | Mwanhuzi   |
| VR002   | Baraka Said | Nyalikungu |
| VR003   | Tunu Elias  | Kisorya    |
+---------+-------------+------------+
+---------------+
| rows_returned |
+---------------+
|             3 |
+---------------+
+--------------------------+
| children_in_the_register |
+--------------------------+
|                        3 |
+--------------------------+
```

Program 4.76: *One card number asked for, and the same box returning the whole register.*

```
1  -- typed: ' UNION SELECT * FROM nurse WHERE 'x'='x
2  SELECT card_no, child_name, village
3  FROM child
4  WHERE card_no = '' UNION SELECT * FROM nurse WHERE 'x'='x';
```

The server replies

```
+---------+------------+----------------------------------------------------------------------+
| card_no | child_name | village                                                              |
+---------+------------+----------------------------------------------------------------------+
| N01     | Neema      | e1e6404a899c09ba917202c9144bcc68790dceb965099c744b065dca591c63ad |
| N02     | Asha       | e1e6404a899c09ba917202c9144bcc68790dceb965099c744b065dca591c63ad |
+---------+------------+----------------------------------------------------------------------+
```

Program 4.77: *The same card number box reaching a table the page never mentions.*

Why the second lookup returns everything: Its condition is now an OR of a comparison that matches no card number and a comparison of one with one, which is true of every row, so every row survives the filter. Nothing was broken and nothing failed: the server did exactly what the statement said.

How far the reach goes: A UNION adds a second query whose rows join the first result set, and it needs the same number of columns, which is why an outsider counts them first. The headings still say card_no and village, and the nurse rows arrive underneath them anyway.

The design decision worth naming: The account this page connects as can read the nurse table, so a statement that reaches it gets the rows. Narrowing that account is the second repair; storing a hash rather than a password is the third.

The register was not broken into; it was asked, in its own language, for everything it holds.

The Master Key: *A typed value that ends the card number early turns the rest of what was typed into syntax, so a condition true of every row returns all 3 children and a UNION returns rows of a table the page never names.*

BUILD Question 185

After the page at Mwanhuzi was rebuilt, the characters that had once returned the whole register returned nothing at all. Neither the characters nor the register had changed. Only the order in which two things reach the server had.

Send the statement to the server first with a placeholder where the card number belongs, then send the same characters as the value, and count what comes back. Then send a real card number through the same prepared statement.

Prepare the lookup with a placeholder instead of a card number.

Execute it with the characters that returned the whole register before.

Execute it again with the card number VR002.

Ask for a count both times, so a result of no rows can be seen.

Model Answer

PREPARE sends the statement with a question mark where the value will go, so the server parses and plans it while the value does not yet exist; EXECUTE hands over the characters as a value, and the count comes back as 0, because no card number equals that string, while a real card number through the same prepared statement returns its 1 row.

```
 1  PREPARE lookup FROM
 2    'SELECT COUNT(*) AS rows_returned FROM child WHERE card_no = ?';
 3
 4  SET @typed = "' OR '1'='1";
 5  EXECUTE lookup USING @typed;
 6
 7  SET @typed = 'VR002';
 8  EXECUTE lookup USING @typed;
 9
10  DEALLOCATE PREPARE lookup;
```

The server replies

```
+---------------+
| rows_returned |
+---------------+
|             0 |
+---------------+
+---------------+
| rows_returned |
+---------------+
|             1 |
+---------------+
```

Program 4.78: *The statement prepared first, and the same characters sent afterwards as a value.*

Why the count is asked for at all: A result set of no rows prints nothing, so a listing would carry silence under it. Asking for a count turns the nothing into a 0 that can be seen, which is the honest way to show a lookup matched no row.

What the placeholder is not: It is not a substitution: nothing is pasted into the statement and no quotes are added. The server holds a plan with a hole in it, and the characters fill that hole as data, so a comparison written inside them is compared, never obeyed.

The same thing in an application: A program writes a placeholder into the statement it prepares and passes the typed value to execute; the server treats it as it treated the value here. Emulation must be off, or the client fills the placeholder in and the value is only escaped.

The characters are unchanged and so is the register; only the moment the value arrived has moved.

The Master Key: *PREPARE sends the statement to the server before the value exists and EXECUTE sends the value afterwards, so characters that returned all 3 children through a joined statement return a countable 0 through a prepared one.*

BUILD Question 186

SHOW GRANTS answered in two lines at Kisorya, and one of those lines named every database on the machine. The account had been created in a hurry when the register went in, and no one had written down what it could do.

Create the clerk account, grant it exactly the privileges its work needs on one table, show what it was really given, take one privilege back, show the grants again, then remove the accounts.

The clerk records doses and reads them back. Nothing else.

Create the clerk account for the local machine, and a second account of the same name reachable from any host.

Grant INSERT and SELECT on the dose table alone, to the local account.

Show what each account holds, revoke INSERT, show it again, then remove both accounts and count what is left.

Model Answer

GRANT INSERT, SELECT on the dose table gives the clerk exactly the 2 privileges the work needs, and SHOW GRANTS proves it: USAGE on everything, which is no privilege at all, and SELECT with INSERT on that one table; the account of the same name reachable from any host holds USAGE alone, because a privilege belongs to a user and a host together, and REVOKE leaves 1 standing.

```
 1  CREATE USER 'clerk'@'localhost' IDENTIFIED BY 'Kd7mQx2Lp4';
 2  CREATE USER 'clerk'@'%'         IDENTIFIED BY 'Zr4tVn8Hs1';
 3
 4  GRANT INSERT, SELECT ON vaccination_register.dose
 5    TO 'clerk'@'localhost';
 6
 7  SHOW GRANTS FOR 'clerk'@'localhost';
 8  SHOW GRANTS FOR 'clerk'@'%';
 9
10  REVOKE INSERT ON vaccination_register.dose
11    FROM 'clerk'@'localhost';
12
13  SHOW GRANTS FOR 'clerk'@'localhost';
14
15  DROP USER 'clerk'@'localhost';
16  DROP USER 'clerk'@'%';
17
18  SELECT COUNT(*) AS accounts_named_clerk
19  FROM mysql.user
20  WHERE user = 'clerk';
```

The server replies

```
+-----------------------------------------------------------------------------------------------
-----------------+
| Grants for clerk@localhost
|
+-----------------------------------------------------------------------------------------------
-----------------+
| GRANT USAGE ON *.* TO `clerk`@`localhost` IDENTIFIED BY PASSWORD
'*A5CFB3B52B470A5E2D01CB80FDB3C01C0AFAFD86' |
| GRANT SELECT, INSERT ON `vaccination_register`.`dose` TO `clerk`@`localhost`
|
+-----------------------------------------------------------------------------------------------
-----------------+
+-----------------------------------------------------------------------------------------------
----------+
| Grants for clerk@%
|
+-----------------------------------------------------------------------------------------------
----------+
| GRANT USAGE ON *.* TO `clerk`@`%` IDENTIFIED BY PASSWORD
'*E595E87DFD49CC59B4BA517EE470C724B641D5CB' |
+-----------------------------------------------------------------------------------------------
----------+
+-----------------------------------------------------------------------------------------------
-----------------+
| Grants for clerk@localhost
|
+-----------------------------------------------------------------------------------------------
-----------------+
| GRANT USAGE ON *.* TO `clerk`@`localhost` IDENTIFIED BY PASSWORD
'*A5CFB3B52B470A5E2D01CB80FDB3C01C0AFAFD86' |
| GRANT SELECT ON `vaccination_register`.`dose` TO `clerk`@`localhost`
|
+-----------------------------------------------------------------------------------------------
-----------------+
+----------------------+
| accounts_named_clerk |
+----------------------+
|                    0 |
+----------------------+
```

Program 4.79: *Two privileges on one table, shown, reduced to one, and the accounts removed.*

Reading what came back: USAGE on everything is what every account starts with and it grants nothing; it is the line saying the account exists. The line under it is the whole of what the clerk may do, and the server lists the privileges in its own order.

Two accounts wearing one name: The second account holds USAGE and nothing else, although the grant named the same user, because it went to that user at the local host. A privilege is per user and per host; the pair is the account.

What each of the two removals does: REVOKE takes back one privilege and leaves SELECT, so the clerk can still read the doses and can no longer add one. DROP USER closes the accounts, and the count of that name is then 0.

What an account may do is not what somebody meant to give it, but what the server lists.

The Master Key: *GRANT INSERT and SELECT on one named table gives an account exactly the 2 privileges its work needs, SHOW GRANTS is where that is verified, and privileges belong to a user at a host, so the same user from another host is another account.*

BUILD Question 187

Sixty-four characters of hexadecimal stand in the password column for Neema at Mwanhuzi, and the same 64 characters stand there for Asha. Nobody typed the same string twice. They typed the same password, and that was enough.

Store the same password for both nurses through the hash function the query language provides, print what is stored, and count how many rows a single guess opens.

Both nurses chose the password Vaccine2024.

Store SHA2 of the password, at 256 bits, in a column of 64 characters.

Print what is stored for each nurse.

Count the rows that one guess at the password would open.

Model Answer

SHA2 turns the password into 64 hexadecimal characters that cannot be read back into it, but it is not a password hash: it is fast, and it is given nothing but the password itself, so the same password stores as exactly the same string for both nurses and one guess tested against the column opens 2 rows at once.

```
 1  CREATE TABLE nurse (
 2    nurse_id   CHAR(3)     NOT NULL,
 3    nurse_name VARCHAR(10) NOT NULL,
 4    pass_hash  CHAR(64)    NOT NULL,
 5    PRIMARY KEY (nurse_id)
 6  ) ENGINE = InnoDB;
 7
 8  INSERT INTO nurse (nurse_id, nurse_name, pass_hash) VALUES
 9    ('N01', 'Neema', SHA2('Vaccine2024', 256)),
10    ('N02', 'Asha',  SHA2('Vaccine2024', 256));
11
12  SELECT nurse_name, pass_hash FROM nurse ORDER BY nurse_id;
13
14  SELECT COUNT(*) AS rows_one_guess_opens
15  FROM nurse
16  WHERE pass_hash = SHA2('Vaccine2024', 256);
```

The server replies

```
+------------+------------------------------------------------------------------+
| nurse_name | pass_hash                                                        |
+------------+------------------------------------------------------------------+
| Neema      | e1e6404a899c09ba917202c9144bcc68790dceb965099c744b065dca591c63ad |
| Asha       | e1e6404a899c09ba917202c9144bcc68790dceb965099c744b065dca591c63ad |
+------------+------------------------------------------------------------------+
+----------------------+
| rows_one_guess_opens |
+----------------------+
|                    2 |
+----------------------+
```

Program 4.80: *The same password stored for two nurses with SHA2, which is a hash but not a password hash.*

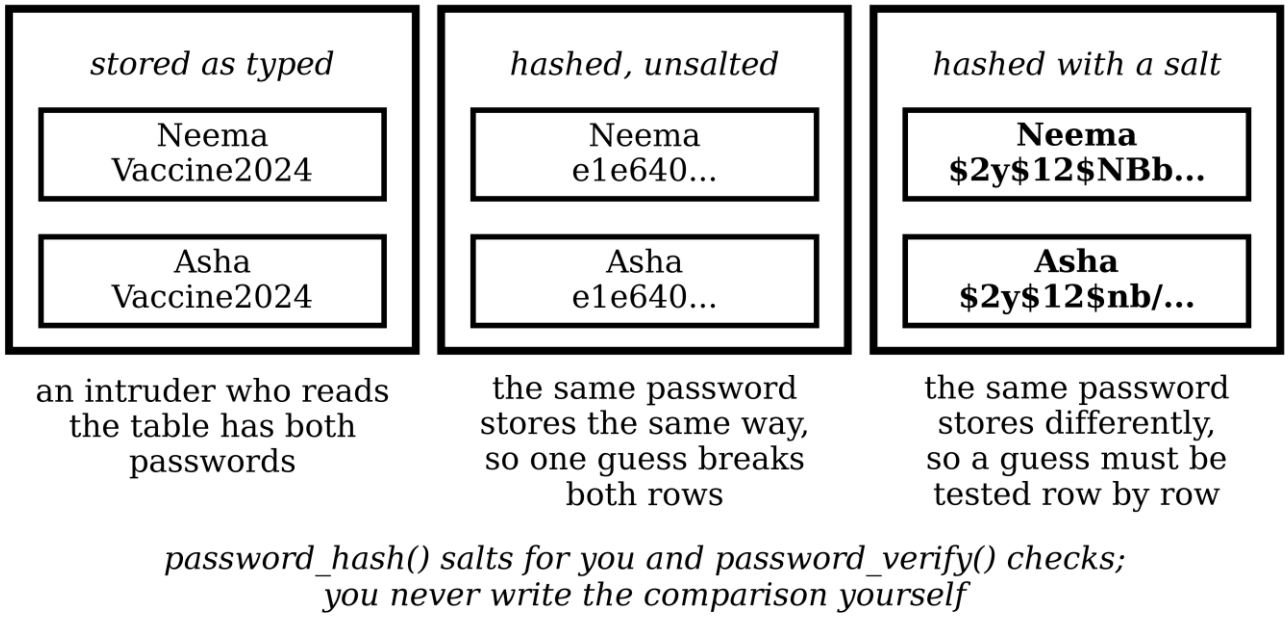


Figure 4.37: *The same two nurses under the three ways a password column can be filled.*

What a hash gives, and what it does not: It gives one direction: the password produces the string and the string does not produce the password. It does not stop guessing, and guessing is how a stolen password column is actually attacked, one candidate at a time against the whole table.

Why the two rows hold the same string: SHA2 is given the password and nothing else, so equal passwords give equal output. The middle column of Figure 4.37 is that: one guess is tested once and opens every row that shares the password, and popular passwords are shared.

What a password hash adds: Two things SHA2 has neither of: a salt, which is fresh random text mixed in for each row so that equal passwords store differently, and deliberate slowness, so that a machine testing millions of candidates a second tests very few instead.

A hash makes a password column unreadable; only a salted, slow password hash makes it expensive.

The Master Key: *SHA2 is a hash and not a password hash, because it is fast and takes nothing but the password, so 2 nurses sharing a password store the identical 64 characters and one guess opens both rows.*

BUILD Question 188

Two stored strings at Nyalikungu came from one password typed twice, and the two do not match. Neither contains the password and neither can be turned back into it, yet the register still lets the right nurse in and refuses the wrong one.

Hash the same password twice with the function written for passwords, show that the two stored strings differ, and check a right password and a wrong one against one of them.

Both nurses chose the password Vaccine2024.

Hash it once for Neema and once for Asha.

Report whether the two stored strings are the same, and how long each one is.

Check the right password and then a wrong one against what is stored for Neema.

Model Answer

password_hash returns a different string on every call, even for the same password, because it invents a fresh salt each time and writes it inside the 60 characters it hands back; password_verify reads that salt out again and does the comparison itself, so the right password is accepted against a string it does not match character for character, and a wrong one is refused.

