```
CREATE OR REPLACE PACKAGE BODY fare_pkg AS
  FUNCTION calc_fare(p_from IN stations.code%TYPE, …)
    RETURN NUMBER IS
  BEGIN
    l_fare := 3.50 + 1.20 * ABS(l_zone_from - l_zone_to);
    RETURN ROUND(l_fare * (1 - get_discount(p_card)), 2);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RAISE_APPLICATION_ERROR(-20001, 'Invalid route');
  END calc_fare;
END fare_pkg;
```

```
SQL> EXEC
booking_api.create_booking(…)
Booking 5009 confirmed: 4.43

PL/SQL procedure successfully
completed.
```

FERNCASTLE RAIL

FNC → WLK

STD · YOUTH · 2025-05-06

Booking 5002 · 4.43

FROM FIRST BLOCK TO PRODUCTION

Oracle PL/SQL Programming

Procedures, packages, triggers, bulk processing, testing, and security

For Oracle Database 19c and later · SQL Developer and SQLcl

BLOCKS · CURSORS · PACKAGES · FORALL · UTPLSQL

THENEXBOOKS

Oracle PL/SQL Programming

From Your First Block to Production-Grade Database Code

Procedures · Packages · Triggers · Bulk Processing · Testing · Security

First Edition

THENEXBOOKS

Oracle PL/SQL Programming
From Your First Block to Production-Grade Database Code

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Oracle Database features vary by release and edition; always consult the Oracle documentation for the version you use. Test every change in a non-production environment first.

Oracle, Java, and MySQL are registered trademarks of Oracle and/or its affiliates. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Oracle Corporation.

Ferncastle Rail, the running example in this book, is fictional, as are all stations, passengers, and bookings in its examples. Any resemblance to real companies or people is coincidental.

Preface

Some of the most important code in the world never leaves the database. Banks calculate interest, airlines price fares, hospitals validate orders, and governments process benefits with code that runs right next to the data it reads and changes. In Oracle Database, that code is written in PL/SQL — a procedural language designed to work hand in hand with SQL, with variables, conditions, loops, error handling, and modular program units built directly around database operations.

PL/SQL rewards developers who understand both halves of its name. Written well, it is fast, safe, and remarkably maintainable: a few packages can enforce business rules for every application that touches the data, process millions of rows efficiently, and keep data consistent through failures. Written poorly, it becomes slow row-by-row loops, swallowed exceptions, injection holes, and triggers nobody understands.

This book teaches PL/SQL the way it is used in well-run teams. We start with the structure of a block and Oracle SQL essentials, build up through cursors, exceptions, procedures, functions, packages, and triggers, then move to collections, bulk processing, dynamic SQL, and modern data types. The final parts cover testing, instrumentation, performance, security, scheduling, integration with applications, and safe deployment — the practices that make database code production-grade.

Who This Book Is For

- **Developers new to Oracle** who need to read, write, and maintain PL/SQL with confidence.
- **Application developers** (Java, Python, .NET, JavaScript) who call PL/SQL and want to understand what happens inside the database.
- **Database developers and DBAs** who want modern, professional habits: bulk processing, packages, testing, and secure dynamic SQL.
- **Students and career changers** preparing for roles that work with Oracle Database.

Some familiarity with SQL helps, but Chapter 3 refreshes the Oracle SQL you need.

What You Will Learn

- How to set up a free Oracle development environment and run PL/SQL in SQL Developer and SQLcl.
- Block structure, variables, data types, `%TYPE` and `%ROWTYPE`, conditions, and loops.
- SQL inside PL/SQL: `SELECT INTO`, DML, transactions, and every kind of cursor.
- Robust exception handling, including custom exceptions and error propagation.
- Procedures, functions, packages, and triggers — and how Oracle manages their dependencies.
- Collections, records, object types, `BULK COLLECT`, `FORALL`, and dynamic SQL.
- Dates, strings, LOBs, and JSON in PL/SQL.

