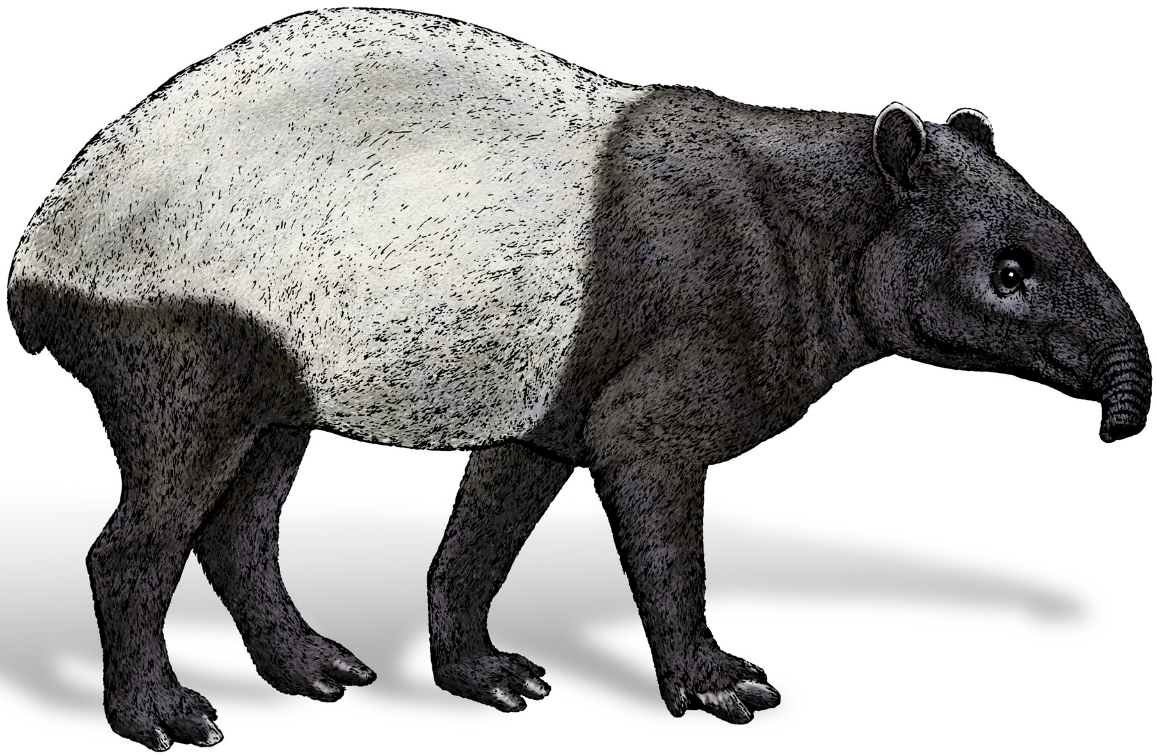
O'REILLY®

3rd Edition

Covers Scala 3

Programming Scala

Scalability = Functional Programming + Objects



Dean Wampler

Foreword by Seth Tisue

O'REILLY®

Programming Scala

Get up to speed on Scala—the JVM, JavaScript, and natively compiled language that offers all the benefits of functional programming, a modern object model, and an advanced type system. Packed with code examples, this comprehensive book shows you how to be productive with the language and ecosystem right away. You'll learn why Scala is ideal for today's highly scalable, data-centric applications, while maximizing developer productivity.

While Java remains popular and Kotlin has become popular, Scala hasn't been sitting still. This third edition covers the new features in Scala 3 with updates throughout the book. *Programming Scala* is ideal for beginning to advanced developers who want a complete understanding of Scala's design philosophy and features with a thoroughly practical focus.

- **Program faster with Scala's succinct and flexible syntax**
- **Dive into basic and advanced functional programming techniques**
- **Build killer big data and distributed apps using Scala's functional combinators and tools like Spark and Akka**
- **Create concise solutions to challenging design problems with the sophisticated type system, mixin composition with traits, pattern matching, and more**

Dean Wampler specializes in data engineering for streaming systems and applications using machine learning. He is a principal software engineer at Domino Data Lab. Dean is the author of several books and reports for O'Reilly, a frequent conference speaker, and a contributor to several open source projects. He has a PhD in Physics from the University of Washington. Find Dean on Twitter @deanwampler.

"Whether you're new to Scala entirely or making the transition from Scala 2 to 3, Dean Wampler is the ideal traveling companion."

—Seth Tisue
Senior Software Engineer, Scala compiler team, Lightbend Inc.

"Dean leaves no question unanswered. Reading this book will enable you to make new connections between concepts you couldn't connect before. Which is to say: you'll learn something."

—Lutz Huehnken
Chief Architect, Hamburg Süd, A Maersk Company

"Dean has succeeded in giving a complete and comprehensive overview of the third major release of the Scala language. Highly recommended!"

—Eric Loots
CTO, Lunatech

PROGRAMMING LANGUAGES

US $69.99 CAN $92.99
ISBN: 978-1-492-07789-3

Twitter: @oreillymedia
facebook.com/oreilly

Praise for *Programming Scala, Third Edition*

"Whether you're new to entirely Scala or making the two to three transition, Dean Wampler is the ideal traveling companion. Some Scala books make you feel like you're back in a classroom. This one makes you feel like you're pair-programming with a helpful expert by your side."

—*Seth Tisue, Senior Software Engineer, Scala Compiler Team, Lightbend Inc.*

"Dean leaves no question unanswered. Rather than telling you only what you need to know to produce working code, he takes an extra step and explains exactly: How is this feature implemented? Is there a more general idea behind it that can provide extra context? Reading this book will enable you to make new connections between concepts you couldn't connect before. Which is to say, you'll learn something."

—*Lutz Huehnken, Chief Architect, Hamburg Süd, A Maersk Company*

"Dean has succeeded in giving a complete and comprehensive overview of the third major release of the Scala language by not only describing all the new features of the language, but also covering what's changed from Scala 2. Highly recommended for both newbies and experienced Scala 2 programmers!"

—*Eric Loots, CTO, Lunatech*

"At his many Strata Data + AI talks and tutorials, Dean made the case for using Scala for data engineering, especially with tools such as Spark and Kafka. He captures his Scala expertise and practical advice here."

—*Ben Lorica, Gradient Flow*

"I've had the great pleasure of working with Dean in a few different roles over the past several years. He is ever the strong advocate for pragmatic, effective approaches for data engineering–especially using Scala as the ideal programming language in that work. This book guides you through why Scala is so compelling and how to use it effectively."

—Paco Nathan, Managing Partner at Derwen, Inc.

"An excellent update to the earlier edition that will help developers understand how to harness the power of Scala 3 in a pragmatic and practical way."

—Ramnivas Laddad, cofounder, Paya Labs, Inc.

THIRD EDITION

Programming Scala

Scalability = Functional Programming + Objects

Dean Wampler

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Programming Scala
by Dean Wampler

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Suzanne McQuade
Development Editor: Michele Cronin
Production Editor: Katherine Tozer
Copyeditor: Sonia Saruba
Proofreader: Athena Lakri
Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

June 2021: Third Edition

Revision History for the Third Edition
2021-05-26: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492077893* for release details.

978-1-492-07789-3

[LSI]

To Peggy Williams Hammond, September 10, 1933–May 11, 2018.

—Dean

Table of Contents

Foreword

Foreword, Third Edition

Forward-looking programming languages don't always make it. Yet Scala is not only surviving but thriving. Some languages never get commercial adoption at all. Those first few companies brave enough to bet their business on your language are hard to find. Other languages get their time in the commercial sun but don't manage to hang on, like Common Lisp and Smalltalk. They live on as influences, their genes still discernable in contemporary languages. That's success of a kind, but not what the creators wanted.

Scala has been defying these trends for well over a decade now. Circa 2008, companies such as Twitter and Foursquare brought Scala out of academia and into the commercial world. Since then, the Scala job market and ecosystem have been sustained not only by independent enthusiasts but by superstar open source projects, such as Spark and Kafka, and companies like those on the Scala Center's advisory board, who collectively employ impressive numbers of Scala programmers.

Can Scala continue to pull it off? Its creator, Martin Odersky, thinks it can and I agree. Scala 3, launching in 2021, is a bold leap into the future of programming. Other languages will be playing catch-up for years to come.

And not for the first time, either. In the years since Scala's initial success, Java emerged from its long torpor with a parade of Scala-inspired language features. Swift and Rust also show Scala's influence. Direct competitors have appeared too. Kotlin remains a contender, while others, such as Ceylon, have already fallen by the wayside.

How much innovation is too much? How much is too little? Explorers must be bold but not foolhardy. Dangers lurk in new seas and on new lands, but you'll never discover anything if you just stay home.

Scala's bet is that being a better Java isn't enough to meet programmers' needs—not even if that better Java is Java itself. For one thing, competing with Java isn't enough anymore. If Scala's growth has leveled off somewhat in recent years, perhaps it's because Java's has too, and because we've already converted all the Java programmers we can hope to convert. We need to show that Scala is also a more-than-worthy alternative to now-mainstream languages, like Python and TypeScript, and insurgent languages, like Rust and Haskell.

The big reason Scala still matters and is worth fighting for is that it fully embraces functional programming. Yes, it's wonderful that Java has added lambdas and pattern matching, features that come from the functional tradition. But functional programming isn't just a bag of disconnected individual features. It's a paradigm shift, a fresh way of thinking. Learning Scala, or any functional language, makes you wonder how you ever programmed any other way.

Learning Scala doesn't mean forgetting everything you already know. Scala fuses the object-oriented and functional programming traditions into a single language you'll never grow out of. And though Scala offers its own vibrant ecosystem of libraries, Scala programmers are also free to leverage vast worlds of Java and JavaScript.

The design of Scala 3 retains the same pragmatism that has been crucial to its success all along. My teammates and I at Lightbend, along with our colleagues at the Scala Center and in Martin's lab, work hard to make migration to new versions smooth, even as we bring you a Christmas-morning's worth of new toys to play with.

It's truly remarkable how much of the Scala ecosystem has already made the leap. Scala 3 only just came out this month, but a rich array of libraries and tooling is already available for it.

Whether you're entirely new to Scala or making the 2 to 3 transition, Dean Wampler is the ideal traveling companion. Some Scala books make you feel like you're back in a classroom. This one makes you feel like you're pair-programming with a helpful expert. The text is bristling with practical know-how, with all of the nuances and need-to-knows for when you're actually at the keyboard, trying to make something run. Dean inspires with how programming in Scala ought to be and is candid about what it is actually like. He gives you tomorrow, and today.

Whatever the future holds for Scala, it will always be known as the language that took functional programming from a daring experiment to a practical, everyday reality.

My fondest wish for this book is that it will find its way into the hands of a new generation of Scala programmers. This new crew will be younger and more diverse than the old guard, and less encumbered by programming's past. Professors: teach your students Scala! I can't wait to see what they'll build.

— Seth Tisue
Senior Software Engineer, Scala Compiler Team, Lightbend, Inc.
Reno, Nevada, May 2021

Foreword, First and Second Edition

If there has been a common theme throughout my career as a programmer, it has been the quest for better abstractions and better tools to support the craft of writing software. Over the years, I have come to value one trait more than any other: composability. If one can write code with good composability, it usually means that other traits we software developers value—such as orthogonality, loose coupling, and high cohesion—are already present. It is all connected.

When I discovered Scala some years ago, the thing that made the biggest impression on me was its composability.

Through some very elegant design choices and simple yet powerful abstractions that were taken from the object-oriented and functional programming worlds, Martin Odersky has managed to create a language with high cohesion and orthogonal, deep abstractions that invites composability in all dimensions of software design. Scala is truly a SCAlable LAnguage that scales with usage, from scripting all the way up to large-scale enterprise applications and middleware.

Scala was born out of academia, but it has grown into a pragmatic and practical language that is very much ready for real-world production use.

What excites me most about this book is that it's so practical. Dean has done a fantastic job, not only by explaining the language through interesting discussions and samples, but also by putting it in the context of the real world. It's written for the programmer who wants to get things done.

I had the pleasure of getting to know Dean some years ago when we were both part of the aspect-oriented programming community. Dean holds a rare mix of deep analytical academic thinking and a pragmatic, get-things-done kind of mentality.

You are about to learn how to write reusable components using mixin and function composition; how to write distributed applications using Akka; how to make effective use of advanced features in Scala, such as macros and higher-kinded types; how to utilize Scala's rich, flexible, and expressive syntax to build domain-specific languages; how to effectively test your Scala code; how to let Scala simplify your big-data problems; and much, much more.

Enjoy the ride. I sure did.

— Jonas Bonér
CTO & cofounder Typesafe
August 2014

Preface

Programming Scala introduces an exciting and powerful language that offers all the benefits of a modern object-oriented programming (OOP) model, functional programming (FP), and an advanced type system. Originally targeted for the Java Virtual Machine (JVM), it now also targets JavaScript and native execution as well. Packed with code examples, this comprehensive book teaches you how to be productive with Scala quickly and explains what makes this language ideal for today's scalable, distributed, component-based applications that run at any scale.

Learn more at *http://programming-scala.org* or at the book's catalog page.

Welcome to Programming Scala, Third Edition

Dean Wampler, April 2021

Programming Scala, second edition was published six years ago, in the fall of 2014. At that time, interest in Scala was surging, driven by two factors.

First, alternative languages for the JVM instead of Java were very appealing. Java's evolution was slow at the time, frustrating developers who wanted improvements like more concise syntax for some constructs and features they saw in other languages, like FP.

Second, big data was a hot sector of the software industry, and some of the most popular tools in that sector, especially Apache Spark and Apache Kafka, were written in Scala and offered concise and elegant Scala APIs.

A lot has changed in six years. Oracle deserves a lot of credit for reinvigorating Java after acquiring it through the purchase of Sun Microsystems. The pace of innovation has improved considerably, and many important features have been added, like support for anonymous functions, called *lambdas*, that addressed the biggest missing feature needed for FP.

Also, the Kotlin language was created by the tool vendor JetBrains, as a "better Java" that isn't as sophisticated as Scala. Kotlin received a big boost when Google endorsed it as the preferred language for Android apps. Around the same time, Apple introduced a language called Swift, primarily for iOS development, that has a very Scala-like syntax, although it does not target the JVM.

Big data drove the emergence of data science as a profession. Actually, this was just a rebranding and refinement of what data analysts and statisticians had been doing for years. The specialties of deep learning (i.e., using neural networks), reinforcement learning, and artificial intelligence are currently the hottest topics in the data world. All fit under the umbrella of machine learning. A large percentage of the popular tools for data science and machine learning are written in Python (or expose Python APIs on top of C++ kernels). As a result, interest in Python is growing strongly again, while Scala's growth in the data world has slowed.

But Scala hasn't been sitting still. The Scala Center at École Polytechnique Fédérale de Lausanne (EPFL) was created to drive the evolution of the language and the core open source tooling for the ecosystem, like build tools and integrated development environments, while Lightbend continues to be the major provider of commercial support for Scala in the enterprise.

The fruits of these labors are many, but Scala version 3 is the most significant result to date. It brings changes to improve the expressiveness and correctness of Scala and remove deprecated and less useful features. Scala 3 is the focus of this edition, whether you are experienced with Scala 2 or brand new to Scala.

Scala 3 continues Scala's unparalleled track record of being a leading-edge language research platform while also remaining pragmatic for widespread industrial use. Scala 3 reworks the industry-leading implicit system so that common idioms are easier to use and understand. This system has propelled the creation of elegant, type-safe APIs that go far beyond what's possible with all other popular languages. The optional braceless syntax makes already-concise Scala code even more pristine, while also appealing to Python data scientists who work with Scala-based data engineering code. Scala's unique, thoughtful combination of FP and OOP is the best I have ever seen. All in all, Scala 3 remains my favorite programming language, concise and elegant, yet powerful when I need it.

Also, Scala is now a viable language for targeting JavaScript applications through Scala.js. Support for Scala as a *native* language (compiled directly to machine object code) is now available through Scala Native. I won't discuss the details of using Scala.js and Scala Native, but the Bibliography lists several resources, like [LiHaoyi2020] and [Whaling2020], respectively.

I currently split my time between the Python-based machine learning world and the Scala-based JVM world. When I use Python, I miss the concision, power, and correctness of Scala. The heavy use of mutation and the incomplete collections lower my productivity and make my code more verbose. However, when I use Scala, I miss the wealth of data-centric libraries available in the Python world.

All things considered, interest in Scala is growing less quickly today, but developers who want Scala's power and elegance are keeping the community vibrant and growing, especially in larger enterprises that are JVM-centered and cloud-based. Who knows what the next five or six years will bring, when it's time for *Programming Scala*, fourth edition?

With each edition of this book, I have attempted to provide a comprehensive introduction to Scala features and core libraries, illustrated with plenty of pragmatic examples and tips based on my years of experience in software development. This edition posed unique challenges because the transition from Scala 2 to 3 requires understanding old and new features, along with the plan for phasing out old features over several Scala 3 releases. I have explained the most important Scala 2 features that you'll need for working with existing code bases, while ignoring some seldom-used features that were dropped in Scala 3 (like procedure syntax). Note that when I refer to Scala 2 features, I'll mean the features as they existed in the last Scala 2 release, 2.13.X, unless otherwise noted.

I have also shortened the previous editions' surveys of Scala libraries. I think this makes the book more useful to you. A Google search is the best way to find the latest and best library for working with JSON, for example. What doesn't change so quickly and what's harder to find on Stack Overflow is the wisdom of how best to leverage Scala for robust, real-world development. Hence, my goal in this edition is to teach you how to use Scala effectively for a wide class of pragmatic problems, without covering every corner case in the language or the most advanced idioms in Scala code.

Finally, I wrote this book for professional programmers. I'll err on the side of tackling deeper technical topics, rather than keeping the material light. There are great alternative books if you prefer less depth. This is a book for if you are serious about mastering Scala professionally.

How to Read This Book

3 The first three chapters provide a fast tour of features without going into much depth. If you are experienced with Scala, skim these chapters to find new Scala 3 features that are introduced. The "3" icon in the lefthand margin makes it easy to find the content specific to Scala 3 throughout the book. If you are new to Scala, make sure you understand all the content in these chapters thoroughly.

Chapters 4–15 go back over the main features in depth. After learning this material, you'll be quite productive working with most Scala code bases. For you experienced readers, Chapters 5 and 6 will be the most interesting because they cover the new ways of abstracting over context (i.e., implicits). Chapters 7–12 are mostly the same for Scala 2 and 3, especially the material that explores Scala as an OOP language. However, you'll find Scala 3 changes throughout all these chapters. Also, all examples shown use the new, optional Scala 3 notation that omits most curly braces.

Chapters 16 and 17 explore the rest of Scala's sophisticated type system. I tried to cover the most important concepts you'll encounter in Chapter 16, with more advanced topics in Chapter 17. You'll find plenty of new Scala 3 content in these chapters.

Finally, pick and choose sections in Chapters 18–24 as you need to understand the concepts they cover. For example, when you encounter the popular, but advanced, subject of category theory, read Chapter 18. When you need to use concurrency and distribution for scalability, read Chapter 19. If you want to balance dynamic and static typing or you need to write domain-specific languages, read Chapter 20 or 21, respectively. If you want more information about tools in the Scala ecosystem and combining Java with Scala code, Chapter 22 offers tips. In a sense, Chapter 23 is a summary chapter that brings together my thoughts on using Scala effectively for long-term, scalable application development. Lastly, Chapter 24 introduces the powerful metaprogramming features of Scala, with significant changes in Scala 3.

For reference, an appendix summarizes optional new syntax conventions compared with traditional syntax. A list of references and an index finish the book.

Welcome to Programming Scala, Second Edition

Dean Wampler, November 2014

Programming Scala, first edition was published five years ago, in the fall of 2009. At the time, it was only the third book dedicated to Scala, and it just missed being the second by a few months. Scala version 2.7.5 was the official release, with version 2.8.0 nearing completion.

A lot has changed since then. At the time of this writing, the Scala version is 2.11.2. Martin Odersky, the creator of Scala, and Jonas Bonér, the creator of Akka, an actor-based concurrency framework, cofounded Typesafe (now Lightbend) to promote the language and tools built on it.

There are also a lot more books about Scala. So do we really need a second edition of this book? Many excellent beginner's guides to Scala are now available. A few advanced books have emerged. The encyclopedic reference remains *Programming in Scala*, second edition, by Odersky et al. (Artima Press).

Yet, I believe *Programming Scala*, second edition remains unique because it is a *comprehensive* guide to the Scala language and ecosystem, a guide for beginners to advanced users, and it retains the focus on the pragmatic concerns of working professionals. These characteristics made the first edition popular.

Scala is now used by many more organizations than in 2009 and most Java developers have now heard of Scala. Several persistent questions have emerged. Isn't Scala complex? Since Java 8 added significant new features found in Scala, why should I switch to Scala?

I'll tackle these and other, real-world concerns. I have often said that I was *seduced by Scala*, warts and all. I hope you'll feel the same way after reading *Programming Scala*, second edition.

Welcome to Programming Scala, First Edition

Dean Wampler and Alex Payne, September 2009

Programming languages become popular for many reasons. Sometimes, programmers on a given platform prefer a particular language, or one is institutionalized by a vendor. Most macOS programmers use Objective-C. Most Windows programmers use C++ and .NET languages. Most embedded-systems developers use C and C++.

Sometimes, popularity derived from technical merit gives way to fashion and fanaticism. C++, Java, and Ruby have been the objects of fanatical devotion among programmers.

Sometimes, a language becomes popular because it fits the needs of its era. Java was initially seen as a perfect fit for browser-based, rich client applications. Smalltalk captured the essence of object-oriented programming as that model of programming entered the mainstream.

Today, concurrency, heterogeneity, always-on services, and ever-shrinking development schedules are driving interest in functional programming. It appears that the dominance of object-oriented programming may be over. Mixing paradigms is becoming popular, even necessary.

We gravitated to Scala from other languages because Scala embodies many of the optimal qualities we want in a general-purpose programming language for the kinds of applications we build today: reliable, high-performance, highly concurrent internet and enterprise applications.

Scala is a multiparadigm language, supporting both object-oriented and functional programming approaches. Scala is scalable, suitable for everything from short scripts up to large-scale, component-based applications. Scala is sophisticated, incorporating state-of-the-art ideas from the halls of computer science departments worldwide. Yet

Scala is practical. Its creator, Martin Odersky, participated in the development of Java for years and understands the needs of professional developers.

We were seduced by Scala, by its concise, elegant, and expressive syntax and by the breadth of tools it put at our disposal. In this book, we strive to demonstrate why all these qualities make Scala a compelling and indispensable programming language.

If you are an experienced developer who wants a fast, thorough introduction to Scala, this book is for you. You may be evaluating Scala as a replacement for or complement to your current languages. Maybe you have already decided to use Scala, and you need to learn its features and how to use it well. Either way, we hope to illuminate this powerful language for you in an accessible way.

We assume that you are well versed in object-oriented programming, but we don't assume that you have prior exposure to functional programming. We assume that you are experienced in one or more other programming languages. We draw parallels to features in Java, C#, Ruby, and other languages. If you know any of these languages, we'll point out similar features in Scala, as well as many features that are new.

Whether you come from an object-oriented or functional programming background, you will see how Scala elegantly combines both paradigms, demonstrating their complementary nature. Based on many examples, you will understand how and when to apply OOP and FP techniques to many different design problems.

In the end, we hope that you too will be seduced by Scala. Even if Scala does not end up becoming your day-to-day language, we hope you will gain insights that you can apply regardless of which language you are using.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Programming Scala* third edition by Dean Wampler (O'Reilly). Copyright 2021 Dean Wampler, 978-1-492-07789-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

Getting the Code Examples

You can download the code examples from GitHub. Unzip the files to a convenient location. See the *README* file in the distribution for instructions on building and using the examples. I'll summarize those instructions in the first chapter.

Some of the example files can be run as scripts using the `scala` command. Others must be compiled into class files. A few files are only compatible with Scala 2, and a few files are additional examples that aren't built by `sbt`, the build tool. To keep these groups separate, I have adopted the following directory structure conventions:

src/main/scala/…/.scala*
: Are all Scala 3 source files built with `sbt`. The standard Scala file extension is *.scala*.

src/main/scala-2/…/.scala*
: Are all Scala 2 source files, some of which won't compile with Scala 3. They are not built with `sbt`.

src/test/…/.scala*
: Are all Scala 3 test source files built and executed with `sbt`.

src/script/…/.scala*
: Are all "Script" files that won't compile with `scalac`. Instead, they are designed for experimentation in the `scala` interactive command-line interface (CLI).

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/programming-scala-3*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://youtube.com/oreillymedia*

Acknowledgments for the Third Edition

Working with early builds of Scala 3, I often ran into unimplemented features and incomplete documentation. The members of the Scala community have provided valuable help while I learned what's new. The Scala Center at EPFL documentation for Dotty provided essential information. Since the second edition was published, the Scala Center as become the flagship organization driving the evolution of the language and core open source tooling for the ecosystem, while Lightbend continues to be the major provider of commercial support for Scala in the enterprise. I'm especially grateful to the reviewers of this edition—Seth Tisue, who also wrote the wonderful foreword for this edition, Daniel Hinojosa, Eric Loots, Ramnivas Laddad, and Lutz Hühnken—and for the advice and feedback from my editors at O'Reilly, Michele Cronin, Katherine Tozer, and Suzanne McQuade.

And special thanks again to Ann, who allowed me to consume so much of our personal time with this project. I love you!

Acknowledgments for the Second Edition

As I worked on this edition of the book, I continued to enjoy the mentoring and feedback from many of my Typesafe colleagues, plus the valuable feedback from people who reviewed the early-access releases. I'm especially grateful to Ramnivas Laddad, Kevin Kilroy, Lutz Hühnken, and Thomas Lockney, who reviewed drafts of the manuscript. Thanks to my longtime colleague and friend, Jonas Bonér, for writing an updated Foreword for the book.

And special thanks to Ann, who allowed me to consume so much of our personal time with this project. I love you!

Acknowledgments for the First Edition

As we developed this book, many people read early drafts and suggested numerous improvements to the text, for which we are eternally grateful. We are especially grateful to Steve Jensen, Ramnivas Laddad, Marcel Molina, Bill Venners, and Jonas Bonér for their extensive feedback.

Much of the feedback we received came through the Safari Rough Cuts releases and the online edition. We are grateful for the feedback provided by (in no particular order) Iulian Dragos, Nikolaj Lindberg, Matt Hellige, David Vydra, Ricky Clarkson, Alex Cruise, Josh Cronemeyer, Tyler Jennings, Alan Supynuk, Tony Hillerson, Roger Vaughn, Arbi Sookazian, Bruce Leidl, Daniel Sobral, Eder Andres Avila, Marek Kubica, Henrik Huttunen, Bhaskar Maddala, Ged Byrne, Derek Mahar, Geoffrey Wiseman, Peter Rawsthorne, Geoffrey Wiseman, Joe Bowbeer, Alexander Battisti, Rob Dickens, Tim MacEachern, Jason Harris, Steven Grady, Bob Follek, Ariel Ortiz, Parth Malwankar, Reid Hochstedler, Jason Zaugg, Jon Hanson, Mario Gleichmann, David Gates, Zef Hemel, Michael Yee, Marius Kreis, Martin Süsskraut, Javier Vegas, Tobias Hauth, Francesco Bochicchio, Stephen Duncan Jr., Patrik Dudits, Jan Niehusmann, Bill Burdick, David Holbrook, Shalom Deitch, Jesper Nordenberg, Esa Laine, Gleb Frank, Simon Andersson, Patrik Dudits, Chris Lewis, Julian Howarth, Dirk Kuzemczak, Henri Gerrits, John Heintz, Stuart Roebuck, and Jungho Kim. Many other readers for whom we only have usernames also provided feedback. We wish to thank Zack, JoshG, ewilligers, abcoates, brad, teto, pjcj, mkleint, dandoyon, Arek, rue, acangiano, vkelman, bryanl, Jeff, mbaxter, pjb3, kxen, hipertracker, ctran, Ram R., cody, Nolan, Joshua, Ajay, Joe, and anonymous contributors. We apologize if we have overlooked anyone!

Our editor, Mike Loukides, knows how to push and prod gently. He's been a great help throughout this crazy process. Many other people at O'Reilly were always there to answer our questions and help us move forward.

We thank Jonas Bonér for writing the Foreword for the book. Jonas is a longtime friend and collaborator from the aspect-oriented programming community. For years, he has done pioneering work in the Java community. Now he is applying his energies to promoting Scala and growing that community.

Bill Venners graciously provided the quote on the back cover. The first published book on Scala, *Programming in Scala*, that he cowrote with Martin Odersky and Lex Spoon, is indispensable for the Scala developer. Bill has also created the wonderful ScalaTest library.

We have learned a lot from fellow developers around the world. Besides Jonas and Bill, Debasish Ghosh, James Iry, Daniel Spiewak, David Pollack, Paul Snively, Ola Bini, Daniel Sobral, Josh Suereth, Robey Pointer, Nathan Hamblen, Jorge Ortiz, and others have illuminated dark corners with their blog entries, forum discussions, and personal conversations.

Dean thanks his colleagues at Object Mentor and several developers at client sites for many stimulating discussions on languages, software design, and the pragmatic issues facing developers in industry. The members of the Chicago Area Scala Enthusiasts (CASE) group have also been a source of valuable feedback and inspiration.

Alex thanks his colleagues at Twitter for their encouragement and superb work in demonstrating Scala's effectiveness as a language. He also thanks the Bay Area Scala Enthusiasts (BASE) for their motivation and community.

Most of all, we thank Martin Odersky and his team for creating Scala.

CHAPTER 1

Zero to Sixty: Introducing Scala

Let's start with a brief look at why you should investigate Scala. Then we'll dive in and write some code.

Why Scala?

Scala is a language that addresses the needs of the modern software developer. It is a statically typed, object-oriented, and functional mixed-platform language with a succinct, elegant, and flexible syntax, a sophisticated type system, and idioms that promote scalability from small tools to large sophisticated applications. So let's consider each of those ideas in more detail:

A Java Virtual Machine (JVM), JavaScript, and native language
: Scala started as a JVM language that exploits the performance and optimizations of the JVM, as well as the rich ecosystem of tools and libraries built around Java. More recently, Scala.js brings Scala to JavaScript, and Scala Native compiles Scala to native machine code, bypassing the JVM and JavaScript runtimes.

Object-oriented programming
: Scala fully supports *object-oriented programming* (OOP). Scala *traits* provide a clean way to implement code using *mixin composition*. Scala provides convenient and familiar OOP consistently for all types, even numeric types, while still enabling highly performant code generation.

Functional programming
: Scala fully supports *functional programming* (FP). FP has emerged as the best tool for thinking about problems of concurrency, big data, and general code correctness. Immutable values, first-class functions, code without side effects, and functional collections all contribute to concise, powerful, and correct code.

A sophisticated type system with static typing
: Scala's rich, static type system goes a long way toward bug-free code where mistakes are caught at compile time. With type inference, Scala code is often as concise as code in dynamically typed languages, yet inherently safer.

A succinct, elegant, and flexible syntax
: Verbose expressions in other languages become concise idioms in Scala. Scala provides several facilities for building *domain-specific languages* (DSLs), APIs that feel native to users.

Scalable architectures
: You can write tiny, single-file tools to large, distributed applications in Scala.

The name *Scala* is a contraction of the words *scalable language*. It is pronounced *scah-lah*, like the Italian word for *staircase*. Hence, the two *a*'s are pronounced the same.

Scala was started by Martin Odersky in 2001. The first public release was January 20, 2004. Martin is a professor in the School of Computer and Communication Sciences at the École Polytechnique Fédérale de Lausanne (EPFL). He spent his graduate years working in the group headed by Niklaus Wirth, of Pascal fame. Martin worked on Pizza, an early functional language on the JVM. He later worked on GJ, a prototype of what later became *generics* in Java, along with Philip Wadler, one of the designers of Haskell. Martin was recruited by Sun Microsystems to produce the reference implementation of `javac` with generics, the ancestor of the Java compiler that ships with the Java Developer Kit (JDK) today.

The Appeal of Scala

The growth of Scala users since it was introduced over 15 years ago confirms my view that Scala is a language for our time. You can leverage the maturity of the JVM and JavaScript ecosystems while enjoying state-of-the-art language features with a concise yet expressive syntax for addressing today's development challenges.

In any field of endeavor, the professionals need sophisticated, powerful tools and techniques. It may take a while to master them, but you make the effort because mastery is the key to your productivity and success.

I believe Scala is a language for professional developers. Not all Scala users are professionals, of course, but Scala is the kind of language a professional in our field needs, rich in features, highly performant, and expressive for a wide class of problems. It will take you a while to master Scala, but once you do, you won't feel constrained by your programming language.

3 Why Scala 3?

If you used Scala before, you used Scala 2, the major version since March 2006! Scala 3 aims to improve Scala in several ways.

First, Scala 3 strengthens Scala's foundations, especially in the type system. Martin Odersky and collaborators have been developing the *dependent object typing* (DOT) calculus, which provides a more sound foundation for Scala's type system. Scala 3 integrates DOT.

Second, Scala 2 has many powerful features, but sometimes they can be hard to use. Scala 3 improves the usability and safety of these features, especially implicits. Other language warts and puzzlers are removed.

Third, Scala 3 improves the consistency and expressiveness of Scala's language constructs and removes unimportant constructs to make the language smaller and more regular. Also, the previous experimental approach to macros is replaced with a new, principled approach to metaprogramming.

We'll call out these changes as we explore the corresponding language features.

Migrating to Scala 3

The Scala team has worked hard to make migration to Scala 3 from Scala 2 as painless as possible, while still allowing the language to make improvements that require breaking changes. Scala 3 uses the same standard library as Scala 2.13, eliminating a class of changes you would otherwise have to make to your code when upgrading. Hence, if necessary, I recommend upgrading to Scala 2.13 first to update your use of the standard library as needed, then upgrade to Scala 3.

In addition, to make the transition to breaking language changes as painless as possible, there are several ways to compile Scala 3 code that allows or disallows deprecated Scala 2 constructs. There are even compiler flags that will do some code rewrites automatically for you! See "Scala 3 Versions" on page 451 and "The scalac Command-Line Tool" on page 454 in Chapter 22 for details.

For a complete guide to migrating to Scala 3, see the Scala Center's Scala 3 Migration Guide.

Installing the Scala Tools You Need

There are many options for installing tools and building Scala projects. See Chapter 22 and the Scala website's Getting Started for more details on available tools and options for starting with Scala. Here, I'll focus on the simplest way to install the tools needed for the book's example code.

The examples target Scala 3 for the JVM. See the Scala.js and Scala Native websites for information on targeting those platforms.

You only need to install two tools:

- A recent Java JDK, version 8 or newer. Newer long-term versions are recommended, like versions 11 or 15 (the latest release at the time of this writing).
- `sbt`, the de facto build tool for Scala.

Follow the instructions for installing the JDK and `sbt` on their respective websites.

When we use the `sbt` command in "Using sbt" on page 5, it will bootstrap everything else needed, including the `scalac` compiler and the `scala` tool for running code.

Building the Code Examples

Now that you have the tools you need, you can download and build the code examples:

Get the code
: Download the code examples as described in "Getting the Code Examples" on page xxvii.

Start `sbt`
: Open a terminal and change to the root directory for the code examples. Type the command **`sbt test`** to download all the library dependencies you need, including the Scala compiler. This will take a while. Then `sbt` will compile the code and run the unit tests. You'll see lots of output, ending with a "success" message. If you run the command again, it should finish very quickly because it won't need to do anything again.

Congratulations! You are ready to get started.