```
 1  <?php
 2  $typed = 'Vaccine2024';
 3
 4  $neema = password_hash($typed, PASSWORD_DEFAULT);
 5  $asha  = password_hash($typed, PASSWORD_DEFAULT);
 6
 7  echo 'same password, same stored string? ',
 8    ($neema === $asha ? 'yes' : 'no'), "\n";
 9  echo 'characters stored for each nurse: ', strlen($neema), "\n";
10  echo 'Neema types Vaccine2024: ',
11    password_verify('Vaccine2024', $neema) ? 'let in' : 'refused', "\n";
12  echo 'Neema types vaccine2024: ',
13    password_verify('vaccine2024', $neema) ? 'let in' : 'refused', "\n";
```

The page prints

```
same password, same stored string? no
characters stored for each nurse: 60
Neema types Vaccine2024: let in
Neema types vaccine2024: refused
```

Program 4.81: *The same password hashed twice, and checked once rightly and once wrongly.*

Why the two strings differ: The salt. A fresh one is generated for each call and stored inside the returned string, so equal passwords no longer store equally, and the third column of Figure 4.37 is what the register holds: a guess must be tested against each row separately.

Why you never write the comparison: The stored string carries the algorithm, its cost setting and the salt, all of which password_verify reads out before comparing. Writing that comparison by hand means reproducing the salt handling, and the point of a salt is that it varies.

What the column has to hold: 60 characters today, in a format meant to change as the default algorithm changes, so declare the column VARCHAR(255) rather than sizing it to what today returns. A column too narrow truncates the string and every login then fails.

Storing a password is never necessary, because checking one needs only what the hash returned.

The Master Key: *password_hash salts each call, so the same password stores as two different strings of 60 characters, and password_verify reads that salt back out and does the comparison, which is why the register can check a password it has never kept.*

HOT Question 189

Double every quote a visitor types, says the fix taped to the monitor at Kisorya, and the register page will be safe. It is not wrong about what doubling does. It is wrong that doubling is the repair.

Judge whether escaping what a user typed is enough to stop SQL injection, and state what a register that has to survive a hostile visitor should actually do.

Model Answer

Escaping is not enough and parameterisation is, so the judgement is to parameterise every value: escape and the statement is still assembled out of text, so one value somebody forgot, one number with no quotes around it, or one character set the escaping was told wrong, and the value crosses back into the syntax, while a parameter cannot cross at all, because the statement was parsed before the value existed.

The case for escaping: It is correct in itself. Doubling an apostrophe does keep the value inside its quotes, and it can be applied to code already written without redesigning it.

Why it fails in practice: It has to be applied to every value, by every author, for ever. One value nobody escaped undoes all the others, and a number joined in without quotes needs no apostrophe to carry a condition. A defence that depends on remembering has a date on it.

The place a placeholder cannot go: A parameter carries a value, never a name. A table, a column, or the column named after ORDER BY cannot be a placeholder, so where the visitor chooses one of those, accept only names from a fixed list written in the program.

The second line, which is the account: The account the page connects as should hold SELECT on the tables the page reads and INSERT where it writes, and nothing on the nurse table at all. Whatever a statement asks for, it is answered only as far as that account reaches.

The third line, which is the message: A page printing the server's own complaint hands over the schema one guess at a time. The visitor gets one neutral reply for every failure; the detail goes to a log the operator reads.

What is still worth doing when all of it fails: Store no password anywhere. A column of salted, slow password hashes is worth far less to whoever takes it than a column of passwords, and it keeps working after every other defence has been got past.

The judgement: Parameterise every value, with emulation off. Use a fixed list of allowed names where a name rather than a value varies. Narrow the account, print no server message to a visitor, and hash the passwords with a salt. Escaping is the answer only where neither of those can be applied, which on this register is nowhere.

What is proposed	What it really protects against	The judgement
double every quote a visitor types	one value, if it is applied to that value, in the right character set, by whoever writes the next page	not the repair; keep it only where a parameter is impossible
send the statement first and the value after	every value in every statement, whatever characters it holds	do this, and turn emulation off so the client is not filling the placeholder
accept only names from a fixed list in the program	the sort column and the table name, which a parameter cannot carry	do this wherever the visitor picks a name rather than a value
grant the page account what the page uses and no more	how far anything that does get through can reach	do this; it bounds the damage rather than preventing it
show the visitor no server message	the mapping of the schema from failed guesses	do this; it hides the map, and repairs nothing on its own
store salted, slow password hashes	the value of the password column once somebody has it	do this; it is what survives every other line being crossed

Table 4.40: *Six proposals for the register page, and what each one is really worth.*

Escaping asks every future author to remember something; parameterisation asks nobody to remember anything.

The Master Key: *Escaping patches the join between a statement and a value and depends on being applied everywhere for ever, while parameterisation removes the join, so parameterise every value, use a fixed list of names where a value is not what varies, narrow the account, print no server message to a visitor, and store salted password hashes.*

Connecting from PHP with PDO

This section is about the four steps that carry a timetable out of a database and onto a page: connect, prepare, execute, fetch. The timetable is 6 periods across the 3 days Mon, Tue and Wed, small enough to count by eye, and every listing here reaches it through PDO as an account that may read it and do nothing else. The section before this one argued why a value must travel separately from the statement that uses it; this one is the code that does it, the loop that reads what comes back, and the catch that keeps a failure off the page.

BINDER Question 190

A line pinned above the office computer at Nyamikoma is all the timetable page has to find its database by. It names a driver, a host, a database and a character set. The last of those four is the one nobody thinks about.

Explain what a PDO connection is given when it is made, name each part of the DSN, and say what belongs in the options array.

Model Answer

A PDO connection is made by giving the constructor four things in order, a DSN, a user name, a password and an array of options, and the DSN is the one string mysql:host=localhost;dbname=school_timetable;charset=utf8mb4, whose first word chooses the driver and whose three settings name the machine, the database and the character set.

What the DSN is made of: The driver name comes first and ends at the colon, and it decides which database the rest of the string is read by. After it come settings separated by semicolons: host for the machine, dbname for the database, charset for the encoding. The user name and the password are separate arguments and no part of it.

Why the character set belongs there: It is applied while the connection is being made, so the first statement is already in it and the driver itself knows the encoding it is working in. A SET NAMES sent afterwards tells only the server, leaves the driver believing something else, and is one more statement to forget.

Why utf8mb4 and not utf8: In MySQL and MariaDB the name utf8 has long meant a three-byte encoding that stops short of the whole of Unicode, and utf8mb4 is the four-byte one that does not. The short one costs nothing until a character outside it arrives.

The three options set on every connection here: ERRMODE_EXCEPTION, so a failure throws instead of quietly returning false; DEFAULT_FETCH_MODE set to FETCH_ASSOC, so a row arrives once, keyed by column name; and EMULATE_PREPARES set to false, so the server prepares the statement rather than the client assembling it.

A connection that carries its own encoding has nothing left to remember.

The Master Key: *A PDO connection takes a DSN, a user, a password and an options array; the DSN names the driver before the colon and the host, the database and the character set after it, and the options that matter are ERRMODE_EXCEPTION, FETCH_ASSOC and EMULATE_PREPARES set to false.*

BINDER Question 191

The lookup at Kiabakari was rewritten in one afternoon, and only one thing moved. The day name used to be inside the statement. It now leaves the program after the statement has already gone, and reaches a server that has finished reading.

Explain what prepare and execute each do and in which order, name the two kinds of placeholder, and state what a placeholder may never stand for.

Model Answer

prepare sends the statement to the server with a placeholder where a value belongs, and the server parses and plans it there and then; execute sends the value afterwards, in an array, to a statement already read;

and because the reading is over before the value arrives, a placeholder may stand for a value only, never for a table name, a column name, or the column named after ORDER BY.

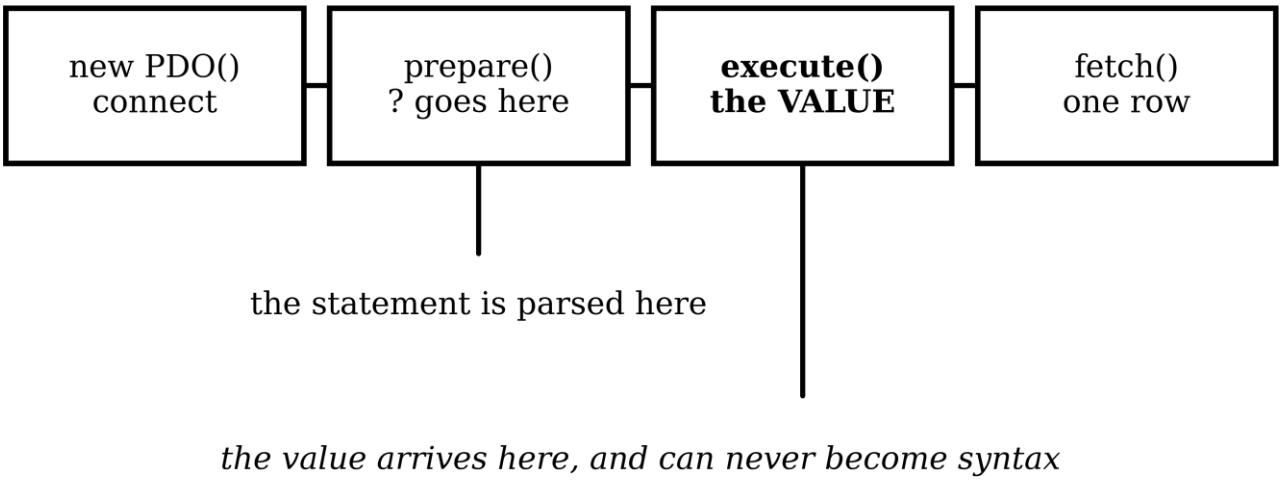


Figure 4.38: *The four steps in the order they happen, and what reaches the server at each of them.*

The four steps in order: Figure 4.38 sets them out left to right. The constructor connects, prepare takes the statement with the question mark in it, execute carries the value, and fetch brings back one row. The wire under the third box is the whole of it: the value arrives there, and can never become syntax.

Two kinds of placeholder: A question mark is positional and is filled from an array in the order the marks appear. A name written with a colon in front of it, such as :day, is filled from an array whose keys are those names. One statement takes one kind or the other, never both.

What execute hands back: It returns true, or it throws. It does not return the rows. Those stay with the statement handle and are taken from it afterwards by fetch or fetchAll, which is why the loop reads the statement variable and not whatever execute returned.

The line a placeholder cannot cross: A name is part of the grammar, and the grammar is settled while the statement is parsed, which is before any value exists. Where a visitor picks the sort column, hold a list of the statements the program will run and let the typed word choose one of them.

Everything a placeholder can and cannot do follows from the order: statement first, value second.

The Master Key: *prepare sends the statement and execute sends the value after it, positionally with question marks or by name with a colon, and a placeholder carries a value but never a table or column name, because names are settled when the statement is parsed.*

REAL Question 192

Alphabetical order was asked for at Kanazi and has never once arrived. Nothing is broken, nothing reaches any log, and the statement that does the sorting looks right on the page. The server has been obeying it exactly.

Read the two listings below. Name the fault in the first, say exactly what the server does with a placeholder after ORDER BY and what it does with one in place of a table name, and explain how the second listing lets a visitor choose the order safely.

Model Answer

The fault is that a sort column is a name and it was sent as a value, so the server prepares the statement, reads the placeholder as a constant expression, orders every row by that same constant and hands the rows back in the order it found them, Maths, English, Physics, History, Biology and Kiswahili rather than alphabetically; the same placeholder in place of a table name is not accepted at all, and the parser stops with SQLSTATE 42000 and server error 1064.

```
<?php
$dsn = 'mysql:host=localhost;dbname=school_timetable;charset=utf8mb4';
$options = [PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
            PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
            PDO::ATTR_EMULATE_PREPARES   => false];
$pdo = new PDO($dsn, 'timetable_app', 'Rk9pQ2vTn6', $options);

$typed = 'subject';

$stmt = $pdo->prepare('SELECT subject FROM period ORDER BY ?');
$stmt->execute([$typed]);
echo 'sorted by a placeholder: ';
while ($row = $stmt->fetch()) {
    echo $row['subject'], ' ';
}
echo "\n";

try {
    $stmt = $pdo->prepare('SELECT subject FROM ?');
    $stmt->execute([$typed]);
} catch (PDOException $e) {
    echo 'the same on the table name: refused, SQLSTATE ',
         $e->getCode(), ', error ', $e->errorInfo[1], "\n";
}
```

The page prints

```
sorted by a placeholder: Maths English Physics History Biology Kiswahili
the same on the table name: refused, SQLSTATE 42000, error 1064
```

Program 4.82: *A sort column sent as a value, and the same placeholder tried on a table name.*

Why nothing complains: ORDER BY takes an expression, and a parameter is a legal expression, so nothing about the statement is wrong. The value was accepted as a value, which is exactly what a placeholder promises. A fault that raises no error is found only by reading the answer.

What the rows are really ordered by: One constant, the same for every row, so every row ties with every other. Ties are settled by whatever order the rows reached the sort, which here is the order they were inserted. Add a period tomorrow and the same listing may print a different order again.

The same mistake, refused: A table name has no expression position in the grammar, so the parser reaches a question mark where a name must be and stops. One rule with two fates: where the grammar can read a value it does, in silence, and where it cannot it refuses outright.

```
<?php
$dsn = 'mysql:host=localhost;dbname=school_timetable;charset=utf8mb4';
$options = [PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
            PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
            PDO::ATTR_EMULATE_PREPARES   => false];
$pdo = new PDO($dsn, 'timetable_app', 'Rk9pQ2vTn6', $options);

$typed = 'subject';

$statements = [
    'subject' => 'SELECT subject FROM period ORDER BY subject',
    'day'     => 'SELECT subject FROM period ORDER BY day_name',
];
$sql = $statements[$typed] ?? $statements['day'];

$stmt = $pdo->prepare($sql);
$stmt->execute();
echo 'chosen from the list: ';
while ($row = $stmt->fetch()) {
    echo $row['subject'], ' ';
}
echo "\n";

$pdo = null;
```

The page prints

```
chosen from the list: Biology English History Kiswahili Maths Physics
```

Program 4.83: *The order chosen from a list of statements the program itself holds.*

How the correction works: The program holds the statements it is willing to run and the typed word chooses one of them by key, so the sort column is always a name the program wrote. Anything not in the list falls back to the default, and Program 4.83 returns the six subjects alphabetically.

Why a list beats a check: A check on typed text has to be right about every character a column name may hold, for ever. A list has to be right about the orders the page actually offers, and it is short enough to read at a glance. The shorter promise is the one that keeps being kept.

The dangerous half of this fault is not the refusal; it is the sort that ran, cost nothing and did nothing.

The Master Key: *A placeholder is a value, so one after ORDER BY sorts every row by the same constant and raises no error at all, while one in place of a table name is refused with error 1064, and the repair is to choose the whole statement from a list the program holds.*

BUILD Question 193

An argument at Nyamikoma about which database the page was really reading ran for a week. Nobody had thought to ask the page. Two lines of output ended it, and the second of the two settled a second argument nobody had started.

Connect to the timetable with PDO, prove which database and which character set the connection is on, and release the connection.

The database school_timetable is on the same machine as the page.

Connect as the account timetable_app, with the three options this book sets on every connection.

Ask the server which database the connection is using and which character set the client is sending, and print both.

Release the connection when the work is done.

Model Answer

The constructor is handed the DSN, the user name, the password and the options array, and either returns a connection or throws, so it sits inside a try; the connection then answers for itself, naming school_timetable and utf8mb4, which is what the DSN asked for and what no later SET NAMES had to be remembered for.