- Unit testing with utPLSQL, logging and instrumentation, performance tuning, and security against SQL injection.
- Scheduling jobs, calling PL/SQL from applications and REST services, and deploying changes safely.

How This Book Is Organized

Part I — PL/SQL Foundations (Chapters 1–5) sets up the environment and teaches blocks, SQL essentials, and declarations. **Part II — Program Logic and Data Access** (Chapters 6–10) covers conditions, loops, SQL in PL/SQL, cursors, and exceptions. **Part III — Program Units** (Chapters 11–15) builds procedures, functions, packages, and triggers and explains how Oracle manages them. **Part IV — Working with Data at Scale** (Chapters 16–20) covers collections, records, bulk processing, dynamic SQL, and modern data types. **Part V — Quality, Performance, and Security** (Chapters 21–24) tests, instruments, tunes, and secures your code. **Part VI — Production and Case Study** (Chapters 25–28) schedules, integrates, deploys, and brings everything together. The appendices provide a cheat sheet, a built-in package and template reference, a production checklist, and a glossary.

Each chapter ends with practice exercises and a **Key Takeaways** box for quick review.

The Running Example: Ferncastle Rail

Throughout the book we build the booking engine of **Ferncastle Rail**, a fictional regional railway with five stations across three fare zones. Passengers book standard or first-class journeys, some with railcard discounts; bookings can be confirmed or cancelled; and every change must be validated, priced correctly, and audited. The data set is deliberately small — five stations, four passengers, eight bookings — so you can check every result by hand, yet it exercises everything real systems need: lookups, calculations, transactions, error handling, packages, triggers, bulk updates, and secure APIs.

Versions and Conventions

Examples target Oracle Database 19c and later. Features introduced in newer releases, such as Oracle Database 23ai, are labeled where they appear. Code appears in `monospace`; output from `DBMS_OUTPUT` is shown in blocks labeled **Output**, and query results are shown as tables. To see `DBMS_OUTPUT` in SQL*Plus or SQLcl, run `SET SERVEROUTPUT ON` first.

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version details, or a useful aside.

WARNING

A common mistake that causes bugs, data corruption, slow code, or security holes.