> For most of the book, we'll use the Scala tools indirectly through `sbt`, which automatically downloads the Scala library and tools we need, including the required third-party dependencies.

More Tips

In your browser, it's useful to bookmark the Scala standard library's Scaladoc documentation. For your convenience, when I mention a type in the library, I'll often include a link to the corresponding Scaladoc entry.

Use the search field at the top of the page to quickly find anything in the docs. The documentation page for each type has a link to view the corresponding source code in Scala's GitHub repository, which is a good way to learn how the library was implemented. Look for a "Source" link.

Any text editor or integrated development environment (IDE) will suffice for working with the examples. Scala plug-ins exist for all the popular editors and IDEs, such as IntelliJ IDEA and Visual Studio Code. Once you install the required Scala plug-in, most environments can open your `sbt` project, automatically importing all the information they need, like the Scala version and library dependencies.

Support for Scala in many IDEs and text editors is now based on the Language Server Protocol (LSP), an open standard started by Microsoft. The Metals project implements LSP for Scala. The Metals website has installation details for your particular IDE or editor. In general, the community for your favorite editor or IDE is your best source of up-to-date information on Scala support.

> If you like working with Scala worksheets, many of the code examples can be converted to worksheets. See the code examples *README* for details.

Using sbt

Let's cover the basics of using `sbt`, which we'll use to build and work with the code examples.

When you start `sbt`, if you don't specify a task to run, `sbt` starts an interactive shell. Let's try that now and see a few of the available tasks.

In the listing that follows, the `$` is the shell command prompt (e.g., `bash`, `zsh`, or the Window's command shell), where you start the `sbt` command, the `>` is the default `sbt` interactive prompt, and the `#` starts a comment. You can type most of these commands in any order:

```
$ sbt
> help       # Describe commands.
> tasks      # Show the most commonly used, available tasks.
> tasks -V   # Show ALL the available tasks.
> compile    # Incrementally compile the code.
> test       # Incrementally compile the code and run the tests.
> clean      # Delete all build artifacts.
> console    # Start the interactive Scala environment.
> run        # Run one of the "main" methods (applications) in the project.
> show x     # Show the value of setting or task "x".
> exit       # Quit the sbt shell (also control-d works).
```

The `sbt` project for the code examples is actually configured to show the following as the `sbt` prompt:

```
sbt:programming-scala-3rd-ed-code-examples>
```

To save space, I'll use the more concise prompt, `>`, when showing `sbt` sessions.

> A handy `sbt` technique is to add a tilde, `~`, at the front of any command. Whenever file changes are saved to disk, the command will be rerun. For example, I use `~test` all the time to keep compiling my code and running my tests. Since, `sbt` uses an incremental compiler, you don't have to wait for a full rebuild every time. Break out of these loops by hitting Return.

The `scala` CLI command has a built-in *REPL* (read, eval, print, loop). This is a historical term, going back to LISP. It's more accurate than *interpreter*, which is sometimes used. Scala code isn't interpreted. It is always compiled and then run, even when using the interactive REPL where bits of code at a time are entered and executed. Hence, I'll use the term REPL when referring to this use of the `scala` CLI. You can invoke it using the `console` command in `sbt`. We'll do this a lot to work with the book's code examples. The Scala REPL prompt is `scala>`. When you see that prompt in code examples, I'm using the REPL.

Before starting the REPL, `sbt console` will build your project and set up the classpath with your compiled artifacts and dependent libraries. This convenience means it's rare to use the `scala` REPL outside of `sbt` because you would have to set up the classpath yourself.

To exit the `sbt` shell, use `exit` or Ctrl-D. To exit the Scala REPL, use `:quit` or Ctrl-D.

> Using the Scala REPL is a very effective way to experiment with code idioms and to learn an API, even non-Scala APIs. Invoking it from `sbt` using the `console` task conveniently adds project dependencies and the compiled project code to the classpath for the REPL.

3 I configured the compiler options for the code examples (in `build.sbt`) to pass `-source:future`. This flag causes warnings to be emitted for constructs that are still allowed in Scala 3.0, but it will be removed in Scala 3.1 or deprecated with planned removal in a subsequent release. I'll cite specific examples of planned transitions as we encounter them. There are several language versions that can be used with the `-source` option. See "Scala 3 Versions" on page 451 for details), especially when starting your own code migrations to Scala 3.

Because I'm using the "aggressive" -source:future option, you'll see warnings when using sbt console that won't appear in other Scala 3 projects that don't use this setting.

Running the Scala Command-Line Tools Using sbt

When the Scala 3 command-line tools are installed separately (see "Command-Line Interface Tools" on page 452 for details), the Scala compiler is called scalac and the REPL is called scala. We will let sbt run them for us, although I'll show you how to run them directly as well.

Let's run a simple Scala program. Consider this "script" from the code examples:

```scala
// src/script/scala/progscala3/introscala/Upper1.scala

class Upper1:
  def convert(strings: Seq[String]): Seq[String] =
    strings.map((s: String) => s.toUpperCase)

val up = new Upper1()
val uppers = up.convert(List("Hello", "World!"))
println(uppers)
```

Most listings, like this one, start with a comment that contains the file path in the code examples, so it's easy for you to locate the file. Not all examples have files, but if you see a listing with no path comment, it often continues where the previous listing left off.

I'll explain the details of this code shortly, but let's focus now on running it.

Change your current working directory to the root of the code examples. Start sbt and run the console task. Then, use the :load command to compile and run the contents of the file. In the next listing, the $ is the terminal's prompt, > is the sbt prompt, scala> is the Scala REPL's prompt, and the ellipses (...) are for suppressed output:

```
$ sbt
...
> console
...
scala> :load src/script/scala/progscala3/introscala/Upper1.scala
List(HELLO, WORLD!)
...
```

And thus we have satisfied the prime directive of the Programming Book Authors Guild, which states that our first program must print "Hello World!"

All the code examples that we'll use in the REPL will have paths that start with *src/script*. However, in most cases you can copy and paste code from any of the source files to the REPL prompt.

If you have the `scala` REPL for Scala installed separately, you can enter `scala` at the terminal prompt, instead of the separate `sbt` and `console` steps. However, most of the example scripts won't run with `scala` outside of `sbt` because `sbt console` includes the libraries and compiled code in the classpath, which most of the scripts need.[1]

Here is a more complete REPL session to give you a sense of what you can do. Here I'll combine `sbt` and `console` into one step (some output elided):

```
$ sbt console
...
scala> :help
The REPL has several commands available:

:help                    print this summary
:load <path>             interpret lines in a file
:quit                    exit the REPL
:type <expression>       evaluate the type of the given expression
:doc <expression>        print the documentation for the given expression
:imports                 show import history
:reset                   reset the REPL to its initial state, ...

scala> val s = "Hello, World!"
val s: String = Hello, World!

scala> println("Hello, World!")
Hello, World!

scala> 1 + 2
val res0: Int = 3

scala> s.con<tab>
concat   contains   containsSlice   contentEquals

scala> s.contains("el")
val res1: Boolean = true

scala> :quit
$    # back at the terminal prompt. "Control-D" also works.
```

The list of commands available and the output of `:help` may change between Scala releases.

1 If you are unfamiliar with the JVM ecosystem, the classpath is a list of locations to search for compiled code, like libraries.

We assigned a string, `"Hello, World!"`, to a variable named `s`, which we declared as an immutable value using the `val` keyword. The `println` method prints a string to the console, followed by a line feed.

When we added two numbers, we didn't assign the result to a variable, so the REPL made up a name for us, `res0`, which we could use in subsequent expressions.

The REPL supports tab completion. The input shown is used to indicate that a tab was typed after `s.con`. The REPL responded with a list of methods on `String` that could be called. The expression was completed with a call to the `contains` method.

The type of something is given by its name, a colon, and then the type. We didn't explicitly specify any type information here because the types could be inferred. When you provide types explicitly or when they are inferred and shown for you, they are called *type declarations*.[2] The output of the REPL shows several examples.

When a type is added to a declaration, the syntax `name: String`, for example, is used instead of `String name`. The latter would be more ambiguous in Scala because of *type inference*, where type information can be omitted from the code yet inferred by the compiler.

> Showing the types in the REPL is very handy for learning the types that Scala infers for particular expressions. It's one example of exploration that the REPL enables.

See "Command-Line Interface Tools" on page 452 for more information about the command-line tools, such as using the `scala` CLI to run compiled code outside of `sbt`.

A Taste of Scala

We've already seen a bit of Scala as we discussed tools, including how to print "Hello World!" The rest of this chapter and the two chapters that follow provide a rapid tour of Scala features. As we go, we'll discuss just enough of the details to understand what's going on, but many of the deeper background details will have to wait for later chapters. Think of this tour as a primer on Scala syntax and a taste of what programming in Scala is like day to day.

2 Sometimes the type information is called an *annotation*, but this is potentially confusing with another concept of annotations that we'll see, so I'll avoid using this term for types. *Type ascriptions* is another term.

When I introduce a type in the Scala library, find its entry in the Scaladoc. Scala 3 uses the Scala 2.13 library with a few minor additions.

Scala follows common comment conventions. A `// comment` goes to the end of a line, while a `/* comment */` can cross line boundaries. Comments intended to be included in Scaladoc documentation use `/** comment */`.

Files named *src/test/scala/.../*Suite.scala* are tests written using MUnit (see "Testing Tools" on page 458). To run all the tests, use the `sbt` command `test`. To run just one particular test, use `testOnly path`, where *`path`* is the fully qualified type name for the test:

```
> testOnly progscala3.objectsystem.equality.EqualitySuite
[info] Compiling 1 Scala source to ...
progscala3.objectsystem.equality.EqualitySuite:
  + The == operator is implemented with the equals method 0.01s
  + The != operator is implemented with the equals method 0.001s
  ...
[info] Passed: Total 14, Failed 0, Errors 0, Passed 14
[success] Total time: 1 s, completed Feb 29, 2020, 5:00:41 PM
>
```

The corresponding source file is *src/test/scala/progscala3/objectsystem/equality/EqualitySuite.scala*. Note that `sbt` follows Apache Maven conventions that directories for compiled source code go under *src/main/scala* and tests go under *src/test/scala*. After that is the *package* definition, *progscala3.objectsystem.equality*, corresponding to file path *progscala3/objectsystem/equality*. Packages organize code hierarchically. The test inside of the file is defined as a class named `EqualitySuite`.

Scala packages, names, and file organization mostly follow Java conventions. Java requires that the directory path and filename must match the declared package and a single public class within the file. Scala doesn't require conformance to these rules, but it is conventional to follow them, especially for larger code bases. The code examples follow these conventions.

Finally, many of the files under *src/main/scala* define entry points (such as `main` methods), the starting points for running those small applications. You can execute them in one of several ways.

First, use `sbt`'s `run` command. It will find all the entry points and prompt you to pick which one. Note that `sbt` will only search *src/main/scala* and *src/main/java*. When you compile and run tests, *src/test/scala* and *src/test/java* are searched. The *src/script* is ignored by `sbt`.

Let's use another example we'll study later in the chapter, *src/main/scala/progscala3/introscala/UpperMain2.scala*. Invoke `run hello world`, where `run` is the `sbt` task and `hello world` are arbitrary arguments that will be passed to a program we'll choose from the list that is printed for us (over 50 choices!). Enter the number shown for `progscala3.introscala.Hello2`:

```
> run hello world
...

Multiple main classes detected, select one to run:

 ...
 [38] progscala3.introscala.Hello2
 ...

38    <--- What you type!

[info] running progscala3.introscala.Hello2 hello world
HELLO WORLD
[success] Total time: 2 s, completed Feb 29, 2020, 5:08:18 PM
```

This program converts the input arguments to uppercase and prints them.

A second way to run this program is to use `runMain` and specify the same fully qualified path to the main class that was shown, `progscala3.introscala.Hello2`. This skips the prompt:

```
> runMain progscala3.introscala.Hello2 hello world again!
...
[info] running progscala3.introscala.Hello2
HELLO WORLD AGAIN!
[success] Total time: 0 s, completed Feb 29, 2020, 5:18:05 PM
>
```

This code is already compiled, so you can also run it outside of `sbt` with the `scala` command. Now the correct classpath must be provided, including all dependencies. This example is easy; the classpath only needs the output root directory for all of the compiled `.class` files. I'm using a shell variable here to fit the line in the space; change the `3.0.0` to match the actual version of Scala used:[3]

[3] I will periodically update the code examples as new Scala releases come out. The version will be set in the file *build.sbt*, the `scalaVersion` string. The other way to tell is to just look at the contents of the `target` directory.

```
$ cp="target/scala-3.0.0/classes/"
$ scala -classpath $cp progscala3.introscala.Hello2 Hello Scala World!
HELLO SCALA WORLD!
```

There's one final alternative we can use. As we'll see shortly, `UpperMain2.scala` defines a single entry point. Because of this, the `scala` command can actually load the source file directly, compile, and run it in one step, without a `scalac` step first. We won't need the `-classpath` argument now, but we will need to specify the file instead of the fully qualified name used previously:

```
$ scala src/main/scala/progscala3/introscala/UpperMain2.scala Hello World!
HELLO WORLD!
```

Let's explore the implementations of these examples. First, here is `Upper1.scala` again:

```scala
// src/script/scala/progscala3/introscala/Upper1.scala

class Upper1:
  def convert(strings: Seq[String]): Seq[String] =
    strings.map((s: String) => s.toUpperCase)

val up = new Upper1()
val uppers = up.convert(List("Hello", "World!"))
println(uppers)
```

We declare a class, `Upper1`, using the `class` keyword, followed by a colon (`:`). The entire class body is indented on the next two lines.

> If you know Scala already, you might ask why are there no curly braces `{…}` and why is a colon (`:`) after the name `Upper1`? I'm using the new *optional braces* syntax that I'll discuss in more depth in "New Scala 3 Syntax—Optional Braces" on page 31.

`Upper1` contains a method called `convert`. Method definitions start with the `def` keyword, followed by the method name and an optional parameter list. The method signature ends with an optional return type. The return type can be inferred in many cases, but adding the return type explicitly, as shown, provides useful documentation for the reader and also avoids occasional surprises from the type inference process.

> I'll use *parameters* to refer to the list of things a method expects to be passed when you call it. I'll use *arguments* to refer to values you actually pass to it when making the call.

Type definitions are specified using `name: type` syntax. The parameter list is `strings: Seq[String]` and the return type of the method is `Seq[String]`, after the parameter list.

An equals sign (=) separates the method signature from the method body. Why an equals sign?

One reason is to reduce ambiguity. If you omit the return type, Scala can infer it. If the method takes no parameters, you can omit the parentheses too. So the equal sign makes parsing unambiguous when either or both of these features are omitted. It's clear where the signature ends and the method body begins.

The `convert` method takes a *sequence* (`Seq`) of zero or more input strings and returns a new sequence, where each of the input strings is converted to uppercase. `Seq` is an abstraction for collections that you can iterate through. The actual kind of sequence returned by this method will be the same kind that was passed into it as an argument, like `Vector` or `List` (both of which are immutable collections).

Collection types like `Seq[T]` are *parameterized types*, where `T` is the type of the elements in the sequence. Scala uses square brackets (`[…]`) for parameterized types, whereas several other languages use angle brackets (`<…>`).

`List[T]` is an immutable linked list. Accessing the head of a `List` is *O(1)*, while accessing an arbitrary element at position *N* is *O(N)*. `Vector[T]` is a *subtype* of `Seq[T]` with almost *O(1)* for all access patterns.

> Scala allows angle brackets to be used in *identifiers*, like method and variable names. For example, defining a "less than" method and naming it `<` is common. To avoid ambiguity, Scala reserves square brackets for parameterized types so that characters like `<` and `>` can be used as identifiers.

Inside the body of `convert`, we use the `map` method to iterate over the elements and apply a transformation to each one and then construct a new collection with the results.

The function passed to the `map` method to do the transformation is an unnamed (anonymous) *function literal* of the form `(parameters) => body`:

```
(s: String) => s.toUpperCase
```

It takes a parameter list with a single `String` named `s`. The body of the function literal is after the "arrow" `=>`. The body calls the `toUpperCase` method on `s`.[4] The result of

4 This method takes no arguments, so parentheses can be omitted.

this call is automatically returned by the function literal. In Scala, the last expression in a function, method, or other block is the return value. (The `return` keyword exists in Scala, but it can only be used in methods, not in anonymous functions like this one. It is only used for early returns in the middle of methods.)

> ## Methods Versus Functions
>
> Following the convention in most OOP languages, the term *method* is used to refer to a function defined within a type. Methods have an implied `this` reference to the object as an additional argument when they are called. Like most OOP languages, the syntax used is `this.method_name(other_args)`.
>
> Functions, like the `(s: String) => s.toUpperCase` example, are not tied to a particular type. Occasionally, I'll use the term *function* to refer to methods and non-methods generically, when the distinction doesn't matter, to avoid the awkwardness of saying "functions or methods."

On the JVM, functions are implemented using JVM lambdas, as the REPL will indicate to you:

```
scala> (s: String) => s.toUpperCase
val res0: String => String = Lambda$7775/0x00000008035fc040@7673711e=
```

Note that the REPL treats this function like any other value and gives it a synthesized name, `res0`, when you don't provide one yourself (e.g., `val f = (s: String) => s.toUpperCase`). Read-only values are declared using the `val` keyword.

Back to `Upper1.scala`, the last two lines, which are outside the class definition, create an instance of `Upper1` named `up`, using `new Upper1()`. Then `up` is used to convert two strings to uppercase. Finally, the resulting sequence `uppers` is printed with `println`. Normally, `println` expects a single string argument, but if you pass it an object, like a `Seq`, the `toString` method will be called. If you run `sbt console`, then copy and paste the contents of the `Upper1.scala` file, the REPL will tell you that the actual type of the `Seq[String]` is `List[String]` (a linked list).

So *src/script/…/Upper1.scala* is intended for copying and pasting (or using `:load`) in the REPL. Let's look at another implementation that is compiled and then run. I added `Main` to the source filename. Note the path to the source file now contains *src/main* instead of *src/script*:

```
// src/main/scala/progscala3/introscala/UpperMain1.scala
package progscala3.introscala                                    ❶

object UpperMain1:
  def main(params: Array[String]): Unit =                        ❷
    print("UpperMain1.main: ")
```

```
    params.map(s => s.toUpperCase).foreach(s => printf("%s ",s))
    println("")

  def main(params: Array[String]): Unit =                            ❸
    print("main: ")
    params.map(s => s.toUpperCase).foreach(s => printf("%s ",s))
    println("")

  @main def Hello(params: String*): Unit =                           ❹
    print("Hello: ")
    params.map(s => s.toUpperCase).foreach(s => printf("%s ",s))
    println("")
```

❶ Declare the package location, `progscala3.introscala`.

❷ Declare a `main` method, a program entry point, inside an `object`. I'll explain what an `object` is shortly.

❸ Declare an alternative `main` entry point as a top-level method, outside any `object`, but scoped to the current package, `progscala3.introscala`.

❹ Declare an entry point method where we can use a different name and we have more flexible options for the argument list.

Packages work much like they do in other languages. They provide a *namespace* for scoping declarations and access to them. Here, the declarations exist in the `progscala3.introscala` package.

You have probably seen classes in other languages that encapsulate *members*, meaning methods, fields (or attributes) that hold state, and so forth. In many languages, the entry point where the program starts is a `main` method. In Java, this method is defined inside a class and declared `static`, meaning it is not tied to any one instance. You can reference any static definition with the syntax `UpperMain1.main`, to use our example.

The pattern of static declarations in classes is so pervasive that Scala builds it into the language. Instead, we declare an `object UpperMain1`, using the `object` keyword. Then we declare `main` and other members using the same syntax we would use in classes. There is no `static` keyword in Scala.

This file has *three* entry points. The first one, `UpperMain1.main`, is how you declare entry points in Scala 2. Following Java conventions, the name `main` is required and it is declared with an `Array[String]` parameter for the user-specified arguments, even if the program takes no arguments or takes specific arguments in a specific order, like an integer followed by two strings. You have to handle parsing the arguments. Also, `Arrays` in Scala are *mutable*, which can be a source of bugs. Using immutable arguments is inherently safer. All these issues are addressed in the last entry point, `Hello`, as we'll discuss in a moment.

Inside `UpperMain1.main`, we print the name of the method first (without a newline), which will be useful for contrasting how these three entry points are invoked. Then we map over the input arguments (`params`), converting them to uppercase and returning a new collection. Finally, we use another collections method called `foreach` to iterate over the new collection and print each string using `printf`, which expects a formatting string and arguments, `s` here, to compose the final string.[5]

Let's run with `UpperMain1.main`:

```
> runMain progscala3.introscala.UpperMain1 Hello World!
UpperMain1.main: HELLO WORLD!
>
```

The method `main` itself is not part of the qualified path, just the enclosing object `UpperMain1`.

3 Scala 3 introduces two new features for greater flexibility. First, you can declare methods, variables, etc., outside objects and classes. This is how the second `main` method is declared, but otherwise it works like `UpperMain1.main`. It is scoped differently, as we can see when we use it:

```
> runMain progscala3.introscala.UpperMain1$package Hello World!
main: HELLO WORLD!
>
```

Note how the definition is scoped to the package plus source filename! If you rename the file to something like *FooBar.scala* and recompile, then the command becomes `runMain progscala3.introscala.FooBar$package`.... Adding the source file to the scope avoids collisions with other definitions in the same package scope, but with different source files. However, having `$package` in the name is inconvenient for Linux and macOS shells like `bash`, so I don't recommend defining an entry point this way.

3 Instead, I recommend the second, new Scala 3 feature, an alternative way of defining entry points, which is shown by our third entry point, the `Hello` method. The `@main` annotation marks this method as an entry point. Note how we refer to it when we run it:

5 This `printf`-style formatting is so common in programming languages, I'll assume it needs no further explanation. If it's new to you, see the link in the paragraph for details.

```
> runMain progscala3.introscala.Hello Hello World!
Hello: HELLO WORLD!
>
```

Now the method name is used. Normally you don't name methods starting with an uppercase letter, but it's useful for entry points if you want invocation commands to look similar to Java invocations like `java...progscala3.introscala.Hello...`. `Hello`, which is also declared outside an object, but this isn't required.

The new `@main` entry points have several advantages. They reduce boilerplate when defining them. They can be defined with parameter lists that match the expected arguments, such as sequences, strings, and integers. Here, we want zero or more string arguments. The `*` in `params: String*` means zero or more `Strings` (called *repeated parameters*), which will be passed to the method body, where `params` is implemented with an immutable `Seq[String]`. Mutable `Arrays` are avoided.

Note that the type of the return value of all three methods is `Unit`. For now, think of `Unit` as analogous to `void` in other languages, meaning nothing useful is returned.

> Because there are three entry points defined in this file, we can't use `scala` to parse and run this file in one step. That's why I used `UpperMain2` earlier, instead. We'll explore that file shortly, where we'll see it has one and only one entry point.

Declaring `UpperMain1` as an `object` makes it a *singleton*, meaning there will always be only one instance of it, which the Scala runtime will create for us. You can't create your own instances with `new`.

Scala makes the *singleton design pattern* a first-class member of the language. In most ways, these `object` declarations are just like other `class` declarations, but they are used when you need one and only one instance to hold some methods and fields, as opposed to the situation where you need multiple instances, each with fields of unique values per instance and methods that operate on a single instance at a time.

The singleton design pattern has drawbacks. It's hard to replace a singleton instance with a *test double* in unit tests, and forcing all computation through a single instance raises concerns about thread safety and limits scalability options. However, we'll see plenty of examples in the book where objects are used effectively.

> To avoid confusion, I'll use *instance*, rather than *object*, when I refer to an instance created from a class with `new` or the single instance of an `object`. Because classes and objects are so similar, I'll use *type* generically for them. All the types we'll see, like `String`, are implemented as classes or objects.

Returning to the implementation details, note the function we passed to `map`:

```
s => s.toUpperCase
```

Our previous example used `(s: String) => s.toUpper(s)`. Most of the time, Scala can infer the types of parameters for function literals because the context provided by `map` tells the compiler what type to expect. So the type declaration `String` isn't needed.

The `foreach` method is used when we want to process each element and perform only side effects, without returning a new value. Here we print a string to *standard output* (without a newline after each one). In contrast, `map` returns a new value for each element (and side effects should be avoided). The last `println` call prints a newline before the program exits.

The notion of side effects means that the function we pass to `foreach` does something to affect the state outside the local context. We could write to a database or to a file, or print to the console, or launch missiles…

Look again at the second line inside each method, how concise it is where we compose operations together. Sequencing transformations lets us create concise, powerful programs, as we'll see over and over again.

We haven't needed to import any library items yet, but Scala imports operate much like similar constructs in other languages. Scala automatically imports many commonly used types and object members, like `Seq`, `List`, `Vector`, and the `print*` methods we used, which are actually methods in an `object` called `scala.Console`. Most of these things that are automatically imported are defined in a library object called `Predef`.

For completeness, let's discuss how to compile and run the example outside `sbt`. First you use `scalac` to compile to a JVM-compatible `.class` file. Often, multiple class files are generated. Then you use `scala` to run it.

If you installed the Scala command-line tools separately (see "Command-Line Interface Tools" on page 452 for details), run the following two commands (ignoring the `$` shell prompt) in a terminal window at the root of the project:

```
$ scalac src/main/scala/progscala3/introscala/UpperMain1.scala
$ scala -classpath . progscala3.introscala.Hello Hello compiled World!
Hello: HELLO COMPILED WORLD!
```

You should now have new directories *progscala3/introscala* with several *.class* and *.tasty* files, including a file named `UpperMain1.class`. Class files are processed by the JVM, and *tasty* files are an intermediate representation used by the compiler. Scala must generate valid JVM byte code and files. For example, the directory structure must match the package structure. The `-classpath .` option adds the current directory to the search classpath, although `.` is the default.

Allowing `sbt` to compile it for us instead, we need a different `-classpath` argument to reflect the directory where `sbt` writes class files:

```
$ scala -classpath target/scala-3.0.0/classes progscala3.introscala.Hello Bye!
BYE!
```

Let's do one last version to see a few other useful ways of working with collections for this scenario. This is the version we ran previously:

```
// src/main/scala/progscala3/introscala/UpperMain2.scala
package progscala3.introscala

@main def Hello2(params: String*): Unit =
  val output = params.map(_.toUpperCase).mkString(" ")
  println(output)
```

Instead of using `foreach` to print each transformed string as before, we map the sequence of strings to a new sequence of strings and then call a convenience method, `mkString`, to concatenate the strings into a final string. There are three `mkString` methods. One takes no arguments. The second version takes a single parameter to specify the delimiter between the elements (`" "` in our example). The third version takes three parameters, a leftmost prefix string, the delimiter, and a rightmost suffix string. Try changing the code to use `mkString("[", ", ", "]")`.

Note the function passed to `map`. The following function literals are essentially the same:

```
s => s.toUpperCase
_.toUpperCase
```

Rather than providing a name for the single argument, we can use _ as a placeholder. This generalizes to functions with two or more arguments, where each use of _ takes the place of one argument. This means that placeholders can't be used if it's necessary to refer to any one of the arguments more than once.

As before, we can run this code with sbt using runMain progscala3.introscala.Hello2…. We also saw previously that we can use the scala command to compile and run it in one step because it has a single entry point:

```
$ scala src/main/scala/progscala3/introscala/UpperMain2.scala last Hello World!
LAST HELLO WORLD!
```

A Sample Application

Let's finish this chapter by exploring several more seductive features of Scala using a sample application. We'll use a simplified hierarchy of geometric shapes, which we will send to another object for drawing on a display. Imagine a scenario where a game engine generates scenes. As the shapes in the scene are completed, they are sent to a display subsystem for drawing.

To begin, we define a Shape class hierarchy:

```
// src/main/scala/progscala3/introscala/shapes/Shapes.scala
package progscala3.introscala.shapes

case class Point(x: Double = 0.0, y: Double = 0.0)                  ❶

abstract class Shape():                                             ❷
  /**
   * Draw the shape.
   * @param f is a function to which the shape will pass a
   * string version of itself to be rendered.
   */
  def draw(f: String => Unit): Unit = f(s"draw: $this")            ❸

case class Circle(center: Point, radius: Double) extends Shape      ❹

case class Rectangle(lowerLeft: Point, height: Double, width: Double) ❺
      extends Shape

case class Triangle(point1: Point, point2: Point, point3: Point)    ❻
      extends Shape
```

❶ Declare a class for two-dimensional points. No members are defined, so we omit the colon (:) at the end of the class signature.

❷ Declare an abstract class for geometric shapes. It needs a colon because it defines a method draw.

❸ Implement a draw method for rendering the shapes. The comment uses the Scaladoc conventions for documenting the method, which are similar to Javadoc conventions.

❹ A circle with a center and radius, which subtypes (`extends`) `Shape`.

❺ A rectangle with a lower-left point, height, and width. To keep it simple, the sides are parallel to the horizontal and vertical axes.

❻ A triangle defined by three points.

Let's unpack what's going on.

The parameter list after the `Point` class name is the list of constructor parameters. In Scala, the whole body of a `class` or `object` is the constructor, so you list the parameters for the constructor after the class name and before the class body.

The `case` keyword before the class declaration causes special handling. First, each constructor parameter is automatically converted to a read-only (immutable) field for `Point` instances. In other words, it's as if we put `val` before each field declaration. When you instantiate a `Point` instance named `point`, you can read the fields using `point.x` and `point.y`, but you can't change their values. Attempting to write `point.y = 3.0` causes a compilation error.

You can also provide default values for constructor and method parameters. The `= 0.0` after each parameter definition specifies `0.0` as the default. Hence, the user doesn't have to provide them explicitly, but they are inferred left to right. This implies that when you define a default value for one parameter, you must also do this for all parameters to its right.

3 Finally, case-class instances are constructed without using `new`, such as `val p = Point(…)`. Scala 3 adds the ability to omit `new` when constructing instances for most noncase classes too. We used `new Upper1()` previously, but omitting `new` would also work. We'll do that from now on, but there are situations we'll see where `new` is still necessary.

Let's use `sbt console` to play with these types. I recommend you do this with most of the book's examples. Recall that `scala>` is the `scala` REPL prompt. When you see a line starting with `// src/script/`, it's not part of the session, but it shows you where you can find this code in the examples distribution.

```
$ sbt
> console
...
// src/script/scala/progscala3/introscala/TryShapes.scala

scala> import progscala3.introscala.shapes.*

scala> val p00 = Point()
val p00: progscala3.introscala.shapes.Point = Point(0.0,0.0)
```

```
scala> val p20 = Point(2.0)
val p20: progscala3.introscala.shapes.Point = Point(2.0,0.0)

scala> val p20b = Point(2.0)
val p20b: progscala3.introscala.shapes.Point = Point(2.0,0.0)

scala> val p02 = Point(y = 2.0)
val p02: progscala3.introscala.shapes.Point = Point(0.0,2.0)

scala> p20 == p20b
val res0: Boolean = true

scala> p20 == p02
val res1: Boolean = false
```

3 Like many other languages, import statements use the * character as a wildcard to import everything in the progscala3.introscala.shapes package. This is a change from Scala 2, where _ was used as the wildcard. However, it is still allowed for backward compatibility, until a future release of Scala 3. Recall that we also saw _ used in function literals as an anonymous placeholder for a parameter, instead of using an explicit name.

In the definition of p00, no arguments are specified, so Scala uses 0.0 for both of them. (However, you must provide the empty parentheses.) When one argument is specified, Scala applies it to the leftmost argument, x, and uses the default value for the remaining argument, as shown for p20 and p20b. We can even specify the arguments by name. The definition of p02 uses the default value for x but specifies the value for y, using Point(y = 2.0).

> I use named arguments like this a lot, even when it isn't required, because Point(x = 0.0, y = 2.0) makes my code much easier to read and understand.

While there is no class body for Point, another feature of the case keyword is that the compiler automatically generates several methods for us, including commonly used toString, equals, and hashCode methods. The output shown for each point—e.g., Point(2.0,0.0)—is the default toString output. The equals and hashCode methods are difficult for most developers to implement correctly, so autogeneration of these methods is a real benefit. However, you can provide your own definitions for any of these methods, if you prefer.

When we asked if p20 == p20b and p20 == p02, Scala invoked the generated equals method, which compares the instances for equality by comparing the fields. (In some languages, == just compares *references*. Do p20 and p20b point to the same spot in memory?)

The last feature of case classes that we'll mention now is that the compiler also generates a *companion object*, a singleton object of the same name, for each case class. In other words, we declared the class Point, and the compiler also created an object Point.

> You can define companions yourself. Any time an object and a class have the same name and they are defined in the same file, they are companions.

The compiler also adds several methods to the companion object automatically, one of which is named apply. It takes the same parameter list as the constructor. When I said earlier that it is unnecessary to use new to create instances of case classes like Point, this works because the companion method Point.apply(...) gets called.

This is true for any instance, either a declared object or an instance of a class, not just for case-class companion objects. If you put an argument list after it, Scala looks for a corresponding apply method to call. Therefore, the following two lines are equivalent:

```
val p1 = Point.apply(1.0, 2.0)    // Point is the companion object here!
val p2 = Point(1.0, 2.0)          // Same!
```

It's a compilation error if no apply method exists for the instance, or the provided argument list is incompatible with what apply expects.[6]

The Point.apply method is effectively a *factory* for constructing Points. The behavior is simple here; it's just like calling the Point class constructor. The companion object generated is equivalent to this:

```
object Point:
  def apply(x: Double = 0.0, y: Double = 0.0) = new Point(x, y)      ❶
  ...
```

❶ Here's our first example where new is still needed. Without it, the compiler would think we are calling Point.apply again on the righthand side, creating an infinite recursion!

6 The name apply originated from early theoretical work on computation, specifically the idea of *function application*.

You can add methods to the companion object, including overloaded `apply` methods. Just declare `object Point:` explicitly and add the methods. The default `apply` method will still be generated, unless you define it explicitly yourself.

A more sophisticated `apply` method might instantiate a different subtype with specialized behavior, depending on the argument supplied. For example, a data structure might have an implementation that is optimal for a small number of elements and a different implementation that is optimal for a larger number of elements. The `apply` method can hide this logic, giving the user a single, simplified interface. Hence, putting an `apply` method on a companion object is a common idiom for defining a factory method for a class hierarchy, whether or not case classes are involved.

We can also define an instance `apply` method in any `class`. It has whatever meaning we decide is appropriate for instances. For example, `Seq.apply(index: Int)` retrieves the element at position `index`, counting from zero.

> To recap, when an argument list is put after an `object` or `class` instance, Scala looks for an `apply` method to call where the parameter list matches the provided arguments. Hence, anything with an `apply` method behaves like a *function*—e.g., `Point(2.0, 3.0)`.
>
> A companion object `apply` method is a factory method for the companion class instances. A class `apply` method has whatever meaning is appropriate for instances of the class; for example, `Seq.apply(index: Int)` returns the item at position `index`.

Continuing with the example, `Shape` is an abstract class. We can't instantiate an abstract class, even if none of the members is abstract. `Shape.draw` is defined, but we only want to instantiate concrete shapes: `Circle`, `Rectangle`, and `Triangle`.

The parameter `f` for `draw` is a function of type `String => Unit`. We saw `Unit` previously. It is a real type, but it behaves roughly like `void` in other languages.

The idea is that callers of `draw` will pass a function that does the actual drawing when given a string representation of the shape. For simplicity, we just use the string returned by `toString`, but a structured format like JSON would make more sense in a real application.

> When a function returns `Unit`, it is totally side-effecting. There's nothing useful returned from the function, so it can only perform side effects on some state, like performing input or output (I/O).