```
<?php
$dsn = 'mysql:host=localhost;dbname=school_timetable;charset=utf8mb4';
$options = [PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
            PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
            PDO::ATTR_EMULATE_PREPARES   => false];
$pdo = null;

try {
    $pdo = new PDO($dsn, 'timetable_app', 'Rk9pQ2vTn6', $options);
    $sql = 'SELECT DATABASE() AS db, @@character_set_client AS cs';
    $row = $pdo->query($sql)->fetch();
    echo 'connected to: ', $row['db'], "\n";
    echo 'characters:   ', $row['cs'], "\n";
} catch (PDOException $e) {
    echo "the timetable is not available just now\n";
}

$pdo = null;
```

The page prints

```
connected to: school_timetable
characters:   utf8mb4
```

Program 4.84: *A connection made, questioned about itself, and released.*

Where the connection is actually made: In the constructor. Making the object opens the connection, so a wrong host, a wrong password or a database that is not there all fail on that one line, and with ERRMODE_EXCEPTION set they fail by throwing.

Why this statement needs no placeholder: Nothing in it came from outside, so query sends it as it stands and hands back a statement to fetch from. The moment a value from a form is involved, that becomes prepare and execute, and there is no third way.

The last line, honestly: PHP releases a PDO connection when the script ends, so assigning null on the last line of a short page changes nothing measurable. Write it where a long script has work left after the database is done with, and know it releases only when nothing else holds the connection, a live statement handle included.

A connection that can be asked what it is doing beats one assumed right.

The Master Key: *The constructor opens the connection and throws if it cannot, and the DSN carries the character set, so the connection can afterwards be asked which database and which encoding it is on and answers school_timetable and utf8mb4.*

BUILD Question 194

The second loop at Kiabakari printed nothing. It was the same statement, the same page, the same result set, with only a blank line between the two loops, and the first loop had printed three rows. Nobody had deleted anything.

Prepare the lookup with a placeholder, execute it with the slot number, print each row as it arrives, then walk the same result set a second time and count what the second walk finds.

The timetable holds 6 periods, two in each of the 3 days Mon, Tue and Wed.

Ask for the first period of each day, sending the slot number as a value.

Print the day and the subject of every row as it arrives.

Walk the same result set again, and count the rows that second walk finds.

Model Answer

The loop prints Mon Maths, Tue Physics and Wed Biology one row at a time, because fetch returns the next row as an array and false when there is none, and the second walk counts 0, because the pointer is now past the last row and the rows were not lost, they were read.

```
 1  <?php
 2  $dsn = 'mysql:host=localhost;dbname=school_timetable;charset=utf8mb4';
 3  $options = [PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
 4              PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
 5              PDO::ATTR_EMULATE_PREPARES   => false];
 6  $pdo = new PDO($dsn, 'timetable_app', 'Rk9pQ2vTn6', $options);
 7
 8  $sql = 'SELECT day_name, subject
 9          FROM period
10          WHERE slot_no = ?
11          ORDER BY period_id';
12
13  $stmt = $pdo->prepare($sql);
14  $stmt->execute([1]);
15
16  while ($row = $stmt->fetch()) {
17      echo $row['day_name'], '  ', $row['subject'], "\n";
18  }
19
20  $second = 0;
21  while ($row = $stmt->fetch()) {
22      $second = $second + 1;
23  }
24  echo 'rows the second walk found: ', $second, "\n";
25
26  $pdo = null;
```

The page prints

```
Mon  Maths
Tue  Physics
Wed  Biology
rows the second walk found: 0
```

Program 4.85: *One result set, walked twice, and counted the second time.*

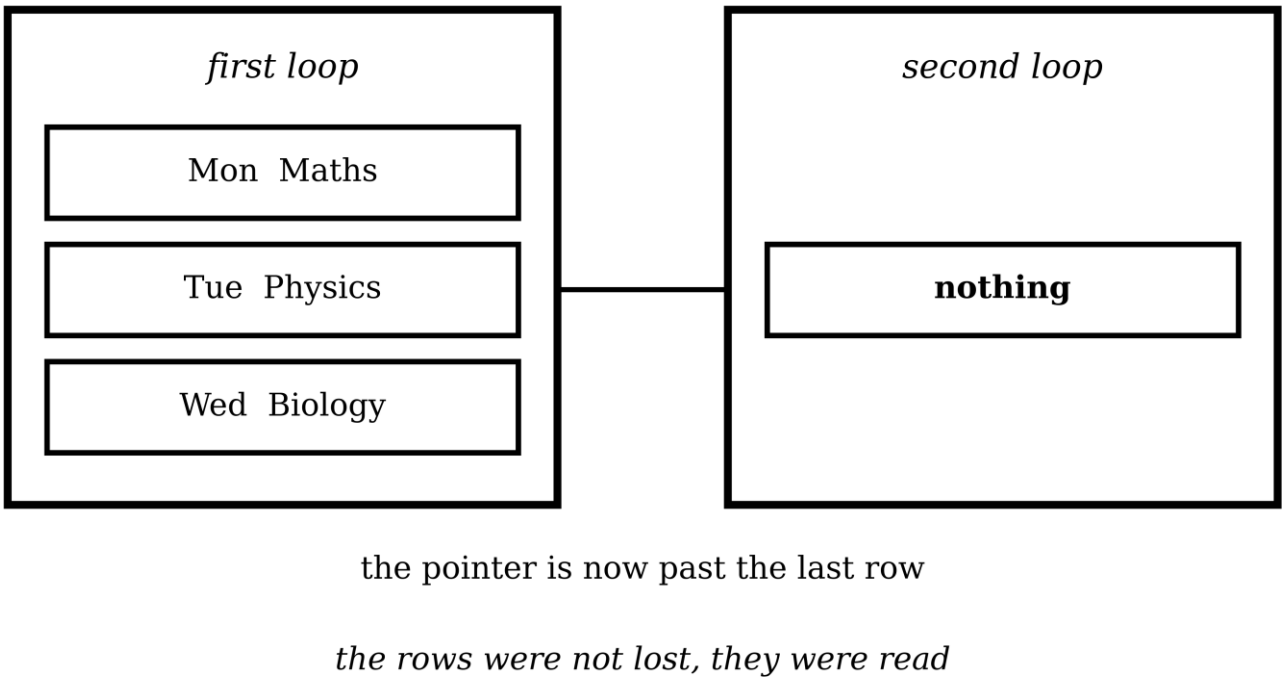


Figure 4.39: *The same result set under two loops, and where the pointer is left standing.*

How the loop reads: The assignment is the condition. fetch is called, its answer is put in the row variable, and that answer is what while tests, so the loop stops of its own accord when false comes back. It needs no counter and no count.

Why the second walk finds nothing: Figure 4.39 draws it. The first loop took Mon Maths, Tue Physics and Wed Biology in turn and left the pointer past the last of them, so the second loop asks for a next row and is told at once that there is none.

When the rows are wanted twice: Read them into an array with fetchAll and walk that as often as you like, or execute the statement again and walk a fresh result set. What cannot be done is walking one set twice, and a printed count is how the emptiness is made visible.

A result set is a queue, not a table: what is taken out is gone from it.

The Master Key: *fetch hands back the next row and false at the end, so a while loop with fetch as its condition walks a result set exactly once, and a second walk of the same statement finds 0 rows, the pointer being past the last one.*

BUILD Question 195

One statement was prepared once at Kanazi and executed three times. The first two answered exactly as the register says they should, Tue slot one and Wed slot two. The third came back with nothing, and nothing was wrong with it.

Write the lookup with named placeholders, prepare it once, execute it for Tue slot 1, for Wed slot 2 and for a day the timetable does not have, and print a count for the last of those so the nothing can be seen.

The lookup takes a day and a slot and returns the subject, the teacher and the room.

Name the placeholders rather than leaving them to their positions.

Prepare once. Execute for Tue slot 1, then for Wed slot 2, then for Fri slot 1.

Fri is not in the timetable, so print a count there rather than a row.

Model Answer

Named placeholders let the values be handed over in an array keyed by name, so the day and the slot cannot be transposed, and the one prepared statement answers Physics with Mr Nzowa in L1 for Tue slot 1, Kiswahili with Mr Bahati in R1 for Wed slot 2, and false for Fri slot 1, printed as a countable 0.

```
<?php
$dsn = 'mysql:host=localhost;dbname=school_timetable;charset=utf8mb4';
$options = [PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
            PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
            PDO::ATTR_EMULATE_PREPARES   => false];
$pdo = new PDO($dsn, 'timetable_app', 'Rk9pQ2vTn6', $options);

$sql = 'SELECT subject, teacher, room
        FROM period
        WHERE day_name = :day AND slot_no = :slot';
$stmt = $pdo->prepare($sql);

$stmt->execute([':day' => 'Tue', ':slot' => 1]);
$row = $stmt->fetch();
echo 'Tue slot 1: ', $row['subject'], ' with ', $row['teacher'],
     ' in ', $row['room'], "\n";

$stmt->execute([':day' => 'Wed', ':slot' => 2]);
$row = $stmt->fetch();
echo 'Wed slot 2: ', $row['subject'], ' with ', $row['teacher'],
     ' in ', $row['room'], "\n";

$stmt->execute([':day' => 'Fri', ':slot' => 1]);
$row = $stmt->fetch();
echo 'periods for Fri slot 1: ', ($row === false ? 0 : 1), "\n";

$pdo = null;
```

The page prints

```
Tue slot 1: Physics with Mr Nzowa in L1
Wed slot 2: Kiswahili with Mr Bahati in R1
periods for Fri slot 1: 0
```

Program 4.86: *One statement prepared once and executed three times, the last of them for a day that is not there.*

Why a name beats a position: With question marks the order of the array alone decides which value lands where, and two values of the same kind make a transposition silent. With names each value is labelled where it is handed over, and the statement can be rewritten without touching the array.

What the second and third executions send: The values, and nothing else. The statement was parsed and planned when it was prepared, so a lookup wanted for every row of a page is prepared outside the loop and executed inside it.

The day that is not there: fetch found no row and returned false, which is what it also returns at the end of a set. Printing that as a 0 is deliberate: a result of no rows prints nothing, and a page showing nothing tells a reader neither that it worked nor that it failed.

One statement, three executions, and the third answer is as useful as the other two.

The Master Key: *Named placeholders are filled from an array keyed by name, a statement is prepared once and executed as often as needed with only the values travelling, and a lookup matching no row returns false, printed as a count so the zero shows.*

BUILD Question 196

A row of this timetable has three columns in it. The array PDO hands back, when nobody has said otherwise, holds six entries, and three of those are the same three values under other keys. Half of what arrives is a copy.

Read the first period of each day into an array in one call, walk that array twice, then fetch one row in the mode PDO falls back on and count the entries in each kind of row.

Read the first period of each day into an array with one call.

Print the subjects out of that array, then print the days out of the same array.

Fetch one row again in FETCH_BOTH, the mode PDO uses when the options array names none.

Count the entries in an associative row and in a FETCH_BOTH row.

Model Answer

fetchAll empties the result set into an ordinary array of 3 rows, which may be walked as often as needed because an array holds no pointer; each row arrives as 3 entries keyed day_name, subject and room under FETCH_ASSOC, and the same row under FETCH_BOTH arrives as 6, every column once under its name and once under its position.

```
<?php
$dsn = 'mysql:host=localhost;dbname=school_timetable;charset=utf8mb4';
$options = [PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
            PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
            PDO::ATTR_EMULATE_PREPARES   => false];
$pdo = new PDO($dsn, 'timetable_app', 'Rk9pQ2vTn6', $options);

$sql = 'SELECT day_name, subject, room
        FROM period
        WHERE slot_no = ?
        ORDER BY period_id';
$stmt = $pdo->prepare($sql);
$stmt->execute([1]);
$rows = $stmt->fetchAll();
echo 'rows held in the array: ', count($rows), "\n";
echo 'entries in one row:     ', count($rows[0]), "\n";
echo 'first pass:  ';
foreach ($rows as $row) { echo $row['subject'], ' '; }
echo "\n";
echo 'second pass: ';
foreach ($rows as $row) { echo $row['day_name'], ' '; }
echo "\n";

$stmt->execute([1]);
$row = $stmt->fetch(PDO::FETCH_BOTH);
echo 'entries in a FETCH_BOTH row: ', count($row), "\n";
echo implode(' ', array_keys($row)), "\n";

$pdo = null;
```

The page prints

```
rows held in the array: 3
entries in one row:     3
first pass:  Maths Physics Biology
second pass: Mon Tue Wed
entries in a FETCH_BOTH row: 6
day_name 0 subject 1 room 2
```

Program 4.87: *The whole set taken in one call and walked twice, then one row in the fallback mode.*

What fetchAll gives you: One call, the whole set, and nothing left in the statement afterwards. The array outlives the statement and the connection, so its rows can be printed twice, counted, or sorted again in the program.

Why the associative row is the one to ask for: Its keys are the column names, so code that reads it says what it is reading and survives a change in the order of the columns. A row of 3 columns is 3 entries, and nothing in it is present twice.

What the fallback costs: FETCH_BOTH doubles every row, keyed by name and by position, so 3 columns become 6 entries. Anything walking the row rather than naming its keys meets every value twice, and the second copy carries nothing the first did not.

A row should arrive in one shape, and the shape is chosen once, in the options array.

The Master Key: *fetchAll returns the whole result set as an ordinary array that may be walked more than once, FETCH_ASSOC gives a row of 3 entries keyed by column name, and the FETCH_BOTH fallback returns the same row as 6 by adding a positional copy of every column.*

BUILD Question 197

One letter is wrong in the statement at Nyamikoma, subjekt where the column is subject. The visitor sees a short sentence with a reference number in it. The operator, reading the log an hour later, sees something else entirely.

Open the connection so that a failure throws, run a statement naming a column that is not there, catch what comes back, write one string to the log and show the visitor another, and print both so they can be compared.

The statement names subjekt, which is not a column of period.

The connection is opened with ERRMODE_EXCEPTION, so the failure throws.

Catch it. Write the server text to a log file, and show the visitor a neutral line with a reference on it.

Print the log line afterwards as well, so that the two strings can be compared.

Model Answer

With ERRMODE_EXCEPTION in the options array the failed prepare throws a PDOException, so the catch runs, and the two strings leaving it are written for two different readers: the visitor gets one sentence and a reference to quote, the log gets the server text, naming the column subjekt and the error number 1054.