Contents

Preface

PART I — PL/SQL FOUNDATIONS

1 What PL/SQL Is and Where It Fits

2 Setting Up: Oracle Database Free, SQL Developer, and SQLcl

3 Oracle SQL Essentials for PL/SQL Developers

4 Your First PL/SQL Blocks: Structure, Variables, and DBMS_OUTPUT

5 Data Types and Declarations: Scalars, %TYPE, %ROWTYPE, and Constants

PART II — PROGRAM LOGIC AND DATA ACCESS

6 Conditional Logic: IF and CASE

7 Loops: Basic, WHILE, FOR, and Cursor FOR Loops

8 SQL Inside PL/SQL: SELECT INTO, DML, and Transaction Control

9 Cursors: Implicit, Explicit, Parameterized, and Cursor Variables

10 Exception Handling

PART III — PROGRAM UNITS

11 Procedures and Parameters

12 Functions

13 Packages

14 Triggers

15 Managing Code in the Database: Dependencies, Compilation, and the Data Dictionary

PART IV — WORKING WITH DATA AT SCALE

16 Collections: Associative Arrays, Nested Tables, and VARRAYs

17 Records and Object Types

18 Bulk Processing with BULK COLLECT and FORALL

19 Dynamic SQL: EXECUTE IMMEDIATE and DBMS_SQL

20 Dates, Strings, LOBs, and JSON in PL/SQL

PART V — QUALITY, PERFORMANCE, AND SECURITY

21 Testing PL/SQL with utPLSQL

22 Debugging, Logging, and Instrumentation

23 Performance Tuning PL/SQL

24 Security: Privileges, Definer and Invoker Rights, and SQL Injection

PART VI — PRODUCTION AND CASE STUDY

25 Scheduling Jobs with DBMS_SCHEDULER

26 Calling PL/SQL from Applications: JDBC, Python, and REST with ORDS

27 Deploying and Evolving Code: Source Control, Scripts, and Edition-Based Redefinition

28 Case Study: The Ferncastle Rail Booking Engine

APPENDICES

A PL/SQL Cheat Sheet

B Built-in Package and Template Reference

C Production Checklist

D Glossary

PART I

PL/SQL Foundations

What PL/SQL is and where it fits, a free development environment, the Oracle SQL you need, and the building blocks of every program: blocks, variables, data types, and declarations.

1 · What PL/SQL Is

2 · Setting Up

3 · Oracle SQL Essentials

4 · Your First Blocks

5 · Data Types and Declarations

CHAPTER 1 · PART I — PL/SQL FOUNDATIONS

What PL/SQL Is and Where It Fits

Oracle Database stores the facts that run banks, hospitals, airlines, and railways. PL/SQL is the language that lets you put careful, reusable logic right next to those facts. This chapter explains what PL/SQL is, how it works with SQL, when it is the right tool, and introduces Ferncastle Rail, the small railway that will travel with you through the whole book.

Part I at a Glance

Part I lays the foundation for everything that follows. In this chapter you will learn what PL/SQL is and why it exists. In Chapter 2 you will install a free Oracle database, connect to it with SQL Developer and SQLcl, and run the setup script that creates the Ferncastle Rail sample schema. Chapter 3 is a focused refresher on the Oracle flavor of SQL, because every useful PL/SQL program talks to the database through SQL. Chapter 4 is where you write and run your first PL/SQL blocks, declare variables, and print results. Chapter 5 completes the part with a careful look at data types and declarations.

What Is PL/SQL?

PL/SQL stands for Procedural Language extensions to SQL. It is Oracle's own programming language, and it lives inside Oracle Database itself. To understand why it exists, it helps to start with what SQL can and cannot do on its own.

SQL (Structured Query Language) is a **declarative** language. When you write a SQL statement, you describe *what* result you want, and the database decides *how* to produce it. "Give me every confirmed booking for May 7, sorted by price" is a perfect SQL question. You never tell the database which index to read first or how to loop over the rows. That is a great strength: the optimizer can choose a clever plan, and your statement stays short and readable.

Real business processes, however, are rarely a single question. Consider what happens when a passenger books a ticket on a railway. The system must look up the two stations, work out the zones, apply a fare rule, check whether the passenger holds a discount railcard, round the result to the cent, insert the booking, perhaps write an audit record, and report a friendly error if any step fails. That is a sequence of steps with decisions and error handling, which is exactly what a **procedural** language is designed to express.

PL/SQL adds the procedural ingredients that SQL lacks:

- **Variables and constants** to hold intermediate values, such as a base fare before the discount is applied.
- **Conditional logic** with `IF` and `CASE`, so the code can take different paths for first-class and standard tickets.
- **Loops** to repeat work, for example once for each booking on a given date.
- **Exception handling**, a structured way to catch and respond to errors instead of letting them crash a whole process.
- **Named, stored program units** such as procedures, functions, packages, and triggers that are compiled once, saved in the database, and reused by every application that connects.

The crucial point is that PL/SQL does not replace SQL. It wraps around it. SQL statements appear directly inside PL/SQL code, with no special library calls and no string building required for ordinary queries. The two languages share the same data types, so a `NUMBER` column in a table and a `NUMBER` variable in your program behave identically. This tight integration is the single most important reason PL/SQL remains popular after more than three decades.