Normally in FP, we prefer *pure* functions that have no side effects and return all their work as their return value. These functions are far easier to reason about, test, compose, and reuse. Side effects are a common source of bugs, so they should be used carefully.

> Use side effects only when necessary and in well-defined places. Keep the rest of the code pure.

`Shape.draw` is another example where a function is passed as an argument, just like we might pass instances of `Strings`, `Points`, etc. We can also return functions from methods and from other functions. Finally, we can assign functions to variables. This means that functions are first class in Scala because they can be used just like strings and other instances. This is a powerful tool for building composable yet flexible software.

When a function accepts other functions as parameters or returns functions as values, it is called a *higher-order function* (HOF).

You could say that `draw` defines a *protocol* that all shapes have to support, but users can customize. It's up to each shape to serialize its state to a string representation through its `toString` method. The `f` method is called by `draw`, which constructs the final string using an *interpolated string*.

An interpolated string starts with `s` before the opening double quote: `s"draw: ${this.toString}"`. It builds the final string by substituting the result of the expression `this.toString` into the larger string. Actually, we don't need to call `toString`; it will be called for us. So we can use just `${this}`. However, now we're just referring to a variable, not a longer expression, so we can drop the curly braces and just write `$this`. Hence, the interpolated string becomes `s"draw: $this"`.

> If you forget the `s` before the interpolated string, you'll get the literal output `draw: $this`, with no interpolation.

Continuing with the example, `Circle`, `Rectangle`, and `Triangle` are concrete subtypes (also called subclasses) of `Shape`. They have no class bodies because `Shape` and the methods generated for `case` classes define all the methods we need, such as the `toString` methods required by `Shape.draw`.

In our simple program, the `f` we will pass to `draw` will just write the string to the console, but in a real application, `f` could parse the string and render the shape to a display, write JSON to a web service, etc.

Even though this will be a single-threaded application, let's anticipate what we might do in a concurrent implementation by defining a set of possible `Messages` that can be exchanged between modules:

```
// src/main/scala/progscala3/introscala/shapes/Messages.scala
package progscala3.introscala.shapes

sealed trait Message                                                   ❶
case class Draw(shape: Shape) extends Message                          ❷
case class Response(message: String) extends Message                   ❸
case object Exit extends Message                                       ❹
```

❶ Declare a `trait` called `Message`. A `trait` is similar to an abstract base class. (We'll explore the differences later.) All messages exchanged are subtypes of `Message`. I explain the `sealed` keyword in a moment.

❷ A message to draw the enclosed `Shape`.

❸ A message with a response to a previous message received from a caller.

❹ Signal termination. `Exit` has no state or behavior of its own, so it is declared a `case object`, since we only need one instance of it. It functions as a signal to trigger a state change, termination in this case.

The `sealed` keyword means that we can only define subtypes of `Message` in the same file. This prevents bugs where users define their own `Message` subtypes that would break the code we're about to see in the next file! These are all the allowed messages, known in advance.

Recall that `Shape` was not declared `sealed` earlier because we intend for people to create their own subtypes of it. There could be an infinite number of `Shape` subtypes, in principle. So, use *sealed hierarchies* when all the possible variants are fixed.

Now that we have defined our shapes and messages types, let's define an `object` for processing messages:

```
// src/main/scala/progscala3/introscala/shapes/ProcessMessages.scala
package progscala3.introscala.shapes

object ProcessMessages:                                                ❶
  def apply(message: Message): Message =                               ❷
    message match                                                      ❸
      case Exit =>
        println(s"ProcessMessage: exiting...")
```

```
      Exit
    case Draw(shape) =>
      shape.draw(str => println(s"ProcessMessage: $str"))
      Response(s"ProcessMessage: $shape drawn")
    case Response(unexpected) =>
      val response = Response(s"ERROR: Unexpected Response: $unexpected")
      println(s"ProcessMessage: $response")
      response
```

❶ We only need one instance, so we use an `object`, but it would be easy enough to make this a `class` and instantiate as many as we need for scalability and other needs.

❷ Define the `apply` method that takes a `Message`, processes it, then returns a new `Message`.

❸ Match on the incoming message to determine what to do with it.

The `apply` method introduces a powerful feature call: *match expressions* with *pattern matching*:

```
message match
  case Exit =>
    expressions
  case Draw(shape) =>
    expressions
  case Response(unexpected) =>
    expressions
```

The whole `message match:...` is an expression, meaning it will return a value, a new `Message` for us to return to the caller. A `match` expression consists only of `case` clauses, which do pattern matching on the message passed into the function, followed by expressions to invoke for a match.

The `match` expressions work a lot like `if/else` expressions but are more powerful and concise. When one of the patterns matches, the block of expressions after the arrow (`=>`) is evaluated, up to the next `case` keyword or the end of the whole expression. Matching is *eager*; the first match wins.

If the case clauses don't cover all possible values that can be passed to the `match` expression, a `MatchError` is thrown at runtime. Fortunately, the compiler can detect and warn you that the case clauses are not *exhaustive*, meaning they don't handle all possible inputs. Note that our `sealed` hierarchy of messages is crucial here. If a user could create a new subtype of `Message`, our `match` expression would no longer cover all possibilities. Hence, a bug would be introduced in this code!

A powerful feature of pattern matching is the ability to extract data from the object matched, sometimes called *deconstruction* (the inverse of construction). Here, when the input message is a Draw, we extract the enclosed Shape and assign it to the variable shape. Similarly, if Response is detected, we extract the message as unexpected, so named because ProcessMessages doesn't expect to receive a Response!

Now let's look at the expressions invoked for each case match:

```
def apply(message: Message): Message =
  message match
    case Exit =>                                                    ❶
      println(s"ProcessMessage: exiting...")
      Exit
    case Draw(shape) =>                                             ❷
      shape.draw(str => println(s"ProcessMessage: $str"))
      Response(s"ProcessMessage: $shape drawn")
    case Response(unexpected) =>                                    ❸
      val response = Response(s"ERROR: Unexpected Response: $unexpected")
      println(s"ProcessMessage: $response")
      response
```

❶ We're done, so print a message that we're exiting and return Exit to the caller.

❷ Call draw on shape, passing it an anonymous function that knows what to do with the string generated by draw. In this case, it just prints the string to the console and sends a Response to the caller.

❸ ProcessMessages doesn't expect to receive a Response message from the caller, so it treats it as an error. It returns a new Response to the caller.

One of the tenets of OOP is that you should never use if or match statements that match on instance type because inheritance hierarchies evolve. When a new subtype is introduced without also fixing these statements, they break. Instead, *polymorphic methods* should be used. So, is the pattern-matching code just discussed an *antipattern*?

Pattern Matching Versus Subtype Polymorphism

Pattern matching plays a central role in FP just as *subtype polymorphism* (i.e., overriding methods in subtypes) plays a central role in OOP. The combination of functional-style pattern matching with polymorphic dispatch, as used here, is a powerful combination that is a benefit of a mixed paradigm language like Scala.

Our match expression only knows about Shape and draw. We don't match on specific subtypes of Shape. This means our code won't break if a user adds a new Shape to the hierarchy.

In contrast, the case clauses match on specific subtypes of `Message`, but we protected ourselves from unexpected change by making `Message` a `sealed` hierarchy. We know by design all the possible `Messages` exchanged.

Hence, we have combined polymorphic dispatch from OOP with pattern matching, a workhorse of FP. This is one way that Scala elegantly integrates these two programming paradigms!

Finally, here is the `ProcessShapesDriver` that runs the example:

```
// src/main/scala/progscala3/introscala/shapes/ProcessShapesDriver.scala
package progscala3.introscala.shapes

@main def ProcessShapesDriver =                                          ❶
  val messages = Seq(                                                    ❷
    Draw(Circle(Point(0.0,0.0), 1.0)),
    Draw(Rectangle(Point(0.0,0.0), 2, 5)),
    Response(s"Say hello to pi: 3.14159"),
    Draw(Triangle(Point(0.0,0.0), Point(2.0,0.0), Point(1.0,2.0))),
    Exit)

  messages.foreach { message =>                                          ❸
    val response = ProcessMessages(message)
    println(response)
  }
```

❶ An entry point for the application. It takes no arguments, and if you provide arguments when you run this application, they will be ignored.

❷ A sequence of messages to send, including a `Response` in the middle that will be considered an error in `ProcessMessages`. The sequence ends with `Exit`.

❸ Iterate through the sequence of messages, call `ProcessMessages.apply()` with each one, then print the response.

Let's try it. Some output elided:

```
> runMain progscala3.introscala.shapes.ProcessShapesDriver
[info] running progscala3.introscala.shapes.ProcessShapesDriver
ProcessMessage: draw: Circle(Point(0.0,0.0),1.0)
Response(ProcessMessage: Circle(Point(0.0,0.0),1.0) drawn)
ProcessMessage: draw: Rectangle(Point(0.0,0.0),2.0,5.0)
Response(ProcessMessage: Rectangle(Point(0.0,0.0),2.0,5.0) drawn)
ProcessMessage: Response(ERROR: Unexpected Response: Say hello to pi: 3.14159)
Response(ERROR: Unexpected Response: Say hello to pi: 3.14159)
ProcessMessage: draw: Triangle(Point(0.0,0.0),Point(2.0,0.0),Point(1.0,2.0))
Response(ProcessMessage: Triangle(Point(0.0,0.0), ...) drawn)
ProcessMessage: exiting...
Exit
[success] ...
```

Make sure you understand how each message was processed and where each line of output came from.

Recap and What's Next

We introduced many of the powerful and concise features of Scala. As you explore Scala, you will find other useful resources. You will find links for libraries, tutorials, and various papers that describe features of the language.

Next we'll continue our introduction to Scala features, emphasizing the various concise and efficient ways of getting lots of work done.

CHAPTER 2
Type Less, Do More

This chapter continues our tour of Scala features that promote succinct, flexible code. We'll discuss organization of files and packages, importing other types, variable and method declarations, a few particularly useful types, and miscellaneous syntax conventions.

3 New Scala 3 Syntax—Optional Braces

If you have prior Scala experience, Scala 3 introduces a new *optional braces* syntax that makes it look a lot more like Python or Haskell, where curly braces, `{…}`, are replaced with *significant indentation*. The examples in the previous chapter and throughout the book use it.

This syntax is more concise and easier to read. It will also appeal to Python developers who learn Scala because it will feel a little more familiar to them (and vice versa). When data scientists who use Python and data engineers who use Scala work together, this can help them collaborate. Also, since many new programmers learn Python as a first language, learning Scala will be that much easier.

There is also a new syntax for control structures like `for` loops and `if` expressions. For example, there is `if condition then…` instead of the older `if (condition)…`. Also, there is `for…do println(…)` instead of `for {…} println(…)`.

The disadvantage of these changes is that they are strictly not necessary. Some breaking changes in Scala 3 are necessary to move the language forward, but you could argue these syntax changes aren't essential. You also have to be careful to use spaces or tabs consistently for indentation. I will mention other pros and cons as we explore examples.

By default, you can mix the old and new syntax conventions. If you omit braces, then indentation is significant. If you want to require use of braces and ignore indentation, perhaps for consistency with an existing Scala 2 code base, use the `-no-indent` compiler flag.

If you want to enforce parentheses around conditionals, as in Scala 2, use the flag `-old-syntax`. If you want allow optional parentheses for conditionals and require the new keywords `then` and `do`, use `-new-syntax`.

Finally, the compiler can rewrite your code to use whichever style you prefer. Add the `-rewrite` compiler flag; for example, use `-rewrite -new-syntax -indent`.

These syntax conventions have been controversial among Scala veterans. I opposed them at first, but now that I have worked with them, I believe the advantages outweigh the disadvantages. I enjoy writing code this way. It makes my code look cleaner, for a language that is already concise. Hence, I chose to use the new conventions throughout this edition of the book.

Table A-1 provides examples of the old versus new syntax.

I'll finish with one feature of the new syntax that you won't use very often. When you have a long method, type, conditional, `match` expression, etc., it might be hard to see where it ends and subsequent definitions begin. You can add an optional `end…` at the same indentation level as the opening tokens. Here are some examples, although they are too small to need the `end` markers:

```
class Foo:
  def uppify(s: String): String =
    s.toUpperCase
  end uppify  // Optional. Note the name of the method is used.
end Foo       // Optional. The name of the type is used.

val i = 1
if i > 0 then
  println(sequence)
end if        // Optional

while i < 10 do
  i += 1
end while     // Optional

for j <- 0 to 10 do   // Loop from 0 to 10, assign each one to "j".
  println(j)
end for       // Optional
```

The `end` keyword is followed by the corresponding identifier or one of the following keywords: `if`, `while`, `for`, `match`, `try`, `new`, `this`, `val`, `given`, or `extension`.[1]

Semicolons

Semicolons are expression delimiters and they are inferred. Scala treats the end of a line as the end of an expression, except when it can infer that the expression continues to the next line, such as in the following example that builds up a string:

```
val str = Seq("STILL", "MORE", "HELLO", "WORLD")
  .map(_.toLowerCase)
  .mkString("[", ", ", "]")
```

Conversely, you can put multiple expressions on the same line, separated by semicolons.

Variable Declarations

Scala allows you to decide whether a variable is immutable (read-only) or not (read-write) when you declare it. We've already seen that an immutable "variable" is declared with the keyword `val`:

```
val seq: Seq[String] = Seq("This", "is", "Scala")
val array: Array[String] = Array("This", "is", "Scala")
```

In Scala, most variables are actually references to heap-allocated objects. Neither `seq` nor `array` can be changed to refer to different objects. Also, the elements of `seq` cannot be modified, as `Seq` is immutable. However, `Array`s are not immutable, so *you can modify the the array elements themselves*:

```
scala> val array: Array[String] = Array("This", "is", "Scala")
val array: Array[String] = Array(This, is, Scala)

scala> array = Array("Bad!")
1 |array = Array("Bad!")
  |^^^^^^^^^^^^^^^^^^^^^
  |Reassignment to val array

scala> array(1) = "still is"

scala> array
val res1: Array[String] = Array(This, still is, Scala)
```

1 The code examples file *src/script/scala/progscala3/IndentationSyntax.scala* has examples for all cases.

Avoid using mutable types like `Array`, as mutation is a common source of bugs in concurrent programs.

A `val` must be initialized when it is declared, except in certain contexts like abstract fields in type declarations.

Similarly, a mutable variable is declared with the keyword `var`, and it must also be initialized immediately (in most cases), even though it can be changed later:

```
scala> var seq2: Seq[String] = Seq("This", "is", "Scala")
var seq2: Seq[String] = List(This, is, Scala)

scala> seq2 = Seq("No", "longer", "Scala")
seq2: Seq[String] = List(No, longer, Scala)
```

We changed `seq2` to refer to a different `Seq` object in memory, but both `Seq` objects are still immutable and cannot be changed.

For performance reasons, some languages treat so-called *primitive* values differently than reference objects. In Java for example, `char`, `byte`, `short`, `int`, `long`, `float`, `double`, and `boolean` values are not heap-allocated, with a pointer to the location in memory. Instead, they are *in-place*, such as inside the allocated memory for an enclosing instance or part of method call *stack frames*. Indeed, there are no objects or references for them, just the raw values.

However, Scala tries to be consistently object-oriented, so these types are actually objects with methods in source code, just like reference types (see "Reference Versus Value Types" on page 252). However, the code generated by the compiler uses underlying primitives when possible, giving you the performance benefit they provide without sacrificing the convenience of object orientation.

Consider the following REPL session, where we define a `Human` class with an immutable name, but a mutable age (because people age, I guess). The parameters are declared with `val` and `var`, respectively, making them both fields in `Human`:

```
// src/script/scala/progscala3/typelessdomore/Human.scala
scala> class Human(val name: String, var age: Int)
// defined class Human

scala> val p = Human("Dean Wampler", 29)
val p: Human = Human@165a128d

scala> p.name
val res0: String = Dean Wampler

scala> p.name = "Buck Trends"
1 |p.name = "Buck Trends"
```

```
  |^^^^^^^^^^^^^^
  |Reassignment to val name

scala> p.name
val res1: String = Dean Wampler

scala> p.age
val res2: Int = 29

scala> p.age = 30

scala> p.age = 30; p.age  // Use semicolon to join two expressions...
val res3: Int = 30

scala> p.age = 31; p.age
val res4: Int = 31
```

> Recall that `var` and `val` only specify whether the reference can be changed to refer to a different instance (`var`) or not (`val`). They don't specify whether or not the instance they reference is mutable.

Use immutable values whenever possible to eliminate a class of bugs caused by mutability. For example, a mutable instance is dangerous as a key in hash-based maps. If the instance is mutated, the output of the `hashCode` method will change, so the corresponding value won't be found at the original location.

More common is unexpected behavior when an instance you are using is being changed by another thread. Borrowing a phrase from quantum physics, these bugs are *spooky action at a distance*. Nothing you are doing locally accounts for the unexpected behavior; it's coming from somewhere else.

These are the most pernicious bugs in multithreaded programs, where synchronized access to a shared, mutable state is required, but difficult to get right. Using immutable values eliminates these issues.

Ranges

Sometimes we need a sequence of numbers from some start to finish. A `Range` is just what we need. You can create ranges for several types: `Int`, `Long`, `Char`, `BigInt`, which represent integers of arbitrary size, and `BigDecimal`, which represents floating-point numbers of arbitrary size.

`Float` and `Double` ranges are not supported because truncation and rounding in floating-point arithmetic makes range calculations error prone.

You can create ranges with an inclusive or exclusive upper bound, and you can specify an interval not equal to one (some output elided to fit):

```
scala> 1 to 10                // Int range inclusive, interval of 1, (1 to 10)
val res0: scala.collection.immutable.Range.Inclusive = Range 1 to 10

scala> 1 until 10             // Int range exclusive, interval of 1, (1 to 9)
val res1: Range = Range 1 until 10

scala> 1 to 10 by 3           // Int range inclusive, every third.
val res2: Range = inexact Range 1 to 10 by 3

scala> (1 to 10 by 3).foreach(println)   // To see the value.
1
4
7
10

scala> 10 to 1 by -3          // Int range inclusive, every third, counting down.
val res3: Range = Range 10 to 1 by -3

scala> 1L to 10L by 3         // Long
val res4: ...immutable.NumericRange[Long] = NumericRange 1 to 10 by 3

scala> ('a' to 'g' by 3).foreach(println)
a
d
g
```

Try creating examples for `BigInt` and `BigDecimal`.

Partial Functions

A `PartialFunction[A,B]` is a special kind of function with its own literal syntax. `A` is the type of the single parameter the function accepts and `B` is the return type.

The literal syntax for a `PartialFunction` consists of only `case` clauses, which we saw in "A Sample Application" on page 20, that do pattern matching on the input to the function. No function parameter is shown explicitly, but when each input is processed, it is passed to the body of the partial function.

For comparison, here is a regular function, `func`, that does pattern matching, and a similar partial function, `pfunc`. Both are adapted from the example we explored in "A Sample Application" on page 20, and I've elided a few details to fit the space:

```
// src/script/scala/progscala3/typelessdomore/FunctionVsPartialFunction.scala
scala> import progscala3.introscala.shapes.*

scala> val func: Message => String = message => message match
     |   case Exit => "Got Exit"
     |   case Draw(shape) => s"Got Draw($shape)"
```

```
     |   case Response(str) => s"Got Response($str)"
val func: progscala3.introscala.shapes.Message => String = Lambda$8843/0x...

scala> val pfunc: PartialFunction[Message, String] =
     |   case Exit => "Got Exit"
     |   case Draw(shape) => s"Got Draw($shape)"
     |   case Response(str) => s"Got Response($str)"
val pfunc: PartialFunction[...shapes.Message, String] = <function1>

scala> func(Draw(Circle(Point(0.0,0.0), 1.0)))
     | pfunc(Draw(Circle(Point(0.0,0.0), 1.0)))
     | func(Response(s"Say hello to pi: 3.14159"))
     | pfunc(Response(s"Say hello to pi: 3.14159"))
val res0: String = Got Draw(Circle(Point(0.0,0.0),1.0))
val res1: String = Got Draw(Circle(Point(0.0,0.0),1.0))
val res2: String = Got Response(Say hello to pi: 3.14159)
val res3: String = Got Response(Say hello to pi: 3.14159)
```

I won't always show the output printed by the REPL for definitions like `func` and `pfunc`, but it's useful to see the differences here.

Function definitions can be a little harder to read than method definition. The function `func` is a named function of type `Message => String`. The equal sign starts the body, `message => message match…`

The partial function, `pfunc`, is simpler. Its type is `PartialFunction[Message, String]`. There is no argument list, just a set of `case` match clauses, which happen to be identical to the clauses in `func`.

The concept of a *partial function* may sound fancy, but it is quite simple. A partial function will handle only some of the possible inputs, not all possible inputs. So don't send it something it doesn't know how to handle. A classic example from mathematics is division, *x/y*, which is undefined when the denominator *y* is 0. Hence, division is a partial function.

If a partial function is called with an input that doesn't match one of the `case` clauses, a `MatchError` is thrown at runtime. Both `func` and `pfunc` are actually total because they handle all possible `Message` arguments. Try commenting out the `case Exit` clauses in both `func` and `pfunc`. You'll get a compiler warning for `func` because the compiler can determine that the match clauses don't handle all possible inputs. It won't complain about `pfunc` because partial matching is by design.

You can test if a `PartialFunction` will match an input using the `isDefinedAt` method. This function avoids the risk of throwing a `MatchError` exception.

You can also chain `PartialFunctions` together: `pf1.orElse(pf2).orElse(pf3)…`. If `pf1` doesn't match, then `pf2` is tried, then `pf3`, etc. A `MatchError` is only thrown if none of them matches.

Let's explore these points with the following example:

```scala
// src/script/scala/progscala3/typelessdomore/PartialFunctions.scala

val pfs: PartialFunction[Matchable,String] =                         ❶
  case s:String => "YES"
val pfd: PartialFunction[Matchable,String] =                         ❷
  case d:Double => "YES"

val pfsd = pfs.orElse(pfd)                                           ❸
```

❶ A partial function that only matches on strings.

❷ A partial function that only matches on doubles.

❸ Combine the two functions to construct a new partial function that matches on strings and doubles.

Let's try these functions. A helper function `tryPF` is used to try the partial function and catch possible `MatchError` exceptions. So a string is returned for both success and failure:

```scala
def tryPF(
    x: Matchable, f: PartialFunction[Matchable,String]): String =
  try f(x)
  catch case _: MatchError => "ERROR!"

assert(tryPF("str", pfs)  == "YES")
assert(tryPF("str", pfd)  == "ERROR!")
assert(tryPF("str", pfsd) == "YES")
assert(tryPF(3.142, pfs)  == "ERROR!")
assert(tryPF(3.142, pfd)  == "YES")
assert(tryPF(3.142, pfsd) == "YES")
assert(tryPF(2, pfs)      == "ERROR!")
assert(tryPF(2, pfd)      == "ERROR!")
assert(tryPF(2, pfsd)     == "ERROR!")

assert(pfs.isDefinedAt("str")  == true)
assert(pfd.isDefinedAt("str")  == false)
assert(pfsd.isDefinedAt("str") == true)
assert(pfs.isDefinedAt(3.142)  == false)
assert(pfd.isDefinedAt(3.142)  == true)
assert(pfsd.isDefinedAt(3.142) == true)
assert(pfs.isDefinedAt(2)      == false)
assert(pfd.isDefinedAt(2)      == false)
assert(pfsd.isDefinedAt(2)     == false)
```

Note that integers are not handled by any combination.

Finally, we can *lift* a partial function into a regular (total) function that returns an `Option` or a `Some(value)` when the partial function is defined for the input argument or `None` when it isn't. This is a type-safe alternative to returning a `value` or `null`, respectively. We can also *unlift* a single-parameter function. Here is a REPL session to see them in action:

```
scala> val fs = pfs.lift
val fs: Any => Option[String] = <function1>

scala> fs("str")
val res0: Option[String] = Some(YES)

scala> fs(3.142)
val res1: Option[String] = None

scala> val pfs2 = fs.unlift
val pfs2: PartialFunction[Any, String] = <function1>

scala> pfs2("str")
val res3: String = YES

scala> tryPF(3.142, pfs2)    // Use tryPF we defined above
val res4: String = ERROR!
```

Method Declarations

Let's explore method definitions, using a modified version of our `Shapes` hierarchy from before.

Method Default and Named Parameters

Here is an updated `Point` case class:

```
// src/main/scala/progscala3/typelessdomore/shapes/Shapes.scala
package progscala3.typelessdomore.shapes

case class Point(x: Double = 0.0, y: Double = 0.0):              ❶
  def shift(deltax: Double = 0.0, deltay: Double = 0.0) =        ❷
    copy(x + deltax, y + deltay)
```

❶ Define `Point` with default initialization values (as before). For case classes, both `x` and `y` are automatically immutable (`val`) fields.

❷ A new `shift` method for creating a new `Point` instance, offset from the existing `Point`.

A `copy` method is also created automatically for case classes. It allows you to construct new instances of a case class while specifying just the fields that are changing. This is very handy for case classes with a lot of fields:

```
scala> val p1 = Point(x = 3.3, y = 4.4)      // Used named arguments.
val p1: Point = Point(3.3,4.4)

scala> val p2 = p1.copy(y = 6.6)             // Copied with a new y value.
val p2: Point = Point(3.3,6.6)
```

Named arguments make client code more readable. They also help avoid bugs when a parameter list has several fields of the same type or it has a lot of parameters. It's easy to pass values in the wrong order. Of course, it's better to avoid such parameter lists in the first place.

Methods with Multiple Parameter Lists

Next, consider the following changes to `Shape.draw()`:

```
abstract class Shape():
  def draw(offset: Point = Point(0.0, 0.0))(f: String => Unit): Unit =
    f(s"draw: offset = $offset, shape = ${this}")
```

`Circle`, `Rectangle`, and `Triangle` are unchanged and not shown.

Now `draw` has two parameter lists, each of which has a single parameter, rather than a single parameter list with two parameters. The first parameter list lets you specify an offset point where the shape will be drawn. It has a default value of `Point(0.0, 0.0)`, meaning no offset. The second parameter list is the same as in the original version of `draw`, a function that does the drawing.

You can have as many parameter lists as you want, but it's rare to use more than two.

So why allow more than one parameter list? Multiple lists promote a very nice block-structure syntax when the last parameter list takes a single function. Here's how we might invoke this new `draw` method to draw a `Circle` at an offset:

```
val s = Circle(Point(0.0, 0.0), 1.0)
s.draw(Point(1.0, 2.0))(str => println(str))
```

Scala lets us replace parentheses with curly braces around a supplied argument (like a function literal) for a parameter list that has a single parameter. So this line can also be written this way:

```
s.draw(Point(1.0, 2.0)){str => println(str)}
```

Suppose the function literal is too long for one line or it has multiple expressions. We can rewrite it this way:

```
s.draw(Point(1.0, 2.0)) { str =>
  println(str)
}
```

Or equivalently:

```
s.draw(Point(1.0, 2.0)) {
  str => println(str)
}
```

3 If you use the traditional curly brace syntax for Scala, it looks like a typical block of code we use with constructs like `if` and `for` expressions, method bodies, etc. However, the {…} block is still a function literal we are passing to `draw`.

So this *syntactic sugar* of using {…} instead of (…) looks better with longer function literals; they look more like the block structure syntax we know.

By default, the new optional braces syntax doesn't work here:

```
scala> s.draw(Point(1.0, 2.0)):
     |   str => println(str)
2 |  str => println(str)
  |      ^
  |      parentheses are required around the parameter of a lambda
  |      This construct can be rewritten automatically under -rewrite.
1 |s.draw(Point(1.0, 2.0)):
  |^
  |not a legal formal parameter
2 |  str => println(str)
```

However, there is a compiler flag, `-language:experimental.fewerBraces`, that enables this capability, but it is experimental because this feature is not fully mature, at least in Scala 3.0.

Back to using parentheses or braces, if we use the default value for `offset`, the first set of parentheses is still required. Otherwise, the function would be parsed as the `offset`, triggering an error.

```
s.draw() {
  str => println(str)
}
```

To be clear, `draw` could just have a single parameter list with two values. If so, the client code would look like this:

```
s.draw(Point(1.0, 2.0), str => println(str))
```

It works, but it's not as elegant. It would also be less convenient for using the default value for the `offset`.

By the way, we can can simplify our expressions even more: `str => println(str)` is an anonymous function that takes a single string argument and passes it to `println`. Although `println` is implemented as a method in the Scala library, it can also be used as a function that takes a single string argument! Hence, the following two lines behave the same:

```
s.draw(Point(1.0, 2.0))(str => println(str))
s.draw(Point(1.0, 2.0))(println)
```

To be clear, these are not identical, but they do the same thing. In the first example, we pass an anonymous function that calls `println`. In the second example, we use `println` as a *named* function directly. Scala handles converting methods to functions in situations like this.

Another advantage of allowing two or more parameter lists is that we can use one or more lists for normal parameters and other lists for *using clauses* (formerly known as *implicit parameter lists*). These are parameter lists declared with the `using` keyword. When the methods are called, we can either explicitly specify arguments for these parameters or we can let the compiler fill them in using suitable values that are in scope. Using clauses provides a more flexible alternative to parameters with default values. Let's explore an example from the Scala library that uses this mechanism, `Futures`.

A Taste of Futures

Our application in "A Sample Application" on page 20 was designed for concurrent execution, drawing the shapes while computing more of them concurrently. However, we made it synchronous for simplicity. Let's look at one tool we could use for concurrency, `scala.concurrent.Future`. A `Future` allows us to encapsulate some work to do, start it running in parallel, then continue with other work. We process the `Future`'s results when they are done. One way to process the results is to provide a *callback* that will be invoked when the result is ready. We'll defer discussion of the rest of the API, as well as other ways of writing concurrent programs, until Chapter 19.

The following example fires off five work items concurrently and handles the results as they finish:

```
// src/script/scala/progscala3/typelessdomore/Futures.scala
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global         ❶
import scala.util.{Failure, Success}

def sleep(millis: Long) = Thread.sleep(millis)                    ❷

(1 to 5).foreach { i =>
  val future = Future {                                            ❸
    val duration = (math.random * 1000).toLong
    sleep(duration)
    if i == 3 then throw RuntimeException(s"$i -> $duration")
    duration
  }
  future.onComplete {                                              ❹
    case Success(result)    => println(s"Success! #$i -> $result")
    case Failure(throwable) => println(s"FAILURE! #$i -> $throwable")
  }
}
```

```
sleep(1000)  // Wait long enough for the "work" to finish.
println("Finished!")
```

❶ We'll discuss this import later in this section.

❷ A `sleep` method to simulate staying busy for an amount of time.

❸ Pass a block of work to the `scala.concurrent.Future.apply` method. It calls `sleep` with a `duration`, a randomly generated number of milliseconds between 0 and 1,000, which it will also return. However, if `i` equals 3, we throw an exception to observe how failures are handled.

❹ Use `onComplete` to assign a partial function to handle the computation result. Notice that the expected output is either `scala.util.Success` wrapping a value or `scala.util.Failure` wrapping an exception.

`Success` and `Failure` are subtypes of `scala.util.Try`, which encapsulates `try {…} catch {…}` clauses with less boilerplate. We can handle successful code and possible exceptions more uniformly. We'll explore these classes further in "Try: When There Is No Do" on page 239.

When we iterate through a `Range` of integers from 1 to 5, inclusive, we construct a `Future` with a block of work to do. `Future.apply` returns a new `Future` instance immediately. The body is executed asynchronously on another thread. The `onComplete` callback we register will be invoked when the body completes.

A final `sleep` call waits before exiting to allow the futures to finish.

A sample run in the REPL might go like this, where the order of the results and the numbers on the righthand side are nondeterministic:

```
Success! #2 -> 178
Success! #1 -> 207
FAILURE! #3 -> java.lang.RuntimeException: 3 -> 617
Success! #5 -> 738
Success! #4 -> 938
Finished!
```

You might wonder about the body of work we're passing to `Future.apply`. Is it a function or something else? Here is part of the declaration of `Future.apply`:

```
apply[T](body: => T)(/* explained below */): Future[T]
```

Note how the type of `body` is declared, `=> T`. This is called a *by-name parameter*. We are passing something that will return a `T` instance, but we want to evaluate `body` *lazily*. Go back to the example body we passed to `Future.apply`. We did not want that code evaluated before it was passed to `Future.apply`. We wanted it evaluated inside the `Future` after construction. This is what by-name parameters do for us. We can pass a block of code that will be evaluated only when needed, similar to passing a function. The implementation of `Future.apply` evaluates this code.

3 OK, let's finally get back to *using clauses*. Recall the second import statement:

```
import scala.concurrent.ExecutionContext.Implicits.global
```

`Future` methods use an `ExecutionContext` to run code in separate threads, providing concurrency. This is a hook you could use to insert your own implementation, if needed. Most `Future` methods take an `ExecutionContext` argument. Here's the complete `Future.apply` declaration (using Scala 3 syntax, although the library is compiled with Scala 2):

```
apply[T](body: => T)(using executor: ExecutionContext): Future[T]
```

In the actual Scala 2 library, the `implicit` keyword is used instead of `using`. The second parameter list is called a *using clause* in Scala 3. It was an *implicit parameter list* in Scala 2.[2]

Because this parameter is in its own parameter list starting with `using` (or `implicit`), users of `Future.apply` don't have to pass a value explicitly. This reduces code boilerplate. We imported the default `ExecutionContext` value, which is declared as `given` (or `implicit` in Scala 2). A value declared with the `given/implicit` keyword means it can be used automatically by the compiler for `using/implicit` parameters. In this case, the given `ExecutionContext.global` uses a thread pool with a *work-stealing algorithm* to balance the load and optimize performance.

We can tailor how threads are used by passing our own `ExecutionContext` explicitly:

```
Future(work)(using someCustomExecutionContext)
```

Alternatively, we can declare our own `given` value that will be used implicitly when `Future.apply` is called:

```
given myEC = MyCustomExecutionContext(arguments)
...
val future = Future(work)
```

Our `given` value will take precedence over `ExecutionContext.global`.

The `Future.onComplete` method we used also has the same using clause:

```
abstract def onComplete[U](
  f: (Try[T]) => U)(using executor: ExecutionContext): Unit
```

So when `ExecutionContext.global` is imported into the current scope, the compiler will use it when methods are called that have a using clause with an `ExecutionCon`

2 If you're new to Scala, this duplication is confusing, but we'll justify these changes starting in Chapter 5.

text parameter, unless we specify a value explicitly. For this to work, only given instances that are type compatible with the parameter will be considered.

If this idea of using clauses, implicits, etc., was a little hard to grasp, know that we'll return to it in Chapter 5. We'll work through the details along with examples of the design problems they help us solve.

Nesting Method Definitions and Recursion

Method definitions can also be *nested*. This is useful when you want to refactor a lengthy method body into smaller methods, but the helper methods aren't needed outside the original method. Nesting them inside the original method means they are invisible to the rest of the code base, including other methods in the enclosing type.

Here is an example for a factorial calculator:

```
// src/script/scala/progscala3/typelessdomore/Factorial.scala

def factorial(i: Int): BigInt =
  def fact(i: Int, accumulator: BigInt): BigInt =
    if i <= 1 then accumulator
    else fact(i - 1, i * accumulator)

  fact(i, BigInt(1))

(0 to 5).foreach(i => println(s"$i: ${factorial(i)}"))
```

The last line prints the following:

```
0: 1
1: 1
2: 2
3: 6
4: 24
5: 120
```

The fact method calls itself recursively, passing an accumulator parameter, where the result of the calculation is accumulated. Note that we return the accumulated value when the counter i reaches 1. (We're ignoring negative integer arguments, which would be invalid. The function just returns 1 for i <= 1.) After the definition of the nested method, factorial calls it with the passed-in value i and the initial accumulator seed value of 1.