```
 1  <?php
 2  $dsn = 'mysql:host=localhost;dbname=school_timetable;charset=utf8mb4';
 3  $options = [PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
 4              PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
 5              PDO::ATTR_EMULATE_PREPARES   => false];
 6  $log = '/tmp/timetable.log';
 7  $pdo = null;
 8
 9  try {
10      $pdo  = new PDO($dsn, 'timetable_app', 'Rk9pQ2vTn6', $options);
11      $stmt = $pdo->prepare('SELECT day_name, subjekt FROM period');
12      $stmt->execute();
13      while ($row = $stmt->fetch()) {
14          echo $row['day_name'], '  ', $row['subjekt'], "\n";
15      }
16  } catch (PDOException $e) {
17      file_put_contents($log, $e->getMessage() . "\n");
18      echo "The timetable cannot be shown just now. Quote TT-19.\n";
19  }
20
21  $pdo = null;
22  echo "and the log now holds:\n";
23  echo file_get_contents($log);
```

The page prints

```
The timetable cannot be shown just now. Quote TT-19.
and the log now holds:
SQLSTATE[42S22]: Column not found: 1054 Unknown column 'subjekt' in 'SELECT'
```

Program 4.88: *One failure, one line to the visitor and another to the log.*

What the other error modes would have done: ERRMODE_SILENT returns false and sets a code nobody reads, so the page walks on and fetches from a statement that was never prepared. ERRMODE_WARNING prints a warning wherever the output goes. Only the exception stops the page where the fault is.

Why the two strings differ: The one on the page is read by somebody who can repair nothing and may not be friendly, so it names no column, no table and no database. The one in the log is read by whoever can repair it, and is worth keeping whole.

What the catch must not do: It must not put the server text on the page, nor swallow the exception and carry on as though the result set were empty. A fault reported as no rows is worse than one reported plainly, because nobody goes looking for it.

A failure is either a repairable fault or a map of the database, and the catch decides which.

The Master Key: *ERRMODE_EXCEPTION makes a failed statement throw, and the catch writes the server text, column name and error number included, to a log for the operator, while the visitor sees one neutral sentence and a reference to quote.*

HOT Question 198

The page at Kiabakari was changed to hold its rows in an array, then changed back a month later by somebody who had read that a loop uses less memory. Both were certain. Neither had asked the driver.

Judge whether a page should walk its rows one at a time with fetch or hold them all with fetchAll, and state the condition that decides it.

Model Answer

Hold the rows with fetchAll wherever they are wanted more than once, must outlive the statement, or are bounded by the query itself, and walk them with fetch otherwise; the memory argument usually offered for the loop does not decide it, because this driver buffers by default, so the whole set reaches the client before the first fetch returns and the loop saves nothing until buffering is switched off deliberately.

The case for the loop: It is the shape the work already has: one row arrives, one row is printed, nothing is kept. There is no array to name and no second pass to keep in step.

The case for the array: The rows outlive the statement and the connection. A second pass over them costs nothing, the count of them is the count of an array, and the result set is finished with the moment the call returns.

The claim that does not survive checking: PDO on MySQL and MariaDB buffers a result set by default, so every row has already been copied into the client before the first fetch is called. Walking a buffered set one row at a time saves no memory whatever; it reads a copy that is there already.

What switching buffering off really buys: Rows then arrive from the server as they are asked for, which does bound the memory a large set needs. The connection is busy until the set is finished, though, and a second query part way through the walk is refused outright.

The set nobody has measured: A query with no limit on a growing table is where holding the rows eventually fails. If the set is not bounded by the query, bound it or read it unbuffered.

The order that is not negotiable either way: The whole set is read before the connection is released, never after. Under a buffered set the wrong order appears to work, because the rows are already in the client and the statement still holds the connection open, which is why the habit survives.

The judgement: Use fetchAll for a set the query bounds, which is nearly every page a school timetable will show, and keep the loop for a set walked once and thrown away. Choose by whether the rows are wanted twice and whether the set is bounded, never by a memory saving a buffered driver has already spent.

What is claimed	What is actually the case here	The judgement
a loop uses less memory than fetchAll	not with the default buffered set, which reaches the client whole before the first fetch returns	true only where buffering has been switched off on purpose
fetchAll is wasteful	it costs one array of the rows the page was going to read anyway	use it wherever the query bounds the set
the rows can be walked again after the loop	they cannot: the pointer is past the last row and nothing is left to fetch	hold them in an array, or execute the statement again
closing early is safe enough	it appears to work only because the set was buffered and the statement still holds the connection	read the whole set, then release; never the other way about

Table 4.41: *Four things said about walking a result set, and what each is worth.*

The loop and the array differ in what the program may do next, not in what the driver already fetched.

The Master Key: *PDO buffers by default, so a loop over a buffered set saves no memory; hold the rows with fetchAll when they are wanted twice or must outlive the statement, walk them with fetch when each row is finished with as it is printed, and switch buffering off only for an unbounded set.*

Parallel and distributed databases

A parallel database and a distributed database are not two points on one scale, and telling them apart is the first thing this section settles. It then cuts a table across sites, by rows and by columns, and shows which cuts can be put back together and which cannot; it says what a copy at every site buys and what it costs; and it names the three promises a distributed system makes to the person writing the query. It closes on the two hard parts, committing one transaction at two sites at once, and deciding what the sites should do when the network between them stops working.

BINDER Question 199

One room at Puge holds one set of records and three processors working over it. Two huts on opposite sides of the estate each hold their own rows. Both are called splitting the load, and only one has split the data.

Distinguish a parallel database system from a distributed database system, and state what decides which of the two a system is.

Model Answer

A parallel database keeps all of its data at one site and sets several processors to work on it at the same time, while a distributed database keeps different parts of its data at different sites and presents the whole of it to the user as one database.

The parallel arrangement: One site holds the data and two or more processors share the work of answering a query, each taking part of the scan or part of the sort. The gain is that one query finishes sooner. Nothing has moved: there is still one stored copy of the table, in one building, on one set of disks.

The distributed arrangement: Bugene holds some of the rows and Rubya holds the rest, each site running its own server over its own disk with a network between them. One management system ties the sites together, so a query names the table and not the site that happens to be holding the part it needs.

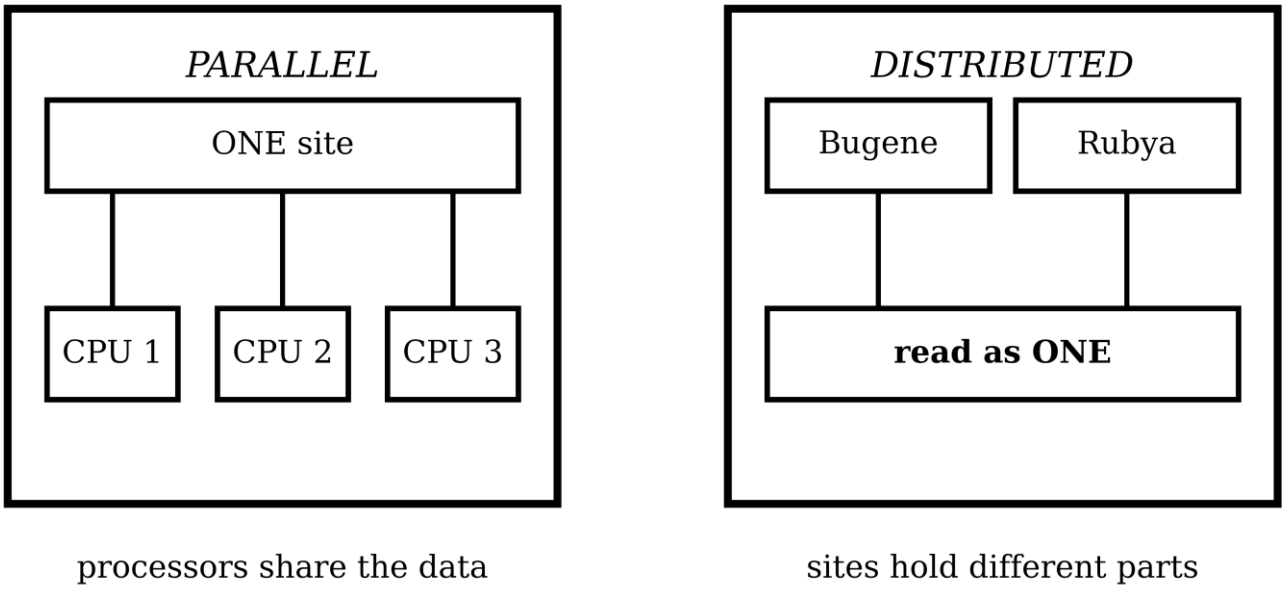


Figure 4.40: *Processors sharing one store of data, against sites each holding a part of it.*

What decides which is which: Ask where the data sits. Many processors over one stored copy is parallel however many processors there are, and parts of the data at different sites is distributed even where each site runs a single processor. The two can be combined, since a site of a distributed system may itself be a parallel machine.

Not merely spread out: Two separate databases at two sites, each queried on its own, are not a distributed database. What makes it one is that the sites are managed together and read as one, so a person who never learns their names still gets an answer covering the whole estate.

A reader who can only say that distributed means spread out has said nothing about sites and nothing that separates the two arrangements.

The Master Key: *Parallel means one site and many processors over shared data, while distributed means many sites, each holding part of the data and all read as one.*

BINDER Question 200

Half the weighbridge table went up to the office at Rubya and half stayed in the hut. Both halves are tidy and neither has lost a column. Laid side by side on the desk, they cannot be matched back into a single line.

Explain horizontal and vertical fragmentation, state the rule every vertical fragment must obey, and say what mixed fragmentation is.

Model Answer

Horizontal fragmentation divides a table by rows, so each fragment holds every column but only some of the rows, while vertical fragmentation divides it by columns, so each fragment holds some of the columns for all of the rows; and every vertical fragment must repeat the primary key, or the fragments can never be put back together.

Horizontal fragmentation: A condition on a column decides which rows go where. Sending every row whose site is Bugene to Bugene, and every row whose site is Rubya to Rubya, gives each weighbridge its own bales with bale_no, site, kg and grade all present. The whole table is the fragments taken together, so a union rebuilds it.

Vertical fragmentation: A choice of columns decides which attributes go where, so one fragment carries the weighing columns and another carries the grading column. Every fragment holds a row for every bale. The whole table is rebuilt by joining the fragments, and a join has to have something to join on.

The clause that is always dropped: The primary key must appear in every vertical fragment. This is the lossless-join test set out earlier in this chapter, now applied across sites: the attributes two fragments share must be a superkey of at least one of them, and bale_no is the only set of columns that always is.

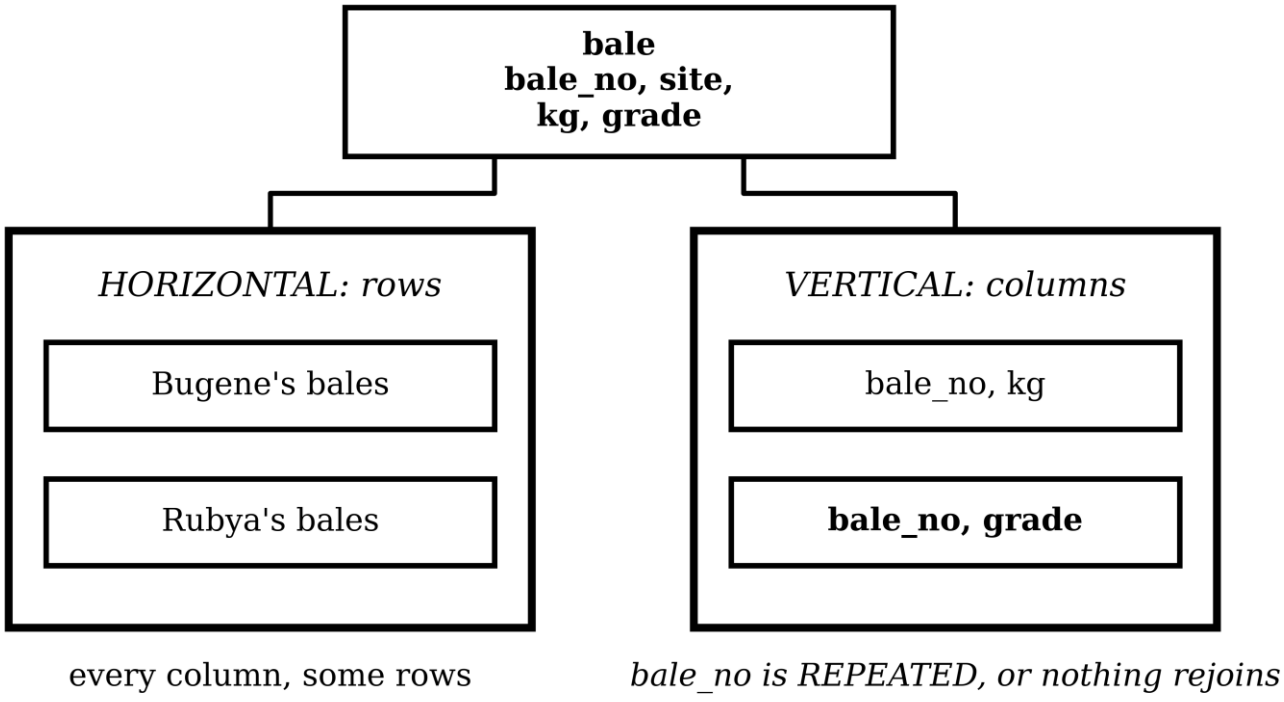


Figure 4.41: *One table cut by rows and cut by columns, with the key repeated on the vertical side.*

Mixed fragmentation: Applying one kind and then the other gives a mixed, or hybrid, fragmentation: split the bales by site, then split each site into weighing columns and grading columns. It is rebuilt by undoing the two steps in reverse, joining the column pieces first and taking the union afterwards.

Both cuts are reversible, but only one is reversible by accident: a horizontal fragment carries the key already, and a vertical fragment carries it only if somebody put it there.

The Master Key: *Horizontal fragmentation splits rows and rejoins by union, vertical fragmentation splits columns and rejoins by join, and a vertical fragment that does not repeat the primary key cannot be rejoined at all.*

BINDER Question 201

Every price change at Puge is telephoned twice, once to Bugene and once to Rubya, because the list lives in one place only. Copying it to both huts would end the telephoning and start something else.

Explain what replication is in a distributed database, and state plainly what it costs.

Model Answer

Replication stores the same data at more than one site, which makes a read local and fast at every site holding a copy, and turns every write into a problem that has to be solved at all of the copies at once.

Full and partial replication: Full replication puts the whole database at every site, so any site can answer any question. Partial replication copies only what a site actually reads, which for this estate means the small table of grades at both weighbridges and the weighings themselves only where they were taken.

What a copy buys: A read is answered from the disk in the same room, with no journey across the network and no dependence on a link that fails in the rains. It also buys survival, since the rows Bugene holds are still readable at Rubya when Bugene is down.

What a copy costs: One update becomes as many updates as there are copies, and the disk is paid for as many times over. The copies also disagree for as long as the update takes to reach them all, so a read at Rubya can return a weight that Bugene has already corrected.

What is being done	One copy, at Puge	A copy at every site
A read at Bugene	Crosses the network to Puge	Answered from the disk in the room
A write	One update, in one place	One update at each of the copies
Puge unreachable	No site can read anything	Every site still reads its own copy
Disk used	One table	One table for every site

Table 4.42: *What a second copy changes, item by item.*

The trade stated plainly: Replication moves cost from reads to writes. A table read everywhere and written rarely, such as the list of grades, is worth copying to every site, and a table written all day, such as the weighings, is not.

Replication is not a feature to be switched on but a trade, and what is traded away is having one place where a value is changed.

The Master Key: *A copy at every site makes every read local and makes every write a distributed problem, so replicate what is read often and written rarely.*

BINDER Question 202

One SELECT typed at Puge names the table and the answer comes back complete. Half of what it carries was weighed at Rubya, three hours away by road, and nobody at the keyboard has been told where any of the rows live.

Name the three kinds of transparency a distributed database offers, and state what each one promises the person using it.

Model Answer

Location transparency promises that the user need not know which site holds the data, fragmentation transparency promises that the user need not know the table was ever split, and replication transparency promises that the user need not know how many copies exist or which of them answered.

What the word is promising: Each kind of transparency names one thing the user does not have to know, because the system knows it instead. What the user writes is then what they would have written against a database standing on one machine in the same room.

Location transparency: The query names bale and never names the site in front of it. Moving the Rubya fragment to Puge becomes a change to the catalogue rather than a change to any statement, so no program has to be rewritten because a server was carried to another building.

Fragmentation transparency: The user writes one SELECT against bale as though it were whole. The system works out that the answer needs the union of two horizontal fragments, or the join of two vertical ones, and does that without being asked and without saying so.

Replication transparency: The user writes one UPDATE. The system applies it to every copy and decides which copy serves a later read, so the number of copies can be raised or lowered without touching anybody else's program.

What transparency does not hide: It hides where the data is, never how far away it is. A query answered across a slow link is still slow, and when the link fails altogether nothing can hide that either, which is what the last two questions here have to settle.

State each as a sentence about what the user is spared and the three cannot be muddled: not knowing where, not knowing that it was split, not knowing how many copies.

The Master Key: *Location transparency hides the site, fragmentation transparency hides the split and replication transparency hides the copies, each being a promise about what the user does not have to know.*

REAL Question 203

A transfer note for one bale sits half written in two places at once, at Bugene and at Rubya. Each clerk has promised to complete it and neither may tear it up. The word that would settle it has not come from Puge.

Trace the transfer through both phases of a two-phase commit, say what each site must answer at each phase, and state exactly what happens if the coordinator fails between the two phases.

Bale B0103 was weighed at Bugene and has now been carted to the yard at Rubya.

One transaction must delete the row at Bugene and insert it at Rubya, both or neither.

The coordinator runs at the estate office at Puge; Bugene and Rubya are the participants.

Model Answer

Puge asks Bugene and Rubya whether each is able to commit and waits for both answers, and only when both say yes does it record the decision and tell them both to commit; if Puge fails after collecting the answers and before sending the decision, each site is left holding its locks and its half of the transfer, able neither to commit nor to abort alone.

Phase one, the prepare: Puge sends a prepare message to both sites. Each does its own half of the work, the delete at Bugene and the insert at Rubya, writes enough into its log to survive a restart, keeps its locks, and answers yes or no. Answering yes gives up the right to abort alone.

What a yes actually promises: Yes means the site can commit whatever happens next, including a power cut, because the work is already in the log. That is why the answer is a promise and not an opinion, and why a site that has said yes may not change its mind.

Phase two, the decision: If both answers are yes, Puge writes the commit into its own log first and then tells both sites to commit. If either answered no, Puge tells both to abort. The decision is taken once, at one site, and applied everywhere, which is what makes the outcome the same at Bugene and Rubya.

Step	At Puge, the coordinator	At Bugene and at Rubya
Phase one	Sends prepare to both sites	Does the work, logs it, holds its locks
The reply	Waits for an answer from both	Answers yes, or answers no
Phase two	Logs the decision, then sends it	Commits, or aborts, as it is told
The end	Waits for both acknowledgements	Releases its locks

Table 4.43: *The two phases, and what each side does in each of them.*

Every participant has to answer: Silence is not consent. A site that cannot be reached leaves Puge waiting, and a coordinator that will not wait for ever has to abort, so one unreachable weighbridge can undo a transfer the other weighbridge was ready to complete.

Where it blocks: If Puge fails after the answers and before the decision, Bugene and Rubya are stuck. Each has said yes, so neither may abort; neither has been told to commit, so neither may commit. Both wait with their locks held and every transaction wanting those rows waits behind them. Two-phase commit is a blocking protocol, and saying it guarantees atomicity without saying this is half an answer.

What it does give: It makes one transaction all-or-nothing across sites, which is atomicity carried past a single server, and it costs two rounds of messages. A stuck participant recovers by asking the others what they were told, and where none of them knows, by waiting for the coordinator to come back.

The protocol is sound and it is not free, because it buys all-or-nothing across sites with a window in which neither site can decide anything.

The Master Key: *Two-phase commit prepares every site, commits only on a unanimous yes, and leaves every participant blocked with its locks held if the coordinator dies between the phases.*

REAL Question 204

Chalked on the board of the weighbridge hut at Bugene are three statements about distributed databases, left by whoever used the room last. Two of them are wrong, and the one that is right sounds more like a complaint than a rule.

Say which of the three chalked statements is true, name exactly what is wrong with each of the other two, and write a corrected version of both.

A. A parallel database spreads the data over several sites, while a distributed database runs several processors over one copy of it.

B. In a vertical fragmentation each column belongs to exactly one fragment, so that no column is ever stored twice.

C. Replication makes reads faster and writes more expensive.

Model Answer

Statement C is the true one; A has the two definitions the wrong way round, and B is wrong because the primary key has to be repeated in every vertical fragment, so exactly one column is stored twice on purpose.

Why C is true: A copy in the room answers a read without crossing the network, so the read gets faster. A write has to reach every copy, so one update becomes as many as there are copies, and until the last of them has landed the copies disagree with one another.

What is wrong with A: The two halves have been exchanged. Written as A has it, a single machine at Puge with three processors over one stored table would be called distributed, and two huts holding a fragment each would be called parallel, which is the confusion the whole distinction exists to prevent.

A corrected: A parallel database runs several processors over data held at one site, while a distributed database holds different parts of the data at different sites and presents them to the user as one database.

What is wrong with B: B describes a cut that cannot be undone. If bale_no sits in the weights fragment and not in the grades fragment, the two have no column in common, so there is nothing to join on and the only way to put them side by side is to pair every weight with every grade.

The clause B has dropped: The attributes two fragments share must be a superkey of at least one of them, which is the lossless-join test of the earlier sections in its distributed form, and the primary key is the one set of columns always available to do that work.

B corrected: In a vertical fragmentation every fragment carries the primary key, and the remaining columns are divided between the fragments so that each of them appears in exactly one.

The two false statements fail differently, one by swapping a pair of definitions and one by dropping a clause, and the dropped clause is what decides whether the pieces are still a database.

The Master Key: *Parallel is many processors at one site while distributed is many sites, and a vertical fragmentation repeats the primary key in every fragment on purpose.*

BUILD Question 205

Cutting a table into pieces is easy. Putting the pieces back and getting exactly what was there before is the part that fails, and with five bales weighed at Bugene and Rubya the failure can be counted rather than argued about.

Build the weighbridge table, fragment it three ways, and count what each rejoin gives back.

bale holds bale_no as its primary key, with the site, the weight in kilogrammes and the grade.

Fragment it horizontally into the rows of Bugene and the rows of Rubya, rejoin with UNION ALL and count.

Fragment it vertically into a weights fragment and a grades fragment, both carrying bale_no, rejoin on bale_no and count.

Fragment it vertically once more, this time leaving bale_no out of the second piece, and count what a rejoin recovers.

Model Answer

The three counts below ask one question three times: the horizontal split gives back the five rows stored, the vertical split carrying bale_no in both pieces gives back the same five, and the vertical split that drops bale_no gives back fifteen.