A First Glimpse

Here is the smallest useful PL/SQL program you will see in this book. It is an **anonymous block**, meaning a piece of PL/SQL code that has no name and is not stored in the database. You send it to the server, it runs once, and it is gone.

first_block.sql

```
SET SERVEROUTPUT ON

BEGIN
  DBMS_OUTPUT.PUT_LINE('Welcome aboard Ferncastle Rail');
END;
/
```

Output

```
Welcome aboard Ferncastle Rail

PL/SQL procedure successfully completed.
```

Even this tiny example shows several ideas you will meet again and again. The keywords `BEGIN` and `END` mark the executable part of the block. `DBMS_OUTPUT.PUT_LINE` is a call to a procedure inside a built-in package named `DBMS_OUTPUT`, which writes a line of text into a buffer. The command `SET SERVEROUTPUT ON` belongs to the client tool, not to PL/SQL, and tells the tool to display that buffer once the block finishes. The lone slash on the last line tells SQL*Plus, SQLcl, or SQL Developer to send the block to the database and run it. Chapter 4 explores each of these pieces in detail; for now, simply notice how little ceremony is required.

SQL and PL/SQL Side by Side

To feel the difference between the two languages, compare two ways of answering a simple question about Ferncastle Rail: how many bookings are confirmed? In SQL the answer is a single statement:

count_confirmed.sql

```
SELECT COUNT(*) AS confirmed_bookings
FROM   bookings
WHERE  status = 'CONFIRMED';
```

CONFIRMED_BOOKINGS
7

In PL/SQL you can run the same query, keep the answer in a variable, and then do something with it, such as make a decision:

count_confirmed_block.sql

```
DECLARE
  v_confirmed  NUMBER;
BEGIN
  SELECT COUNT(*)
  INTO   v_confirmed
  FROM   bookings
  WHERE  status = 'CONFIRMED';

  IF v_confirmed > 5 THEN
    DBMS_OUTPUT.PUT_LINE('Busy week: ' || v_confirmed || ' confirmed bookings');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Quiet week: ' || v_confirmed || ' confirmed bookings');
  END IF;
END;
/
```

Output

```
Busy week: 7 confirmed bookings

PL/SQL procedure successfully completed.
```

The SQL statement inside the block is almost identical to the stand-alone query. The only addition is the `INTO` clause, which delivers the result into the variable `v_confirmed`. After that, ordinary procedural code takes over. The `||` operator joins pieces of text together, and the number 7 is converted to text automatically. You will be able to run this block yourself as soon as you have completed the setup in Chapter 2.

How PL/SQL Runs Inside the Database

Most programming languages run on a client machine or an application server and send SQL to the database over a network. PL/SQL is different: it runs *inside* the database server process. Understanding this architecture explains both PL/SQL's strengths and some of its performance rules.

Every Oracle database contains two cooperating engines. The **SQL engine** parses, optimizes, and executes SQL statements. The **PL/SQL engine** executes procedural code: assignments, conditions, loops, and calls between program units. When your block reaches a SQL statement, the PL/SQL engine hands it to the SQL engine, waits for the result, and continues. When a SQL statement calls a PL/SQL function, the hand-off happens in the other direction. Each hand-off is called a **context switch**.

Context switches are fast, but they are not free. A loop that issues one `INSERT` per row for a million rows performs a million switches. Later in the book, in Chapter 18, you will learn bulk processing with `BULK COLLECT` and `FORALL`, which reduce that cost dramatically by moving many rows per switch. Keep the mental picture in mind: two engines in the same process, passing work back and forth.

Compilation and Storage

When you create a named program unit such as a procedure, Oracle **compiles** it immediately. The compiler checks your syntax, verifies that every table, column, and other program you reference actually exists, and confirms that you have permission to use them. If all is well, Oracle stores both the source code and the compiled form in the **data dictionary**, the set of internal tables in which the database describes itself. From then on, any session with the right privileges can call the unit without recompiling it.

By default, PL/SQL is compiled to an intermediate form that the PL/SQL engine interprets. Oracle can also compile units to native machine code, controlled by a setting named `PLSQL_CODE_TYPE`. Native compilation can help computation-heavy code, but most PL/SQL spends its time inside SQL statements, so the difference is often modest. Chapter 23 returns to this topic with measurements in mind.