Notice that we use i as a parameter name twice, first in the factorial method and again in the nested fact method. The use of i as a parameter name for fact shadows the outer use of i as a parameter name for factorial. This is fine because we don't need the outer value of i inside fact. We only use it the first time we call fact, at the end of factorial.

Like a local variable declaration in a method, a nested method is also only visible inside the enclosing method.

Look at the return types for the two functions. I used `scala.math.BigInt` because factorials grow in size quickly. We don't need the return type declaration on `factorial` because it will be inferred from the return type of `fact`.

However, we must declare the return type for `fact`. Scala provides *local type inference*, meaning local to some scope, as opposed to global. This is sufficient to infer method return types in most cases, but not when they are recursive.

You might be a little nervous about a recursive function. Aren't we at risk of blowing up the stack? The JVM and some other runtime environments don't do *tail-call optimizations*, which would convert a tail recursive function into a loop. This would prevent stack overflow and also make execution faster by eliminating the overhead of function invocations.

The term *tail recursive* means that the recursive call is the last thing done in an expression. If we make the recursive call, then add something to the result, for example, that would not be a tail call. This doesn't mean that a recursion that is not a tail call is disallowed, just that we can't optimize it into a loop.

Recursion is a hallmark of FP and a powerful tool for writing elegant implementations of many algorithms. Hence, the Scala compiler does limited tail-call optimizations itself. It will handle functions that call themselves, but not *mutual recursion* (i.e., "a calls b calls a calls b," etc.).

Still, you might want to know if you got it right and the compiler did in fact perform the optimization. No one wants a blown stack in production. Fortunately, the compiler can tell you if you got it wrong if you add an annotation, `tailrec`, as shown in this refined version of `factorial`:

```
// src/script/scala/progscala3/typelessdomore/FactorialTailrec.scala
import scala.annotation.tailrec

def factorial(i: Int): BigInt =
  @tailrec
  def fact(i: Int, accumulator: BigInt): BigInt =
    if i <= 1 then accumulator
    else fact(i - 1, i * accumulator)

  fact(i, BigInt(1))

(0 to 5).foreach(i => println(s"$i: ${factorial(i)}"))
```

If `fact` is not actually tail recursive, the compiler will throw an error. Consider this attempt to write a naïve recursive implementation of Fibonacci sequences:

```
// src/script/scala/progscala3/typelessdomore/FibonacciTailrec.scala
scala> import scala.annotation.tailrec

scala> @tailrec
     | def fibonacci(i: Int): BigInt =
     |   if i <= 1 then BigInt(1)
     |   else fibonacci(i - 2) + fibonacci(i - 1)
4 |  else fibonacci(i - 2) + fibonacci(i - 1)
  |                          ^^^^^^^^^^^^^^^^
  |                 Cannot rewrite recursive call: it is not in tail position
4 |  else fibonacci(i - 2) + fibonacci(i - 1)
  |       ^^^^^^^^^^^^^^^^
  |       Cannot rewrite recursive call: it is not in tail position
```

We are attempting to make two recursive calls, not one, and then do something with the returned values, in this case add them. So this function is not tail recursive. (It is naïve because it is possible to write a tail recursive implementation.)

Finally, the nested function can see anything in scope, including arguments passed to the outer function. Note the use of `n` in `count` in the next example:

```
// src/script/scala/progscala3/typelessdomore/CountTo.scala
import scala.annotation.tailrec

def countTo(n: Int): Unit =
  @tailrec
  def count(i: Int): Unit =
    if (i <= n) then
      println(i)
      count(i + 1)
  count(1)

countTo(5)
```

Inferring Type Information

Statically typed languages provide wonderful compile-time safety, but they can be verbose if all the type information has to be explicitly provided. Scala's type inference removes most of this explicit detail, but where it is still required, it can provide an additional benefit of documentation for the reader.

Some FP languages, like Haskell, can infer almost all types because they do global type inference. Scala can't do this, in part because it has to support subtype polymorphism for object-oriented inheritance, which makes type inference harder.

We've already seen examples of Scala's type inference. Here are two more examples, showing different ways to declare a `Map`:

```
scala> val map1: Map[Int, String] = Map.empty
val map1: Map[Int, String] = Map()

scala> val map2 = Map.empty[Int, String]
val map1: Map[Int, String] = Map()
```

The second form is more idiomatic most of the time. `Map` is actually an abstract type with concrete subtypes, so you'll sometimes make declarations like this for a `TreeMap`:

```
scala> import scala.collection.immutable.TreeMap
scala> val map3: Map[Int, String] = TreeMap.empty
val map3: Map[Int, String] = Map()
```

We'll explore Scala's type hierarchy in Chapter 13.

Let's look at a few examples of cases we haven't seen yet where explicit types are required. First, we look at overloaded methods:

```
// src/script/scala/progscala3/typelessdomore/MethodOverloadedReturn.scala

case class Money(value: BigDecimal)
case object Money:
  def apply(s: String): Money = apply(BigDecimal(s.toDouble))
  def apply(d: Double): Money = apply(BigDecimal(d))
```

While the `Money` constructor expects a `BigDecimal`, we want the user to have the convenience of passing a `String` or a `Double`.[3] Note that we have added two more `apply` methods to the companion object. Both call the `apply(value: BigDecimal)` method the compiler automatically generates for the companion object, corresponding to the primary constructor `Money(value: BigDecimal)`.

The two methods have explicit return types. If you try removing them, you'll get a compiler error, "Overloaded or recursive method apply needs return type."

3 If a bad string is used, like `"x"`, a `java.lang.NumberFormatException` will be thrown by `String.toDouble`.

When Explicit Type Annotations Are Required

In practical terms, you have to provide explicit type declarations for the following situations in Scala:

- Abstract `var` or `val` declarations in an abstract class or trait.
- All method parameters (e.g., `def deposit(amount: Money) = …`).
- Method return types in the following cases:
  — When you explicitly call `return` in a method (even at the end).
  — When a method is recursive.
  — When two or more methods are overloaded (have the same name) and one of them calls another. The `calling` method needs a return type declaration.
  — When the inferred return type would be more general than you intended (e.g., `Any`).

The last case is somewhat rare, fortunately.

> The `Any` type is the root of the Scala type hierarchy. If a block of code is inferred to return a value of type `Any` unexpectedly, chances are good that the code is more general than you intended so that `Any` is the only common *supertype* of all possible values.

Repeated Parameter Lists

Scala supports methods that take repeated parameters. Other languages call them *variable argument lists* (*varargs* for short) or refer to the methods that use them as *variadic methods*. We briefly saw an example in "A Taste of Scala" on page 9 while discussing `@main` entry points. Consider this contrived example that computes the mean of `Doubles`:

```
// src/script/scala/progscala3/typelessdomore/RepeatedParameters.scala

object Mean1:
  def calc1a(ds: Double*): Double = calc1b(ds)
  def calc1b(ds: Seq[Double]): Double = ds.sum/ds.size

  def calc2a(ds: Double*): Double = ds.sum/ds.size
  def calc2b(ds: Seq[Double]): Double = calc2a(ds*)
```

(We'll ignore errors from an empty list.) The syntax `ds: Double*` means zero or more `Doubles`. When `calc1a` calls `calc1b`, it just passes `ds`. The repeated parameters are implemented with a `Seq[Double]`. The pair `calc2a` and `calc2b` shows how to pass a sequence as a repeated parameter list, using the `(ds*)` syntax. Scala 2 code used the syntax `(ds: _*)` to pass a sequence to a function expecting a repeated parameters list. Scala 3.0 allows this syntax as well, for backward compatibility, but not Scala 3.1.

Why have two functions? Users can pick what's most convenient to use. However, there are disadvantages too. The API footprint is larger with two methods instead of one. Is the convenience really worth it?

Assuming you want a pair of methods, why not use the same name? In particular, `apply` would be more convenient than ad hoc names like `calc1b`. Let's try it in the REPL:

```
scala> object Mean2:
     |   def apply(ds: Double*): Double = apply(ds)
     |   def apply(ds: Seq[Double]): Double = ds.sum/ds.size
3 |  def apply(ds: Seq[Double]): Double = ds.sum/ds.size
  |      ^
  |Double definition:
  |def apply(ds: Double*): Double in object Mean2 at line 2 and
  |def apply(ds: Seq[Double]): Double in object Mean2 at line 3
  |have the same type after erasure.
  |
  |Consider adding a @targetName annotation to one of the conflicting definitions
  |for disambiguation.
```

Because `ds: Double*` is implemented with a sequence, the methods look identical at the byte code level. The error message suggests one fix using the annotation `@targetName`. I'll discuss that option in "Defining Operators" on page 71. Here I'll describe a common idiom to break the ambiguity: add a first `Double` parameter in the first `apply`, then use a repeated parameter list for the rest of the parameters:

```
object Mean:
  def apply(d: Double, ds: Double*): Double = apply(d +: ds)
  def apply(ds: Seq[Double]): Double = ds.sum/ds.size
```

The first version happens to fix a bug we have ignored—that the method fails badly if an empty list of `Doubles` is provided. When calling the second `apply`, the first one constructs a new sequence by prepending `d` to the `ds` sequence using `d +: ds`. You'll see this used a lot in Scala. We'll explore it some more in "Operator Precedence Rules" on page 78.

Finally, `Nil` is an object representing an empty sequence with any type of element. Try `1.1 +: Nil` in the REPL, then prepend `2.2` to the returned sequence.

Language Keywords

Table 2-1 lists the reserved keywords and symbols in Scala, which are used for defining constructs like conditionals and declaring variables. Most of them are reserved for exclusive use. Those that aren't are marked with *(soft)*, which means they can be used as regular identifiers, such as method and variable names and type aliases, when they are used outside a narrow context. All of the soft keywords are new in Scala 3, but not all new keywords are soft, such as `given` and `then`. The reason for treating most of them as soft is to avoid breaking older code that happens to use them as identifiers.

Table 2-1. Reserved keywords and symbols

Word	Description	More details
`abstract`	Make a declaration abstract.	Chapter 9
`as`	(soft) Provide an alias for an imported and exported names.	"Importing Types and Their Members" on page 64
`case`	Start a case clause in a `match` expression. Define a case class.	Chapter 4
`catch`	Start a clause for catching thrown exceptions.	"Using try, catch, and finally Clauses" on page 90
`class`	Start a class declaration.	Chapter 9
`def`	Start a method declaration.	"Method Declarations" on page 39
`do`	New syntax for `while` and `for` loops without braces. Old Scala 2 `do…while` loop.	"Scala Conditional Expressions" on page 83
`derives`	(soft) Used in type class derivation.	"Type Class Derivation" on page 158
`else`	Start an `else` clause for an `if` expression.	"Scala Conditional Expressions" on page 83
`end`	(soft) Optional marker for the end of a block when using the braceless syntax.	"New Scala 3 Syntax—Optional Braces" on page 31
`enum`	Start an enumeration declaration.	"Sealed Class Hierarchies and Enumerations" on page 62
`export`	Export members of private fields as part of a type's interface.	"Export Clauses" on page 263
`extends`	Indicates that the class or trait that follows is the supertype of the class or trait being declared.	"Supertypes" on page 261
`extension`	(soft) Marks one or more extension methods for a type.	"Extension Methods" on page 139
`false`	`Boolean` false.	"Boolean Literals" on page 55
`final`	Apply to a type to prohibit creating subtypes from it. Apply to a member to prohibit overriding it in a subtype.	"Overriding Methods? The Template Method Pattern" on page 251
`finally`	Start a clause that is executed after the corresponding `try` clause, whether or not an exception is thrown by the `try` clause.	"Using try, catch, and finally Clauses" on page 90

Word	Description	More details
`for`	Start a `for` comprehension (loop).	"for Comprehensions" on page 86
`forSome`	Used in Scala 2 for existential type declarations to constrain the allowed concrete types that can be used. Dropped in Scala 3.	"Existential Types (Obsolete)" on page 367
`given`	Mark a definition as eligible for a using clause.	Chapter 5
`if`	Start an `if` clause.	"Scala Conditional Expressions" on page 83
`implicit`	Legacy alternative to `given` and `using` constructs.	Chapter 5
`import`	Import one or more identifiers into the current scope.	"Importing Types and Their Members" on page 64
`infix`	(soft) Mark a method or type as suitable for infix notation.	"Defining Operators" on page 71
`inline`	(soft) Tell the compiler to expand the definition inline.	"Inline" on page 491
`lazy`	Defer evaluation of a `val`.	"Lazy Values" on page 97
`match`	Start a pattern-matching expression.	Chapter 4
`new`	Create a new instance of a class.	"When new Is Optional" on page 102
`null`	Value of a reference variable that has not been assigned a value.	"Option, Some, and None: Avoiding Nulls" on page 60
`object`	Start a singleton declaration: a `class` with only one instance.	Chapter 9
`opaque`	(soft) Declare a special type member with zero runtime overhead.	"Opaque Types and Value Classes" on page 253
`open`	(soft) Declare a concrete class open for subtyping.	"Open Versus Closed Types" on page 247
`override`	Override a concrete member of a type, as long as the original is not marked `final`.	"Overriding Methods? The Template Method Pattern" on page 251
`package`	Start a package scope declaration.	"Organizing Code in Files and Namespaces" on page 63
`private`	Restrict visibility of a declaration.	Chapter 15
`protected`	Restrict visibility of a declaration.	Chapter 15
`requires`	Dropped in Scala 3. Was used for self-typing.	"Self-Type Declarations" on page 382
`return`	Return from a method.	"A Taste of Scala" on page 9
`sealed`	Apply to a supertype to require all subtypes to be declared in the same source file.	"Sealed Class Hierarchies and Enumerations" on page 62
`super`	Analogous to `this`, but binds to the supertype.	"Linearization of a Type Hierarchy" on page 301
`then`	New syntax for `if` expressions	"Scala Conditional Expressions" on page 83
`this`	Refer to the enclosing instance. The method name for auxiliary constructors.	"Constructors in Scala" on page 262
`throw`	Throw an exception.	"Using try, catch, and finally Clauses" on page 90

Word	Description	More details
`trait`	Start an abstract type declaration, used as a base type for concrete types or as a mixin module that adds additional state and behavior to other types.	Chapter 10
`transparent`	(soft) Mark a trait to suppress including it as part of an inferred type. Also used with inlining of code.	"Transparent Traits" on page 281, "Inline" on page 491
`true`	`Boolean` true.	"Boolean Literals" on page 55
`try`	Start a block that may throw exceptions to enable catching them.	"Using try, catch, and finally Clauses" on page 90
`type`	Start a type member declaration.	"Parameterized Types Versus Abstract Type Members" on page 66
`using`	(soft) Scala 3 alternative to `implicit` for using clauses.	Chapter 5
`val`	Start a read-only variable declaration.	"Variable Declarations" on page 33
`var`	Start a read-write variable declaration.	"Variable Declarations" on page 33
`while`	Start a `while` loop.	"Scala while Loops" on page 90
`with`	Include the trait that follows in the type being declared or the instance being instantiated.	Chapter 10
`yield`	Return an element in a `for` comprehension that becomes part of a sequence.	"Yielding New Values" on page 87
`:`	Separator between an identifier and a type declaration.	"A Taste of Scala" on page 9
`=`	Assignment.	"A Taste of Scala" on page 9
`?`	The wildcard for type parameters.	"Givens and Imports" on page 159
`*`	(soft) The wildcard for import and export statements, a marker for repeated parameters.	"Importing Types and Their Members" on page 64
`+`	(soft) Marks covariant types.	"Parameterized Types Versus Abstract Type Members" on page 66
`-`	(soft) Marks contravariant types.	"Parameterized Types Versus Abstract Type Members" on page 66
`_`	The anonymous placeholder for function literal arguments and a way to suppress some imports.	"Anonymous Functions, Lambdas, and Closures" on page 190, "Importing Types and Their Members" on page 64
`<-`	Part of `for` comprehension generator expressions.	"for Comprehensions" on page 86
`<:`	Constrain a type parameter with an upper bound.	"Type Bounds" on page 349
`>:`	Constrain a type parameter with a lower bound.	"Type Bounds" on page 349
`#`	Project a nested type.	"Type Projections" on page 385
`@`	Mark use of an annotation.	"Annotations" on page 468
`=>`	In function literals, separates the parameter list from the function body.	"Anonymous Functions, Lambdas, and Closures" on page 190
`=>>`	In type lambdas, separates the parameter list from the body.	"Type Lambdas" on page 391

Word	Description	More details
?=>	In context function types, separates the parameter list from the body.	"Context Functions" on page 172
|	(soft) Indicates alternatives in pattern matches.	"Values, Variables, and Types in Matches" on page 107

Some APIs written in other languages use names that are reserved keywords in Scala, for example, `java.util.Scanner.match`. To avoid a compilation error, surround the name with single back quotes (*backticks*) (e.g., ``java.util.Scanner.`match` ``).

Literal Values

We've seen a few *literal values* already, such as `val book = "Programming Scala"`, where we initialized a `val book` with a `String` literal, and `(s: String) => s.toUpperCase`, an example of a function literal. Let's discuss all the literals supported by Scala.

Numeric Literals

3 Scala 3 expanded the ways that *numeric literals* can be written and used as initializers. Consider these examples:

```
val i: Int = 123                         // decimal
val x: Long = 0x123L                     // hexadecimal (291 decimal)
val f: Float = 123_456.789F              // 123456.789
val d: Double = 123_456_789.0123         // 123456789.0123
val y: BigInt = 0x123_a4b                // 1194571
val z: BigDecimal = 123_456_789.0123     // 123456789.0123
```

Scala allows underscores to make long numbers easier to read. They can appear anywhere in the literal (except between `0x`), not just between every third character.

Hexadecimal numbers start with `0x` followed by one or more digits and the letters `a` through `f` and `A` through `F`.

Indicate a negative number by prefixing the literal with a `-` sign.

For `Long` literals, you must append the `L` character at the end of the literal, unless you are assigning the value to a variable declared to be `Long`. Otherwise, `Int` is inferred. Lowercase `l` is allowed but discouraged because it's easy to misread it as the number `1`. The valid values for an integer literal are bounded by the type of the variable to which the value will be assigned. Table 2-2 defines the limits, which are inclusive.

Table 2-2. Ranges of allowed values for integer literals (boundaries are inclusive)

Target type	Minimum (inclusive)	Maximum (inclusive)
`Long`	-2^{63}	$2^{63} - 1$
`Int`	-2^{31}	$2^{31} - 1$
`Short`	-2^{15}	$2^{15} - 1$
`Char`	0	$2^{16} - 1$
`Byte`	-2^{7}	$2^{7} - 1$

A compile-time error occurs if an integer literal is outside these ranges.

Floating-point literals are expressions with an optional minus sign, zero or more digits and underscores, followed by a period (`.`), followed by one or more digits. For `Float` literals, append the `F` or `f` character at the end of the literal. Otherwise, a `Double` is assumed. You can optionally append a `D` or `d` for a `Double`.

Floating-point literals can be expressed with or without exponentials. The format of the exponential part is `e` or `E`, followed by an optional `+` or `-`, followed by one or more digits.

Here are some example floating-point literals where `Double` is inferred unless the declared variable is `Float`, or an `f` or `F` suffix is used:

```
0.14          // leading 0 required
3.14, 3.14f, 3.14F, 3.14d, 3.14D
3e5, 3E5
3.14e+5, 3.14e-5, 3.14e-5f, 3.14e-5F, 3.14e-5d, 3.14e-5D
```

At least one digit must appear after the period, and `3.` and `3.e5` are disallowed. Use `3.0` and `3.0e5` instead. Otherwise it would be ambiguous; do you mean some method `e5` on the `Int` value of `3` or do you mean floating point literal `3.0e5`?

`Float` consists of all IEEE 754 32-bit, single-precision binary floating-point values. `Double` consists of all IEEE 754 64-bit, double-precision binary floating-point values.

3 Scala 3 introduced a mechanism to allow using numeric literals for library and user-defined types like `BigInt` and `BigDecimal`. It is implemented with a trait called `FromDigits`.[4]

Boolean Literals

The *Boolean literals* are `true` and `false`. The type of the variable to which they are assigned is inferred to be `Boolean`:

4 "Internal DSLs" on page 440 shows an example for a custom `Money` type.

```
scala> val (t, f) = (true, false)
val t: Boolean = true
val f: Boolean = false
```

Character Literals

A *character literal* is either a *printable* Unicode character or an escape sequence, written between single quotes. A character with a Unicode value between 0 and 255 may also be represented by an octal escape; that is, a backslash (\) followed by a sequence of up to three octal characters. It is a compile-time error if a backslash character in a character or string literal does not start a valid escape sequence.

Here are some examples:

```
'A', '\u0041'  // 'A' in Unicode
'\n', '\012'    // '\n' in octal
'\t'
```

Releases of Scala before 2.13 allowed three Unicode arrow characters to be used instead of two-character ASCII equivalents. These alternatives are now deprecated: ⇒ for `=>`, → for `->`, and ← for `<-`. You'll see them used in older code, but I'll avoid them in the book's examples.

The valid escape sequences are shown in Table 2-3.

Table 2-3. Character escape sequences

Sequence	Meaning
`\b`	Backspace (BS)
`\t`	Horizontal tab (HT)
`\n`	Line feed (LF)
`\f`	Form feed (FF)
`\r`	Carriage return (CR)
`\"`	Double quote (")
`\'`	Single quote (')
`\\`	Backslash (\)
`\u{0000-FFFF}`	Unicode hex value

String Literals

A *string literal* is a sequence of characters enclosed in double quotes or triples of double quotes (`"""..."""`).

For string literals in double quotes, the allowed characters are the same as the character literals. However, if a double quote (") character appears in the string, it must be escaped with a \ character. Here are some examples:

```
"Programming\nScala"
"He exclaimed, \"Scala is great!\""
"First\tSecond"
```

Triple-quoted string literals support *multiline* strings; the line feeds will be part of the string. They can include any characters, including one or two double quotes together, but not three together. They are useful for strings with backslash (\) characters that don't form valid Unicode or escape sequences (those listed in Table 2-3). *Regular expressions*, which use lots of escaped characters with special meanings, are a good example. Conversely, if escape sequences appear, they aren't interpreted.

Here are four example strings:

```
"""Programming\nScala"""
"""He exclaimed, "Scala is great!""""
"""First line\n
Second line\t

Fourth line"""
"""^\s*(\d{4})-(\d{2})-(\d{2})\s+(\w*)\s*$"""
```

The last example describes a *regular expression*, which we'll discuss in "Matching on Regular Expressions" on page 119. Try converting the triple quotes to single quotes in the REPL. What errors are reported?

When using multiline strings in code, you'll want to indent the substrings for proper code formatting, yet you probably don't want that extra whitespace in the actual string output. `String.stripMargin` solves this problem. It removes all whitespace in the substrings up to and including the first occurrence of a vertical bar (`|`) character:

```
// src/script/scala/progscala3/typelessdomore/MultilineStrings.scala
scala> val welcome = s"""Welcome!
     |   Hello!
     |   * (Gratuitous Star character!!)
     |   |This line has a margin indicator.
     |   |  This line has some extra whitespace.""".stripMargin
val welcome: String = Welcome!
  Hello!
  * (Gratuitous Star character!!)
This line has a margin indicator.
  This line has some extra whitespace.
```

Note on each line where leading whitespace is removed and where it isn't.

If you want to use a different leading character than `|`, use the overloaded version of `stripMargin` that takes a `Char` (character) parameter. If the whole string has a prefix or suffix you want to remove (but not on individual lines), there are corresponding `stripPrefix` and `stripSuffix` methods too:

```
scala> "<hello> <world>".stripPrefix("<").stripSuffix(">")
val res0: String = hello> <world
```

The < and > inside the string are not removed.

Symbol Literals

3 Scala supports symbols, which are *interned strings*, meaning that two symbols with the same character sequence will actually refer to the same object in memory. A Scala 2 literal syntax for them uses a leading, single quote, `'mysymbol`, but this syntax is deprecated in Scala 3. If you want to continue using this syntax, use the language import `import language.deprecated.symbolLiterals` or use `Symbol("mysymbol")` instead.

Function Literals

As we've seen already, `(i: Int, d: Double) => (i+d).toString` is a *function literal*. It has the type `Function2[Int,Double,String]`, where the last type is the return type.

You can even use the literal syntax for a type declaration. The following declarations are equivalent:

```
val f1: (Int, Double) => String        = (i, d) => (i+d).toString
val f2: Function2[Int, Double, String] = (i, d) => (i+d).toString
```

Tuples

Often, declaring a class to hold instances with two or more values is more than you need. You could put those values in a collection, but then you lose their specific type information. Scala implements *tuples* of values, where the individual types are retained. The tuple syntax uses a comma-separated list of values surrounded by parentheses.

Here is an example of a tuple declaration and how we can access the elements inside it. Starting with the declaration, we can use the syntax to construct a three-element tuple. We can use the same syntax for the type too:

```
// src/script/scala/progscala3/typelessdomore/Tuples.scala

scala> val tup = ("Hello", 1, 2.3)
val tup: (String, Int, Double) = (Hello,1,2.3)

scala> val tup2: (String, Int, Double) = ("World", 4, 5.6)
val tup2: (String, Int, Double) = (World,4,5.6)
```

3 We can retrieve the first element with the `_1` method and similarly for the rest of them. Tuple indexing with these methods is one-based, by historical convention, not zero-based. However, Scala 3 adds the ability to access the elements like we can access

elements in arrays and sequences, with zero-based indexing, `tup(0)`, etc. Let's use both approaches to retrieve the three elements:

```
scala> (tup._1, tup(0))
val res7: (String, String) = (Hello,Hello)

scala> (tup._2, tup(1))
val res8: (Int, Int) = (1,1)

scala> (tup._3, tup(2))
val res9: (Double, Double) = (2.3,2.3)

scala> (tup._4, tup(3))
1 |(tup._4, tup(3))
  | ^^^^^^
  | value _4 is not a member of (String, Int, Double) - did you mean tup._1?
```

The last line shows what happens if we ask for nonexistent elements.

Finally, we can grab all three elements separately with pattern matching:

```
scala> val (s, i, d) = tup
val s: String = Hello
val i: Int = 1
val d: Double = 2.3
```

Try removing the `d` in the first line. Try adding a fourth variable. What happens in both cases?

Two-element tuples, sometimes called *pairs* for short, are so commonly used there is a special way of creating them:

```
scala> 1 -> "one"
val res3: (Int, String) = (1,one)

scala> (1, "one")          // Like all other tuples
val res4: (Int, String) = (1,one)

scala> Tuple2(1, "one")    // Rarely used
val res5: (Int, String) = (1,one)
```

For example, maps are often constructed with key-value pairs as follows:

```
// src/script/scala/progscala3/typelessdomore/StateCapitalsSubset.scala

scala> val stateCapitals = Map(
     |   "Alabama" -> "Montgomery",
     |   "Alaska"  -> "Juneau",
     |   // ...
     |   "Wyoming" -> "Cheyenne")
val stateCapitals: Map[String, String] =
  Map(Alabama -> Montgomery, Alaska -> Juneau, Wyoming -> Cheyenne)
```

Option, Some, and None: Avoiding Nulls

Let's discuss three useful types that express a very useful concept, when we may or may not have a value.

Most languages have a special keyword for reference variables when they are not assigned a valid value. Scala uses `null`. Nulls are a giant source of nasty bugs across most languages. What `null` signals is that we don't have a value in a given situation. If the value is not `null`, we do have a value. Why not express this situation explicitly with the type system and exploit type checking to avoid `NullPointerExceptions`?

`Option` lets us express this situation explicitly without using `null`. `Option` is an abstract class with two concrete subtypes: `Some`, for when we have a value, and `None`, when we don't. Think of an `Option` as a special kind of collection with zero or one value.

You can see `Option`, `Some`, and `None` in action using the map of state capitals in the United States that we declared in the previous section:

```
scala> stateCapitals.get("Alabama")
     | stateCapitals.get("Wyoming")
     | stateCapitals.get("Unknown")
val res6: Option[String] = Some(Montgomery)
val res7: Option[String] = Some(Cheyenne)
val res8: Option[String] = None

scala> stateCapitals.getOrElse("Alabama", "Oops1")
     | stateCapitals.getOrElse("Wyoming", "Oops2")
     | stateCapitals.getOrElse("Unknown", "Oops3")
val res9: String = Montgomery
val res10: String = Cheyenne
val res11: String = Oops3
```

`Map.get` returns an `Option[T]`, where `T` is `String` in this case. Either a `Some` wrapping the value is returned or a `None` when no value for the specified key is found.

In contrast, similar methods in other languages just return a `T` value, when found, or `null`.

By returning an `Option`, we can't "forget" that we have to verify that something was returned. In other words, the fact that a value may not exist for a given key is enshrined in the return type for the method declaration. This also provides clear documentation for the user of `Map.get` about what can be returned.

The second group uses `Map.getOrElse`. This method returns either the value found for the key or it returns the second argument passed in, which functions as the default value to return.

So `getOrElse` is more convenient, as you don't need to process the `Option`, as long as a suitable default value exists.

To reiterate, because the `Map.get` method returns an `Option`, it automatically documents for the reader that there may not be an item matching the specified key. The map handles this situation by returning a `None`.

Also, thanks to Scala's static typing, you can't make the mistake of "forgetting" that an `Option` is returned and attempting to call a method supported by the type of the value inside the `Option`. You must extract the value first or handle the `None` case. Without an option return type, when a method just returns a value, it's easy to forget to check for `null` before calling methods on the returned object.

> Never write methods that can return `null`. Instead, return `Option`, so the user learns the possible behavior through the type signature and the user's code must properly handle the `Some` and `None` cases.

When You Really Can't Avoid Nulls

Because Scala runs on the JVM, JavaScript, and native environments, it must interoperate with other libraries, which means Scala has to support `null`, as many of these libraries have methods that can return `null`.

Scala has a `Null` type that is a subtype of all `AnyRef` types. Suppose you have a Java `HashMap` to access:

```
// src/script/scala/progscala3/typelessdomore/Null.scala

import java.util.HashMap as JHashMap                    ❶

val jhm = JHashMap[String,String]()
jhm.put("one", "1")

val one1: String = jhm.get("one")                       ❷
val one2: String | Null = jhm.get("one")                ❸

val two1: String = jhm.get("two")                       ❹
val two2: String | Null = jhm.get("two")
```

❶ Import the Java `HashMap`, but give it an alias so it doesn't shadow Scala's `HashMap`.

❷ Return the string `"1"`.

❸ Declare explicitly that `one2` is of type `String` or `Null`. The value will still be `"1"` in this case.

❹ These two values will equal `null`.

3 The type `String | Null` is called a *union type*. It tells the reader that the value could be either a `String` or `null`.[5]

There is an optional and experimental feature to enable aggressive null checking. It is experimental because the Scala compiler team is still developing this feature, so avoid it in production code. You can enable this feature with the compiler flag `-Yexplicit-nulls`, after which the declarations of `one1` and `two1` will be disallowed because the compiler knows you are referring to a Java library where `null` could be returned. For more details, see the *explicit nulls* documentation. If you try this same code in a REPL with this flag enabled, you'll see the following:

```
$ scala -Yexplicit-nulls
...
scala> val one1: String = jhm.get("one")
1 |val one1: String = jhm.get("one")
  |                   ^^^^^^^^^^^^^^
  |                   Found:    String | UncheckedNull
  |                   Required: String

...
```

Tony Hoare invented the null reference in 1965 while working on a language called ALGOL W. He has called its invention his "billion dollar" mistake. Use `Option` instead.

Sealed Class Hierarchies and Enumerations

While we're discussing `Option`, let's discuss a useful design feature it uses. A key point about `Option` is that there are really only two valid subtypes. Either we have a value, the `Some` case, or we don't, the `None` case. There are no other subtypes of `Option` that would be valid. So we would really like to prevent users from creating their own.

Scala 2 and 3 have a keyword `sealed` for this purpose. `Option` could be declared as follows:

```
sealed abstract class Option[+A] {...}
case class Some[+A](a: A) extends Option[A] {...}
case object None extends Option[Nothing] {...}
```

The `sealed` keyword tells the compiler that all subtypes must be declared in the same source file. `Some` and `None` are declared in the same file with `Option` in the Scala library. This technique effectively prevents additional subtypes of `Option`.

5 Union types are new to Scala 3. We'll explore them in depth in "Union and Intersection Types" on page 279.

None has an interesting declaration. It is a case class with only one instance, so it is declared `case object`. The `Nothing` type along with the `Null` type are subtypes of all other types in Scala. I'll say something about `Nothing`, if you get what I mean, in more detail in "Sequences" on page 200.

You can also declare a type `final` if you want to prevent users from subtyping it.

3 This same constraint on subtyping can now be achieved more concisely in Scala 3 with the new `enum` syntax that we'll explore in "Enumerations and Algebraic Data Types" on page 79. Here's a teaser of what `Option` would look like defined as an `enum`:

```
enum Option[+A] {
  case Some(a: A) {...}
  case None {...}
  ...
}
```

We'll see more examples of `enum`s and sealed hierarchies, which help us carefully craft our types for optimal utility, robustness, and type safety.

Organizing Code in Files and Namespaces

Scala has a `package` concept for namespaces. While inspired by packages in Java, filenames do not have to match the type names, and the package structure does not have to match the directory structure. So you can define packages in files independent of their "physical" location.

The following example defines a class `MyClass` in a package `com.example.mypkg` using the most common syntax:

```
// src/main/scala/progscala3/typelessdomore/Package1.scala
package com.example.mypkg

class MyClass:
  def mymethod(s: String): String = s
```

Scala also supports a block-structured syntax for declaring package scope:

```
// src/main/scala/progscala3/typelessdomore/Package2.scala
package com:
  package example:                 // Subpackage of "com"
    package pkg1:                  // Subpackage of "example"
      class Class11:               // Class inside "com.example.pkg1"
        def m = "m11"

      class Class12:               // Class inside "com.example.pkg1"
        def m = "m12"

    package pkg2:                  // Subpackage of "example"
      class Class21:               // Class inside "com.example.pkg2"
```

```
      def m = "m21"
      def makeClass11 = pkg1.Class11()

      def makeClass12 = pkg1.Class12()

  package pkg3.pkg31.pkg311:    // More concise nesting of packages
    class Class311:
      def m = "m21"
```

The comments explain the organization. The `makeClass11` and `makeClass12` methods in `Class21` illustrate how to reference a type in the sibling package, `pkg1`. You can also reference these classes by their full paths, `com.example.pkg1.Class11` and `com.example.pkg1.Class12`, respectively.

Here the *root* package is the first one declared, `com`. The root package for Scala's library classes is named `scala`.

Although the package declaration syntax is flexible, one limitation is that packages cannot be defined within classes and objects, which wouldn't make much sense anyway.

Importing Types and Their Members

To use declarations in packages, you have to import them. However, Scala offers flexible options for how items are imported:

```
import scala.math.*                                     (1)
import scala.io.Source                                  (2)
import scala.io.Source.*                                (3)
import scala.collection.immutable.{List, Map}           (4)
import scala.collection.immutable.Vector                (5)
import collection.immutable.Vector
```

1. Import everything in a package, using a star (`*`) as a wildcard.
2. Import an individual type.
3. Import all members of the `Source` object.
4. Selectively import two types.
5. This line and the next are effectively the same. You can omit `scala`.

> I always write `import scala...` for Scala library imports. At a glance, I can tell it is importing from the Scala library and not some other library with a package path beginning with `util`, `collection`, etc.