```
 1  CREATE TABLE bale (
 2    bale_no CHAR(6)      NOT NULL,
 3    site    VARCHAR(8)   NOT NULL,
 4    kg      DECIMAL(6,2) NOT NULL,
 5    grade   CHAR(1)      NOT NULL,
 6    PRIMARY KEY (bale_no)
 7  ) ENGINE = InnoDB;
 8
 9  INSERT INTO bale (bale_no, site, kg, grade) VALUES
10    ('B0101', 'Bugene', 118.50, 'A'),
11    ('B0102', 'Bugene', 121.00, 'B'),
12    ('B0103', 'Bugene', 117.25, 'A'),
13    ('R0201', 'Rubya',  119.75, 'B'),
14    ('R0202', 'Rubya',  120.40, 'C');
15
16  CREATE TABLE bugene_bale LIKE bale;
17  CREATE TABLE rubya_bale  LIKE bale;
18
19  INSERT INTO bugene_bale SELECT * FROM bale WHERE site = 'Bugene';
20  INSERT INTO rubya_bale  SELECT * FROM bale WHERE site = 'Rubya';
21
22  SELECT * FROM bale;
23
24  SELECT COUNT(*) AS rows_recovered
25  FROM (SELECT * FROM bugene_bale
26        UNION ALL
27        SELECT * FROM rubya_bale) AS rejoined;
```

The server replies

```
+---------+--------+--------+-------+
| bale_no | site   | kg     | grade |
+---------+--------+--------+-------+
| B0101   | Bugene | 118.50 | A     |
| B0102   | Bugene | 121.00 | B     |
| B0103   | Bugene | 117.25 | A     |
| R0201   | Rubya  | 119.75 | B     |
| R0202   | Rubya  | 120.40 | C     |
+---------+--------+--------+-------+
+----------------+
| rows_recovered |
+----------------+
|              5 |
+----------------+
```

Program 4.89: *A horizontal fragmentation by site, rejoined with UNION ALL.*

```
 1  CREATE TABLE bale_weight (
 2    bale_no CHAR(6)      NOT NULL,
 3    site    VARCHAR(8)   NOT NULL,
 4    kg      DECIMAL(6,2) NOT NULL,
 5    PRIMARY KEY (bale_no)
 6  ) ENGINE = InnoDB;
 7
 8  CREATE TABLE bale_grade (
 9    bale_no CHAR(6) NOT NULL,
10    grade   CHAR(1) NOT NULL,
11    PRIMARY KEY (bale_no)
12  ) ENGINE = InnoDB;
13
14  INSERT INTO bale_weight SELECT bale_no, site, kg FROM bale;
15  INSERT INTO bale_grade  SELECT bale_no, grade    FROM bale;
16
17  SELECT COUNT(*) AS rows_recovered
18  FROM bale_weight w
19  JOIN bale_grade g ON w.bale_no = g.bale_no;
20
21  CREATE TABLE grade_only (
22    grade CHAR(1) NOT NULL,
23    PRIMARY KEY (grade)
24  ) ENGINE = InnoDB;
25
26  INSERT INTO grade_only SELECT DISTINCT grade FROM bale;
27  SELECT COUNT(*) AS rows_recovered
28  FROM bale_weight w
29  CROSS JOIN grade_only g;
```

The server replies