Because the compiler checks references at compile time, the database also tracks **dependencies**. If a stored procedure reads the `bookings` table and someone later changes that table, Oracle marks the procedure as needing recompilation. Chapter 15 explains this dependency tracking and how to inspect it in the data dictionary.

> **NOTE**
>
> PL/SQL runs on the database server, so `DBMS_OUTPUT` does not print directly to your screen. It writes to a buffer in your database session, and your client tool reads that buffer after the call completes. This is why output appears all at once at the end of a long-running block, not line by line as it executes.

The Building Blocks: Kinds of Program Units

Everything in PL/SQL is organized as a **block**, a unit of code with an optional declaration section, a mandatory executable section, and an optional exception-handling section. Blocks can be anonymous, as in the examples above, or named and stored in the database. The table below introduces the kinds of program units you will build in this book and points to the chapter where each one is covered in depth.

Program unit	What it is	Ferncastle Rail example	Chapter
Anonymous block	Unnamed code sent to the server and run once	A script that prints daily revenue	4
Procedure	Named unit that performs an action	Cancel a booking	11
Function	Named unit that returns a value	Calculate a fare	12
Package	A named group of related procedures, functions, types, and variables	`fare_pkg` and `booking_api`	13
Trigger	Code that runs automatically when an event occurs	Audit every change to a booking	14
Object type	A user-defined type with attributes and methods	A journey with its own fare method	17

Packages deserve a special mention because they are the backbone of serious PL/SQL development. A **package** has two parts: a **specification**, which declares what the outside world may call, and a **body**, which contains the implementation. This separation lets you change how something works without disturbing the programs that use it. In Ferncastle Rail, the fare logic lives in `fare_pkg` and the booking operations live in `booking_api`. By Chapter 28 you will have built both in full.

Why Put Logic in the Database?

Developers who come from other languages sometimes ask why anyone would write business logic inside a database at all. Application servers are easy to scale, modern languages have rich ecosystems, and keeping the database "just for storage" sounds tidy. There is no single right answer for every project, but PL/SQL offers advantages that are hard to match elsewhere.

Proximity to the Data

The most obvious benefit is that PL/SQL runs where the data lives. A process that reads thousands of rows, makes decisions, and writes results can do all of that without moving a single row across the network. When an application server performs the same work, each step can involve a network round trip, and round trips add up. A booking routine that needs five queries and two inserts costs seven round trips from a remote client, but only one when the client calls a single stored procedure that does everything on the server.

One Set of Rules for Every Application

Organizations rarely have just one program touching their data. Ferncastle Rail might have a website, a mobile app, ticket machines at stations, a nightly reporting job, and a customer service tool. If each of these calculates fares independently, sooner or later they will disagree. One team rounds differently, another forgets the senior discount, and passengers are charged different prices for the same journey. When the fare rule lives in `fare_pkg.calc_fare` inside the database, every application calls the same code and gets the same answer.

Security Through a Narrow Interface

PL/SQL lets you grant users the right to *run* a procedure without granting them direct access to the tables it touches. A ticket machine account could be allowed to execute `booking_api.create_booking` and nothing else. It cannot update prices directly, read other passengers' email addresses, or delete history. This pattern, sometimes described as a "thick database" or "smart database" approach, turns the database into a well-guarded service with a small, controlled surface. Chapter 24 shows how definer rights and invoker rights make this work.

Transactions and Integrity

Oracle's transaction model is one of the most robust in the industry, and PL/SQL works with it naturally. A stored procedure can perform several related changes and either commit all of them or roll all of them back. Constraints, triggers, and procedures together can guarantee that a booking never references a passenger who does not exist, and that a cancelled booking is never cancelled twice. When this logic sits in the database, no application can bypass it by accident.