Scala uses * as the wildcard for all items in the enclosing scope. What if you want to import a method named * in a math package? Use backticks: `import foo.math.`*``.

You can put import statements almost anywhere, so you can scope their visibility to just where they are needed, you can rename types as you import them, and you can suppress the visibility of unwanted types:

```
def stuffWithCollections() =
  import scala.collection.immutable.{
    BitSet as _,                     ❶
    LazyList,                        ❷
    HashMap as HMap }                ❸
  // Do stuff with LazyList, HMap...
```

❶ Alias `BitSet` to `_`, which makes it invisible. Use this technique when you want to import everything except a few items.

❷ Import `LazyList`, so it can be referenced simply as `LazyList` without the package prefix.

❸ Import `HashMap` but give it an alias. Note the `as` keyword. Use this technique to avoid shadowing other items with the same name. This is used a lot when mixing Java and Scala types that have the same name, such as collection types.

> Recall from Chapter 1 that Scala 2 uses _ as the import wildcard, instead of *. Scala 2 also uses => instead of `as` for aliasing an imported item. Both are still allowed in Scala 3.0, but they will be removed in a future release.

Because this import statement is inside `stuffWithBigInteger`, the imported items are not visible outside the method.

Package Imports and Package Objects

Sometimes it's nice to give the user one import statement for a public API that brings in all types, as well as constants and methods not attached to a type. For example:

```
import progscala3.typelessdomore.api.*
```

This is simple to do; just define anything you need under the package:

```
// src/main/scala/progscala3/typelessdomore/TopLevelDeclarations.scala
package progscala3.typelessdomore.api

val DefaulCount = 5
def countTo(limit: Int = DefaulCount) = (0 to limit).foreach(println)

class Class1:
```

```
    def m = "cm1"

  object Object1:
    def m = "om1"
```

In Scala 2, definitions that aren't types had to be declared inside a *package object*, like this:

```
// src/main/scala-2/progscala3/typelessdomore/PackageObjects.scala
package progscala3.typelessdomore   // Notice, no ".api"

package object api {
  val DefaultCount = 5
  def countTo(limit: Int = DefaultCount) = (0 to limit).foreach(println)

  class Class1 {
    def m = "cm1"
  }

  object Object1 {
    def m = "om1"
  }
}
```

Package objects are still supported in Scala 3, but they are deprecated.

Parameterized Types Versus Abstract Type Members

We mentioned in "A Taste of Scala" on page 9 that Scala supports *parameterized types* where square brackets (`[...]`) enclose the type parameter, for example `Seq[T]`.

Because we can plug in almost any type for a type parameter `T`, this feature is called *parametric polymorphism*. Generic implementations of the `List` methods can be used with instances of any type `T` (the parameter), causing polymorphic behavior (for all `List[T]`).

Consider the declaration of `Map`, which is written as follows, where `K` is the keys type and `V` is the values type.

```
trait Map[K, +V] extends Iterable[(K, V)] with ...
```

The `+` in front of the `V` means that `Map[K, V2]` is a subtype of `Map[K, V1]` for any `V2` that is a subtype of `V1`. This is called *covariant typing*. It is a reasonably intuitive idea. If we have a function `f(map: Map[String, Any])`, it makes sense that passing a `Map[String, Double]` to it should work fine because the function has to assume values of `Any`, a supertype of `Double`.

In contrast, the key `K` is *invariant*. We can't pass `Map[Any, Any]` to `f`, nor any `Map[S, Any]` for some subtype or supertype `S` of `String`.

If there is a dash (`-`) in front of a type parameter, the relationship goes the other way; `Foo[B]` would be a supertype of `Foo[A]` if `B` is a subtype of `A` and the declaration is `Foo[-A]` (called *contravariant typing*). This is less intuitive, but also not as important to understand now. We'll see how it is important for function types in "Parameterized Types" on page 347.

Scala supports another type of abstraction mechanism called *abstract type members*, which can be applied to many of the same design problems for which parameterized types are used. However, they are not redundant mechanisms. Each has strengths and weaknesses for certain design problems.

Abstract type members are declared as members of other types, just like abstract methods and fields. Here is an example that uses an abstract type member in a supertype, then makes the type concrete in subtypes, where it becomes an alias for other types:

```
// src/main/scala/progscala3/typelessdomore/BulkReaderAbstractTypes.scala
package progscala3.typelessdomore
import scala.io.Source

abstract class BulkReader:
  type In                                                           ❶
  /** The source of data to read. */
  val source: In
  /** Read source and return a sequence of Strings */
  def read: Seq[String]

case class StringBulkReader(source: String) extends BulkReader:     ❷
  type In = String
  def read: Seq[String] = Seq(source)

case class FileBulkReader(source: Source) extends BulkReader:       ❸
  type In = Source
  def read: Seq[String] = source.getLines.toVector
```

❶ Abstract type member, similar to an abstract field or method.

❷ Concrete subtype of `BulkReader` where `In` is defined as an alias for `String`. Note that the type of the `source` parameter passed to `StringBulkReader` must match.

❸ Concrete subtype of `BulkReader` where `In` is defined to be an alias for `Source`, the Scala library type for reading sources like files. `Source.getLines` returns an iterator, which we can read into a `Vector` with `toVector`.

Strictly speaking, we don't need to declare the `source` field in the supertype, but I put it there to show you that the concrete case classes can make it a constructor parameter, where the specific type is specified.

We've seen many other abstract types, such as traits. A *type member*, abstract or concrete, is declared with the `type` keyword.

Let's try these readers:

```
// src/script/scala/progscala3/typelessdomore/BulkReader.scala

scala> import progscala3.typelessdomore.{StringBulkReader, FileBulkReader}
     | import scala.io.Source

scala> val strings = StringBulkReader("Hello Scala!").read
val strings: Seq[String] = List(Hello Scala!)

scala> val lines = FileBulkReader(Source.fromFile("README.md")).read
val lines: Seq[String] = Vector(# Programming Scala, 3rd Edition, ...)

scala> lines(0)    // look at two lines...
     | lines(2)
val res2: String = # Programming Scala, 3rd Edition
val res3: String = ## README for the Code Examples
```

The abstract type member `BulkReader.In` is used in an analogous way to a type parameter in a parameterized type. As an exercise, try rewriting the example to use type parameters, `BulkReader[In]`.

So what are the advantages of using abstract type members instead of parameterized types? Parameterized types are best for when the type parameter has no relationship with the parameterized type, like mapping over a `Seq[A]`, which behaves uniformly for when `A` is `Int`, `String`, `Person`, or anything else. A type member works best when it evolves in parallel with the enclosing type, as in our `BulkReader` example, where the type member must match the behaviors expressed by the enclosing type, specifically the `read` method. Sometimes this characteristic is called *family polymorphism* or *covariant specialization*.

All concrete type members are aliases for other types. In fact, it's sometimes convenient to define a type member for a complicated type just to simplify using it. For a simple example, suppose you use `(String, Double)` tuples a lot in some code. You could either declare a class for it or use a type alias as a simple alternative:

```
// src/script/scala/progscala3/typelessdomore/Rec.scala

scala> type Rec = (String, Double)
// defined alias type Rec = (String, Double)

scala> def transform(record: Rec): Rec = (record._1.toUpperCase, 2*record._2)
def transform(record: Rec): Rec
```

```
scala> val rec2 = transform(("hello", 10))
val rec2: Rec = (HELLO,20.0)
```

Notice that a tuple literal is used as the argument to `transform`.

Recap and What's Next

We covered a lot of practical ground, such as literals, keywords, file organization, and imports. We learned how to declare variables, classes, and member types and methods. We learned about `Option` as a better tool than `null`, plus other useful techniques. In the next chapter, we will finish our fast tour of the Scala basics before we dive into more detailed explanations of Scala's features.

CHAPTER 3
Rounding Out the Basics

Let's finish our survey of essential basics in Scala.

Defining Operators

Almost all *operators* are actually methods. Consider this most basic of examples:

```
1 + 2
```

The plus sign between the numbers is a method on the `Int` type.

Scala doesn't have special primitives for numbers and Booleans that are distinct from types you define. They are regular types: `Float`, `Double`, `Int`, `Long`, `Short`, `Byte`, `Char`, and `Boolean`. Hence, they can have methods.

Therefore, `+` is actually a method implemented by `Int`. We can write `1.+(2)`, although it looks strange.

Fortunately, Scala also supports *infix operator notation*. When a method takes one argument and the name uses only nonalphanumeric characters, we can drop the period and parentheses to write the expression we want, `1 + 2`.

This is *infix notation* because `+` is between the object and argument. It is also called *operator notation* because it is especially popular when writing libraries where mathematics operator notation is convenient.

> Actually, they don't always behave identically, due to *operator precedence rules*. While `1 + 2 * 3 = 7`, `1.+(2)*3 = 9`. When present, the period binds before the star.

Recall in "Tuples" on page 58, we used `x -> y` to create a tuple `(x, y)`. This is also implemented as a method using a special library utility type called `ArrowAssoc`, defined in `Predef`. We'll explore this type in "Extension Methods" on page 139.

Infix operator notation isn't limited to methods that look like operators, meaning their names don't have alphanumeric characters. It's not uncommon to see `Seq(1,2,3) foreach println` in code, for example.

3 Scala 2 imposed no constraints on using infix operator notation for any methods, but excessive use of this feature can lead to code that is hard to read and sometimes hard to parse. Therefore, Scala 3 deprecates the use of infix operator notation for methods with alphanumeric names, meaning names that contain letters, numbers, $, and _ characters.

However, exceptions are allowed if one of the following is true:

- The method is declared with the `infix` keyword.
- The method was compiled with Scala 2.
- Use of the method is followed with an opening curly brace.
- The method is invoked with backticks.

A deprecation warning will be issued otherwise, but only starting with Scala 3.1, to ease migration. Because the Scala 2 library is used by Scala 3.0, all the common uses of infix notation, such as methods on collections like `map` and `foreach`, will work as before, but the long-term goal is to greatly reduce this practice.

Here is an example of the rules where `append` is not declared `infix`, but `combine` is:

```
// src/script/scala/progscala3/rounding/InfixMethod.scala

case class Foo(str: String):
  def append(s: String): Foo = copy(str + s)
  infix def combine(s:String): Foo = append(s)

Foo("one").append("two")          ❶
Foo("one") append {"two"}         ❷
Foo("one") `append` "two"
Foo("one") append "two"           ❸

Foo("one") combine "two"          ❹
```

❶ Normal usage.

❷ This line and the next one are accepted, but the usage looks odd.

❸ Triggers a deprecation warning starting with Scala 3.1.

❹ No warning, because `combine` is declared `infix`.

The keyword `infix` is a soft modifier. As we learned in "Language Keywords" on page 51, that means `infix` is treated as a regular identifier when used in any other context.

You can also define your own operator methods with symbolic names. Suppose you want to allow users to work with directory and file paths by appending strings using `/`, the file separator for Unix-derived systems. Consider the following implementation:

```
// src/main/scala/progscala3/rounding/Path.scala
package progscala3.rounding

import scala.annotation.targetName
import java.io.File

case class Path(
    value: String, separator: String = Path.defaultSeparator):   ❶
  val file = File(value)
  override def toString: String = file.getPath                     ❷

  @targetName("concat") def / (node: String): Path =               ❸
    copy(value + separator + node)                                 ❹

  infix def append(node: String): Path = /(node)                   ❺

object Path:
  val defaultSeparator = sys.props("file.separator")
```

❶ Use the operating system default path separator string as the default separator when constructing the actual path and a corresponding `java.io.File` instance.

❷ How to override the default `toString` method. Here, I use the path string from `File`.

❸ I'll explain the `@targetName` annotation in a moment.

❹ Use the case-class copy method to create a new instance, changing only the `value`.

❺ A method that can be used with infix notation.

Now users can work with paths and create `File` instances as follows:

```
scala> import progscala3.rounding.Path

scala> val one  = Path("one")
val one: progscala3.rounding.Path = one
```

```
scala> val three = one / "two" / "three"
val three: progscala3.rounding.Path = one/two/three

scala> three.file
val res0: java.io.File = one/two/three

scala> val threeb = one./("two")./("three")
val threeb: progscala3.rounding.Path = one/two/three

scala> three == threeb
val res1: Boolean = true

scala> one concat "two"
1 |one concat "two"
  |^^^^^^^^^^
  |value concat is not a member of progscala3.rounding.Path

scala> one append "two"
val res2: progscala3.rounding.Path = one/two
```

On Windows, the character \ would be used as the default separator. This method is designed to be used with infix notation. It looks odd to use normal invocation syntax.

In Scala 3, the `@targetName` annotation is optional, but suggested for operator methods that might be called from Java.

In this example, `concat` is the name the compiler will use internally when it generates byte code. This is the name you would use if you wanted to call the method from code in another language, like Java, which doesn't support invoking methods with symbolic names. However, the name `concat` can't be used in Scala code, as shown in the session. It only affects the byte code produced by the compiler that is visible to other languages.

The `infix` keyword on `append` allows us to use it as an operator. The keyword is not required for methods with names that only use *operator characters*, like `*` and `/` because support for symbolic operators has always existed for the particular purpose of allowing intuitive, infix expressions, like `a * b` and `path1 / path2`.

Types can also be written with infix notation, when useful. The same rules for when to explicitly use the `infix` keyword apply:

```
// src/script/scala/progscala3/rounding/InfixType.scala
import scala.annotation.targetName

@targetName("TIEFighter") case class <+>[A,B](a: A, b: B)   ❶
val ab1: Int <+> String = 1 <+> "one"                       ❷
val ab2: Int <+> String = <+>(1, "one")                     ❸

infix case class tie[A,B](a: A, b: B)                       ❹
```

```
val ab3: Int tie String = 1 tie "one"
val ab4: Int tie String = tie(1, "one")
```

❶ A type declaration inspired by *Star Wars* with two type parameters and an operator name.

❷ An attempt to use infix notation on both sides, but we get an error that `<+>` is not a method on `Int`. We'll solve this problem in Chapter 5.

❸ This declaration works, with the noninfix notation on the righthand side.

❹ These three lines behave the same, but we need `infix` now if we want to use the type with infix notation because the name is alphanumeric.

To recap:

- Mark alphanumeric types and methods with `infix` if you want to allow their use with infix notation, but limit your use of this feature.
- Annotate symbolic operator definitions with `@targetName("some_name")`.

While dropping the punctuation for infix expressions can sometimes make your code less cluttered, it quickly leads to expressions that are hard to understand. Use this feature with discretion.

3 The `@targetName` annotation can also work around a problem with JVM *type erasure*. Consider `Seq[T]`. For historical reasons, the specific parameter type for `T` is erased in JVM byte code. This causes problems with definitions that differ only in type parameters:

```
// src/script/scala/progscala3/rounding/TypeErasureProblem.scala

scala> object O:
     |   def m(is: Seq[Int]): Int = is.sum
     |   def m(ss: Seq[String]): Int = ss.length
     |
3 |  def m(ss: Seq[String]): Int = ss.length
  |      ^
  |Double definition:
  |def m(is: Seq[Int]): Int in object O at line 2 and
  |def m(ss: Seq[String]): Int in object O at line 3
  |have the same type after erasure.
  |
  |Consider adding a @targetName annotation to one of the conflicting definitions
  |for disambiguation.
```

The `Int` versus `String` type information is lost in the byte code. The last message tells us what to do:

```scala
// src/script/scala/progscala3/rounding/TypeErasureTargetNameFix.scala

import scala.annotation.targetName
object O:
  @targetName("m_seq_int")
  def m(is: Seq[Int]): Int = is.sum
  @targetName("m_seq_string")
  def m(ss: Seq[String]): Int = ss.length
```

Now the two methods have unique names in the generated byte code. Only one method needs to be annotated, or more generally, *N – 1* of *N* overloaded methods.

Allowed Characters in Identifiers

Here is a summary of the rules for characters in identifiers:

Characters
: Scala allows all the printable ASCII characters, including letters, digits, the underscore (_), and the dollar sign ($), with the exceptions of the parenthetical characters, (,), [,], {, and }, and the delimiter characters, `, ’, ', ", ., ,, and ;. Scala allows the Unicode characters between \u0020 and \u007F that are not in the sets just shown, such as mathematical symbols, the operator characters like / and <, and some other symbols. This includes whitespace characters.

Keywords can’t be used
: We listed the keywords in “Language Keywords” on page 51. Recall that some of them are combinations of operator and punctuation characters. For example, a single underscore (_) is a keyword!

Plain identifiers—combinations of letters, digits, `$`, _, and operators
: A *plain identifier* can begin with a letter or underscore, followed by more letters, digits, underscores, and dollar signs. Unicode-equivalent characters are also allowed. Scala reserves the dollar sign for internal use, so you shouldn’t use it in your own identifiers, although this isn’t prevented by the compiler. After an underscore, you can have either letters and digits, or a sequence of operator characters. The underscore is important. It tells the compiler to treat all the characters up to the next whitespace as part of the identifier. For example, `val xyz_++= = 1` assigns the variable `xyz_++=` the value `1`, while the expression `val xyz++= = 1` won’t compile because the identifier could also be interpreted as `xyz ++=`, which looks like an attempt to append something to `xyz`. Similarly, if you have operator characters after the underscore, you can’t mix them with letters and digits. This restriction prevents ambiguous expressions like this: `abc_-123`. Is that an identifier `abc_-123` or an attempt to subtract `123` from `abc_`?

Plain identifiers—operators
: If an identifier begins with an operator character, the rest of the characters must be operator characters.

Backtick literals
: An identifier can also be an arbitrary string between two backtick characters. For example, you could give your test methods names like this: `` def `test that addition works` = assert(1 + 1 == 2) ``. (Using this trick for literate test names is the one use I can think of for this otherwise questionable technique for using whitespace in identifiers.) Also use back quotes to invoke a method or variable in a non-Scala API when the name is identical to a Scala keyword—e.g., `` java.net.Proxy.`type`() ``.

Pattern-matching identifiers
: In pattern-matching expressions (for example, "A Sample Application" on page 20), tokens that begin with a lowercase letter are parsed as *variable identifiers*, while tokens that begin with an uppercase letter are parsed as *constant identifiers* (such as class names). This restriction prevents some ambiguities because of the very succinct variable syntax that is used (e.g., no `val` keyword is present).

Once you know that all operators are methods, it's easier to reason about unfamiliar Scala code. You don't have to worry about special cases when you see new operators. We've seen several examples where infix expressions like `matrix1 * matrix2` were used, which are actually just ordinary method invocations.

This flexible method naming gives you the power to write libraries that feel like a natural extension of Scala itself. You can write a new math library with numeric types that accept all the usual mathematical operators. The possibilities are constrained by just a few limitations for method names.

> Avoid making up operator symbols when an established alphanumeric name exists because the latter is easier to understand and remember, especially for beginners reading your code.

Methods with Empty Parameter Lists

Scala is flexible about whether or not parentheses are defined for methods with no parameters.

3 If a method takes no parameters, you can define it without parentheses. Callers must invoke the method without parentheses. (Scala 2 was more forgiving about inconsistent invocation.) Conversely, if you add empty parentheses to your definition, callers must add the parentheses.

For example, `Seq.size` has no parentheses, so you write `Seq(1, 2, 3).size`. If you try `Seq(1, 2, 3).size()`, you'll get an error.

However, exceptions are made for no-parameter methods in non-Scala libraries. For example, the `length` method for `java.lang.String` is defined with parentheses because Java requires them, but Scala lets you write either `"hello".length()` or `"hello".length`.

A convention in the Scala community is to omit parentheses for methods that have no side effects, like returning a field value. The size of a collection might be a precomputed, immutable field in the object, but even if it is computed on demand, calling `size` behaves like a reader method. However, when the method performs side effects or does extensive work, the convention is to add parentheses to provide a hint to the reader of nontrivial activity, for example `myFileReader.readLines()`.

Operator Precedence Rules

So if an expression like `2.0 * 4.0 / 3.0 * 5.0` is actually a series of method calls on `Doubles`, what are the operator precedence rules? Here they are in order from lowest to highest precedence:

1. All letters
2. `|`
3. `^`
4. `&`
5. `< >`
6. `= !`
7. `:`
8. `+ -`
9. `* / %`
10. All other special characters

Characters on the same line have the same precedence. An exception is `=` when it's used for assignment, in which case it has the lowest precedence.

Because `*` and `/` have the same precedence, the two lines in the following `scala` session behave the same:

```
scala> 2.0 * 4.0 / 3.0 * 5.0
res0: Double = 13.333333333333332
```

```
scala> (((2.0 * 4.0) / 3.0) * 5.0)
res1: Double = 13.333333333333332
```

Usually, method invocations using infix operator notation simply bind in left-to-right order (i.e., they are *left-associative*). However, not all methods work this way! Any method with a name that ends with a colon (`:`) binds to the right when used in infix notation, while all other methods bind to the left. For example, you can prepend an element to a `Seq` using the `+:` method (sometimes called *cons*, which is short for "constructor," a term from Lisp):

```
scala> val seq = Seq('b', 'c', 'd')
val seq: Seq[Char] = List(b, c, d)

scala> val seq2 = 'a' +: seq
val seq2: Seq[Char] = List(a, b, c, d)

scala> val seq3 = 'z'.+:(seq2)
1 |val seq3 = 'z'.+:(seq2)
  |           ^^^^^^
  |           value +: is not a member of Char

scala> val seq3 = seq2.+:('z')
val seq3: Seq[Char] = List(z, a, b, c, d)
```

Note that if we don't use infix notation, we have to put `seq2` on the left.

> Any method whose name ends with a `:` binds to the right, not the left, in infix operator notation.

3 Enumerations and Algebraic Data Types

While it's common to declare a type hierarchy to represent all the possible types of some parent abstraction, sometimes we know the list of them is fixed.

Enumerations are very useful in this case. Here are simple and more advanced enumerations for the days of the week:

```
// src/script/scala/progscala3/rounding/WeekDay.scala
enum WeekDaySimple:                                               ❶
  case Sun, Mon, Tue, Wed, Thu, Fri, Sat

enum WeekDay(val fullName: String):                               ❷
  case Sun extends WeekDay("Sunday")                              ❸
  case Mon extends WeekDay("Monday")
  case Tue extends WeekDay("Tuesday")
  case Wed extends WeekDay("Wednesday")
  case Thu extends WeekDay("Thursday")
```

```
  case Fri extends WeekDay("Friday")
  case Sat extends WeekDay("Saturday")

  def isWorkingDay: Boolean = ! (this == Sat || this == Sun)           ❹

import WeekDay.*

val sorted = WeekDay.values.sortBy(_.ordinal).toSeq                    ❺
assert(sorted == List(Sun, Mon, Tue, Wed, Thu, Fri, Sat))

assert(Sun.fullName == "Sunday")
assert(Sun.ordinal == 0)                                               ❻
assert(Sun.isWorkingDay == false)
assert(WeekDay.valueOf("Sun") == WeekDay.Sun)                          ❼
```

❶ Declare an enumeration, similar to declaring a class. The allowed values are declared with `case`.

❷ An alternative declaration with a field `fullName`. Declare fields with `val` if you want them to be accessible (e.g., `WeekDay.Sun.fullName`).

❸ The values are declared using the `case` keyword, and `fullName` is set.

❹ You can define methods.

❺ The `WeekDay.values` order does not match the declaration order, so we sort by the `ordinal`, a unique number for each case in declaration order, starting at `0`.

❻ Since `Sun` was declared first, its ordinal value is `0`.

❼ You can lookup an enumeration value by its name.

This is the new syntax for enumerations introduced in Scala 3.[1] We also saw a teaser example of an enumeration in "Sealed Class Hierarchies and Enumerations" on page 62, where we discussed an alternative approach, sealed type hierarchies. The new syntax lends itself to a more concise definition of *algebraic data types* (ADTs—not to be confused with *abstract data types*). An ADT is "algebraic" in the sense that transformations obey well-defined properties (think of addition with integers as an example). For example, transforming an element or combining two of them with an operation can only yield another element in the set.

Consider the following example that shows two ways to define an ADT for *tree data structures*, one using a `sealed` type hierarchy and one using an enumeration:

1 You can find an example that uses the Scala 2 syntax in the code examples, *src/script/scala-2/progscala3/rounding/WeekDay.scala*.

```scala
// src/script/scala/progscala3/rounding/TreeADT.scala

object SealedADT:
  sealed trait Tree[T]                                          ❶
  final case class Branch[T](                                   ❷
    left: Tree[T], right: Tree[T]) extends Tree[T]
  final case class Leaf[T](elem: T) extends Tree[T]             ❸

  val tree = Branch(
    Branch(
      Leaf(1),
      Leaf(2)),
    Branch(
      Leaf(3),
      Branch(Leaf(4),Leaf(5))))

object EnumADT:
  enum Tree[T]:                                                 ❹
    case Branch(left: Tree[T], right: Tree[T])
    case Leaf(elem: T)

  import Tree.*                                                 ❺
  val tree = Branch(
    Branch(
      Leaf(1),
      Leaf(2)),
    Branch(
      Leaf(3),
      Branch(Leaf(4),Leaf(5))))

SealedADT.tree                                                  ❻
EnumADT.tree
```

❶ Use a sealed type hierarchy. Valid for Scala 2 and 3.

❷ One subtype, a branch with left and right children.

❸ The other subtype, a leaf node.

❹ Scala 3 syntax using the new `enum` construct. It is much more concise.

❺ The elements of the `enum`, `Branch`, and `Leaf` need to be imported. They are nested under `Tree`, which is under `EnumADT`. In `SealedADT`, all three types were at the same level of nesting, directly under `SealedADT`.

❻ Is the output the same for these two lines?

The `enum` syntax provides the same benefits as sealed type hierarchies, but with less code.

The types of the tree values are slightly different (note the Branch versus Tree):

```
scala> SealedADT.tree
val res1: SealedADT.Branch[Int] = Branch(...)

scala> EnumADT.tree
val res2: EnumADT.Tree[Int] = Branch(...)
```

One last point: you may have noticed that Branch and Leaf don't extend Tree in EnumADT, while in WeekDay, each day extends WeekDay. For Branch and Leaf, extending Tree is inferred by the compiler, although we could add this explicitly. For WeekDay, each day must extend WeekDay to provide a value for the fullName field declared by WeekDay.

Interpolated Strings

We introduced interpolated strings in "A Sample Application" on page 20. Let's explore them further.

A String of the form s"foo ${bar}" will have the value of expression bar, converted to a String and inserted in place of ${bar}. If the expression bar returns an instance of a type other than String, the appropriate toString method will be invoked, if one exists. It is an error if it can't be converted to a String.

If bar is just a variable reference, the curly braces can be omitted. For example:

```
val name = "Buck Trends"
println(s"Hello, $name")
```

The standard library provides two other kinds of interpolated strings. One provides printf formatting and uses the prefix f. The other is called *raw* interpolated strings. It doesn't expand escape characters, like \n.

Suppose we're generating financial reports and we want to show floating-point numbers to two decimal places. Here's an example:

```
val gross   = 100000F
val net     = 64000F
val percent = (net / gross) * 100
println(f"$$${gross}%.2f vs. $$${net}%.2f or ${percent}%.1f%%")
```

The output of the last line is the following:

```
$100000.00 vs. $64000.00 or 64.0%
```

Scala uses Java's Formatter class for printf formatting. The embedded references to expressions use the same ${…} syntax as before, but printf formatting directives trail them with no spaces.

Two dollar signs, `$$`, are used to print a literal US dollar sign, and two percent signs, `%%`, are used to print a literal percent sign. The expression `${gross}%.2f` formats the value of `gross` as a floating-point number with two digits after the decimal point.

The types of variables used must match the format expressions, but some implicit conversions are performed. An `Int` expression in a floating point context is allowed. It just pads with zeros. However, attempting to use `Double` or `Float` in an `Int` context causes a compilation error due to the truncation that would be required.

While Scala uses Java strings, in certain contexts the Scala compiler will wrap a Java `String` with extra methods defined in `scala.collection.StringOps`. One of those extra methods is an *instance method* called `format`. You call it on the format string itself, then pass as arguments the values to be incorporated into the string. For example:

```
scala> val s = "%02d: name = %s".format(5, "Dean Wampler")
val s: String = "05: name = Dean Wampler"
```

In this example, we asked for a two-digit integer, padded with leading zeros.

The final version of the built-in string interpolation capabilities is the raw format that doesn't expand escape sequences. Consider these examples:

```
scala> val name = "Dean Wampler"
val name: String = "Dean Wampler"

scala> val multiLine = s"123\n$name\n456"
val multiLine: String = 123
Dean Wampler
456

scala> val multiLineRaw = raw"123\n$name\n456"
val multiLineRaw: String = 123\nDean Wampler\n456
```

Finally, we can define our own string interpolators, but we'll need to learn more about *context abstractions* first. See "Build Your Own String Interpolator" on page 142 for details.

Scala Conditional Expressions

Scala conditionals start with the `if` keyword. They are expressions, meaning they return a value that you can assign to a variable. In many languages, `if` conditionals are statements, which can only perform side-effect operations.

Here is an example using the new Scala 3 optional syntax for conditionals:

```
// src/script/scala/progscala3/rounding/If.scala

(0 until 6).map { n =>
  if n%2 == 0 then
```

```
    s"$n is even"
  else if n%3 == 0 then
    s"$n is divisible by 3"
  else
    n.toString
}
```

As discussed in "New Scala 3 Syntax—Optional Braces" on page 31, the `then` keyword is required only if you pass the `-new-syntax` flag to the compiler or REPL. (This is used in the code examples `build.sbt` file.) However, if you don't use that flag, you must wrap the predicate expressions, like `n%2 == 0`, in parentheses. If you use `-old-syntax` instead, then parentheses are required and `then` is disallowed.

The bodies of each clause are so concise, we can write them on the same line as the `if` or `else` expressions:

```
(0 until 6).map { n =>
  if n%2 == 0 then s"$n is even"
  else if n%3 == 0 then s"$n is divisible by 3"
  else n.toString
}
```

Here are the same examples using the original control syntax, with and without curly braces:

```
// src/script/scala-2/progscala3/rounding/If.scala

(0 until 6).map { n =>
  if (n%2 == 0) {
    s"$n is even"
  } else if (n%3 == 0) {
    s"$n is divisible by 3"
  } else {
    n
  }
}

(0 until 6).map { n =>
  if (n%2 == 0) s"$n is even"
  else if (n%3 == 0) s"$n is divisible by 3"
  else n
}
```

What is the type of the returned value if objects of different types are returned by different branches? The type will be the *least upper bound* of all the branches, the closest supertype that matches all the potential values from each clause.

In the following example, the least upper bound is `Option[String]` because the three branches return either `Some[String]` or None. The returned sequence is of type `IndexedSeq[Option[String]]`:

```
// src/script/scala/progscala3/rounding/IfTyped.scala

scala> val seq = (0 until 6) map { n =>
     |   if n%2 == 0 then Some(n.toString)
     |   else None
     | }
val seq: IndexedSeq[Option[String]] = Vector(Some(0), None, Some(2), ...)
```

Conditional and Comparison Operators

Table 3-1 lists the operators that can be used in conditional expressions.

Table 3-1. Conditional and comparison operators

Operator	Operation	Description
`&&`	and	The values on the left and right of the operator are true. The righthand side is only evaluated if the lefthand side is true.
`||`	or	At least one of the values on the left or right is true. The righthand side is only evaluated if the lefthand side is false.
`>`	greater than	The value on the left is greater than the value on the right.
`>=`	greater than or equal to	The value on the left is greater than or equal to the value on the right.
`<`	less than	The value on the left is less than the value on the right.
`<=`	less than or equal to	The value on the left is less than or equal to the value on the right.
`==`	equal to	The value on the left is equivalent to the value on the right.
`!=`	not equal to	The value on the left is not equivalent to the value on the right.

The `&&` and `||` operators are *short-circuiting*. They stop evaluating expressions as soon as the answer is known. This is handy when you must work with `null` values:

```
scala> val s: String|Null = null
val s: String | Null = null

scala> val okay = s != null && s.length > 5
val okay: Boolean = false
```

Calling `s.length` would throw a `NullPointerException` without the `s != null` test first. Note that we don't use `if` here because we just want to know the `Boolean` value of the expression.

The equivalence operators, `==` and its negation `!=`, check for logical equivalence between instances, such as comparing field values. The `equals` method for the type on the lefthand side is invoked for this purpose. You can implement this method yourself, but it's uncommon to do so because most of the time you compare case-class instances where the compiler generated `equals` automatically for you! Most of the Scala library types also define `equals`.

If you need to determine if two values are identical references, use the `eq` method or its negation, `ne`.

See "Equality of Instances" on page 292 for more details.

for Comprehensions

Another familiar control structure that's particularly feature rich in Scala is the `for` loop, called `for` *comprehension*. They are expressions, not statements.

The term *comprehension* comes from set theory and has been used in several FP languages. The term expresses the idea that we define a set or other collection by enumerating the members explicitly or by specifying the properties that all members satisfy. *List comprehension* in Python is a similar concept.

for Loops

Let's start with a basic `for` expression. As for `if` expressions, I use the new format options consistently in the code examples, except where noted:

```
// src/script/scala/progscala3/rounding/BasicFor.scala

for
  i <- 0 until 10            // Recall "until" means 10 is exclusive.
do println(i)
```

Since there is one expression inside the `for…do`, you can put the expression on the same line after the `for`, and you can even put everything on one line:

```
for i <- 0 until 10
do println(i)

for i <- 0 until 10 do println(i)
```

As you might guess, this code says, "For every integer between 0 inclusive and 10 exclusive, print it on a new line."

The `do` keyword indicates that nothing will be returned. Only side effects are performed. These kinds of `for` comprehensions are sometimes called `for` *loops*.

The original Scala 2 syntax is still supported, where parentheses or curly braces are required and `do` is not used. The examples are written as follows:

```
// src/script/scala-2/progscala3/rounding/BasicFor.scala

for (i <- 0 until 10)
  println(i)

for (i <- 0 until 10) println(i)
```

For all the `for` comprehension forms we'll examine, neither the `-new-syntax` nor the `-old-syntax` flag affect which syntax is allowed or restricted. Both are always allowed.

> From now on, I'll only show Scala 3 syntax, but you can find Scala 2 versions of some examples in the code examples under the directory *src/*/scala-2/progscala3/...* and a table of differences in Table A-1.

Generators

The expression `i <- 0 until 10` is called a *generator*, so named because it generates individual values in some way. The left arrow operator (`<-`) is used to iterate through any instance that supports iterative access to elements, such as `Seq` and `Vector`, and also `Set` and `Map`, where order isn't guaranteed:

```
scala> for i <- Set(0,2,1,2,3,4,4,5) do print(s"$i|")
0|5|1|2|3|4|

scala> for (key, value) <- Map("one" -> 1, "two" -> 2, "three" -> 3)
     | do println(s"$key -> $value")
one -> 1
two -> 2
three -> 3
```

Guards: Filtering Values

We can add `if` expressions, called *guards*, to filter elements:

```
// src/script/scala/progscala3/rounding/GuardFor.scala

for
  n <- 0 to 6                // Recall "to" means 6 is inclusive.
  if n%2 == 0
do println(n)
```

The output is the numbers 0, 2, 4, and 6. Note the sense of filtering; the guards express what to keep, not remove.

Yielding New Values

So far our `for` loops have only performed side effects, writing to output. Usually, we want to return a new collection, making our `for` expressions comprehensions rather than loops. We use the `yield` keyword to express this intent:

```
// src/script/scala/progscala3/rounding/YieldingFor.scala

val evens = for
```

```scala
  n <- 0 to 10
  if n%2 == 0
yield n

assert(evens == Vector(0, 2, 4, 6, 8, 10))
```

Each iteration through the `for` expression yields a new value held by `n`. These are accumulated into a new collection that is assigned to the variable `evens`.