```
+----------------+
| rows_recovered |
+----------------+
|              5 |
+----------------+
+----------------+
| rows_recovered |
+----------------+
|             15 |
+----------------+
```

Program 4.90: *Two vertical fragmentations, one carrying bale_no in both pieces and one not.*

Where these tables would really stand: bugene_bale would stand at Bugene and rubya_bale at Rubya, and the union is what a distributed system does for the user under fragmentation transparency. One server can show the algebra, never the network.

Reading the first two counts: Five rows went in, five come back from the union and five from the join on bale_no. Nothing was lost or invented, and site travels with the weight because a vertical fragmentation must place every column.

Why the third count is fifteen: grade_only is the grade column alone, which as a projection holds the three distinct grades. It shares no column with the weights, so the only pairing left is every row with every row: five weights against three grades makes fifteen.

The one decision worth naming: The horizontal fragments are declared with CREATE TABLE LIKE, which copies the columns and the primary key of the table being fragmented, so a fragment cannot drift from its parent.

A fragmentation is only one if the pieces rebuild what they came from, which is a claim to count rather than assume.

The Master Key: *Count what a rejoin returns: the original number means the fragmentation is sound, and more than that means the key was left out of a piece.*

HOT Question 206

The microwave link between Bugene and Rubya drops most afternoons in the rains. A lorry is standing on the plate at Bugene and the operator is waiting for the machine. It can answer him now or it can be sure, and not both.

Judge what each weighbridge should do while the link between them is down, and state the condition that decides it.

Model Answer

Decide by asking whether an answer given now and later found wrong can be put right from what was written down: where it can, let both sites keep working on their own rows and reconcile when the link returns, and where it cannot, let one of them refuse until it can be sure.

What the CAP result actually says: It is a statement about what happens during a partition, and only then. When the network splits a system into groups that cannot reach each other, each group must either answer from data it cannot confirm is current, or refuse to answer until it can. Consistency and availability are what it chooses between, at that moment and at no other.

What it does not say: It is not a menu of three properties from which two are picked on a calm day. With the link up a system can be consistent and available at the same time, and usually is. Partition tolerance is not picked either, since a network that can drop messages is a fact about the estate rather than a design choice.

The case for answering anyway: The weighbridge cannot stop. A weight not recorded now is lost for good, because the lorry moves off and the reading cannot be taken again. Each site records what it sees and the two records are merged when the link comes back up.

The case for refusing: If both sites can issue bale numbers while cut off, two bales can leave the estate carrying the same number, and once the link returns nobody can say which weight belongs to which bale. A refusal costs a delay a person can work around, and a collision costs a fact nobody can recover.

The line between the two: Ask whether the damage is lateness or wrongness. A total at Bugene that does not yet include the afternoon at Rubya is merely late, and it corrects itself. Two rows claiming one key are wrong, and no message arriving afterwards repairs them.

The judgement: Let both weighbridges stay available and record locally, and take away the reason to refuse by making the key carry the site, so Bugene issues B numbers and Rubya issues R numbers and the two can never collide. The condition is whether the sites can be stopped by design from making decisions that contradict each other: where they can, choose availability, and where they cannot, refuse and stay consistent.

Availability and consistency are not qualities a system simply has, they are the two answers open to it while the network is broken.

The Master Key: *During a partition a system must choose between answering with data it cannot confirm and refusing until it can, and the choice belongs to whichever of the two errors can still be repaired afterwards.*

HOT Question 207

A quotation has arrived at Puge for the software that would keep a second server at Rubya in step with Bugene. The software costs more than the server does. The manager wants to know what the difference is buying him.

Judge whether a two-weighbridge estate should run a distributed database at all, rather than one database at Puge that both weighbridges connect to, and state the condition that decides it.

Model Answer

Distribute only where a site must keep working while cut off from the others, or where the data is too large to be served from one place; this estate meets the first test and not the second, so fragment horizontally by site, replicate only the small tables everybody reads, and keep every transaction inside one site.

What one database at Puge gives: One copy, one commit, one backup and one set of locks. Everything this chapter says about a transaction holds without qualification, there is no protocol between sites to go wrong, and the person who looks after it has one server to understand rather than three.

How the single site fails: When the link to Puge drops, both weighbridges stop, and they stop on the afternoons when the link is worst and the lorries are queuing. A single site does not escape a partition, it loses to it outright, because there is nothing at the hut to answer from.

What distributing buys: Each weighbridge keeps recording while cut off, since its own rows are on its own disk. Reads are local, and a third weighbridge is added by giving it its own fragment rather than by buying a larger machine for the office.

What distributing costs: A transaction that writes at two sites needs two-phase commit, with the blocking window that comes with it. Recovery has to work out which site failed and when, a backup has to be taken across sites at one instant, and there are now three machines where there was one.

The cost that is usually forgotten: The estate has to be able to run the thing. A distributed installation needs somebody who can read two logs at once and reason about a failure that appears only when the network is slow, and that person is scarcer in a district than the hardware.

The test that decides it: Ask two questions. Can a site do useful work while cut off, and does any single transaction have to write at more than one site? The first argues for distributing, the second against, because every such write costs a protocol that can block.

The judgement: Distribute, but shallowly. Give each weighbridge the horizontal fragment of its own bales, copy the small grade and price tables to both, and keep every write inside one site so two-phase commit is never reached. Puge reads the union of the fragments for its reports.

The question is never whether distribution is better, but whether the estate has a problem only distribution solves, and here it has exactly one.

The Master Key: *Distribute when a site must keep working while cut off from the others, and design the fragments so that no transaction ever has to write at two sites at once.*

NoSQL, data warehousing and big data

This last section leaves the relational table for a while. It expands the name NoSQL, which is not a refusal of SQL, and sets the four families side by side by the only thing that truly separates them, the shape of one stored item; it states BASE against ACID and says what a reader may see while the copies still disagree; and it shards a table by a key, including the key that ruins the arrangement. It then builds the other half of the answer, a warehouse with a fact table at its centre and the words held one step away, before closing the chapter on big data and on what data mining is a name for.

BINDER Question 208

A poster at the Igalula club office lists the league software as NoSQL, and the youngest clerk reads it aloud as no SQL. Two of the four stores behind that word answer questions in a language he could almost type unaided.

Say what the name NoSQL is short for, explain what these systems actually are, and correct two things students are commonly told about them.

Model Answer

NoSQL is read as not only SQL: it names stores that hold data in some shape other than a table of rows, and two of the four families have a flagship product whose query language was built on purpose to look like SQL.

What the name is short for: The expansion is not only SQL. The word was coined for stores built to do the one thing relational systems then did badly, spreading one body of data across many ordinary machines. It says something has been added beside SQL, not that SQL was thrown out.

The query language is not the dividing line: Cassandra is queried in CQL, in which SELECT won, lost FROM standing WHERE team_id = 3 is a real statement, and Couchbase is queried in SQL++, which is SQL with documents where rows would be. What such a language gives up is not the syntax but the freedom to join anything to anything cheaply.

What actually changes: One stored item stops being a row. It becomes a key with a value, a document, a wide row of columns, or a node with edges, and everything else these systems do differently follows from that single decision.

Schema-less is only half true: A document store lets two documents in one collection carry different fields, so the word fits it. A wide-column store such as Cassandra insists that columns and their types be declared before any row is written. Students are often taught, wrongly, that being schema-less defines the whole family.

Transactions are not the dividing line either: Several of these stores have carried ACID transactions for years. The honest contrast is about what is cheap and what is usual, never about what is possible.

Read the name as not only SQL and the rest of this section follows.

The Master Key: *NoSQL means not only SQL: stores whose stored item is not a row, some of them queried in a language deliberately shaped like SQL.*

BINDER Question 209

Four stores hold one fact about Itobo, that it has won nine matches. One keeps it under a key, one inside a document, one along a wide row, one as an arrow drawn between two clubs. The fact never changes. Its container does.

Name the four families of NoSQL store, show what one stored item looks like in each, and give a query that fetches it.

Model Answer

The four families are key-value, document, column family, also called wide-column, and graph, and what separates them is the shape of one stored item: a key with a value the store never opens, a document of named fields, a row of columns gathered into families, or a node joined to others by stored edges.

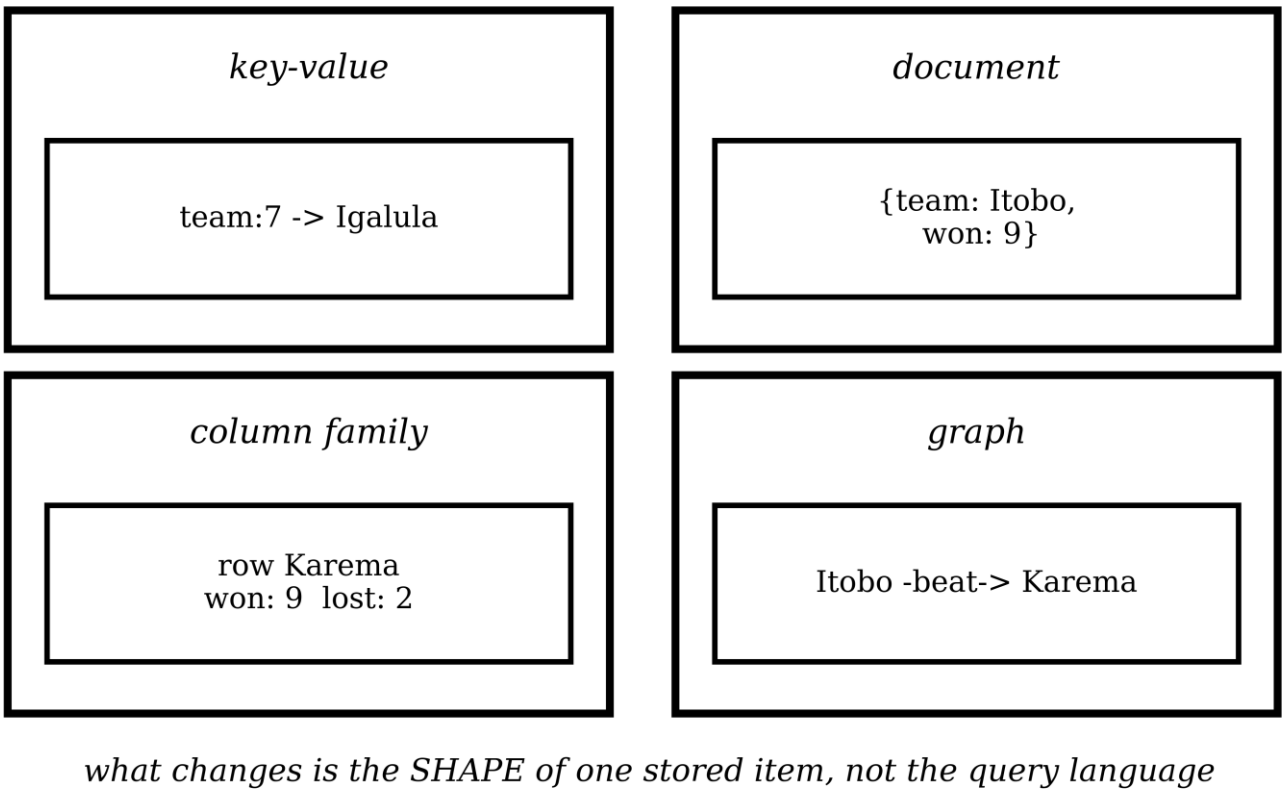


Figure 4.42: *One item from the league table, stored four ways.*

Key-value: The store keeps a key and a value it treats as a lump, so the only question it answers is the one naming the key. Asking for team:7 returns Igalula at once; asking which clubs have won more than eight matches cannot be asked of this family at all.

Document: The fields inside the document are visible to the query, so the store can be asked for every document whose won field is nine. Two documents in one collection need not carry the same fields, which is what schema-less properly means.

Column family: A row is found by its row key and its columns are gathered into families, declared before any row is written. The query language is CQL, and the line in the table below is a CQL statement a reader of this chapter can already read, which is the point of the name.

Graph: The edge is stored as a thing in its own right, alongside the nodes it joins, so the question of who Itobo beat is answered by walking one edge rather than by pairing a table with itself. The gain grows with the number of steps the question takes.

Family	One stored item	A query that fetches it	A product
Key-value	team:7 -> Igalula	GET team:7	Redis
Document	{team: Itobo, won: 9}	db.standing.find({won: 9})	MongoDB
Column family	row Karema, won 9, lost 2	SELECT won, lost FROM standing WHERE team_id = 3;	Cassandra
Graph	Itobo -beat-> Karema	MATCH (a)-[:BEAT]->(b) RETURN a.name, b.name	Neo4j

Table 4.44: *One stored item and one query, in each of the four families.*

Reading the four panels together: Nothing in the figure is about a language. Each panel shows one item as the store holds it, and the query beside it in the table is the shape that item forces on the question.

Name the family, then say what one stored item is in it, and the use case follows without being memorised.

The Master Key: *Key-value stores a key and a value it never opens, document stores named fields, column family stores wide declared rows, and graph stores the edges themselves.*

BINDER Question 210

Refreshing the league table at Karema gives Itobo nine points; refreshing a second later gives eight. Nothing was deleted. The second answer came from a machine the first update had not yet reached, and both were doing exactly what they promised.

State what BASE stands for, explain each of its three parts, and say what a person reading the league table may actually see under it.

Model Answer

BASE is basically available, soft state, eventually consistent: the store answers what it can while some machines are unreachable, the value one machine holds may change with nobody writing to it, and the copies are promised to agree in the end, with no deadline on the word end.

Basically available: An answer is preferred to a refusal. With machines missing or slow the store still replies, and the reply may be older than the truth, which is why the word basically sits in the name.

Soft state: The value on one machine is not settled merely because nobody is writing. Updates already sent are still working their way across, so the state moves on its own, and what was read a moment ago may already be stale.

Eventually consistent: Stop writing and every copy converges on the same value. Convergence is the promise, timing is not part of it, and the window has no stated limit, so a design that must never be wrong cannot rest on this.

What a reader sees meanwhile: Points that go backwards on a second refresh, two supporters seeing different tables at the same instant, a result just entered missing from the very next read. All three are permitted, and none is a fault anybody could report.

Set against ACID: ACID, set out earlier in this chapter, promises that nobody sees a state breaking the rules. BASE permits exactly that for a while in exchange for answering while the network is broken; this is the trade CAP describes.

A default, not a law: Many of these stores let one query demand that a majority of copies agree first, buying back consistency for that query and paying in waiting. BASE is what the store does when nobody asks for more.

Give the three words, then the three things a reader might see, and the model is explained rather than recited.

The Master Key: *BASE is basically available, soft state and eventually consistent, and eventually is a promise that the copies converge, with no deadline on when.*

BINDER Question 211

Somebody split the season records across three machines by the first letter of the club name. Igalula and Itobo landed together, Karema landed alone, and the third machine was given nothing to hold. The idle machine cost as much as the busy one.

Explain what sharding is, say exactly what the shard key does, and name the failure that a badly chosen shard key produces.

Model Answer

Sharding is horizontal fragmentation used to gain capacity: the rows of one table are divided across machines, whole rows on each, and the value of one chosen column, the shard key, decides which machine a row goes to; choose that column badly and one shard takes nearly all the work while the rest stand idle.

What one shard holds: Every column and some of the rows, which is the row cut of the previous section. What is new is the purpose: fragmentation puts rows near the people who read them, and sharding spreads them because one machine can no longer serve them.

How the key decides: The store computes the shard from the key, either by range, so keys near each other sit together, or by hash, so they are scattered. A query naming the shard key goes to one machine; a query that does not goes to every machine, and the answers are merged.

The hot shard: A key with few values, or one whose newest value takes all of today's writes, loads one machine and leaves the rest idle. Sharding league records by season sends every write of the current season to one shard, and the first letter of a club name gives one machine two clubs and another none.

What makes a key good: Many values, used evenly, and named by the queries that matter. The last pulls against the first, since a key scattered by hash spreads writes beautifully and makes every range query visit every shard.

Sharding is not replication: Sharding divides the data and replication copies it, and real installations do both. Splitting alone gains capacity and no safety at all: lose one shard and the rows on it are simply gone.

Name what decides the shard first, because the shard key is the whole design and the whole failure.

The Master Key: *Sharding divides rows across machines by a shard key, and a key with few values or with all of today's writes in it makes one shard hot while the others idle.*

REAL Question 212

Saturday evening, and the Itobo secretary enters three results one after another, each done before he has put the pen down. On Monday the association asks for the crowd at every match ever played, and the same machine goes quiet.

Trace the Saturday work and the Monday report through the league's data, say how the two differ in the shape of the work, and explain why a warehouse is denormalised from the day it is designed.

The operational database holds one row for each result and one row for each club standing.

On Saturday three results are entered, and each entry writes the result and adjusts two standings.

On Monday the association asks for the total crowd by ground and by month, across the ten seasons held.

Model Answer

The Saturday work is many small writes that must each be right the moment they finish, and the Monday report is one enormous read that touches every row ever stored and prints a handful of lines; that difference in the shape of the work, rather than the words operational and analytical, is why the two are given separate databases and why the warehouse is denormalised from the start.

Tracing the Saturday: One result written is one row inserted and two standings adjusted, so three rows for one match and nine for the evening. Each of the three sits inside its own transaction, each finishes in a moment, and each must be correct immediately, because the next entry reads what the last one wrote.

Tracing the Monday: The report reads six match rows for each of the ten seasons, sixty in all, groups them by two grounds and two months, and prints four lines. It writes nothing, and nobody minds if it takes a minute, because no result is waiting on it.

The shape, stated plainly: Many small writes against few enormous reads. Sixty rows is a small number and the shape shows in it already: the Saturday work touches three rows and must be instant, the Monday work touches everything and returns almost nothing.

What is being compared	The operational database, OLTP	The warehouse, OLAP
One unit of work	Enter one result	Report ten seasons of crowds
Rows touched	Three	Every match row held
How often it runs	Many times an evening	Once a month
What must hold at the end	The standing is right now	The answer matches the last load
How the schema is shaped	Normalised, few repeated words	Denormalised on purpose
What is kept	The current standing, overwritten	Every season ever loaded

Table 4.45: *The two kinds of work, item by item.*

Why they are separated: A report reading every row competes for the same machine as the desk trying to write, and the indexes that make a point lookup fast are not those that help a full scan. The operational database also overwrites a standing as it changes, so the history a report needs is kept only where nothing is overwritten.

Why the warehouse starts denormalised: Normalisation exists to stop one fact being updated in one place and left stale in another. A warehouse is not updated by hand at all; it is loaded in batches from the operational system, so the anomaly it guards against cannot arise, while the cost it charges, a join for every descriptive word, is charged on every report.

When a warehouse is worth building: Build one when the reports have grown heavy enough to disturb the evening's work, or when the operational database has already overwritten the history they need. Below that one well indexed database answers both, and a second copy is a second thing to keep right.

The two databases hold the same league and are shaped by opposite demands, which is why one design cannot serve both once the reports grow.

The Master Key: *OLTP is many small writes that must be right at once and OLAP is few enormous reads, and a warehouse is denormalised on purpose because nothing updates it by hand.*

REAL Question 213

A drawing pinned up in the Karema clubhouse has one box in the middle and four around it, wired like a wheel. Somebody has written the club names into the middle box, and that one decision has undone the whole design.

Set out the league's warehouse as a star schema, say exactly what may and may not appear in the fact table, and say what a snowflake changes and what it costs.

match_fact is to hold one row for each match played.

The measures are the goals scored and the crowd that came through the gate.

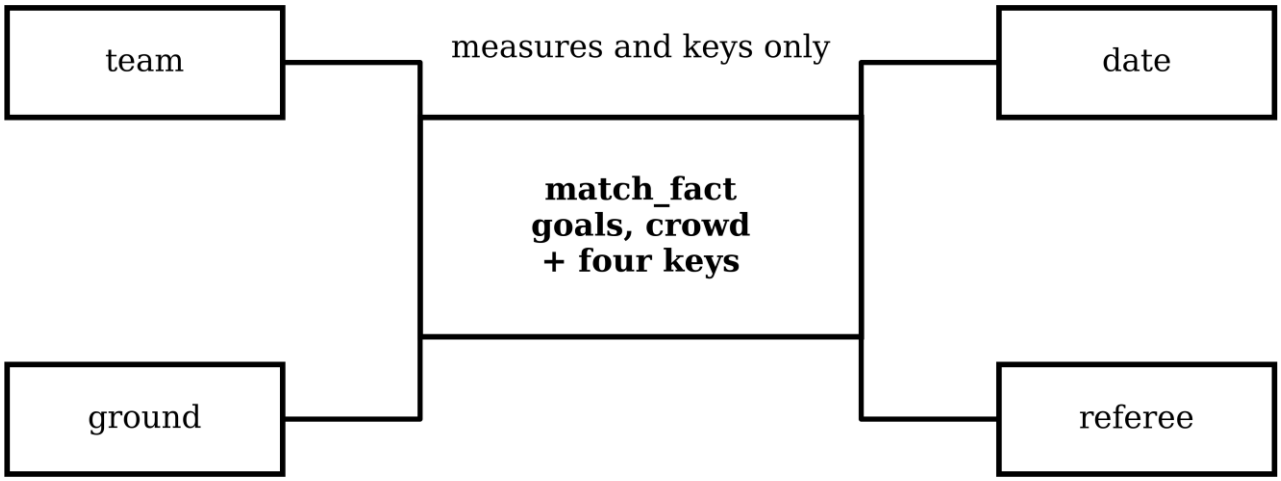
The words a report prints, the club, the month, the ground and the referee, must be reachable and must not sit in the fact table.

Model Answer

A star schema is one fact table at the centre holding the measures and one foreign key for each dimension, surrounded by dimension tables holding every descriptive word, and a snowflake is that same star with one or more dimensions normalised into further tables, which saves a little space and costs an extra join on every query wanting the words.

The fact table: A fact table holds measures and foreign keys, and no descriptions. Here that is goals and crowd, the four keys to team, date, ground and referee, and the match identifier, seven columns of which not one can be read aloud as a word. This is the sentence to write down.

The dimension tables: The words live here, in tables that are wide, flat and small: the club names in team, the calendar spelled out in date, the ground names in ground, the officials in referee. A dimension repeats itself without apology, since May is written against every match played in May.



every word a reader sees lives in a dimension

Figure 4.43: *The league warehouse as a star, with every word one step out from the centre.*

The grain, decided first: One row of the fact table stands for one match, and that is the grain. It is settled before any column is chosen because every measure has to be true at it: a crowd figure belongs to a match, so recording it against a club and a season instead would count the same people twice.

What the snowflake changes: Normalise the ground dimension into ground and town and the schema snowflakes: the town name is then stored once rather than once for each ground in it. A report wanting the town now joins the fact to ground and ground to town, two joins where the star needed one.

Which one to draw: Draw the star unless asked otherwise. It is what the phrase asks for, it shows the fact and dimension split in one picture, and the space a snowflake saves is nothing beside a fact table holding a row for every match of every season.

The error in the drawing: Club names written into the middle box put a word in the fact table, so the same name is stored once for every match a club plays, and correcting a spelling becomes an edit of the largest table in the warehouse instead of one row in team.

The star and the snowflake differ in one dimension being split, and the fact table is the same in both, which is where the rule about descriptions lives.

The Master Key: *A fact table holds measures and foreign keys and no descriptions, a star keeps each dimension in one table, and a snowflake normalises a dimension and pays a join for it.*

BUILD Question 214

Seven columns, and not one of them a word. The middle table of the league's warehouse looks unreadable on purpose, and everything a person could read aloud has been moved one step outwards, into the four small tables around it.

Build the warehouse: the four dimension tables, then the fact table with its four foreign keys and two measures, and one report joining the fact to two of the dimensions.

team, date, ground and referee each hold an integer key of their own and the words that go with it.

match_fact holds one row for each match: the four keys, the goals scored by the home club, and the crowd.

The league is three clubs and six matches in the season being loaded, and the ground at Karema is being repaired, so its home matches are played at Itobo.

The report is to print goals and crowd by club and by ground.

Model Answer

The two listings build the star the figure draws: four dimensions carrying every word, then match_fact carrying six rows of keys and measures, and the report at the end turns those keys back into words by joining two of the dimensions to the fact.