When PL/SQL Is Not the Right Tool

A balanced view matters. PL/SQL is excellent at data-centric logic, but it is not the answer to every problem, and you will design better systems if you recognize its limits.

- **User interfaces.** PL/SQL can generate web pages, and tools such as Oracle APEX build complete applications on top of it, but most teams build their interfaces in languages designed for that purpose.
- **Heavy computation unrelated to data.** Image processing, machine-learning training, or complex numeric simulations usually run better in specialized environments.
- **Portability across database vendors.** PL/SQL is specific to Oracle. If your product must run on several different databases, logic in the database becomes harder to share.
- **Calls to many external services.** PL/SQL can make HTTP calls, but orchestrating many external APIs is often simpler in an application tier.
- **Team skills.** Code that nobody on the team can maintain is a liability, whatever language it is written in.

The practical rule followed by many experienced teams is simple: keep data rules close to the data, and keep presentation and integration logic in the application tier. Fare calculation, booking validation, auditing, and reporting belong naturally in PL/SQL. Rendering a seat map or sending a push notification to a phone usually does not.

> **TIP**
> When you are unsure where a piece of logic belongs, ask a simple question: "If a second application started using this data tomorrow, would it need the same rule?" If the answer is yes, the rule probably belongs in the database, where every application will share it.

Versions and Editions Used in This Book

Oracle Database has evolved continuously, and PL/SQL has gained features in nearly every release. This book targets **Oracle Database 19c**, a long-term release that is widely deployed in production, and everything later. Wherever a feature requires **Oracle Database 23ai** (the release previously called 23c), we label it clearly. Examples include the SQL `BOOLEAN` data type and the `IF EXISTS` and `IF NOT EXISTS` clauses for DDL statements.

For practice, you will use **Oracle Database Free**, the no-cost developer edition of recent Oracle releases. It runs on a laptop or in a container, includes the full PL/SQL language, and is more than sufficient for every example in this book. Chapter 2 walks you through installing it and connecting with Oracle's free tools: **SQL Developer**, a graphical integrated development environment; **SQLcl**, a modern command-line interface; and **SQL*Plus**, the classic command-line tool that still ships with every Oracle installation.

When behavior differs between versions in ways that matter, we say so. When a detail depends on settings or versions in ways too varied to describe exactly, we point you to the Oracle documentation. That documentation is free, thorough, and well worth bookmarking; the *PL/SQL Language Reference* and the *PL/SQL Packages and Types Reference* are the two volumes you will consult most often.

Meet Ferncastle Rail

Learning a language is easier when the examples connect to each other. Instead of a new toy table in every chapter, this book uses a single running example: **Ferncastle Rail**, a small fictional regional railway. Its data fits on one page, so you can check every result by eye, yet it is rich enough to demonstrate lookups, calculations, transactions, errors, auditing, bulk processing, and testing. All of its tables belong to a database user named `FERNCASTLE`; in the examples we simply use the unqualified table names.

Stations

The railway serves five stations. Each has a numeric ID, a three-letter code that passengers see on tickets, a name, and a fare **zone**. Zones grow outward from the city center, and the number of zones a journey crosses drives its price.

station_id	code	name	zone
10	FNC	Ferncastle Central	1
20	HBR	Harborview	1
30	MLD	Millford	2
40	ASH	Ashby Junction	2
50	WLK	Westlake	3

Passengers

Four passengers are registered. Each may hold a **railcard**, a discount card: `YOUTH` for younger travelers, `SENIOR` for older ones, or `NONE`. Notice that Dev Rao has no email address. That missing value, a `NULL` in database terms, is deliberate; it will help us demonstrate how SQL and PL/SQL handle unknown data.

passenger_id	full_name	email	railcard
1	Ada Lin	ada.lin@example.com	NONE
2	Ben Hart	ben.hart@example.com	YOUTH
3	Cora Diaz	cora.diaz@example.com	SENIOR
4	Dev Rao	(null)	NONE