The type of collection returned by a `for` comprehension is inferred from the type of collection being iterated over. Here, we started with a `Range`, but the comprehension actually returns a `Vector`.

In the following example, a `Vector[Int]` is converted to a `Vector[String]`:

```scala
// src/script/scala/progscala3/rounding/YieldingForVector.scala

val odds = for
  number <- Vector(1,2,3,4,5)
  if number % 2 == 1
yield number.toString

assert(odds == Vector("1", "3", "5"))
```

Expanded Scope and Value Definitions

You can define immutable values inside the `for` expressions without using the `val` keyword, like `fn` in the following example that uses the `WeekDay` enumeration we defined earlier in this chapter:

```scala
// src/script/scala/progscala3/rounding/ScopedFor.scala

import progscala3.rounding.WeekDay

val days = for
  day <- WeekDay.values
  if day.isWorkingDay
  fn = day.fullName
yield fn

assert(days.toSeq.sorted ==
  Seq("Friday", "Monday", "Thursday", "Tuesday", "Wednesday"))
```

In this case, the `for` comprehension now returns an `Array[String]` because `WeekDay.values` returns an `Array[WeekDay]`. Because `Arrays` are Java `Arrays` and Java doesn't define a useful `equals` method, we convert to a `Seq` with `toSeq` and perform the assertion check.

Now let's consider a powerful use of `Option` with `for` comprehensions. Recall we discussed `Option` as a better alternative to using `null`. It's also useful to recognize that

Option behaves like a special kind of collection, limited to zero or one elements. In fact, we can "comprehend" it too:

```
// src/script/scala/progscala3/rounding/ScopedOptionFor.scala

import progscala3.rounding.WeekDay
import progscala3.rounding.WeekDay.*

val dayOptions = Seq(
  Some(Mon), None, Some(Tue), Some(Wed), None,
  Some(Thu), Some(Fri), Some(Sat), Some(Sun), None)

val goodDays1 = for          // First pass
  dayOpt <- dayOptions
  day <- dayOpt
  fn   = day.fullName
yield fn
assert(goodDays1 ==
  Seq("Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday", "Sunday"))
```

Imagine that we call some services to return days of the week. The services return Options because some of them can return a day of the week, while others can't. Some services can return a value like Some(Tue), for example, while others return None. Now we want to remove and ignore the None values.

In the first expression of the for comprehension for goodDays1, each element extracted is an Option, assigned to dayOpt. The next line uses the arrow to extract the value in the option and assign it to day.

But wait! Doesn't None throw an exception if you try to extract a value from it? Yes, but the comprehension effectively checks for this case and skips the Nones. It's as if we added an explicit if dayOpt != None before the second line.

Hence, we construct a collection with only values from Some instances.

This can be written more concisely:

```
val goodDays2 = for          // second, more concise pass
  case Some(day) <- dayOptions
  fn = day.fullName
yield fn
assert(goodDays2 ==
  Seq("Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday", "Sunday"))
```

This version makes the filtering even cleaner and more concise, using pattern matching. The expression case Some(day) <- dayOptions only succeeds when the instance is a Some, skipping the None values, and it extracts the value into day, all in one step. We'll explore pattern matching in depth in Chapter 4.

To recap, use a generator clause (with the left arrow, `<-`) when you are iterating through a collection and extracting values. Use an assignment (with the equals sign, `=`) when you are assigning a value from an expression that doesn't involve iteration. `for` comprehensions are required to start with a generator clause. If you really need to define a value first, put it before the comprehension.

When working with loops in many languages, they provide `break` and `continue` keywords for breaking out of a loop completely or continuing to the next iteration, respectively. Scala doesn't have either of these keywords, but when writing idiomatic Scala code, they aren't missed. Use conditional expressions to test if a loop should continue, or make use of recursion. Better yet, filter your collections ahead of time to eliminate complex conditions within your loops.

Scala while Loops

The `while` loop is less frequently used. It executes a block of code while a condition is true:

```
// src/script/scala/progscala3/rounding/While.scala

var count = 0
while count < 10
do
  count += 1
  println(count)

assert(count == 10)
```

Scala 3 dropped the `do-while` construct in Scala 2 because it was rarely used. It can be rewritten using `while`, although awkwardly:

```
// src/script/scala/progscala3/rounding/DoWhileAlternative.scala

var count = 0
while
  count += 1
  println(count)
  count < 10
do {}
assert(count == 10)
```

Using try, catch, and finally Clauses

Through its use of functional constructs and strong typing, Scala encourages a coding style that lessens the need for exceptions and exception handling. However, exceptions are still supported, in part because they are common in non-Scala libraries.

Unlike Java, Scala does not have *checked exceptions*. Java's checked exceptions are treated as unchecked by Scala. There is also no `throws` clause on method declarations. However, there is a `@throws` annotation that is useful for Java interoperability. See "Annotations" on page 468.

You throw an exception by writing `throw MyException(…)`. To catch exceptions, Scala uses pattern matching to specify the exceptions to be caught.

The following example implements a common application scenario—resource management. We want to open files and process them in some way. In this case, we'll just count the lines. However, we must handle a few error scenarios. The file might not exist, perhaps because the user misspelled the filenames. Also, something might go wrong while processing the file. (We'll trigger an arbitrary failure to test what happens.) We need to ensure that we close all open file handles, whether or not we process the files successfully:

```
// src/main/scala/progscala3/rounding/TryCatch.scala
package progscala3.rounding
import scala.io.Source                                              ❶
import scala.util.control.NonFatal

/** Usage: scala rounding.TryCatch filename1 filename2 ... */
@main def TryCatch(fileNames: String*) =                            ❷
  fileNames.foreach { fileName =>
    var source: Option[Source] = None                               ❸
    try                                                             ❹
      source = Some(Source.fromFile(fileName))                      ❺
      val size = source.get.getLines.size
      println(s"file $fileName has $size lines")
    catch
      case NonFatal(ex) => println(s"Non fatal exception! $ex")     ❻
    finally
      for s <- source do                                            ❼
        println(s"Closing $fileName...")
        s.close
  }
```

❶ Import `scala.io.Source` for reading input and `scala.util.control.NonFatal` for matching on *nonfatal* exceptions (i.e., those where it's reasonable to attempt recovery).

❷ Use the `@main` annotation to mark the method as the program entry point. The arguments we expect are zero or more strings.

❸ Declare the `source` to be an `Option`, so we can tell in the `finally` clause if we successfully created an instance or not. We use a mutable variable, but it's hidden inside the implementation, and thread safety isn't a concern in this code.

4. Start of the `try` clause.

5. `Source.fromFile` will throw a `java.io.FileNotFoundException` if the file doesn't exist. Otherwise, wrap the returned `Source` instance in a `Some`. Calling `get` on the next line is safe because if we're here, we know we have a `Some`. If `source` were still a `None`, an exception would be thrown by `get`.

6. Catch nonfatal errors. For example, out of memory would be fatal.

7. Use a `for` comprehension to extract the `Source` instance from the `Some` and close it. If `source` is `None`, then nothing happens.

Note the `catch` clause. Scala uses pattern matching to specify the exceptions you want to catch. In this case, the clause `case NonFatal(ex) =>…` `scala.util.control.NonFatal` matches on and extracts any exception that isn't considered fatal, binding the exception instance to `ex`.

The `finally` clause is used to ensure proper resource cleanup in one place. Without it, we would have to repeat the logic at the end of the `try` clause and the `catch` clause to ensure our file handles are closed. Here we use a `for` comprehension to extract the `Source` from the option. If the option is actually a `None`, nothing happens; the block with the `close` call is not invoked. Note that since this is the main method, the handles would be cleaned up anyway on exit, but you'll want to close resources in other contexts.

When resources need to be cleaned up, whether or not the resource is used successfully, put the cleanup logic in a `finally` clause.

This program is already compiled by `sbt`, and we can run it from the `sbt` prompt using the `runMain` task, which lets us pass arguments. I have elided some output:

```
> runMain progscala3.rounding.TryCatch README.md foo/bar
file README.md has 148 lines
Closing README.md...
Non fatal exception! java.io.FileNotFoundException: foo/bar (...)
```

While I'll rarely use `null` in this book, for reasons we saw in "Option, Some, and None: Avoiding Nulls" on page 60, there are times when you might use `null` very carefully instead of `Option`, like in the previous example, in order to simplify the code:

```
// src/script/scala/progscala3/rounding/Uninitialized.scala
import scala.io.Source
```

```scala
import scala.compiletime.uninitialized                                    ❶

case class LineLoader(file: java.io.File):
  private var source: Source = uninitialized                              ❷
  val lines = try
    source = Source.fromFile("README.md")
    source.getLines.toSeq
  finally
    if source != null then source.close

val ll = LineLoader(java.io.File("README.md"))
assert(ll.lines.take(1) == List("# Programming Scala, 3rd Edition"))
```

❶ Import a special `uninitialized` value.

❷ Use it when initializing a `var` *field* to `null`.

In Scala 2, `_` was used for uninitialized `var` fields. This is deprecated in Scala 3 because `uninitialized` makes the intention more clear. For `var`s declared in methods, you have to use `null`. Concrete `val`s must always be initialized.

Automatic resource management is a common pattern. Let's use a Scala library facility, `scala.util.Using`, for this purpose.[2] Then we'll actually implement our own version to illustrate some powerful capabilities in Scala and better understand how the library version works.

```scala
// src/main/scala/progscala3/rounding/FileSizes.scala
package progscala3.rounding

import scala.util.Using
import scala.io.Source

/** Usage: scala rounding.FileSizes filename1 filename2 ... */
@main def FileSizes(fileNames: String*) =
  val sizes = fileNames.map { fileName =>
    Using.resource(Source.fromFile(fileName)) { source =>
      source.getLines.size
    }
  }
  println(s"Returned sizes: ${sizes.mkString(", ")}")
  println(s"Total size: ${sizes.sum}")
```

This simple program also counts the number of lines in the files specified on the command line. However, if a file is not readable or doesn't exist, an exception is thrown and processing stops. No other results are produced, unlike the preceding `TryCatch` example, which continues processing the arguments specified.

2 Not to be confused with the keyword `using` that we discussed in "A Taste of Futures" on page 42.

See the scala.util.Using documentation for a few other ways this utility can be used. For more sophisticated approaches to error handling, see "Retry Failing Tasks with Cats and Scala".

Call by Name, Call by Value

Now let's implement our own *application resource manager* to learn a few powerful techniques that Scala provides for us. This implementation will build on the TryCatch example:

```
// src/main/scala/progscala3/rounding/TryCatchARM.scala
package progscala3.rounding
import scala.language.reflectiveCalls
import reflect.Selectable.reflectiveSelectable
import scala.util.control.NonFatal
import scala.io.Source

object manage:
  def apply[R <: { def close():Unit }, T](resource: => R)(f: R => T): T =
    var res: Option[R] = None
    try
      res = Some(resource)         // Only reference "resource" once!!
      f(res.get)                   // Return the T instance
    catch
      case NonFatal(ex) =>
        println(s"manage.apply(): Non fatal exception! $ex")
        throw ex
    finally
      res match
        case Some(resource) =>
          println(s"Closing resource...")
          res.get.close()
        case None => // do nothing

/** Usage: scala rounding.TryCatchARM filename1 filename2 ... */
@main def TryCatchARM(fileNames: String*) =
  val sizes = fileNames.map { fileName =>
    try
      val size = manage(Source.fromFile(fileName)) { source =>
        source.getLines.size
      }
      println(s"file $fileName has $size lines")
      size
    catch
      case NonFatal(ex) =>
        println(s"caught $ex")
        0
  }
  println("Returned sizes: " + (sizes.mkString(", ")))
```

The output will be similar what we saw for TryCatch.

This is a lovely little bit of *separation of concerns*, but to implement it, we used a few new power tools.

First, we named our object `manage` rather than `Manage`. Normally, you follow the convention of using a leading uppercase letter for type names, but in this case we will use `manage` like a *function*. We want client code to look like we're using a built-in operator, similar to a `while` loop. This is another example of Scala's tools for building little DSLs.

That `manage.apply` method declaration is hairy looking. Let's deconstruct it. Here is the signature again, spread over several lines and annotated:

```
def apply[
  R <: { def close():Unit },       ❶
  T ]                              ❷
  (resource: => R)                 ❸
  (f: R => T) = ...                ❹
```

❶ Two new things are shown here. `R` is the type of the resource we'll manage. The `<:` means `R` is a subtype of something else. In this case, *any type used for `R` must contain a `close():Unit` method.* We declare this using a *structural type* defined with the braces. What would be more intuitive, especially if you are new to structural types, would be for all resources to implement a `Closable` interface that defines a `close():Unit` method. Then we could say `R <: Closable`. Instead, structural types let us use reflection and plug in any type that has a `close():Unit` method (like `Source`). Reflection has a lot of overhead and structural types are a bit scary, so reflection is an *optional feature*. Hence, we added the first two import statements to tell the compiler to enable this feature.

❷ `T` will be the type returned by the anonymous function passed in to do work with the resource.

❸ It looks like `resource` is a function with an unusual declaration. Actually, `resource` is a *by-name* parameter, which we first encountered in "A Taste of Futures" on page 42.

❹ Finally we have a second parameter list containing a function for the work to do with the resource. This function will take the resource as an argument and return a result of type `T`.

Recapping point 1, here is how the `apply` method declaration would look if we could assume that all resources implement a `Closable` abstraction:

```
object manage:
  def apply[R <: Closable, T](resource: => R)(f: R => T) =
    ...
```

The line, `res = Some(resource)`, is the only place `resource` is evaluated, which is important because it is a by-name parameter. We learned in "A Taste of Futures" on page 42 that they are lazily evaluated, only when used, but they are evaluated every time they are referenced, just like a function call would be. The thing we pass as `resource` inside `TryCatchARM`, `Source.fromFile(fileName)`, should only be evaluated *once* inside `apply` to construct the `Source` for a file. The code correctly evaluates it once.

So you have to use by-name parameters carefully, but their virtue is the ability to control when and even if a block of code is evaluated. We'll see another example shortly where we will evaluate a by-name parameter repeatedly for a good reason.

To recap, it's as if the `res =` … line is actually this:

```
res = Some(Source.fromFile(fileName))
```

After constructing `res`, it is passed to the work function `f`.

See how `manage` is used in `TryCatchARM`. It looks like a built-in control structure with one parameter list that creates the `Source`, and a second parameter list that is a block of code that works with the `Source`. So using `manage` looks something like a conventional `while` statement.

Like most languages, Scala normally uses *call-by-value* semantics. If we write `val source = Source.fromFile(fileName)`, it is evaluated immediately.

Supporting idiomatic code like our use of `manage` is the reason that Scala offers by-name parameters, without which we would have to pass an anonymous function that looks ugly:

```
manage(() => Source.fromFile(fileName)) { source =>
```

Then, within `manage.apply`, our reference to `resource` would now be a function call:

```
val res = Some(resource())
```

OK, that's not a terrible burden, but *call by name* enables a syntax for building our own control structures, like our `manage` utility.

Here is another example using call by name, this time repeatedly evaluating *two* by-name parameters to implement a `while`-like loop construct called `continue`:

```
// src/script/scala/progscala3/rounding/CallByName.scala
import scala.annotation.tailrec

@tailrec                                                              ❶
def continue(conditional: => Boolean)(body: => Unit): Unit =          ❷
  if conditional then                                                 ❸
    body
    continue(conditional)(body)
```

```
var count = 0
continue (count < 5) {                                          ❹
  println(s"at $count")
  count += 1
}
assert(count == 5)
```

❶ Ensure the implementation is tail recursive.

❷ Define a `continue` function that accepts two argument lists. The first list takes a single, by-name parameter that is the conditional. The second list takes a single, by-name value that is the body to be evaluated for each iteration.

❸ Evaluate the condition. If true, evaluate the body and call `continue` recursively.

❹ Try it!

So by-name parameters are evaluated every time they are referenced. In a sense, they are *lazy* because evaluation is deferred, but possibly repeated over and over again. Scala also provides lazy values, which are initialized once, but only when used.

Notice that our `continue` implementation shows how loop constructs can be replaced with recursion.

Unfortunately, this ability to define our own control structures doesn't work as nicely with the new braceless syntax. We have to use parentheses and braces, as shown. If `continue` really behaved like `while` or similar built-in constructs, we would be able to use it with the same syntax `while` supports. However, a future release of Scala 3 may support it.

By the way, by-name parameters are a less obvious example of type erasure, which we discussed previously. Suppose we tried to add a second definition of `continue` that stops if an integer by-name parameter goes negative:

```
def continue(conditional: => Boolean)(body: => Unit): Unit = ...
def continue(nonNegative: => Int)(body: => Unit): Unit = ...
```

These two definitions are considered identical because the implementation type of a by-name parameter is a function type, `Function0`. The `0` is because these functions take no arguments, but they return a value, of type `Boolean` or `Int` in our case. Hence, they have a type parameter for the return type. You can remove the ambiguity here using `@targetName` as before.

Lazy Values

By-name parameters show us that lazy evaluation is useful, but they are evaluated every time they are referenced.

There are times when you want to evaluate an expression once to initialize a field in an instance, but you want to defer that invocation until the value is actually needed. In other words, on-demand evaluation. This is useful when:

- The expression is expensive (e.g., opening a database connection) and you want to avoid the overhead until the value is actually needed, which could be never.
- You want to improve startup times for modules by deferring work that isn't needed immediately.
- A field in an instance needs to be initialized lazily so that other initializations can happen first.

We'll explore the last scenario when we discuss "Initializing Abstract Fields" on page 305.

Here is a sketch of an example using a `lazy val`:

```
// src/script/scala/progscala3/rounding/LazyInitVal.scala

case class DBConnection():
  println("In constructor")
  type MySQLConnection = String
  lazy val connection: MySQLConnection =
    // Connect to the database
    println("Connected!")
    "DB"
```

The `lazy` keyword indicates that evaluation will be deferred until the value is accessed.

Let's try it. Notice when the `println` statements are executed:

```
scala> val dbc = DBConnection()
In constructor
val dbc: DBConnection = DBConnection()

scala> dbc.connection
Connected!
val res4: dbc.MySQLConnection = DB

scala> dbc.connection
val res5: dbc.MySQLConnection = DB
```

So how is a `lazy val` different from a method call? We see that "Connected!" was only printed once, whereas if `connection` were a method, the body would be executed *every* time and we would see "Connected!" printed each time. Furthermore, we didn't see that message until we referenced `connection` the first time.

One-time evaluation makes little sense for a mutable field. Therefore, the `lazy` keyword is not allowed on `var`s.

Lazy values are implemented with the equivalent of a guard. When client code references a lazy value, the reference is intercepted by the guard to check if initialization is required. This guard step is really only essential the first time the value is referenced, so that the value is initialized first before the access is allowed to proceed. Unfortunately, there is no easy way to eliminate these checks for subsequent calls. So lazy values incur overhead that eager values don't. Therefore, you should only use lazy values when initialization is expensive, especially if the value may not actually be used. There are also some circumstances where careful ordering of initialization dependencies is most easily implemented by making some values lazy (see "Initializing Abstract Fields" on page 305).

There is a `@threadUnsafe` annotation you can add to a `lazy val` (in package `scala.annotation`). It causes the initialization to use a faster mechanism that is not thread-safe, so use it with caution.

Traits: Interfaces and Mixins in Scala

Until now, I have emphasized the power of FP in Scala. I waited until now to discuss Scala's features for OOP, such as how abstractions and concrete implementations are defined and how inheritance is supported. We've seen some details in passing, like abstract and case classes and objects, but now it's time to cover these concepts.

Scala uses traits to define abstractions. We'll explore most details in Chapter 10, but for now, think of them as interfaces for declaring abstract member fields, methods, and types, with the option of defining any or all of them, too.

Traits enable true separation of concerns and composition of behaviors (*mixins*).

Here is a typical enterprise developer task, adding logging. Let's start with a service:

```
// src/script/scala/progscala3/rounding/Traits.scala
import util.Random

open class Service(name: String):                                    ❶
  def work(i: Int): (Int, Int) = (i, Random.between(0, 1000))

val service1 = new Service("one")
(1 to 3) foreach (i => println(s"Result: ${service1.work(i)}"))
```

❶ This is a concrete class, but we intend to extend it—that is, to create subtypes from it. In Scala 3, you declare such concrete classes with `open`.

3 The `open` keyword is optional in Scala 3.0. It indicates that subtypes can be derived from this concrete class. See "Classes Open for Extension" on page 250 for details.

The output of the last line is the following:

```
Result: (1,975)
Result: (2,286)
Result: (3,453)
```

Now we want to mix in a standard logging library. For simplicity, we'll just use `println`.

Here is an `enum` for the logging level and two traits, one that defines the abstraction and the other that implements the abstraction for logging to standard output:

```
enum Level:                                                          ❶
  case Info, Warn, Error
  def ==(other: Level): Boolean = this.ordinal == other.ordinal
  def >=(other: Level): Boolean = this.ordinal >= other.ordinal

trait Logging:
  import Level.*

  def level: Level                                                   ❷
  def log(level: Level, message: String): Unit

  final def info(message: String): Unit =                           ❸
    if level >= Info then log(Info, message)
  final def warn(message: String): Unit =
    if level >= Warn then log(Warn, message)
  final def error(message: String): Unit =
    if level >= Error then log(Error, message)

trait StdoutLogging extends Logging:                                 ❹
  def log(level: Level, message: String) =
    println(s"${level.toString.toUpperCase}: $message")
```

❶ For simplicity, just consider three levels and define two of the possible comparison operators for them, relying on the built-in `ordinal` value for each case. Note how concisely we could write the three cases with just one `case` keyword.

❷ Implementers will need to define the current logging level and the `log` method.

❸ The three methods `info`, `warning`, and `error` are declared `final`. If the current logging level is less than or equal to the level for each method, then call the general `log` method, which subtypes must implement.

❹ Log to standard output.

Finally, let's declare a service that "mixes in" logging and use it:

```
case class LoggedService(name: String, level: Level)
    extends Service(name) with StdoutLogging:
  override def work(i: Int): (Int, Int) =
    info(s"Starting work: i = $i")
    val result = super.work(i)
```

```
        info(s"Ending work: result = $result")
        result

    val service2 = LoggedService("two", Level.Info)
    (1 to 3) foreach (i => println(s"Result:  ${service2.work(i)}"))
```

Overriding the `work` method allows us to log when we enter and before we leave the method. Scala requires the `override` keyword when you override a concrete method in a supertype. This prevents mistakes when you didn't know you were overriding a method, for example from a library supertype. It also catches misspelled method names that aren't actually overrides! Note how we access the supertype `work` method, using `super.work`.

Here is the output (the numbers will vary randomly):

```
INFO:    Starting work: i = 1
INFO:    Ending work: result = (1,737)
Result:  (1,737)
INFO:    Starting work: i = 2
INFO:    Ending work: result = (2,310)
Result:  (2,310)
INFO:    Starting work: i = 3
INFO:    Ending work: result = (3,273)
Result:  (3,273)
```

> Be very careful about overriding concrete methods! In this case, we don't change the behavior of the supertype method. We just log activity, then call the supertype method, then log again. We are careful to return the result unchanged that was returned by the supertype method.

To mix in traits while constructing an instance as shown, we use the `with` keyword. We can mix in as many as we want. Some traits might not modify existing behavior at all and just add new useful, but independent, methods.

In this example, we modify the behavior of `work`, in order to inject logging, but we are not changing its *contract* with clients, that is, its external behavior.[3]

There's one more detail you might have noticed; the `Logging.level` method was not defined in `LoggedService`, was it? In fact, it was defined, using the field `level` in the constructor argument list. In Scala, an abstract method that takes no arguments can be implemented by a `val` field in a subtype. The reason this is safe is because the method signature only says that some instance of the declared type will be returned, possibly a different instance on every invocation (like `math.random` works). However,

3 That's not strictly true, in the sense that the extra I/O has changed the code's interaction with the "world."

if we use a `val`, only a single instance will ever be returned, but that satisfies the method's "specification."

> A corollary is this; when declaring an abstract field in a supertype, consider using a no-parameter method declaration instead. This gives concrete implementations greater flexibility to use either a `val` or a method to implement it.

There is a lot more to discuss about traits and mixin composition, as we'll see.

3 When `new` Is Optional

In "A Sample Application" on page 20, we saw that `new` can be omitted when constructing most instances, even for noncase classes. For case classes, the `apply` method in the companion object is used. Other objects offer custom `apply` methods, like `Seq.apply`, where `Seq` itself isn't concrete. For all other types, you had to use `new` in Scala 2.

Scala 3 extends the case-class scheme to all concrete classes. It generates a *synthetic object* with `apply` methods corresponding to the constructors in the class, even for library types compiled in other languages and Scala 2. *Auxiliary* (or secondary) *constructors* are uncommon in Scala types, so we'll wait until "Constructors in Scala" on page 262 to discuss them in detail, but here is an example:

```
// src/script/scala/progscala3/typelessdomore/OptionalNew.scala

class Person(name: String, age: Int):
  def this() = this("unknown", 0)                                   (1)
```

(1) Auxiliary constructors are named `this`. They must call another constructor.

This feature is called *universal apply methods*, in the sense of using `apply` to create things. These `apply` methods are called *constructor proxies*. For example:

```
import java.io.File
val file = File("README.md")     // No "new" needed, even for Java classes!
```

The motivation for this feature is to provide more uniform syntax.

A few rules to keep in mind:

- If a class already has a companion object (i.e., user-defined), the synthetic object won't be generated.
- If the object already has an `apply` method with a parameter list matching a constructor, then a constructor proxy for it won't be generated.

- When a constructor takes no arguments, rewrite `new Foo` with `Foo()`. Omitting the parentheses would be ambiguous for the compiler.
- For a type `Foo` with a companion object, you should still write `new Foo(...)` inside the object's `apply` methods when you want to call a constructor. Writing `Foo(...)` without `new` will be interpreted as `Foo.apply(...)`, if the arguments match one of the `apply` method's parameter lists, leading to infinite recursion! This has always been necessary in Scala, of course, but it bears repeating in this context.
- Anonymous classes require `new`.

An *anonymous* class is useful when you need just one instance of something, so defining a named class is not necessary. It is created from a trait or a class, where any abstract members are implemented within the declaration:

```
trait Welcome:
  def hello(name: String): Unit

val hello = new Welcome:
  def hello(name: String): Unit = println(s"Hello: $name")
```

There is no synthesized object for `Welcome` because it is not concrete, nor is one created for the anonymous class *on the fly*, so `new` is required.

For case-class companion objects, only the primary constructor gets an autogenerated `apply` method, while in synthetic objects, all constructors get a corresponding `apply` method (*constructor proxy*). This difference is because it's more common in Java libraries to define and use multiple constructors, and there is no concept of primary versus auxiliary constructors. Hence, all of them need to be supported.

Recap and What's Next

We've covered a lot of ground in these first three chapters. We learned how flexible and concise Scala code can be. In this chapter, we learned some powerful constructs for defining DSLs and for manipulating data, such as `for` comprehensions. Finally, we learned more about enumerations and the basic capabilities of `traits`.

You should now be able to read quite a lot of Scala code, but there's plenty more about the language to learn. Next we'll begin a deeper dive into Scala features.

CHAPTER 4

Pattern Matching

Scala's pattern matching provides deep inspection and decomposition of objects in a variety of ways. It's one of my favorite features in Scala. For your own types, you can follow a protocol that allows you to control the visibility of internal state and how to expose it to users. The terms *extraction* and *destructuring* are sometimes used for this capability.

Pattern matching can be used in several code contexts, as we've already seen in "A Sample Application" on page 20 and "Partial Functions" on page 36. We'll start with a change in Scala 3 for better type safety, followed by a quick tour of common and straightforward usage examples, then explore more advanced scenarios. We'll cover a few more pattern-matching features in later chapters, once we have the background to understand them.

3 Safer Pattern Matching with Matchable

Let's begin with an important change in Scala 3's type system that is designed to make compile-time checking of pattern-matching expressions more robust.

Scala 3 introduced an immutable wrapper around `Arrays` called `scala.IArray`. `Arrays` in Java are mutable, so this is intended as a safer way to work with them. In fact, `IArray` is a type alias for `Array` to avoid the overhead of wrapping arrays, which means that pattern matching introduces a hole in the abstraction. Using the Scala 3.0 REPL without the `-source:future` setting, observe the following:

```
// src/script/scala/progscala3/patternmatching/Matchable.scala
scala> val iarray = IArray(1,2,3,4,5)
     | iarray match
     |   case a: Array[Int] => a(2) = 300 // Scala 3 warning!!
     | println(iarray)
val iarray: opaques.IArray[Int] = Array(1, 2, 300, 4, 5)
```

There are other examples where this can occur. To close this loophole, The Scala type system now has a trait called Matchable. It fits into the type hierarchy as follows:

```
abstract class Any:
  def isInstanceOf
  def getClass
  def asInstanceOf      // Cast to a new type: myAny.asInstanceOf[String]
  def ==
  def !=
  def ##   // Alias for hashCode
  def equals
  def hashCode
  def toString

trait Matchable extends Any

class AnyVal extends Any, Matchable

class AnyRef extends Any, Matchable
```

Note that `Matchable` is a *marker trait*, as it currently has no members. However, a future release of Scala may move `getClass` and `isInstanceOf` to `Matchable`, as they are closely associated with pattern matching.

The intent is that pattern matching can only occur on values of type `Matchable`, not `Any`. Since almost all types are subtypes of `AnyRef` and `AnyVal`, they already satisfy this constraint, but attempting to pattern match on the following types will trigger warnings in future Scala 3 releases or when using `-source:future` with Scala 3.0:

- Type `Any`. Use `Matchable` instead, when possible.
- Type parameters and abstract types without bounds. Add `<: Matchable`.
- Type parameters and abstract types bounded only by *universal traits*. Add `<: Matchable`.

We'll discuss universal traits in "Value Classes" on page 258. We can ignore them for now. As an example of the second bullet, consider the following method definition in a REPL session with the `-source:future` flag restored:

```
scala> def examine[T](seq: Seq[T]): Seq[String] = seq map {
     |   case i: Int => s"Int: $i"
     |   case other => s"Other: $other"
     | }
2 |  case i: Int => s"Int: $i"
  |          ^^^
  |          pattern selector should be an instance of Matchable,
  |          but it has unmatchable type T instead
```

Now the type parameter `T` needs a bound:

```
scala> def examine[T <: Matchable](seq: Seq[T]): Seq[String] = seq map {
     |   case i: Int => s"Int: $i"
     |   case other => s"Other: $other"
     | }
def examine[T <: Matchable](seq: Seq[T]): Seq[String]

scala> val seq = Seq(1, "two", 3, 4.4)
     | examine(seq)
val seq: Seq[Matchable] = List(1, two, 3, 4.4)
val res0: Seq[String] = List(Int: 1, Other: two, Int: 3, Other: 4.4)
```

Notice the inferred common supertype of the values in the sequence, `seq`. In Scala 2, it would be `Any`.

Back to `IArray`, the example at the beginning now triggers a warning because the `IArray` alias is not bounded by `Matchable`:

```
scala> val iarray = IArray(1,2,3,4,5)
     | iarray match
     |   case a: Array[Int] => a(2) = 300
     |
3 |  case a: Array[Int] => a(2) = 300
  |          ^^^^^^^^^^
  |          pattern selector should be an instance of Matchable,
  |          but it has unmatchable type opaques.IArray[Int] instead
```

`IArray` is considered an abstract type by the compiler. Abstract types are not bounded by `Matchable`, which is why we now get the warning we want.

This is a significant change that will break a lot of existing code. Hence, warnings will only be issued starting in a future Scala 3 release or when compiling with `-source:future`.

Values, Variables, and Types in Matches

Let's cover several kinds of matches. The following example matches on specific values, all values of specific types, and it shows one way of writing a default clause that matches anything:

```
// src/script/scala/progscala3/patternmatching/MatchVariable.scala

val seq = Seq(1, 2, 3.14, 5.5F, "one", "four", true, (6, 7))     ❶
val result = seq.map {
  case 1                   => "int 1"                             ❷
  case i: Int              => s"other int: $i"
  case d: (Double | Float) => s"a double or float: $d"            ❸
  case "one"               => "string one"                        ❹
  case s: String           => s"other string: $s"
  case (x, y)              => s"tuple: ($x, $y)"                  ❺
  case unexpected          => s"unexpected value: $unexpected"    ❻
}
```

```
assert(result == Seq(
  "int 1", "other int: 2",
  "a double or float: 3.14", "a double or float: 5.5",
  "string one", "other string: four",
  "unexpected value: true",
  "tuple: (6, 7)"))
```

❶ Because of the mix of values, `seq` is of type `Seq[Matchable]`.

❷ If one or more case clauses specify particular values of a type, they need to occur before more general clauses that just match on the type. So we first check if the anonymous value is an `Int` equal to `1`. If so, we simply return the string `"int 1"`. If the value is another `Int` value, the next clause matches. In this case, the value is *cast* to `Int` and assigned to the variable `i`, which is used to construct a string.

❸ Match on any `Double` *or* `Float` value. Using `|` is convenient when two or more cases are handled the same way. However, for this to work, the logic after the `=>` must be type compatible for all matched types. In this case, the interpolated string works fine.

❹ Two case clauses for strings.

❺ Match on a two-element tuple where the elements are of any type, and extract the elements into the variables `x` and `y`.

❻ Match all other inputs. The variable `unexpected` has an arbitrary name. Because no type declaration is given, `Matchable` is inferred. This functions as the default clause. The Boolean value from the sequence `seq` is assigned to `unexpected`.

3 We passed a partial function to `Seq.map()`. Recall that the literal syntax requires `case` statements, and we have put the partial function inside parentheses or braces to pass it to `map`. However, this function is *effectively total,* because the last clause matches any `Matchable`. (It would be `Any` in Scala 2.) This means it wouldn't match instances of the few other types that aren't `Matchables`, like `IArray`, but these types are no longer candidates for pattern matching. From now on, I'll just call partial functions like this *total.*

Don't use clauses with specific floating-point literal values because matching on floating-point literals is a bad idea. Rounding errors mean two values that you might expect to be the same may actually differ.

Matches are eager, so more specific clauses must appear before less specific clauses. Otherwise, the more specific clauses will never get the chance to match. So the clauses matching on particular values of types must come before clauses matching on the type (i.e., on any value of the type). The default clause shown must be the last one.

Fortunately, the compiler will issue an "Unreachable case" warning if you make this mistake. Try switching the two `Int` clauses to see what happens.

`Match` clauses are expressions, so they return a value. In this example, all clauses return strings, so the return type of the `match` expression (and the partial function) is `String`. Hence, the return type of the `map` call is `List[String]`. The compiler infers the least upper bound, the closest supertype, for the types of values returned by all the `case` clauses.

This is a contrived example, of course. When designing pattern-matching expressions, be wary of relying on a default `case` clause. Under what circumstances would "none of the above" be the correct answer? It may indicate that your design could be refined so you know more precisely all the possible matches that might occur, like a sealed type hierarchy or `enum`, which we'll discuss further. In fact, as we go through this chapter, you'll see more realistic scenarios and no default clauses.

Here is a similar example that passes an anonymous function to `map`, rather than a partial function, plus some other changes:

```
// src/script/scala/progscala3/patternmatching/MatchVariable2.scala

val seq2 = Seq(1, 2, 3.14, "one", (6, 7))
val result2 = seq2.map { x => x match
  case _: Int   => s"int: $x"                          ❶
  case _        => s"unexpected value: $x"             ❷
}
assert(result2 == Seq(
  "int: 1", "int: 2", "unexpected value: 3.14",
  "unexpected value: one", "unexpected value: (6,7)"))
```

❶ Use _ for the variable name, meaning we don't capture it.