```
 1  CREATE TABLE team (
 2    team_id   INT         NOT NULL,
 3    team_name VARCHAR(12) NOT NULL,
 4    PRIMARY KEY (team_id)
 5  ) ENGINE = InnoDB;
 6  CREATE TABLE date (
 7    date_id    INT        NOT NULL,
 8    played_on  DATE       NOT NULL,
 9    month_name VARCHAR(9) NOT NULL,
10    PRIMARY KEY (date_id)
11  ) ENGINE = InnoDB;
12  CREATE TABLE ground (
13    ground_id   INT         NOT NULL,
14    ground_name VARCHAR(12) NOT NULL,
15    PRIMARY KEY (ground_id)
16  ) ENGINE = InnoDB;
17  CREATE TABLE referee (
18    referee_id   INT         NOT NULL,
19    referee_name VARCHAR(12) NOT NULL,
20    PRIMARY KEY (referee_id)
21  ) ENGINE = InnoDB;
22  INSERT INTO team VALUES (1, 'Igalula'), (2, 'Itobo'), (3, 'Karema');
23  INSERT INTO ground VALUES (1, 'Igalula'), (2, 'Itobo');
24  INSERT INTO referee VALUES (1, 'Neema'), (2, 'Juma'), (3, 'Salum');
25  INSERT INTO date VALUES
26    (1, '2025-05-03', 'May'),  (2, '2025-05-10', 'May'),
27    (3, '2025-05-17', 'May'),  (4, '2025-05-24', 'May'),
28    (5, '2025-06-07', 'June'), (6, '2025-06-14', 'June');
29  SELECT * FROM date;
```

The server replies

```
+---------+------------+------------+
| date_id | played_on  | month_name |
+---------+------------+------------+
|       1 | 2025-05-03 | May        |
|       2 | 2025-05-10 | May        |
|       3 | 2025-05-17 | May        |
|       4 | 2025-05-24 | May        |
|       5 | 2025-06-07 | June       |
|       6 | 2025-06-14 | June       |
+---------+------------+------------+
```

Program 4.91: *The four dimensions, and the calendar written out in words.*

```
 1  CREATE TABLE match_fact (
 2    match_id   INT NOT NULL,
 3    team_id    INT NOT NULL,
 4    date_id    INT NOT NULL,
 5    ground_id  INT NOT NULL,
 6    referee_id INT NOT NULL,
 7    goals      INT NOT NULL,
 8    crowd      INT NOT NULL,
 9    PRIMARY KEY (match_id),
10    FOREIGN KEY (team_id)    REFERENCES team (team_id),
11    FOREIGN KEY (date_id)    REFERENCES date (date_id),
12    FOREIGN KEY (ground_id)  REFERENCES ground (ground_id),
13    FOREIGN KEY (referee_id) REFERENCES referee (referee_id)
14  ) ENGINE = InnoDB;
15
16  INSERT INTO match_fact VALUES
17    (1, 1, 1, 1, 1, 2, 1200), (2, 2, 2, 2, 2, 1,  900),
18    (3, 3, 3, 2, 3, 3,  800), (4, 1, 4, 1, 2, 0, 1400),
19    (5, 2, 5, 2, 1, 2, 1000), (6, 3, 6, 2, 3, 1,  700);
20
21  SELECT * FROM match_fact;
22
23  SELECT t.team_name, g.ground_name,
24         SUM(f.goals) AS goals, SUM(f.crowd) AS crowd
25  FROM match_fact f
26  JOIN team t   ON f.team_id   = t.team_id
27  JOIN ground g ON f.ground_id = g.ground_id
28  GROUP BY t.team_name, g.ground_name
29  ORDER BY t.team_name;
```

The server replies