Bookings

Eight bookings have been made for four travel days in May 2025. Each records the passenger, the origin and destination station codes, the travel date, the fare class (`STD` for standard or `FIRST` for first class), the status, and the price that was charged. One booking, 5005, has been cancelled.

booking_id	passenger	from	to	travel_date	class	status	price
5001	1	FNC	MLD	2025-05-06	STD	CONFIRMED	4.70
5002	2	FNC	WLK	2025-05-06	STD	CONFIRMED	4.43
5003	3	HBR	ASH	2025-05-07	FIRST	CONFIRMED	4.94
5004	4	MLD	WLK	2025-05-07	FIRST	CONFIRMED	7.05
5005	1	WLK	FNC	2025-05-08	STD	CANCELLED	5.90

booking_id	passenger	from	to	travel_date	class	status	price
5006	2	ASH	MLD	2025-05-08	STD	CONFIRMED	2.63
5007	3	FNC	HBR	2025-05-09	STD	CONFIRMED	2.45
5008	4	FNC	WLK	2025-05-09	FIRST	CONFIRMED	8.85

A few totals are worth remembering because they reappear throughout the book and make handy sanity checks. Seven bookings are confirmed, and together they earned 35.05. Including the cancelled booking, all eight prices add up to 40.95. Per passenger, confirmed revenue is 4.70 for Ada Lin, 7.06 for Ben Hart, 7.39 for Cora Diaz, and 15.90 for Dev Rao. Per travel date, confirmed revenue is 9.13 on May 6, 11.99 on May 7, 2.63 on May 8, and 11.30 on May 9. Whenever one of your programs prints a total, comparing it with these numbers will tell you at once whether the program is right.

The Fare Rule

Every price in the bookings table was produced by a single fare rule, which you will implement in PL/SQL step by step as the book progresses:

1. Start with a **base fare** of 3.50 plus 1.20 for each zone crossed: `3.50 + 1.20 × ABS(zone_from - zone_to)`. A journey within one zone costs 3.50.
2. For first class, multiply by 1.5.
3. Apply the railcard discount: 25% off for `YOUTH`, 30% off for `SENIOR`, nothing for `NONE`.
4. Round to two decimal places, with halves rounded away from zero, which is how Oracle's `ROUND` function behaves.

Working a few bookings through by hand is the best way to understand the rule, and it is also exactly the kind of reasoning you will later encode as tests. The table shows four representative cases.

Booking	Zones	Base	After class	After railcard	Rounded
5001 FNC→MLD STD, NONE	1 → 2	4.70	4.70	4.70	4.70
5002 FNC→WLK STD, YOUTH	1 → 3	5.90	5.90	4.425	4.43
5003 HBR→ASH FIRST, SENIOR	1 → 2	4.70	7.05	4.935	4.94
5006 ASH→MLD STD, YOUTH	2 → 2	3.50	3.50	2.625	2.63

Booking 5002 shows why rounding rules matter. Before rounding, the fare is 4.425, exactly halfway between 4.42 and 4.43. Oracle's `ROUND` sends halves away from zero, so the passenger pays 4.43. Some other languages round halves to the nearest even digit by default, which would produce 4.42. A one-cent disagreement between a website and a ticket machine is exactly the kind of inconsistency that putting the rule in one place prevents.

A Preview of the Destination

By Chapter 13 the fare rule will live in a package function named `fare_pkg.calc_fare`, which takes the origin code, destination code, fare class, and railcard, and returns the price. Once it exists, any SQL statement or PL/SQL program can call it. The following query is a preview; it will not run until you have built the package.

preview_calc_fare.sql (runs after Chapter 13)

```
SELECT fare_pkg.calc_fare('FNC', 'WLK', 'STD', 'YOUTH') AS fare
FROM   dual;
```

FARE
4.43

The result matches booking 5002, as it should. Around this function the book builds a complete booking engine: `booking_api` with `create_booking`, `cancel_booking`, and `get_passenger_total`; a sequence named `booking_seq` that hands out new booking numbers starting at 5009; an audit table, `booking_audit`, filled by a trigger named `bookings_aud_trg`; a logging package named `app_log`; and custom exceptions such as `e_invalid_route` and `e_already_cancelled`. Chapter 28 assembles all of these into a finished case study.