❷ Catch-all clause that also uses `x` instead of capturing to a new variable.

The first case clause doesn't need to capture the variable because it doesn't exploit the fact that the value is an `Int`. For example, it doesn't call `Int` methods. Otherwise, just using `x` wouldn't be sufficient, as it has type `Matchable`.

Once again, braces are used around the whole anonymous function, but the optional braces syntax is used inside the function for the `match` expression. In general, using a partial function is more concise because we eliminate the need for `x => x match`.

> When you use pattern matching with any of the collection methods, like `map` and `foreach`, use a partial function.

There are a few rules and gotchas to keep in mind for `case` clauses. The compiler assumes that a term that begins with a lowercase letter is the name of a variable that will hold a matched value. If the term starts with a capital letter, it will expect to find a definition already in scope.

This lowercase rule can cause surprises, as shown in the following example. The intention is to pass some value to a method, then see if that value matches an element in the collection:

```
// src/script/scala/progscala3/patternmatching/MatchSurprise.scala

def checkYBad(y: Int): Seq[String] =
  for x <- Seq(99, 100, 101)
  yield x match
    case y => "found y!"
    case i: Int => "int: "+i  // Unreachable case!
```

The first case clause is supposed to match on the value passed in as `y`, but this is what we actually get:

```
def checkBad(y: Int): Seq[String]
10 |      case i: Int => "int: "+i  // Unreachable case!
   |           ^^^^^^
   |           Unreachable case
```

We treat warnings as errors in our `built.sbt` settings, but if we didn't, then calling `checkY(100)` would return `found y!` for all three numbers.

The `case y` clause means "match anything because there is no type declaration, and assign it to this *new* variable named `y`." The `y` in the clause is not interpreted as a reference to the method parameter `y`. Rather, it *shadows* that definition. Hence, this clause is actually a default, match-all clause and we will never reach the second `case` clause.

There are two solutions. First, we could use capital `Y`, although it looks odd to have a method parameter start with a capital letter:

```
def checkYGood1(Y: Int): Seq[String] =
  for x <- Seq(99, 100, 101)
  yield x match
    case Y => "found y!"
    case i: Int => "int: "+i
```

Calling `checkYGood1(100)` returns `List(int: 99, found y!, int: 101)`.

The second solution is to use backticks to indicate we really want to match against the value held by `y`:

```
def checkYGood2(y: Int): Seq[String] =
  for x <- Seq(99, 100, 101)
  yield x match
```

```
  case `y` => "found y!"
  case i: Int => "int: "+i
```

> In `case` clauses, a term that begins with a lowercase letter is assumed to be the name of a new variable that will hold an extracted value. To refer to a previously defined variable, enclose it in backticks or start the name with a capital letter.

Finally, most `match` expressions should be exhaustive:

```
// src/script/scala/progscala3/patternmatching/MatchExhaustive.scala

scala> val seq3 = Seq(Some(1), None, Some(2), None)
val seq3: Seq[Option[Int]] = List(Some(1), None, Some(2), None)

scala> val result3 = seq3.map {
     |   case Some(i)  => s"int $i"
     | }
5 |  case Some(i)  => s"int $i"
  |  ^
  |  match may not be exhaustive.
  |
  |  It would fail on pattern case: None
```

The compiler knows that the elements of `seq3` are of type `Option[Int]`, which could include `None` elements. At runtime, a `MatchError` will be thrown if a `None` is encountered. The fix is straightforward:

```
// src/script/scala/progscala3/patternmatching/MatchExhaustiveFix.scala

scala> val result3 = seq3.map {
     |   case Some(i)  => s"int $i"
     |   case None     => ""
     | }
val result3: Seq[String] = List(int 1, "", int 2, "")
```

"Problems in Pattern Bindings" on page 124 will discuss additional points about exhaustive matching.

Matching on Sequences

Let's examine the classic idiom for iterating through a `Seq` using pattern matching and recursion and, along the way, learn some useful fundamentals about sequences:

```
// src/script/scala/progscala3/patternmatching/MatchSeq.scala

def seqToString[T](seq: Seq[T]): String = seq match      ❶
  case head +: tail => s"($head +: ${seqToString(tail)})"  ❷
  case Nil => "Nil"                                        ❸
```

❶ Define a recursive method that constructs a `String` from a `Seq[T]` for some type `T`, which will be inferred from the sequence passed in. The body is a single `match` expression.

❷ There are two match clauses and they are exhaustive. The first matches on any nonempty `Seq`, extracting the first element as `head` and the rest of the `Seq` as `tail`. These are common names for the parts of a `Seq`, which has `head` and `tail` methods. However, here these terms are used as variable names. The body of the clause constructs a `String` with the head followed by `+:` followed by the result of calling `seqToString` on the tail, all surrounded by parentheses, `()`. Note this method is recursive, but not tail recursive.

❸ The only other possible case is an empty `Seq`. We can use the special case object for an empty `List`, `Nil`, to match all the empty cases. This clause terminates the recursion. Note that any type of `Seq` can always be interpreted as terminating with a `Nil`, or we could use an empty instance of the actual type (examples follow).

The operator `+:` is the cons (construction) operator for sequences. Recall that methods that end with a colon (`:`) bind to the right, toward the `Seq` tail. However, `+:` in this `case` clause is actually an `object` named `+:`, so we have a nice syntax symmetry between *construction* of sequences, like `val seq = 1 +: 2 +: Nil`, and *deconstruction*, like `case 1 +: 2 +: Nil =>`.... We'll see later in this chapter how an `object` is used to implement deconstruction.

These two clauses are mutually exclusive, so they could be written with the `Nil` clause first.

Now let's try it with various empty and nonempty sequences:

```
scala> seqToString(Seq(1, 2, 3))
     | seqToString(Seq.empty[Int])
val res0: String = (1 +: (2 +: (3 +: Nil)))
val res1: String = Nil

scala> seqToString(Vector(1, 2, 3))
     | seqToString(Vector.empty[Int])
val res2: String = (1 +: (2 +: (3 +: Nil)))
val res3: String = Nil

scala> seqToString(Map("one" -> 1, "two" -> 2, "three" -> 3).toSeq)
     | seqToString(Map.empty[String,Int].toSeq)
val res4: String = ((one,1) +: ((two,2) +: ((three,3) +: Nil)))
val res5: String = Nil
```

Note the common idiom for constructing an empty collection, like `Vector.empty[Int]`. The `empty` methods are in the companion objects.

Map is not a subtype of Seq because it doesn't guarantee a particular order when you iterate over it. Calling Map.toSeq creates a sequence of key-value tuples that happen to be in insertion order, which is a side effect of the implementation for small Maps and not true for arbitrary maps. The nonempty Map output shows parentheses from the tuples as well as the parentheses added by seqToString.

Note the output for the nonempty Seq (actually List) and Vector. They show the hierarchical structure implied by a linked list, with a head and a tail:

```
(1 +: (2 +: (3 +: Nil)))
```

So we process sequences with just two case clauses and recursion. This implies something fundamental about all sequences: they are either empty or not. That sounds trite, but once you recognize fundamental structural patterns like this, it gives you a surprisingly general tool for "divide and conquer." The idiom used by processSeq is widely reusable.

To demonstrate the construction versus destruction symmetry, we can copy and paste the output of the previous examples to reconstruct the original objects. However, we have to add quotes around strings:

```
scala> val is = (1 +: (2 +: (3 +: Nil)))
val is: List[Int] = List(1, 2, 3)

scala> val kvs = (("one",1) +: (("two",2) +: (("three",3) +: Nil)))
val kvs: List[(String, Int)] = List((one,1), (two,2), (three,3))

scala> val map = Map(kvs*)
val map: Map[String, Int] = Map(one -> 1, two -> 2, three -> 3)
```

The Map.apply method expects a repeated parameter list of two-element tuples. In order to use the sequence kvs, we use the * idiom so the compiler converts the sequence to a repeated parameter list.

Try removing the parentheses that we added in the preceding string output.

For completeness, there is an analog of +: that can be used to process the sequence elements in reverse, :+:

```
// src/script/scala/progscala3/patternmatching/MatchReverseSeq.scala

scala> def reverseSeqToString[T](l: Seq[T]): String = l match
     |   case prefix :+ end => s"${reverseSeqToString(prefix)} :+ $end)"
     |   case Nil => "Nil"

scala> reverseSeqToString(Vector(1, 2, 3, 4, 5))
val res6: String = (((((Nil :+ 1) :+ 2) :+ 3) :+ 4) :+ 5)
```

Note that Nil comes first this time in the output. A Vector is used for the input sequence to remind you that accessing a nonhead element is *O(1)* for a Vector, but

O(N) for a `List` of size *N*! Hence, `reverseSeqToString` is *O(N)* for a `Vector` of size *N* and $O(N^2)$ for a `List` of size *N*!

As before, you could use this output to reconstruct the collection:

```
scala> val revList1 = (((((Nil :+ 1) :+ 2) :+ 3) :+ 4) :+ 5)
val revList1: List[Int] = List(1, 2, 3, 4, 5)       // but List is returned!

scala> val revList2 = Nil :+ 1 :+ 2 :+ 3 :+ 4 :+ 5  // unnecessary () removed
val revList2: List[Int] = List(1, 2, 3, 4, 5)

scala> val revList3 = Vector.empty[Int] :+ 1 :+ 2 :+ 3 :+ 4 :+ 5
val revList3: Vector[Int] = Vector(1, 2, 3, 4, 5)   // how to get a Vector
```

Pattern Matching on Repeated Parameters

Speaking of repeated parameter lists, you can also use them in pattern matching:

```
// src/script/scala/progscala3/patternmatching/MatchRepeatedParams.scala

scala> def matchThree(seq: Seq[Int]) = seq match
     |   case Seq(h1, h2, rest*) =>    // same as h1 +: h2 +: rest => ...
     |     println(s"head 1 = $h1, head 2 = $h2, the rest = $rest")
     |   case _ => println(s"Other! $seq")

scala> matchThree(Seq(1,2,3,4))
     | matchThree(Seq(1,2,3))
     | matchThree(Seq(1,2))
     | matchThree(Seq(1))
head 1 = 1, head 2 = 2, the rest = List(3, 4)
head 1 = 1, head 2 = 2, the rest = List(3)
head 1 = 1, head 2 = 2, the rest = List()
Other! List(1)
```

3 We see another way to match on sequences. If we don't need `rest`, we can use the placeholder, `_`, that is `case Seq(h1, h2, _*)`. In Scala 2, `rest*` was written `rest @ _*`. The Scala 3 syntax is more consistent with other uses of repeated parameters.

Matching on Tuples

Tuples are also easy to match on, using their literal syntax:

```
// src/script/scala/progscala3/patternmatching/MatchTuple.scala

val langs = Seq(
  ("Scala",   "Martin", "Odersky"),
  ("Clojure", "Rich",   "Hickey"),
  ("Lisp",    "John",   "McCarthy"))

val results = langs.map {
  case ("Scala", _, _) => "Scala"                                    ❶
```

```
    case (lang, first, last) => s"$lang, creator $first $last"    ❷
  }
```

❶ Match a three-element tuple where the first element is the string "Scala" and we ignore the second and third arguments.

❷ Match any three-element tuple, where the elements could be any type, but they are inferred to be `Strings` due to the input `langs`. Extract the elements into variables `lang`, `first`, and `last`.

A tuple can be taken apart into its constituent elements. We can match on literal values within the tuple, at any positions we want, and we can ignore elements we don't care about.

3 In Scala 3, tuples have enhanced features to make them more like linked lists, but where the specific type of each element is preserved. Compare the following example to the preceding implementation of `seqToString`, where `*:` replaces `+:` as the operator:

```
scala> langs.map {
     |   case "Scala" *: first *: last *: EmptyTuple =>
     |     s"Scala -> $first -> $last"
     |   case lang *: rest => s"$lang -> $rest"
     | }
val res0: Seq[String] = List(Scala -> Martin -> Odersky,
 Clojure -> (Rich,Hickey), Lisp -> (John,McCarthy))
```

The analog of `Nil` for tuples is `EmptyTuple`. The second case clause can handle *any* tuple with one or more elements. Let's create a new list by prepending `EmptyTuple` itself and a one-element tuple:

```
scala> val l2 = EmptyTuple +: ("Indo-European" *: EmptyTuple) +: langs
val l2: Seq[Tuple] = List((), (Indo-European,), (Scala,Martin,Odersky),
 (Clojure,Rich,Hickey), (Lisp,John,McCarthy))

scala> l2.map {
     |   case "Scala" *: first *: last *: EmptyTuple =>
     |     s"Scala -> $first -> $last"
     |   case lang *: rest => s"$lang -> $rest"
     |   case EmptyTuple => EmptyTuple.toString
     | }
val res1: Seq[String] = List((), Indo-European -> (),
 Scala -> Martin -> Odersky, Clojure -> (Rich,Hickey), Lisp -> (John,McCarthy))
```

You might think that `("Indo-European")` would be enough to construct a one-element tuple, but the compiler just interprets the parentheses as unnecessary wrappers around the string! `("Indo-European" *: EmptyTuple)` does the trick.

Just as we can construct pairs (two-element tuples) with `->`, we can deconstruct them that way too:

```
// src/script/scala/progscala3/patternmatching/MatchPair.scala

val langs2 = Seq("Scala" -> "Odersky", "Clojure" -> "Hickey")

val results = langs2.map {
  case "Scala" -> _ => "Scala"                              ❶
  case lang -> last => s"$lang: $last"                      ❷
}
assert(results == Seq("Scala", "Clojure: Hickey"))
```

❶ Match on a tuple with the string "Scala" as the first element and anything as the second element.

❷ Match on any other, two-element tuple.

Recall that I said `+:` in patterns is actually an `object` in the `scala.collection` package. Similarly, there is an `*:` `object` and a type alias for `->` to `Tuple2.type` (effectively the companion `object` for the `Tuple2` case class) in the `scala` package.

Parameter Untupling

Consider this example using tuples:

```
// src/script/scala/progscala3/patternmatching/ParameterUntupling.scala

val tuples = Seq((1,2,3), (4,5,6), (7,8,9))
val counts1 = tuples.map {    // result: List(6, 15, 24)
  case (x, y, z) => x + y + z
}
```

A disadvantage of the case syntax inside the anonymous function is the implication that it's not exhaustive, when we know it is for the `tuples` sequence. It is also a bit inconvenient to add `case`. Scala 3 introduces *parameter untupling* that simplifies special cases like this. We can drop the `case` keyword:

```
val counts2 = tuples.map {
  (x, y, z) => x + y + z
}
```

We can even use anonymous variables:

```
val counts3 = tuples.map(_+_+_)
```

However, this untupling only works for one level of decomposition:

```
scala> val tuples2 = Seq((1,(2,3)), (4,(5,6)), (7,(8,9)))
     | val counts2b = tuples2.map {
     |   (x, (y, z)) => x + y + z
     | }
     |
3 |  (x, (y, z)) => x + y + z
```

```
    |       ^^^^^^
    |       not a legal formal parameter
```

Use `case` for such, uh, cases.

Guards in Case Clauses

Matching on literal values is very useful, but sometimes you need a little additional logic:

```
// src/script/scala/progscala3/patternmatching/MatchGuard.scala

val results = Seq(1,2,3,4).map {
  case e if e%2 == 0 => s"even: $e"                          ❶
  case o             => s"odd:  $o"                          ❷
}
assert(results == Seq("odd:  1", "even: 2", "odd:  3", "even: 4"))
```

❶ Match only if `e` is even.

❷ Match the only other possibility, that `o` is odd.

Note that we didn't need parentheses around the condition in the `if` expression, just as we don't need them in `for` comprehensions. In Scala 2, this was true for guard clause syntax too.

Matching on Case Classes and Enums

It's no coincidence that the same `case` keyword is used for declaring special classes and for `case` expressions in `match` expressions. The features of case classes were designed to enable convenient pattern matching. The compiler implements pattern matching and extraction for us. We can use it with nested objects, and we can bind variables at any level of the extraction, which we are seeing for the first time now:

```
// src/script/scala/progscala3/patternmatching/MatchDeep.scala

case class Address(street: String, city: String)
case class Person(name: String, age: Int, address: Address)

val alice   = Person("Alice",   25, Address("1 Scala Lane", "Chicago"))
val bob     = Person("Bob",     29, Address("2 Java Ave.",  "Miami"))
val charlie = Person("Charlie", 32, Address("3 Python Ct.", "Boston"))

val results = Seq(alice, bob, charlie).map {
  case p @ Person("Alice", age, a @ Address(_, "Chicago")) =>      ❶
    s"Hi Alice! $p"
  case p @ Person("Bob", 29, a @ Address(street, city)) =>         ❷
    s"Hi ${p.name}! age ${p.age}, in ${a}"
  case p @ Person(name, age, Address(street, city)) =>             ❸
```

```
    s"Who are you, $name (age: $age, city = $city)?"
}
assert(results == Seq(
  "Hi Alice! Person(Alice,25,Address(1 Scala Lane,Chicago))",
  "Hi Bob! age 29, in Address(2 Java Ave.,Miami)",
  "Who are you, Charlie (age: 32, city = Boston)?"))
```

❶ Match on any person named “Alice”, of any age at any street address in Chicago. Use `p @` to bind variable `p` to the whole `Person`, while also extracting fields inside the instance, in this case `age`. Similarly, use `a @` to bind `a` to the whole `Address` while also binding `street` and `city` inside the `Address`.

❷ Match on any person named “Bob”, age 29 at any street and city. Bind `p` the whole `Person` instance and `a` to the nested `Address` instance.

❸ Match on any person, binding `p` to the `Person` instance and `name`, `age`, `street`, and `city` to the nested fields.

If you aren’t extracting fields from the `Person` instance, we can just write `p: Person => …`

3 This nested matching can go arbitrarily deep. Consider this example that revisits the `enum Tree[T]` algebraic data type from “Enumerations and Algebraic Data Types” on page 79. Recall the `enum` definition, which also supports “automatic” pattern matching:

```
// src/main/scala/progscala3/patternmatching/MatchTreeADTEnum.scala
package progscala3.patternmatching

enum Tree[T]:
  case Branch(left: Tree[T], right: Tree[T])
  case Leaf(elem: T)
```

Here we do deep matching on particular structures:

```
// src/script/scala/progscala3/patternmatching/MatchTreeADTDeep.scala
import progscala3.patternmatching.Tree
import Tree.{Branch, Leaf}

val tree1 = Branch(
  Branch(Leaf(1), Leaf(2)),
  Branch(Leaf(3), Branch(Leaf(4), Leaf(5))))
val tree2 = Branch(Leaf(6), Leaf(7))

for t <- Seq(tree1, tree2, Leaf(8))
yield t match
  case Branch(
    l @ Branch(_,_),
    r @ Branch(rl @ Leaf(rli), rr @ Branch(_,_))) =>
      s"l=$l, r=$r, rl=$rl, rli=$rli, rr=$rr"
```

```
    case Branch(l, r) => s"Other Branch($l, $r)"
    case Leaf(x) => s"Other Leaf($x)"
```

The same extraction could be done for the alternative version we defined using a sealed class hierarchy in the original example. We'll try it in "Sealed Hierarchies and Exhaustive Matches" on page 121.

The last two case clauses are relatively easy to understand. The first one is highly tuned to match `tree1`, although it uses `_` to ignore some parts of the tree. In particular, note that it isn't sufficient to write `l @ Branch`. We need to write `l @ Branch(_,_)`. Try removing the `(_,_)` here and you'll notice the first case no longer matches `tree1`, without any obvious explanation.

> If a nested pattern `match` expression doesn't match when you think it should, make sure that you capture the full structure, like `l @ Branch(_,_)` instead of `l @ Branch`.

It's worth experimenting with this example to capture different parts of the trees, so you develop an intuition about what works, what doesn't, and how to debug `match` expressions.

Here's an example using tuples. Imagine we have a sequence of `(String,Double)` tuples for the names and prices of items in a store, and we want to print them with their index. The `Seq.zipWithIndex` method is handy here:

```
// src/script/scala/progscala3/patternmatching/MatchDeepTuple.scala

val itemsCosts = Seq(("Pencil", 0.52), ("Paper", 1.35), ("Notebook", 2.43))

val results = itemsCosts.zipWithIndex.map {
  case ((item, cost), index) => s"$index: $item costs $cost each"
}
assert(results == Seq(
  "0: Pencil costs 0.52 each",
  "1: Paper costs 1.35 each",
  "2: Notebook costs 2.43 each"))
```

Note that `zipWithIndex` returns a sequence of tuples of the form `(element, index)`, or `((name, cost), index)` in this case. We matched on this form to extract the three elements and construct a string with them. I write code like this *a lot*.

Matching on Regular Expressions

Regular expressions (or *regexes*) are convenient for extracting data from strings that have a particular structure. Here is an example:

```scala
// src/script/scala/progscala3/patternmatching/MatchRegex.scala

val BookExtractorRE = """Book: title=([^,]+),\s+author=(.+)""".r      ❶
val MagazineExtractorRE = """Magazine: title=([^,]+),\s+issue=(.+)""".r

val catalog = Seq(
  "Book: title=Programming Scala Third Edition, author=Dean Wampler",
  "Magazine: title=The New Yorker, issue=January 2021",
  "Unknown: text=Who put this here??"
)

val results = catalog.map {
  case BookExtractorRE(title, author) =>                                ❷
    s"""Book "$title", written by $author"""
  case MagazineExtractorRE(title, issue) =>
    s"""Magazine "$title", issue $issue"""
  case entry => s"Unrecognized entry: $entry"
}
assert(results == Seq(
  """Book "Programming Scala Third Edition", written by Dean Wampler""",
  """Magazine "The New Yorker", issue January 2021""",
  "Unrecognized entry: Unknown: text=Who put this here??"))
```

❶ Match a book string, with two *capture groups* (note the parentheses), one for the title and one for the author. Calling the `r` method on a string creates a regex from it. Also match a magazine string, with capture groups for the title and issue (date).

❷ Use the regular expressions much like using case classes, where the string matched by each capture group is assigned to a variable.

Because regexes use backslashes for constructs beyond the normal ASCII control characters, you should either use triple-quoted strings for them, as shown, or use raw interpolated strings, such as `raw"foo\sbar".r`. Otherwise, you must escape these backslashes; for example `"foo\\sbar".r`. You can also define regular expressions by creating new instances of the `Regex` class, as in `new Regex("""\W+""")`.

> Using interpolation in triple-quoted strings doesn't work cleanly for the regex escape sequences. You still need to escape these sequences (e.g., `s"""$first\\s+$second""".r` instead of `s"""$first\s+$second""".r`). If you aren't using interpolation, escaping isn't necessary.

`scala.util.matching.Regex` defines several methods for other manipulations, such as finding and replacing matches.

Matching on Interpolated Strings

If you know the strings have an exact format, such as a precise number of spaces, you can even use interpolated strings for pattern matching. Let's reuse the `catalog`:

```
// src/script/scala/progscala3/patternmatching/MatchInterpolatedString.scala

val results = catalog.map {
  case s"""Book: title=$t, author=$a""" => ("Book" -> (t -> a))
  case s"""Magazine: title=$t, issue=$d""" => ("Magazine" -> (t -> d))
  case item => ("Unrecognized", item)
}
assert(results == Seq(
  ("Book", ("Programming Scala Third Edition", "Dean Wampler")),
  ("Magazine", ("The New Yorker", "January 2020")),
  ("Unrecognized", "Unknown: text=Who put this here??")))
```

Sealed Hierarchies and Exhaustive Matches

Let's revisit the need for exhaustive matches and consider the situation where we have an `enum` or the equivalent `sealed` class hierarchy.

First, let's use the `enum Tree[T]` definition from earlier. We can pattern match on the leafs and branches knowing we'll never be surprised to see something else:

```
// src/script/scala/progscala3/patternmatching/MatchTreeADTExhaustive.scala
import progscala3.patternmatching.Tree
import Tree.{Branch, Leaf}

val enumSeq: Seq[Tree[Int]] = Seq(Leaf(0), Branch(Leaf(6), Leaf(7)))
val tree1 = for t <- enumSeq yield t match
  case Branch(left, right) => (left, right)
  case Leaf(value) => value
assert(tree1 == List(0, (Leaf(6),Leaf(7))))
```

Because it's not possible for a user of `Tree` to add another `case` to the `enum`, these `match` expressions can never break. They will always remain exhaustive.

As an exercise, change the `case Branch` to recurse on `left` and `right` (you'll need to define a method), then use a deeper tree example.

Let's try a corresponding sealed hierarchy:

```
// src/main/scala/progscala3/patternmatching/MatchTreeADTSealed.scala
package progscala3.patternmatching

sealed trait STree[T]               // "S" for "sealed"
case class SBranch[T](left: STree[T], right: STree[T]) extends STree[T]
case class SLeaf[T](elem: T) extends STree[T]
```

The match code is essentially identical:

```
import progscala3.patternmatching.{STree, SBranch, SLeaf}

val sealedSeq: Seq[STree[Int]] = Seq(SLeaf(0), SBranch(SLeaf(6), SLeaf(7)))
val tree2 = for t <- sealedSeq yield t match
  case SBranch(left, right) => (left, right)
  case SLeaf(value) => value
assert(tree2 == List(0, (SLeaf(6),SLeaf(7))))
```

A corollary is to avoid using `sealed` hierarchies and `enums` when the type hierarchy needs to evolve. Instead, use an "open" object-oriented type hierarchy with polymorphic methods instead of `match` expressions. We discussed this trade-off in "A Sample Application" on page 20.

Chaining Match Expressions

Scala 3 changed the parsing rules for `match` expressions to allow chaining, as in this contrived example:

```
// src/script/scala/progscala3/patternmatching/MatchChaining.scala

scala> for opt <- Seq(Some(1), None)
     | yield opt match {
     |   case None => ""
     |   case Some(i) => i.toString
     | } match {  // matches on the String returned from the previous match
     |   case "" => false
     |   case _ => true
     | }
val res10: Seq[Boolean] = List(true, false)
```

Pattern Matching Outside Match Expressions

Pattern matching is not restricted to `match` expressions. You can use it in assignment statements, called *pattern bindings*:

```
// src/script/scala/progscala3/patternmatching/Assignments.scala

scala> case class Address(street: String, city: String, country: String)
scala> case class Person(name: String, age: Int, address: Address)

scala> val addr = Address("1 Scala Way", "CA", "USA")
scala> val dean = Person("Dean", 29, addr)
val addr: Address = Address(1 Scala Way,CA,USA)
val dean: Person = Person(Dean,29,Address(1 Scala Way,CA,USA))

scala> val Person(name, age, Address(_, state, _)) = dean
val name: String = Dean
val age: Int = 29
val state: String = CA
```

They work in `for` comprehensions:

```
scala> val people = (0 to 4).map {
     |   i => Person(s"Name$i", 10+i, Address(s"$i Main Street", "CA", "USA"))
     | }
val people: IndexedSeq[Person] = Vector(Person(Name0,10,Address(...)), ...)

scala> val nas = for
     |   Person(name, age, Address(_, state, _)) <- people
     | yield (name, age, state)
val nas: IndexedSeq[(String, Int, String)] =
  Vector((Name0,10,CA), (Name1,11,CA), ...)
```

Suppose we have a function that takes a sequence of doubles and returns the count, sum, average, minimum value, and maximum value in a tuple:

```
// src/script/scala/progscala3/patternmatching/AssignmentsTuples.scala

/** Return the count, sum, average, minimum value, and maximum value. */
def stats(seq: Seq[Double]): (Int, Double, Double, Double, Double) =
  assert(seq.size > 0)
  val sum = seq.sum
  (seq.size, sum, sum/seq.size, seq.min, seq.max)

val (count, sum, avg, min, max) = stats((0 until 100).map(_.toDouble))
```

Pattern bindings can be used with interpolated strings:

```
// src/script/scala/progscala3/patternmatching/AssignmentsInterpStrs.scala

val str = """Book: "Programming Scala", by Dean Wampler"""
val s"""Book: "$title", by $author""" = str : @unchecked
assert(title == "Programming Scala" && author == "Dean Wampler")
```

I'll explain the need for `@unchecked` in a moment.

Finally, we can use pattern bindings with a regular expression to decompose a string. Here's an example for parsing (simple!) SQL strings:

```
// src/script/scala/progscala3/patternmatching/AssignmentsRegex.scala

scala> val c = """\*|[\w, ]+"""  // cols
     | val t = """\w+"""         // table
     | val o = """.*"""          // other substrings
     | val selectRE =
     |   s"""SELECT\\s*(DISTINCT)?\\s+($c)\\s*FROM\\s+($t)\\s*($o)?;""".r

scala> val selectRE(distinct, cols, table, otherClauses) =
     |   "SELECT DISTINCT col1 FROM atable WHERE col1 = 'foo';": @unchecked
val distinct: String = DISTINCT
val cols: String = "col1 "
val table: String = atable
val otherClauses: String = WHERE col1 = 'foo'
```

See the source file for other examples. Because I used string interpolation, I had to add extra backslashes (e.g., `\\s` instead of `\s`) in the last regular expression.

Next I'll explain why the `@unchecked` type annotation was used.

Problems in Pattern Bindings

In general, keep in mind that pattern matching will throw `MatchError` exceptions when the match fails. This can make your code fragile when used in assignments because it's harder to make them exhaustive. In the previous interpolated string and regex examples, the `String` type for the righthand side values can't ensure that the matches will succeed.

3 Assume I didn't have the `: @unchecked` type declaration. In Scala 2 and 3.0, both examples would compile and work without `MatchErrors`. Starting in a future Scala 3 release or when compiling with `-source:future`, the examples fail to compile, for example:

```
scala> val selectRE(distinct, cols, table, otherClauses) =
     |   "SELECT DISTINCT col1 FROM atable WHERE col1 = 'foo';"
     |
2 |  "SELECT DISTINCT col1 FROM atable WHERE col1 = 'foo';"
  |  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |pattern's type String is more specialized than the righthand side
  |expression's type String
  |
  |If the narrowing is intentional, this can be communicated by adding
  |`: @unchecked` after the expression.
```

This compile-time enforcement makes your code more robust, but if you *know* the declaration is safe, you can add the `@unchecked` type declaration, as we did earlier, and the compiler will not complain.

However, if we silence these warnings, we may get runtime `MatchErrors`. Consider the following examples with sequences:

```
// src/script/scala/progscala3/patternmatching/AssignmentsFragile.scala

scala> val h4a +: h4b +: t4 = Seq(1,2,3,4) : @unchecked
val h4a: Int = 1
val h4b: Int = 2
val t4: Seq[Int] = List(3, 4)

scala> val h2a +: h2b +: t2 = Seq(1,2) : @unchecked
val h2a: Int = 1
val h2b: Int = 2
val t2: Seq[Int] = List()

scala> val h1a +: h1b +: t1 = Seq(1) : @unchecked     // MatchError!
```

```
scala.MatchError: List(1) (of class scala.collection.immutable.$colon$colon)
  ...
```

`Seq` doesn't constrain the number of elements, so the lefthand matches may work or fail. The compiler can't verify at compile time if the match will succeed or throw a `MatchError`, so it will report a warning unless the `@unchecked` type annotation is added as shown. Sure enough, while the first two cases succeed, the last one raises a `MatchError`.

Pattern Matching as Filtering in for Comprehensions

However, in a `for` comprehension, matching that isn't exhaustive functions as a filter instead:

```
// src/script/scala/progscala3/patternmatching/MatchForFiltering.scala

scala> val elems = Seq((1, 2), "hello", (3, 4), 1, 2.2, (5, 6))
val elems: Seq[Matchable] = List((1,2), hello, (3,4), 1, 2.2, (5,6))

scala> val what1 = for (case (x, y) <- elems) yield (y, x)         ❶
     | val what2 = for  case (x, y) <- elems  yield (y, x)
val what1: Seq[(Any, Any)] = List((2,1), (4,3), (6,5))
val what2: Seq[(Any, Any)] = List((2,1), (4,3), (6,5))
```

❶ The `case` keyword is required for matching and filtering. The parentheses are optional.

Note that the inferred common supertype for the elements in `elems` is `Matchable`, not `Any`. For `what1` and `what2`, the inferred type is a tuple—a subtype of `Matchable`. The tuple members can be `Any`.

3 The `case` keyword was not required for Scala 2 or 3.0. Starting with a future Scala 3 release or compiling with `-source:future` will trigger the "narrowing" warning if you omit the case keyword:

```
scala> val nope = for (x, y) <- elems yield (y, x)
1 |val nope = for (x, y) <- elems yield (y, x)
  |               ^^^^^^
  |pattern's type (Any, Any) is more specialized than the right hand side
  |expression's type Matchable
  |
  |If the narrowing is intentional, this can be communicated by writing `case`
  |before the full pattern.
[source,scala]
```

When we discussed exhaustive matching previously, we used an example of a sequence of `Option` values. We can filter out values in a sequence using pattern matching:

```
scala> val seq = Seq(None, Some(1), None, Some(2.2), None, None, Some("three"))
scala> val filtered = for case Some(x) <- seq yield x
val filtered: Seq[Matchable] = List(1, 2.2, three)
```

Pattern Matching and Erasure

Consider the following example, where we attempt to discriminate between the inputs `List[Double]` and `List[String]`:

```
// src/script/scala/progscala3/patternmatching/MatchTypesErasure.scala

scala> val results = Seq(Seq(5.5,5.6,5.7), Seq("a", "b")).map {
     |   case seqd: Seq[Double] => ("seq double", seqd)   // Erasure warning
     |   case seqs: Seq[String] => ("seq string", seqs)   // Erasure warning
     |   case other             => ("unknown!", other)
     | }
2 |  case seqd: Seq[Double] => ("seq double", seqd)   // Erasure warning
  |       ^^^^^^^^^^^^^^^^^
  |       the type test for Seq[Double] cannot be checked at runtime
3 |  case seqs: Seq[String] => ("seq string", seqs)   // Erasure warning
  |       ^^^^^^^^^^^^^^^^^
  |       the type test for Seq[String] cannot be checked at runtime
```

These warnings result from type erasure, where the information about the actual types used for the type parameters is not retained in the compiler output. Hence, while we can tell at runtime that the object is a `Seq`, we can't check that it is a `Seq[Double]` or a `Seq[String]`. In fact, if we neglect the warning, the second `case` clause for `Seq[String]` is unreachable. The first clause matches for all `Seq`s.

One ugly workaround is to match on the collection first, then use a nested match on the head element to determine the type. We now have to handle an empty sequence too:

```
// src/script/scala/progscala3/patternmatching/MatchTypesFix.scala

def doSeqMatch[T <: Matchable](seq: Seq[T]): String = seq match
  case Nil => ""
  case head +: _ => head match
    case _ : Double => "Double"
    case _ : String => "String"
    case _ => "Unmatched seq element"

val results = Seq(Seq(5.5,5.6), Nil, Seq("a","b")).map(seq => doSeqMatch(seq))
assert(results == Seq("Double", "", "String"))
```

Extractors

So how does pattern matching and destructuring or extraction work? Scala defines a pair of `object` methods that are implemented automatically for case classes and for many types in the Scala library. You can implement these extractors yourself to customize the behavior for your types. When those methods are available on suitable types, they can be used in pattern-matching clauses.

However, you will rarely need to implement your own extractors. You also don't need to understand the implementation details to use pattern matching effectively. Therefore, you can safely skip the rest of this chapter now and return to this discussion later, when needed.

unapply Method

Recall that the companion object for a case class has at least one factory method named `apply`, which is used for construction. Using symmetry arguments, we might infer that there must be another method generated called `unapply`, which is used for deconstruction or extraction. Indeed, there is an `unapply` method, and it is invoked in pattern-match expressions for most types.