```
+----------+---------+---------+-----------+------------+-------+-------+
| match_id | team_id | date_id | ground_id | referee_id | goals | crowd |
+----------+---------+---------+-----------+------------+-------+-------+
|        1 |       1 |       1 |         1 |          1 |     2 |  1200 |
|        2 |       2 |       2 |         2 |          2 |     1 |   900 |
|        3 |       3 |       3 |         2 |          3 |     3 |   800 |
|        4 |       1 |       4 |         1 |          2 |     0 |  1400 |
|        5 |       2 |       5 |         2 |          1 |     2 |  1000 |
|        6 |       3 |       6 |         2 |          3 |     1 |   700 |
+----------+---------+---------+-----------+------------+-------+-------+
+-----------+-------------+-------+-------+
| team_name | ground_name | goals | crowd |
+-----------+-------------+-------+-------+
| Igalula   | Igalula     |     2 |  2600 |
| Itobo     | Itobo       |     3 |  1900 |
| Karema    | Itobo       |     4 |  1500 |
+-----------+-------------+-------+-------+
```

Program 4.92: *The fact table, and the report that turns its keys back into words.*

What the specification asked for: Two measures, four foreign keys and no descriptive column anywhere in the middle. Seven integer columns satisfy it, and the plain SELECT proves it: six rows in which there is nothing a reader could read aloud, which is exactly what a fact table should look like.

Reading the report: Three lines come back, one for each club, totalling nine goals and six thousand through the gates. The line for Karema names the ground at Itobo, because that is where its home matches were played, and the two joined dimensions are the only reason any word appears at all.

The one decision worth naming: The date is a dimension table rather than a column, and it spells the calendar out, so a report groups by month without computing anything from a date. Storing the month as a word is redundant, and in a warehouse that is the point.

The report is the whole design in one statement: measures from the middle, words from the edge, joined on keys.

The Master Key: *Build the dimensions first, give the fact table nothing but measures and foreign keys, and let every word a report prints come from a join.*

HOT Question 215

A company at Itobo offers to move the league off its tables and into documents, one for each club, with no fixed columns to alter when somebody wants a new field. The offer is real. So is the join it takes away.

Judge whether the league should replace its relational tables with a document store, and state the condition that decides it.

Model Answer

Keep the tables. A document store earns its place where items genuinely differ in shape and are read whole by their own key, and where one machine can no longer take the writes; a league of three clubs meets neither condition, and the questions it asks are aggregates across every club at once, which is the work a relational engine does best.

What is really on offer: One document for each club, holding everything about it, fetched in a single read with no join, and fields added to one document without altering anything else. Those are real gains, worth saying plainly before arguing against them.

Where the offer is strongest: Items that truly differ, one club with a women's side and three sponsors and another with neither, and a reading pattern that fetches one whole item by key. A page showing one club is then one read where the tables need several.

What the league actually asks: Order every club by points. Total the crowd by month. Find the clubs that have never lost at home. Not one fetches a single item by key; each reads across everything and computes, which is the work SQL was designed for and the work a document store leaves to the program.

The correction that matters: It is not that a document store cannot join or hold a transaction, because several do both, so arguing the case on capability loses it. The real difference is what is cheap, what is default, and how much work the store does rather than the program.

The cost nobody quotes: A club name inside every document mentioning that club is a repeated fact, so correcting a spelling becomes many writes rather than one. That is the update anomaly of the earlier sections, bought back deliberately, and bearable only where data is written once and read often.

The condition that decides it: Ask three questions. Do the items differ in shape? Is the common read a fetch of one whole item by key? Has one machine stopped coping with the writes? Two yes answers make the move worth costing; one makes it a preference.

The judgement: Refuse the offer for the database itself and revisit it for one job only. If the standings page is ever read by thousands at once, put a key-value store in front of it holding the finished table under one key, refreshed after each match, which buys the speed without moving the records.

The choice is not between old and modern but between a store that computes across items and one that hands back a single item quickly.

The Master Key: *Move to a document store when items differ in shape, are read whole by key, and one machine can no longer cope, never because the tables feel dated.*

HOT Question 216

The league secretary has been told that the club records are big data and belong on a cluster. Ten seasons of results fit on one screen with room to spare. Somewhere between those two sentences sits the question of what the word means.

Judge whether the league's records are big data, define the term properly, and say what data mining could and could not honestly find in them.

Model Answer

They are not big data, and calling them so buys the league nothing: big data names the point at which the ordinary arrangement, one database on one machine, can no longer hold what arrives or process it in the time available, and sixty rows are nowhere near that line.

The three Vs, and the two usually added: Volume, velocity and variety are the three: how much there is, how fast it arrives and how fast an answer is wanted, and how many shapes it comes in. Veracity, how far it can be trusted, and value, whether anything useful can be got from it, are the two commonly added.

The V	What it names	The league measured against it
Volume	How much there is to hold	Sixty rows in ten seasons
Velocity	How fast it arrives, how fast an answer is wanted	Six results in a season
Variety	How many shapes it arrives in	One shape, a result
Veracity	How far it can be trusted	One secretary, one book
Value	Whether anything useful can be got from it	A crowd figure for each match

Table 4.46: *The five Vs, and the league set against each of them.*

Why the Vs are a description and not a threshold: No number of gigabytes turns data into big data. The sign is that the ordinary arrangement stops working: one machine cannot hold it, or one pass cannot finish in the time the answer is wanted, so the tools change to many machines working on parts of it at once.

Where the district would meet it: A turnstile counting every entry at every ground in the region, second by second, with photographs and messages about the matches alongside, would arrive faster than one machine could absorb and in several shapes at once. That is a change of kind, and it is not what the league has.

Data mining is a task, not a tool: Mining is the search for patterns nobody asked for, and it is done with several kinds of task: sorting items into known classes, grouping items that behave alike, finding things that occur together, and picking out the one item too far from the rest. A query answers a question already held; mining proposes one.

What it could honestly find here: Very little. An apparent link between an afternoon start and a larger crowd, drawn from sixty matches, could easily be chance, and even a real link is not a cause, since school holidays could produce both. A pattern from few rows is a suggestion to look further, never a finding.

The judgement: Do not buy a cluster. Keep loading the warehouse season after season, because history nobody kept is the one loss that cannot be repaired later, and put the question again when the data starts arriving faster than one machine can take it. Mine when there is enough of it that a pattern could not easily have arisen by accident.

The word big is about what the ordinary arrangement can still do, so the honest first move is always to count what there is.

The Master Key: *Big data begins where one machine and one database stop coping, measured by volume, velocity and variety, with veracity and value often added, and mining is the search for patterns nobody asked for.*

INDEX

A

Abstract class, 96-109, 173-184. *see also* Abstraction; Interface
  compared with an interface, 96-109
  in Python, the ABC module, 179, 182
Abstraction, 70, 71, 96-109, 173-184. *see also* Abstract class
  compared with encapsulation, 173-175
Access control (database), 498-508. *see also* SQL injection
  GRANT and REVOKE, 498, 503, 504
  least privilege, 498
Access modifiers, 45, 70-80. *see also* Encapsulation
  public, private and protected, 73-75
  package-private (default), 73-75
ACID properties, 457-468. *see also* Transaction
  atomicity, 459-461, 466
  consistency, 459-462
  isolation, 459-461, 466
  durability, 459-462
Ageing, 259-268. *see also* Starvation
Aggregate functions, 425-433. *see also* GROUP BY
  COUNT, 426-430
  SUM and AVG, 426, 430-432
  MIN and MAX, 426, 432, 433
  and NULL, 425-428
ALTER TABLE, 397-406, 467. *see also* CREATE TABLE
Anomalies, 361-372. *see also* Normalisation
  insertion, deletion and modification, 370
Argument, *see* Parameter
Armstrong's axioms, 361-370. *see also* Closure of an attribute set
Array (Java), 29-44, 105, 106. *see also* List (Python)
  declaration and length, 34-36, 40
  two dimensional, 34, 35
  enhanced for loop, 34-37, 40
Associative entity, 353-360. *see also* Cardinality
Attribute (entity-relationship), 329, 330, 339-345. *see also* Entity
  simple, composite and derived, 340, 343
  multivalued, 340-343
  key attribute, 340
Autocommit, 457-468, 477. *see also* COMMIT and ROLLBACK
Average waiting time, 250-268. *see also* CPU scheduling

B

Backup and recovery, 315, 318, 457-468. *see also* Transaction
Banker's algorithm, 276-284. *see also* Deadlock
  safe sequence, 280-282
  need matrix, 280, 281
Batch operating system, 217-224
Best fit, 290-293. *see also* First fit; Worst fit
Big data, 529-538. *see also* Data warehousing
Bitwise operators, 19-28. *see also* Operators
Boyce-Codd normal form, 380-388. *see also* Normalisation
Burst time, 250-268. *see also* CPU scheduling
Bytecode, 1-7, 62, 75. *see also* JVM (Java Virtual Machine)

C

Candidate key, 326-332, 363, 366, 385. *see also* Primary key
Cardinality, 346-354. *see also* Participation
  one-to-one, one-to-many, many-to-many, 346, 349-351
  resolving many-to-many, 353, 354, 360
Check constraint, 397-406. *see also* Constraint
Class, 56-69, 161-172. *see also* Object
  fields and methods, 59, 63, 64
  in Java, 56-69
  in Python, 161-172
  class against instance variable, 163, 164, 170
Class file, *see* Bytecode
Closure of an attribute set, 361-370. *see also* Armstrong's axioms
Collections (Python), 141-150. *see also* Array (Java)
  list, 142, 147-149
  tuple, 141, 142
  dictionary, 142, 143, 150
  set, 141, 142, 149
  comprehension, 141-150
Comment, 8, 130
COMMIT and ROLLBACK, 457-468, 479, 480. *see also* Transaction
  savepoint, 464-466
Composite key, 326-332, 358-360, 373. *see also* Primary key
Concurrency control, 315, 469-482. *see also* Locking
  lost update, 469, 470, 481
  dirty read, 470-472
  non-repeatable read and phantom, 469-472
Console input and output, 8-18, 122-130, 136-139. *see also* Scanner class
  System.out.println in Java, 12, 16-18
  print and input in Python, 124-126, 130
  formatted output, 91, 92, 138-140
Constraint, 397-406. *see also* Integrity rules
  NOT NULL, 400, 401, 404
  UNIQUE, 401, 404
  DEFAULT, 397, 401
  named and dropped, 401-404
Constructor, 56-69, 89, 90, 161-172. *see also* Class
  in Java, and overloading, 56, 60
  in Python, __init__, 167-170
  chaining with this and super, 61
Context switch, 232, 242-249, 266. *see also* Process
Control flow, 29-44, 56, 141-150. *see also* Loop
Convoy effect, 255, 256. *see also* CPU scheduling
CPU scheduling, 250-268. *see also* Gantt chart
  first come first served, 250, 253, 254
  shortest job first, 250, 258, 259
  shortest remaining time, 259-262
  round robin and the quantum, 259, 267, 268
  priority, and starvation, 261, 265, 266
  preemptive and non-preemptive, 259-262
CREATE TABLE, 397-406, 467, 468. *see also* Data types (SQL)
Critical section, 269-275, 283. *see also* Race condition

D

Data types (Java), 8-18, 22, 23
  primitive and reference, 9-11, 18
  casting and widening, 10, 11
Data types (SQL), 335-337, 397-406
  CHAR against VARCHAR, 398
  DECIMAL for money, 398-400
  DATE and DATETIME, 398-400
Data warehousing, 529-538. *see also* NoSQL
  OLTP against OLAP, 533

Database design, 333-339. *see also* Entity-relationship diagram
  conceptual, logical and physical, 333-336
Database management system, 313-320. *see also* Database models
  what it adds over files, 313, 317, 318
  three level architecture, 316
Database models, 319-326. *see also* Relational model
  hierarchical and network, 320, 321, 324
  relational, 323-325
Deadlock, 276-285. *see also* Banker's algorithm
  the four conditions, 276, 277, 280
  resource allocation graph, 276, 277
  prevention, avoidance and detection, 278, 279, 284
Debugging, 1, 110-121, 197-210. *see also* Exception
Decomposition, 377, 378, 388-396. *see also* Normalisation
  lossless join, 388-391
  dependency preservation, 388, 392
DELETE, 405-415. *see also* INSERT
  compared with TRUNCATE and DROP, 407, 408, 414
Demand paging, 294-302. *see also* Virtual memory
Denormalisation, 388-396, 532, 533. *see also* Normalisation
Device management, 212, 213, 303-312
  device driver, 213
  buffering, 303-312
Direct memory access, 225-234. *see also* Interrupt
  compared with programmed and interrupt-driven I/O, 230, 231
Disk scheduling, 303-312
  FCFS and SSTF, 307-310
  SCAN, C-SCAN and LOOK, 308-310
Dispatcher, 244, 247, 250-268. *see also* Schedulers, the three
Distributed database, 520-528, 532. *see also* NoSQL
  fragmentation and replication, 521-523
  two-phase commit, 523, 524
  the CAP theorem, 528
Distribution transparency, 520-528. *see also* Distributed database
  location, fragmentation and replication, 522, 523
DROP, 397-408, 496. *see also* DELETE

E

Encapsulation, 70-81, 173-184. *see also* Access modifiers
  in Java, private fields and accessors, 77-80
  in Python, the single and double underscore, 173-175
  compared with abstraction, 71, 173-175
Entity, 339-348. *see also* Attribute (entity-relationship)
  strong and weak, 346-352
  entity type against instance, 339-341, 344
Entity-relationship diagram, 339-345. *see also* Cardinality
  mapping to tables, 353-360
  notation, 339-341
Exception, 110-121, 197-210. *see also* Debugging
  try, catch and finally, 111-113, 119, 120
  try, except and finally in Python, 197-199
  checked and unchecked, 110, 119, 120
  throw and throws, 113
  raise, 199, 205
  custom exception classes, 203
EXPLAIN, 483-497. *see also* Query optimisation

F

File system, 237, 303-312
  directory structure, 303-305
  allocation methods, 305-307
  journalling, 312
final, 26, 70-80, 95, 96. *see also* Access modifiers
First fit, 289-291. *see also* Best fit; Worst fit
First normal form, 371-380. *see also* Normalisation
  repeating groups, 371-374
Foreign key, 326-332, 354-357. *see also* Primary key
  referential integrity, 328, 329, 332
  ON DELETE and ON UPDATE, 405, 406
Fourth normal form, 367, 370, 380-387. *see also* Normalisation
Fragmentation (memory), 285-295. *see also* Paging
  internal, 288, 292
  external, and compaction, 288, 292, 293
Function (Python), 151-162. *see also* Method
  def and return, 152, 159, 160
  docstring, 156-159
  default and keyword arguments, 154, 155
Functional dependency, 361-370, 388. *see also* Normalisation
  full, partial and transitive, 363
  prime and non-prime attributes, 363

G

Gantt chart, 250-268. *see also* CPU scheduling
Garbage collection, 4, 56-69. *see also* Object
GROUP BY, 416, 425-433. *see also* Aggregate functions
  HAVING against WHERE, 425-427

H

HAVING, 416, 425-427. *see also* GROUP BY; WHERE

I

If statement, 29-44, 141-150. *see also* Switch statement
  nested, and else-if, 44, 147, 148
Indentation, in Python, 122-130
Index (database), 483-497. *see also* Query optimisation
  B-tree, 483-497
  when an index does not help, 483, 484, 497
Inheritance, 81-95, 185-196. *see also* Polymorphism
  in Java, extends and super, 81-95
  in Python, and multiple inheritance, 187, 196
  single, multilevel and hierarchical, 93, 94
  is-a against has-a, 94, 95, 196
INSERT, 407-414, 453-455, 498, 503. *see also* UPDATE
  multiple rows, 410, 411, 414
Integrity rules, 315, 326-332. *see also* Constraint
  entity integrity, 328, 329
  referential integrity, 328, 329, 332
  domain integrity, 328
Interface, 96-110. *see also* Abstract class
  implements, 97, 98
  why Java has no multiple inheritance of classes, 109
Interpreter, 122-130, 158. *see also* JVM (Java Virtual Machine)
  compared with a compiler, 122, 128-130
Interrupt, 225-233. *see also* System call
Isolation levels, 469-482. *see also* Concurrency control
  read uncommitted and read committed, 470, 476-478
  repeatable read, 470, 478-481
  serialisable, 470, 481, 482

J

Java Development Kit (JDK), 1-7. *see also* JVM (Java Virtual Machine)
Java Runtime Environment (JRE), 1-7. *see also* JVM (Java Virtual Machine)
Join, 434-443. *see also* Subquery
  inner, 435-438
  left and right outer, 435, 441, 442
  full outer, and its absence in MySQL, 442, 443

self join, 435, 436, 443
cross join, 435, 438, 439
Just-in-time compilation, 1-7. *see also* JVM (Java Virtual Machine)
JVM (Java Virtual Machine), 1-7, 36, 103. *see also* Bytecode
write once, run anywhere, 4

K

Kernel, 225-233, 238, 239. *see also* Shell
kernel mode and user mode, 225, 226, 232, 233
monolithic, microkernel and hybrid, 237-240
Key, *see* Primary key; Foreign key; Candidate key; Composite key; Superkey

L

List (Python), 137, 141-151, 155. *see also* Array (Java); Collections (Python)
Locking, 469-482, 523. *see also* Concurrency control
shared and exclusive, 482
deadlock in a database, 472, 473
row and table level, 469-482
Loop, 29-44, 141-150
while and do-while, 32-34
for, and the enhanced for, 34-37, 40
for over a collection in Python, 141, 148
break and continue, 31, 32, 146

M

Magic method (dunder), 161-172, 195
Memory management, 212, 285-293. *see also* Fragmentation (memory)
contiguous allocation and partitions, 286-288
first fit, best fit and worst fit, 290-293
compaction, 292, 293
Method, 45-55. *see also* Parameter
declaration and return type, 45-47
overloading, 46, 47, 55
static against instance, 45, 55
the main method, 8-18
Method overriding, 81-95, 99, 185-196. *see also* Polymorphism
compared with overloading, 190, 195, 196
the @Override annotation, 85-87
Method resolution order, 185-196. *see also* Inheritance
Module (Python), 151-161
import and from-import, 155, 156
__name__ and __main__, 155-157
Multiprogramming and multitasking, 217-225. *see also* Operating system
Multivalued dependency, 367, 370, 380-387. *see also* Fourth normal form
Mutual exclusion, 269-276, 280. *see also* Semaphore

N

Normalisation, 371-387. *see also* Functional dependency
unnormalised form, 371-376
first normal form, 372-374, 380
second normal form, 371-373, 376
third normal form, 380, 381, 384, 385
Boyce-Codd normal form, 380-386
fourth normal form, 380-383
NoSQL, 529-538. *see also* Data warehousing
document, key-value, column and graph, 531, 537
compared with relational, 529, 537
BASE against ACID, 529-532
NULL, 415-430. *see also* Aggregate functions; SELECT
IS NULL and IS NOT NULL, 416, 417, 420
in aggregates, 426-430
three-valued logic, 416-418

O

Object, 56-69, 131-134, 161-172. *see also* Class
creating with new, 56-58
reference against value, 57-59, 68
Operating system, 211-220. *see also* Kernel
what it is not, 213, 214
functions of, 212, 213, 216
types of, 218, 219, 223, 224
Operating system architecture, 234-241. *see also* Kernel
monolithic, 234, 237-240
layered, 235-238
microkernel, 238-240
hybrid, 238-241
Operators, 19-28, 131-140
arithmetic, and integer division, 19, 20, 27
relational and logical, 21-25
increment and decrement, 19-21
precedence and associativity, 25, 135
the conditional operator, 22, 27
identity against equality in Python, 132, 133
ORDER BY, 415-426, 510-512. *see also* SELECT
ASC and DESC, 423, 424
Overloading, 45-56, 60, 99. *see also* Method overriding

P

Package (Java), 70-80. *see also* Access modifiers
Page replacement, 258, 294-302. *see also* Paging
FIFO, LRU and optimal, 298-301
Belady's anomaly, 300, 301
Paging, 294-303. *see also* Virtual memory
page table and frames, 294-297
page fault, 296-299
compared with segmentation, 294-296, 302
Parameter, 45-55, 85-87, 151-160. *see also* Method
formal and actual, 45
pass by value, 45-55
default and keyword arguments in Python, 154, 155
Participation, 346-352. *see also* Cardinality
Password storage, 498-509. *see also* Access control (database)
salting, 507, 508
compared with encryption, 498-508
PDO (PHP Data Objects), 509-519. *see also* SQL injection
the DSN and the connection, 509, 512, 513
prepared statements, 509, 510, 515
fetch modes, 516-519
error handling, 512, 518
Polymorphism, 96-109, 185-196. *see also* Method overriding
compile-time and run-time, 98-100
upcasting and instanceof, 101-104
duck typing in Python, 189, 190
Prepared statement, 498-519. *see also* SQL injection
Primary key, 326-332, 354-359. *see also* Foreign key
surrogate and natural, 331, 332
AUTO_INCREMENT, 407-409, 414
Process, 242-249. *see also* Thread
process states, 247
process control block, 242-246
compared with a program, 242, 243, 249
compared with a thread, 242, 243, 249
Process synchronisation, 269-275, 282. *see also* Semaphore
the bounded buffer, 269, 273
busy waiting, 274, 275

Programmed input and output, *see* Direct memory access
Property decorator, 173-184. *see also* Encapsulation

Q

Quantum (time slice), 259-268. *see also* CPU scheduling
Query optimisation, 483-497. *see also* Index (database)
  reading a query plan, 488-491
  sargable predicates, 484

R

Race condition, 269-275. *see also* Critical section
Ready queue, 242-249, 252. *see also* Process
Real-time operating system, 217-224. *see also* Operating system
Recursion, 45-55, 151-160. *see also* Method
Referential action, 405-414. *see also* Foreign key
Relational model, 326-332. *see also* Integrity rules
  relation, tuple, attribute and domain, 326-329
  the three integrity rules, 326-328, 332
Relationship, 334, 339-345, 348-350, 354. *see also* Cardinality
  degree: unary, binary and ternary, 341, 344, 345

S

Scanner class, 8-18, 40, 66, 119
Schedulers, the three, 250-268. *see also* Dispatcher
Scheduling, *see* CPU scheduling; Disk scheduling; Schedulers, the three
Schema, 333-338, 396-398, 467, 468, 530. *see also* Database design
Scope of a variable, 45-55, 151-160. *see also* Variable
Second normal form, 371-385. *see also* Normalisation
  partial dependency, 372, 373, 376
Seek time, 303-312. *see also* Disk scheduling
Segmentation, 294-296, 302. *see also* Paging
SELECT, 415-424. *see also* ORDER BY
  the WHERE clause, 415, 416, 422
  DISTINCT, 416-418
  LIKE and wildcards, 422
  BETWEEN and IN, 422
  LIMIT, 416, 423
self, 161-172, 186, 194, 201. *see also* Class
Semaphore, 269-275, 283. *see also* Mutual exclusion
  binary and counting, 270-274
  wait and signal, 271
Sharding, 520-528, 532. *see also* Distributed database
Shell, 225-233. *see also* Kernel
  command line and graphical, 221
SQL injection, 498-508. *see also* PDO (PHP Data Objects)
  prevented by prepared statements, 498, 499, 502
Standard library, 151-161. *see also* Module (Python)
Star schema, 533, 534. *see also* Data warehousing
Starvation, 258-268, 285, 310. *see also* Ageing
static, 45-55, 75, 76. *see also* Method
  static method against instance method, 47, 48, 51
  static variable, 49-52
Storage engine, 457-468
Stored procedure and function, 444-457. *see also* View
  parameters and DELIMITER, 455, 456
  trigger, 455, 456
String, 8-18, 90-92, 131-140
  immutability, 11, 133, 134
  comparison with equals, 11, 12, 132
  formatting and f-strings, 138-140
Subquery, 444-456. *see also* Join
  correlated, 444, 445, 449
  IN, EXISTS and ANY, 450-452
super, 81-95, 185-196. *see also* Inheritance
Superkey, 326-332, 365, 366, 381. *see also* Candidate key
  alternate key, 327, 331, 332
Swapping, 285-302. *see also* Virtual memory
Switch statement, 29-45. *see also* If statement
  fall-through and break, 31, 32, 38
System call, 225-233. *see also* Kernel
  categories of, 227, 228

T

Table alias, and the self join, 434-444. *see also* Join
Testing, 110-121. *see also* Debugging
Third normal form, 380-388. *see also* Normalisation
  transitive dependency, 381-384
this, 47, 48, 52, 56-69, 89. *see also* self
Thrashing, 294-302. *see also* Working set
Thread, 219, 220, 242-249. *see also* Process
  compared with a process, 242, 245-247
  multithreading, 219, 220
Three level architecture, 316, 333-338. *see also* Database management system
  data independence, 315, 316, 319
Throughput, 240, 250-268, 279. *see also* Turnaround time
Time-sharing system, 217-224. *see also* Multiprogramming and multitasking
Transaction, 457-472. *see also* ACID properties; Concurrency control
  the ACID properties, 457-461
  COMMIT and ROLLBACK, 458, 466, 467
  states of, 457-468
Trigger, *see* Stored procedure and function
Turnaround time, 250-268. *see also* Average waiting time

U

UNION, 434-443, 451, 452, 501. *see also* Join
Unnormalised form, 371-379. *see also* Normalisation
UPDATE, 407-415, 459, 479-482. *see also* DELETE
  the missing WHERE clause, 407, 408, 411, 412

V

Variable, 8-21, 131-140
  declaration and initialisation, 131
  naming and identifiers, 13-17
  scope, 45, 153
View, 319, 444-456. *see also* Stored procedure and function
  updatable views, 453
  why a view is not a table, 455
Virtual memory, 294-302. *see also* Paging
  demand paging, 296, 297
  thrashing, 301, 302

W

Weak entity, 346-358. *see also* Entity
WHERE, 411, 412, 415-427, 439, 440. *see also* HAVING; SELECT
  compared with HAVING, 425-433
Working set, 294-302. *see also* Thrashing
Worst fit, 290, 291. *see also* Best fit; First fit
Wrapper class, 8-18, 115