Where PL/SQL Fits in a Modern Architecture

It is useful to picture how Ferncastle Rail's systems might connect. A passenger opens the railway's website. The web server, written in Java, Python, or JavaScript, receives the request and calls a stored procedure, `booking_api.create_booking`, through a database driver. The procedure validates the route, calls `fare_pkg.calc_fare`, inserts the booking, and returns the new booking number. A trigger records the change in `booking_audit`. Overnight, a scheduled job written in PL/SQL summarizes the day's revenue. The station ticket machines use the same procedure through a REST endpoint published by Oracle REST Data Services.

In this picture, PL/SQL is not a competitor to the application languages; it is a partner. The application tier handles users, sessions, screens, and integrations. The database tier guards the data and enforces the rules. Chapter 26 shows how to call PL/SQL from Java through JDBC, from Python, and over HTTP with ORDS, and Chapter 25 covers scheduling with `DBMS_SCHEDULER`.

How This Book Is Organized

The book has six parts, each building on the one before.

- **Part I — PL/SQL Foundations** (Chapters 1–5) sets up your environment, refreshes Oracle SQL, and introduces blocks, variables, and data types.

- **Part II — Program Logic and Data Access** (Chapters 6–10) covers conditions, loops, SQL inside PL/SQL, cursors, and exception handling.
- **Part III — Program Units** (Chapters 11–15) turns code into stored procedures, functions, packages, and triggers, and explains how Oracle manages them.
- **Part IV — Working with Data at Scale** (Chapters 16–20) introduces collections, records, object types, bulk processing, dynamic SQL, and the handling of dates, strings, large objects, and JSON.
- **Part V — Quality, Performance, and Security** (Chapters 21–24) covers automated testing with utPLSQL, debugging and logging, performance tuning, and security.
- **Part VI — Production and Case Study** (Chapters 25–28) schedules jobs, connects applications, deploys and evolves code safely, and finishes with the complete Ferncastle Rail booking engine.

If you are new to PL/SQL, read the chapters in order; each one assumes the ones before it. If you already have some experience, feel free to skim Part I and dive into the topics you need, using the Ferncastle Rail setup script from Chapter 2 so that the examples still work. The appendices provide a cheat sheet, a reference to the built-in packages and templates used in the book, a production checklist, and a glossary.

Conventions Used in This Book

A few conventions make the examples easier to read.

- **Code style.** SQL and PL/SQL keywords appear in uppercase (`SELECT`, `BEGIN`), while names of tables, columns, and variables appear in lowercase. Oracle itself is not case-sensitive for unquoted names, so this is purely for readability.
- **Naming prefixes.** Local variables start with `v_`, constants with `c_`, parameters with `p_`, package-level (global) variables with `g_`, and user-defined exceptions with `e_`. Prefixes prevent a common bug in which a variable has the same name as a column.
- **File captions.** A caption above each code listing names a file, such as `count_confirmed.sql`, so you can save your work in an organized way.
- **Output.** Text printed with `DBMS_OUTPUT` appears in a block captioned *Output*. Query results appear as tables with the column headings Oracle would use; Oracle shows unquoted column aliases in uppercase.
- **Callouts.** Tips offer practical shortcuts, Notes add background, and Warnings point out mistakes that can cost you time or data.

Every chapter ends with a short Practice section and a list of Key Takeaways. The practice exercises use Ferncastle Rail data, so you can always check your answers against the known totals listed above.