There are several ways to implement `unapply`, specifically what is returned from it. We'll start with the return type used most often: an `Option` wrapping a tuple. Then we'll discuss other options for return types.

Consider again `Person` and `Address` from before:

```
person match
  case Person(name, age, Address(street, city)) => ...
  ...
```

Scala looks for `Person.unapply(...)` and `Address.unapply(...)` and calls them. They return an `Option[(...)]`, where the tuple type corresponds to the number of values and their types that can be extracted from the instance.

By default for case classes, the compiler implements `unapply` to return all the fields declared in the constructor argument list. That will be three fields for `Person`, of types `String`, `Int`, and `Address`, and two fields for `Address`, both of type `String`. So the `Person` companion object has methods that would look like this:

```
object Person:
  def apply(name: String, age: Int, address: Address) =
    new Person(name, age, address)
  def unapply(p: Person): Some[(String,Int,Address)] =
    Some((p.name, p.age, p.address))
```

Why is an `Option` used if the compiler already knows that the object is a `Person`? Scala allows an implementation of `unapply` to veto the match for some reason and return `None`, in which case Scala will attempt to use the next `case` clause. Also, we don't have to expose all fields of the instance if we don't want to. We could suppress our `age`, if we're embarrassed by it. We could even add additional values to the returned tuples.

When a Some wrapping a tuple is returned by an unapply, the compiler extracts the tuple elements for use in the case clause or assignment, such as comparison with literal values, binding to variables, or dropping them for _ placeholders.

However, note that the simple compiler-generated Person.unapply never fails, so Some[…] is used as the return type, rather than Option[…].

The unapply methods are invoked recursively when necessary, so the nested Address instance is processed first, then Person.

Recall the head +: tail expression we used previously. Now let's understand how it actually works. We've seen that the +: (cons) operator can be used to construct a new sequence by prepending an element to an existing sequence, and we can construct an entire sequence from scratch this way:

```
val list = 1 +: 2 +: 3 +: 4 +: Nil
```

Because +: is a method that binds to the right, we first prepend 4 to Nil, then prepend 3 to that list, and so forth.

If the construction of sequences is done with a method named +:, how can extraction be done with the same syntax, so that we have uniform syntax for *construction* and *deconstruction/extraction*?

To do that, the Scala library defines a special singleton object named +:. Yes, that's the name. Like methods, types can have names with a wide variety of characters.

It has just one method, the unapply method the compiler needs for our extraction case statement. The declaration of unapply is conceptually as follows (some details removed):

```
def unapply[H, Coll](collection: Coll): Option[(H, Coll)]
```

The head is of type H, which is inferred, and some collection type Coll, which represents the type of the tail collection. So an Option of a two-element tuple with the head and tail is returned.

We learned in "Defining Operators" on page 71 that types can be used with infix notation, so head +: tail is valid syntax, equivalent to +:(head, tail). In fact, we can use the normal notation in a case clause:

```
scala> def seqToString2[T](seq: Seq[T]): String = seq match
     |   case +:(head, tail) => s"($head +: ${seqToString2(tail)})"
     |   case Nil => "Nil"

scala> seqToString2(Seq(1,2,3,4))
val res0: String = (1 +: (2 +: (3 +: (4 +: Nil))))
```

Here's another example, just to drive home the point:

```scala
// src/script/scala/progscala3/patternmatching/Infix.scala

infix case class And[A,B](a: A, b: B)

val and1: And[String,Int] = And("Foo", 1)
val and2: String And Int  = And("Bar", 2)
// val and3: String And Int  = "Baz" And 3  // ERROR

val results = Seq(and1, and2).map {
  case s And i => s"$s and $i"
}
assert(results == Seq("Foo and 1", "Bar and 2"))
```

We mentioned earlier that you can pattern match pairs with `->`. This feature is implemented with a `val` defined in `Predef`, `->`. This is an alias for `Tuple2.type`, which subtypes `Product2`, which defines an `unapply` method that is used for these pattern-matching expressions.

Alternatives to Option Return Values

While it is common to return an `Option` from `unapply`, any type with the following signature is allowed, which `Option` also implements:

```scala
def isEmpty: Boolean
def get: T
```

A `Boolean` can also be returned or a `Product` type, which is a supertype of tuples, for example. Here's an example using `Boolean` where we want to discriminate between two kinds of strings and the match is really implementing a true versus false analysis:

```scala
// src/script/scala/progscala3/patternmatching/UnapplyBoolean.scala

object ScalaSearch:                                                      // 1
  def unapply(s: String): Boolean = s.toLowerCase.contains("scala")

val books = Seq(
  "Programming Scala",
  "JavaScript: The Good Parts",
  "Scala Cookbook").zipWithIndex   // add an "index"

val result = for s <- books yield s match                                // 2
  case (ScalaSearch(), index) => s"$index: found Scala"                  // 3
  case (_, index) => s"$index: no Scala"

assert(result == Seq("0: found Scala", "1: no Scala", "2: found Scala"))
```

1. Define an object with an `unapply` method that takes a string, converts to lowercase, and returns the result of a predicate; does it contain "scala"?

❷ Try it on a list of strings, where the first case match succeeds only when the string contains "scala."

❸ Empty parentheses required.

Other single values can be returned. Here is an example that converts a Scala `Map` to a Java `HashMap`:

```
// src/script/scala/progscala3/patternmatching/UnapplySingleValue.scala

import java.util.{HashMap as JHashMap}

case class JHashMapWrapper[K,V](jmap: JHashMap[K,V])
object JHashMapWrapper:
  def unapply[K,V](map: Map[K,V]): JHashMapWrapper[K,V] =
    val jmap = new JHashMap[K,V]()
    for (k,v) <- map do jmap.put(k, v)
    new JHashMapWrapper(jmap)
```

In action:

```
scala> val map = Map("one" -> 1, "two" -> 2)
val map: Map[String, Int] = Map(one -> 1, two -> 2)

scala> map match
     |   case JHashMapWrapper(jmap) => jmap
val res0: java.util.HashMap[String, Int] = {one=1, two=2}
```

However, it's not possible to implement a similar extractor for Java's `HashSet` and combine them into one `match` expression (because there are two possible return values, not one):

```
// src/script/scala/progscala3/patternmatching/UnapplySingleValue2.scala
scala> ...
scala> val map = Map("one" -> 1, "two" -> 2)
scala> val set = map.keySet
scala> for x <- Seq(map, set) yield x match
     |   case JHashMapWrapper(jmap) => jmap
     |   case JHashSetWrapper(jset) => jset
... errors ...
```

See the source file for the full details. The Scala collections already have tools for converting between Scala and Java collections. See "Conversions Between Scala and Java Collections" on page 463 for details.

Another option for `unapply` is to return a `Product`, or more specifically an object that mixes in this trait, which is an abstraction for types when it is useful to treat the member fields uniformly, such as retrieving them by index or iterating over them. Tuples implement `Product`. We can use it as a way to provide several return values extracted by `unapply`:

```scala
// src/script/scala/progscala3/patternmatching/UnapplyProduct.scala

class Words(words: Seq[String], index: Int) extends Product:    // 1
  def _1 = words                                                  // 2
  def _2 = index

  def canEqual(that: Any): Boolean = ???                          // 3
  def productArity: Int = ???
  def productElement(n: Int): Any = ???

object Words:
  def unapply(si: (String, Int)): Words =                         // 4
    val words = si._1.split("""\W+""").toSeq                      // 5
    new Words(words, si._2)

val books = Seq(
  "Programming Scala",
  "JavaScript: The Good Parts",
  "Scala Cookbook").zipWithIndex   // add an "index"

val result = books.map {
  case Words(words, index) => s"$index: count = ${words.size}"
}
assert(result == Seq("0: count = 2", "1: count = 4", "2: count = 2"))
```

1. Now we need a class `Words` to hold the results when a match succeeds. `Words` implements `Product`.

2. Define two methods for retrieving the first and second items. Note the method names are the same as for two-element tuples.

3. The `Product` trait declares these methods too, so we have to provide definitions, but we don't need working implementations. This is because `Product` is actually a marker trait for our purposes. All we really need is for `Words` to mixin this type. So we simply invoke the `???` method defined in `Predef`, which always throws `NotImplementedError`.

4. Matches on a tuple of `String` and `Int`.

5. Split the string on runs of whitespace.

unapplySeq Method

When you want to return a sequence of extracted items, rather than a fixed number of them, use `unapplySeq`. It turns out the `Seq` companion object implements `apply` and `unapplySeq`, but not `unapply`:

```scala
def apply[A](elems: A*): Seq[A]
final def unapplySeq[A](x: Seq[A]): UnapplySeqWrapper[A]
```

`UnapplySeqWrapper` is a helper class.

Matching with `unapplySeq` is invoked in this variation of our previous example for `+:`, where we examine a sliding window of pairs of elements at a time:

```scala
// src/script/scala/progscala3/patternmatching/MatchUnapplySeq.scala

// Process pairs
def windows[T](seq: Seq[T]): String = seq match
  case Seq(head1, head2, tail*) =>                                    // 1
    s"($head1, $head2), " + windows(seq.tail)                         // 2
  case Seq(head, tail*) =>                                            // 3
    s"($head, _), " + windows(tail)
  case Nil => "Nil"                                                   // 4

val nonEmptyList   = List(1, 2, 3, 4, 5)
val emptyList      = Nil
val nonEmptyMap    = Map("one" -> 1, "two" -> 2, "three" -> 3)

val results = Seq(nonEmptyList, emptyList, nonEmptyMap.toSeq).map {
  seq => windows(seq)
}
assert(results == Seq(
  "(1, 2), (2, 3), (3, 4), (4, 5), (5, _), Nil",
  "Nil",
  "((one,1), (two,2)), ((two,2), (three,3)), ((three,3), _), Nil"))
```

1. It looks like we're calling `Seq.apply(…)`, but in a match clause, we're actually calling `Seq.unapplySeq`. We grab the first two elements separately, and the rest of the repeated parameters list as the tail.

2. Format a string with the first two elements, then move the window by one (not two) by calling `seq.tail`, which is also equivalent to `head2 +: tail`.

3. We also need a match for a one-element sequence, such as near the end, or we won't have exhaustive matching. This time we use the tail in the recursive call, although we actually know that this call to `windows(tail)` will simply return `Nil`.

4. The `Nil` case terminates the recursion.

We could rewrite the second case statement to skip the final invocation of `windows(tail)`, but I left it as is for simplicity.

We could still use the `+:` matching we saw before, which is more elegant and what I would do:

```scala
// src/script/scala/progscala3/patternmatching/MatchWithoutUnapplySeq.scala

val nonEmptyList   = List(1, 2, 3, 4, 5)
val emptyList      = Nil
val nonEmptyMap    = Map("one" -> 1, "two" -> 2, "three" -> 3)

// Process pairs
def windows2[T](seq: Seq[T]): String = seq match
  case head1 +: head2 +: _ => s"($head1, $head2), " + windows2(seq.tail)
  case head +: tail => s"($head, _), " + windows2(tail)
  case Nil => "Nil"

val results = Seq(nonEmptyList, emptyList, nonEmptyMap.toSeq).map {
  seq => windows2(seq)
}
assert(results == Seq(
  "(1, 2), (2, 3), (3, 4), (4, 5), (5, _), Nil",
  "Nil",
  "((one,1), (two,2)), ((two,2), (three,3)), ((three,3), _), Nil"))
```

Working with sliding windows is actually so useful that `Seq` gives us two methods to create them:

```scala
scala> val seq = 0 to 5
val seq: scala.collection.immutable.Range.Inclusive = Range 0 to 5

scala> seq.sliding(2).foreach(println)
ArraySeq(0, 1)
ArraySeq(1, 2)
ArraySeq(2, 3)
ArraySeq(3, 4)

scala> seq.sliding(3,2).foreach(println)
ArraySeq(0, 1, 2)
ArraySeq(2, 3, 4)
```

Both `sliding` methods return an iterator, meaning they are lazy and don't immediately make a copy of the collection, which is desirable for large collections. The second method takes a `stride` argument, which is how many steps to go for the next sliding window. The default is one step. Note that none of the sliding windows contain our last element, 5.

Implementing unapplySeq

Let's implement an `unapplySeq` method adapted from the preceding `Words` example. We'll tokenize the words as before but also remove all words shorter than a specified value:

```scala
// src/script/scala/progscala3/patternmatching/UnapplySeq.scala

object Tokenize:
```

```scala
  // def unapplySeq(s: String): Option[Seq[String]] = Some(tokenize(s))❶
  def unapplySeq(lim_s: (Int,String)): Option[Seq[String]] =          ❷
    val (limit, s) = lim_s
    if limit > s.length then None
    else
      val seq = tokenize(s).filter(_.length >= limit)
      Some(seq)

  def tokenize(s: String): Seq[String] = s.split("""\W+""").toSeq      ❸

val message = "This is Programming Scala v3"
val limits = Seq(1, 3, 20, 100)

val results = for limit <- limits yield (limit, message) match
  case Tokenize() => s"No words of length >= $limit!"
  case Tokenize(a, b, c, d*) => s"limit: $limit => $a, $b, $c, d=$d"   ❹
  case x => s"limit: $limit => Tokenize refused! x=$x"

assert(results == Seq(
  "limit: 1 => This, is, Programming, d=ArraySeq(Scala, v3)",
  "limit: 3 => This, Programming, Scala, d=ArraySeq()",
  "No words of length >= 20!",
  "limit: 100 => Tokenize refused! x=(100,This is Programming Scala v3)"))
```

❶ If we didn't match on the `limit` value, this is what the declaration would be.

❷ We match on a tuple with the limit for word size and the string of words. If successful, we return `Some(Seq(words))`, where the words are filtered for those with a length of at least `limit`. We consider it unsuccessful and return `None` when the input `limit` is greater than the length of the input string.

❸ Split on whitespace.

❹ Capture the first three words returned and the rest of them as a repeated parameters list (`d`).

Try simplifying this example to not do length filtering. Uncomment the line for comment 1 and work from there.

Recap and What's Next

Along with `for` comprehensions, pattern matching makes idiomatic Scala code concise, yet powerful. It provides a protocol for extracting data inside data structures in a principled way, one you can control by implementing custom `unapply` and `unapplySeq` methods. These methods let you extract that information while hiding other details. In fact, the information returned by `unapply` might be a transformation of the actual fields in the instance.

Pattern matching is a hallmark of many functional languages. It is a flexible and concise technique for extracting data from data structures. We saw examples of pattern matching in `case` clauses and how to use pattern matching in other expressions too.

The next chapter discusses a unique, powerful, but controversial feature in Scala—context abstractions, formerly known as *implicits*, which are a set of tools for building intuitive DSLs, reducing boilerplate, and making APIs both easier to use and more amenable to customization.

CHAPTER 5

Abstracting Over Context: Type Classes and Extension Methods

3 In previous editions of this book, this chapter was titled “Implicits” after the mechanism used to implement many powerful idioms in Scala. Scala 3 begins the migration to new language constructs that emphasize purpose over mechanism, to make both learning and using these idioms easier and to address some shortcomings of the prior implementation. The transition will happen over several 3.X releases of Scala to make it easier, especially for existing code bases. Therefore, I will cover both the Scala 2 and 3 techniques, while emphasizing the latter.[1]

All of these idioms fall under the umbrella *abstracting over context*. We saw a few examples already, such as the `ExecutionContext` parameters needed in many `Future` methods, discussed in “A Taste of Futures” on page 42. We’ll see many more idioms now in this chapter and the next. In all cases, the idea of *context* will be some situation where an extension to a type, a transformation to a new type, or an insertion of values automatically is desired for easier programming. Frankly, in all cases, it would be possible to live without the tools described here, but it would require more work on the user’s part. This raises an important point, though. Make sure you use these tools judiciously; all constructs have pros and cons.

How we define and use context abstractions are the most significant changes introduced in Scala 3 that will impact how your Scala code looks in the future (assuming you stick with brace syntax). They are designed to make the purpose and application of these abstractions more clear. The underlying implicit mechanism is still there, but it’s now easier to use for specific purposes. The changes not only make intention

1 Because this chapter and the next one are extensively devoted to new Scala 3 features, I won’t use the “3” icon in the margins again until Chapter 7.

more clear but also eliminate some boilerplate previously required when using implicits, as well as fix other drawbacks of the Scala 2 idioms.

Four Changes

If you know Scala 2 implicits, the changes in Scala 3 can be summarized as follows:[2]

Given instances
: Instead of declaring implicit terms (i.e., `vals` or methods) to be used to resolve implicit parameters, the new `given` clause specifies how to synthesize the required term from a type. The change de-emphasizes the previous distinction where we had to know when to declare a method, an instance, or a type. Now, most of the time we will specify that a particular type should be used to satisfy the need for a value, and the compiler will do the rest.

Using clauses
: Instead of using the keyword `implicit` to declare an implicit parameter list for a method, we use the keyword `using`, and now it is also used when providing parameters explicitly. The changes eliminate several ambiguities, and they allow a method definition to have more than one implicit parameter list, now called *using clauses*.

Given imports
: When we use wildcards in import statements, they no longer import `given` instances along with everything else. Instead, we use the `given` keyword to specify when `givens` should be imported.

Implicit conversions
: For the special case of `given` terms that are used for implicit conversions between types, they are now declared as `given` instances of a standard `Conversion` class. All other forms of implicit conversions will be phased out.

This chapter explores the context abstractions for extending types with additional state and behavior using *extension methods* and *type classes*, which are defined using `given` instances. We'll also cover `given` imports and implicit conversions. In Chapter 6, we'll explore using clauses and the specific idioms they support.

2 Adapted from this Dotty documentation.

Extension Methods

In Scala 2, if we wanted to simulate adding new methods to existing types, we had to do an implicit conversion to a wrapper type that implements the method. Scala 3 adds extension methods that allow us to extend a type with new methods without conversion. By themselves, extension methods only allow us to add one or more methods, but not new fields for additional state, nor is there a mechanism for implementing a common abstraction. We'll revisit those limitations when we discuss type classes.

But first, why not just modify the original source code? You may not have that option, for example if it's a third-party library. Also, adding too many methods and fields to classes makes them very difficult to maintain. Every modification to an existing type forces users to recompile their code, at least. This is especially annoying if the changes involve functionality they don't even use.

Context abstractions help us avoid the temptation to create types that contain lots of utility methods that are used only occasionally. Our types can avoid *mixing concerns*. For example, if some users want `toJSON` methods on a hierarchy of types, like our `Shapes` in "A Sample Application" on page 20, then only those users are affected.

Hence, the goal is to enable ad hoc additions to types in a principled way, where types remain focused on their core abstractions, while additional behaviors are added separately and only where needed. Global modifications that affect all users are minimized.

However, a drawback of this separation of concerns is that the separate functionality needs to track the evolution of the type hierarchy. If a field is renamed, the compiler will catch it for us. If a new field is added, for example, `Shape.color`, it will be easy to miss.

Let's explore an example. Recall that we used the pair construction idiom, `a -> b`, to create tuples `(a, b)`, which is popular for creating `Map` instances:

```
val map = Map("one" -> 1, "two" -> 2)
```

In Scala 2, this is done using an implicit conversion to a library type `ArrowAssoc` in `Predef` (some details omitted for simplicity):

```
implicit final class ArrowAssoc[A](private val self: A) {
  def -> [B](y: B): (A, B) = (self, y)
}
```

When the compiler sees the expression `"one" -> 1`, it sees that `String` does not have the `->` method. However, `ArrowAssoc[T]` is in scope, it has this method, *and* the class is declared `implicit`. So the compiler will emit code to create an instance of `ArrowAssoc[String]`, with the string `"one"` passed as the `self` argument. Then `->(1)` is called to construct and return the tuple `("one", 1)`.

If `ArrowAssoc` were not declared `implicit`, the compiler would not attempt to use it for this purpose.

Let's reimplement this using a Scala 3 extension method, using two ways to define it. To avoid ambiguity with `->`, I'll use `~>` and `~~>` instead, but they work identically:

```
// src/script/scala/progscala3/contexts/ArrowAssocExtension.scala

scala> extension [A] (a: A)
     |   def ~>[B](b: B): (A, B) = (a, b)
def ~>[A](a: A): [B](b: B): (A, B)                              ❶

scala> extension [A,B] (a: A)
     |   def ~~>(b: B): (A, B) = (a, b)
def ~~>[A, B](a: A)(b: B): (A, B)

scala> "one" ~> 1
val res0: (String, Int) = (one,1)

scala> "two" ~~> 2
val res1: (String, Int) = (two,2)

scala> ~>("ONE")(1.1)
val res2: (String, Double) = (ONE,1.1)

scala> ~~>("TWO")(2.2)
val res3: (String, Double) = (TWO,2.2)
```

❶ Note the method signatures returned by the REPL for both definitions.

The syntax for defining an extension method starts with the `extension` keyword, followed by type parameters, an argument list for the type being extended, and one or more methods.

The difference between the two methods is where we specify the `B` type parameter. The first definition for `~>` is how type parameters are specified for regular type and method definitions, where `A` is needed for the type being extended and `B` is only needed on the method. The second syntax for `~~>` is an alternative, where all the type parameters are specified after the `extension` keyword. The value `a` is used to refer to the instance of the extended type, `A`.

From the method signatures shown, we see that each method has two parameter lists. The first list is for the target instance of type `A`. The second list is for the instance of `B`. In fact, both methods can be called like any other method, as shown in the last two examples.

So when the compiler sees `"one" ~> 1`, it finds the `~>` method in scope and emits code to call it. Using an implicit conversion to wrap the `a` value in a new instance, as required in Scala 2, is no longer necessary. Extension methods are a more straightforward mechanism than implicit conversions in Scala 2.

Let's complete an example we started in "Defining Operators" on page 71, where we showed that parameterized types with two parameters can be written with infix notation. At the time, we didn't know how to support using the same type name as an operator for constructing instances. Specifically, we defined a type `<+>` allowing declarations like `Int <+> String`, but we couldn't define a value of this type using the same literal syntax, for example, `2 <+> "two"`. Now we can do this by defining an extension method `<+>` as follows:

```
// src/script/scala/progscala3/contexts/InfixTypeRevisited.scala
import scala.annotation.targetName

@targetName("TIEFighter") case class <+>[A,B](a: A, b: B)         ❶
extension [A] (a: A) def <+>[B](b: B): A <+> B = new <+>(a, b)    ❷

val ab1: Int <+> String = 1 <+> "one"                              ❸
val ab2: Int <+> String = <+>(1, "one")                            ❹
```

❶ The same case class defined in "Defining Operators" on page 71.

❷ The extension method definition. When only one method is defined, you can define it on the same line, as shown. Note that `new` must be used on the righthand side to disambiguate between the type `<+>` and the method (but we're pushing the limits of readability).

❸ This line failed to compile before, but now the extension method is applied to `Int` and invoked with the `String` argument `"one"`.

❹ Constructing a case-class instance the old-fashioned way.

With extension methods, we get the ability to call methods like `->` when we need them, while keeping types as focused and uncluttered as possible.

So far we have extended classes. What about extension methods on `objects`? An object can be thought of as a singleton. To get its type, use `Foo.type`:

```
// src/script/scala/progscala3/contexts/ObjectExtensionMethods.scala
scala> object Foo:
     |   def one: Int = 1
     |
     | extension (foo: Foo.type)
     |   def add(i: Int): Int = i + foo.one
def add(foo: Foo.type)(i: Int): Int

scala> Foo.one
     | Foo.add(10)
```

```
val res0: Int = 1
val res1: Int = 11
```

Note the method signature returned by the REPL. `Foo.type`, the `Foo` object's type, is being extended. Incidentally, the type of a case-class companion object is the case-class name with `.type`, such as `Person.type` for the `Person` case class we used in Chapter 4.

Build Your Own String Interpolator

The code examples contain another example of extension methods that are used to implement a string interpolator for parsing simple SQL queries such as `sql"SELECT name, salary FROM employees;"`. See *src/main/scala/progscala3/contexts/SQLStringInterpolator.scala*. It uses the same mechanism as Scala's built-in interpolators, `s"…"`, `f"…"`, and `raw"…"` (see "Interpolated Strings" on page 82). The `sql` method is implemented as an extension method for the library type `scala.StringContext`.

The example illustrates two important points. First, we can use extension methods (and type classes, which follow next) to enhance library code that we don't own or control! Second, string interpolators are not required to return a new string. They can return any type we want.

Type Classes

The next step beyond extension methods is to implement an abstraction, so all type extensions are done uniformly. A term that is popular for these kinds of extensions is *type class*, which comes from the Haskell language, where this idea was pioneered. The word *class* in this context is not the same as a Scala `class`, which can be confusing. For this reason, some people like to spell it *typeclass*, to reinforce the distinction even more.

Type classes will also give us the ability to have state across all instances of a type, like the state you might keep in a type's companion object. I'll call it *type-level state* for convenience. We won't gain the ability to add *fields* to individual instances of a class (i.e., *instance-level state*). When that's required, you'll need to add the fields to the original type definition or use mixin traits.

As an example, consider our `Shape` hierarchy from "A Sample Application" on page 20. We want the ability to call `someShape.toJSON` that returns a JSON representation appropriate for each type. Let's examine the pros and cons of using a type class to implement this feature.

Scala 3 Type Classes

A type class is declared with a trait that defines the abstraction. It can have any extension (instance) methods, as well as type-level members, meaning across all instances. A type class provides another way to implement mixin composition (see "Traits: Interfaces and Mixins in Scala" on page 99). The trait for the abstraction is valuable for ensuring that all "instances" of the type class follow the same protocol uniformly. We will define *one* instance of the type class for each type that needs the `toJSON` method. Each instance will customize the implementation as required for the corresponding type.

To keep things simple, we'll return JSON-formatted strings, not objects from some JSON library:

```
// src/main/scala/progscala3/contexts/json/ToJSON.scala
package progscala3.contexts.json

trait ToJSON[T]:
  extension (t: T) def toJSON(name: String = "", level: Int = 0): String

  protected val indent = "  "
  protected def indentation(level: Int): (String,String) =
    (indent * level, indent * (level+1))
  protected def handleName(name: String): String =
    if name.length > 0 then s""""$name": """ else ""
```

This is the Scala 3 type class pattern. We define a trait with a type parameter. It has one extension method, `toJSON`, the public method users care about. This is an instance method for instances of the target type `T`. The `protected` methods, `indentation` and `handleName`, and the `indent` value, are implementation details. They are type-level members, not instance-level members, which is why these two methods are *not* extension methods.

Now create instances of the type class, one for `Point` and one each for the `Shape` types, with implementations for `Rectangle` and `Triangle` omitted:

```
// src/main/scala/progscala3/contexts/typeclass/new1/ToJSONTypeClasses.scala
package progscala3.contexts.typeclass.new1

import progscala3.introscala.shapes.{Point, Shape, Circle, Rectangle, Triangle}
import progscala3.contexts.json.ToJSON

given ToJSON[Point] with                                                    ❶
  extension (point: Point)
    def toJSON(name: String = "", level: Int = 0): String =
      val (outdent, indent) = indentation(level)
      s"""${handleName(name)}{
        |${indent}"x": "${point.x}",
        |${indent}"y": "${point.y}"
        |$outdent}""".stripMargin
```

```
given ToJSON[Circle] with                                            ❷
  extension (circle: Circle)
    def toJSON(name: String = "", level: Int = 0): String =
      val (outdent, indent) = indentation(level)
      s"""${handleName(name)}{
        |${indent}${circle.center.toJSON("center", level + 1)},
        |${indent}"radius": ${circle.radius}
        |$outdent}""".stripMargin
```

❶ The `given` keyword declares an instance of the type class, `ToJSON[Point]`. The extension method for `ToJSON` is implemented. Note that `with` is used to start the body where the abstract members of `ToJSON` are implemented.

❷ A given for `ToJSON[Circle]`.

Here is an entry point to try it:

```
@main def TryJSONTypeClasses() =
  println(s"summon[ToJSON[Point]] = ${summon[ToJSON[Point]]}")        ❶
  println(s"summon[ToJSON[Circle]] = ${summon[ToJSON[Circle]]}")
  println(Circle(Point(1.0,2.0), 1.0).toJSON("circle", 0))
  println(Rectangle(Point(2.0,3.0), 2, 5).toJSON("rectangle", 0))
  println(Triangle(
    Point(0.0,0.0), Point(2.0,0.0), Point(1.0,2.0)).toJSON("triangle", 0))
```

❶ The `summon` method is described ahead.

Running `TryJSONTypeClasses` prints the following:

```
> runMain progscala3.contexts.typeclass.new1.TryJSONTypeClasses
...
summon[ToJSON[Point]] = ...given_ToJSON_Point...
summon[ToJSON[Circle]] = ...given_ToJSON_Circle...
"circle": {
  "center": {
    "x": "1.0",
    "y": "2.0"
  },
  "radius": 1.0
}
...
```

The first two lines of output show us the names generated by the compiler: `given_ToJSON_Point` and `given_ToJSON_Circle`, respectively (with other details omitted). Each of these given instances is an `object`. They can be used directly, although because these naming conventions are a compiler implementation detail, it's more robust to use the `Predef.summon` method. If you know Scala 2 implicits, `summon` works the same as the `implicitly` method. You specify a type parameter and it returns the given instance or *implicit value* in scope for that type. We'll see more examples where `summon` is used as we go.

Not all `given` instances are type class instances. We'll see other examples in this chapter and the next.

There's a problem with our current implementation. What if we have a `Seq[Shape]` and we want to use `toJSON`?

```
Seq(Circle(Point(1.0,2.0), 1.0), Rectangle(Point(2.0,3.0), 2, 5)).map(
  shape => shape.toJSON("shape", 0))
```

We get an error for `shape.toJSON("shape", 0))` that `toJSON` is not a member of `Shape`. We didn't explicitly define a given for `ToJSON[Shape]`. Even if we did, the usual object-oriented method dispatch rules do not work for extension methods!

What if we add a given for `Shape` that pattern matches on the type of `Shape`?

```
given ToJSON[Shape] with
  extension (shape: Shape) def toJSON(name: String, level: Int): String =
    shape match
      case c: Circle    => c.toJSON(name, level)
      case r: Rectangle => r.toJSON(name, level)
      case t: Triangle  => t.toJSON(name, level)
```

This compiles, but we get an infinite recursion at runtime! This is because the same `ToJSON[Shape].toJSON` method is called recursively, not the more specific methods for `Circle`, and so forth.

Instead, let's call the compiler generated `toJSON` implementations directly using the `summon` method. We'll use a completely new implementation, just showing what's new:

```
// src/main/scala/progscala3/contexts/typeclass/new2/ToJSONTypeClasses.scala

given ToJSON[Shape] with
  extension (shape: Shape)
    def toJSON(name: String = "", level: Int = 0): String =
      shape match
        case c: Circle    =>
          summon[ToJSON[Circle]].toJSON(c)(name, level)
        case r: Rectangle =>
          summon[ToJSON[Rectangle]].toJSON(r)(name, level)
        case t: Triangle  =>
          summon[ToJSON[Triangle]].toJSON(t)(name, level)

@main def TryJSONTypeClasses() =
  val c = Circle(Point(1.0,2.0), 1.0)
  val r = Rectangle(Point(2.0,3.0), 2, 5)
  val t = Triangle(Point(0.0,0.0), Point(2.0,0.0), Point(1.0,2.0))
  println("==== Use shape.toJSON:")
  Seq(c, r, t).foreach(s => println(s.toJSON("shape", 0)))
  println("==== call toJSON on each shape explicitly:")
```

```
println(c.toJSON("circle", 0))
println(r.toJSON("rectangle", 0))
println(t.toJSON("triangle", 0))
```

The output of …contexts.typeclass.new2.TryJSONTypeClasses (not shown) verifies that calling shape.toJSON, and the more specific circle.toJSON, now both work as desired.

An alternative to using summon or the compiler-generated name is to provide names for the given instances. A final variant is shown next (just for Triangle) and the updated ToJSON[Shape]:

```
// src/main/scala/progscala3/contexts/typeclass/new3/ToJSONTypeClasses.scala

given triangleToJSON: ToJSON[Triangle] with                          ❶
  def toJSON2(
      tri: Triangle, name: String = "", level: Int = 0): String =    ❷
    val (outdent, indent) = indentation(level)
    s"""${handleName(name)}{
      |${indent}${tri.point1.toJSON("point1", level + 1)},
      |${indent}${tri.point2.toJSON("point2", level + 1)},
      |${indent}${tri.point3.toJSON("point3", level + 1)},
      |$outdent}""".stripMargin
  extension (tri: Triangle)
    def toJSON(name: String = "", level: Int = 0): String =
      toJSON2(tri, name, level)                                      ❸

given ToJSON[Shape] with
  extension (shape: Shape)
    def toJSON(name: String = "", level: Int = 0): String =
      shape match
        case c: Circle    => circleToJSON.toJSON2(c, name, level)    ❹
        case r: Rectangle => rectangleToJSON.toJSON2(r, name, level)
        case t: Triangle  => triangleToJSON.toJSON2(t, name, level)
```

❶ The given instance is now named triangleToJSON, instead of the synthesized name given_ToJSON_Triangle. Note the name: Type syntax, like normal variable declarations.

❷ A type-level helper method. Note the first argument is a Triangle instance.

❸ The extension method now calls the helper method.

❹ Cleaner syntax. The helper methods are used; however, the type-level toJSON2 method is now part of the public abstraction for each given instance, which could be confusing.

> Polymorphic dispatch (i.e., object-oriented method dispatch) does *not* work for extension methods! This is problematic in all the `ToJSON[Shape]` variants, which match on the particular subtype of `Shape`. This code will break as soon as a new `Shape` subtype is added (e.g., `Polygon`)!

This example illuminates the trade-offs between choosing type classes with extension methods versus object-oriented polymorphic methods. A `Shape.toJSON` method is a very good candidate for a regular method, declared abstract in `Shape` and implemented in the concrete subtypes. If you need this method frequently in your domain classes, it might be worth the disadvantage of expanding the API footprint and the implementation size of your types. Furthermore, the `match` expressions in the `ToJSON[Shape]` implementations are very fragile. Because `Shape` is deliberately open for extension, meaning it is designed to support whatever subtypes users desire, the `match` expressions in the different `ToJSON[Shape]` implementations will break as soon as new `Shape` subtypes are added!

Three Kinds of Polymorphism

To recap, here are the three kinds of polymorphism we have encountered:

1. Extension methods implement *ad hoc polymorphism* because the polymorphic behavior of `toJSON` is not tied to the type system hierarchy. `Point` and the `Shapes` are not related in the type hierarchy (ignoring `Any` and `AnyRef` at the top of the type hierarchy), but we defined `toJSON` with consistent behavior for all of them.
2. Traditional overriding of methods in subtypes is *subtype polymorphism*, which allows supertypes to declare abstractions that are defined in subtypes. We had to hack around this missing feature for `toJSON`! This mechanism is also the only way to support instance-level fields, either defined in the core type hierarchy or using mixin composition with traits.
3. For completeness, we encountered *parametric polymorphism* in "Parameterized Types Versus Abstract Type Members" on page 66, where types like `Seq[A]` and methods like `map` behave uniformly for any type `A`.

Let's explore another example that illustrates the differences between instance-level extension methods and type-level members. A second example will also help us internalize all the new details we're learning about given instances and type classes.

It makes sense to think of type-level members as the analogs of companion object members. Each given instance for some type `T` is an `object`, so in a way, `T` gets an additional companion object for each type class instance created.

Let's look at type classes for `Semigroup` and `Monoid`. `Semigroup` generalizes the notion of addition or composition. You know how addition works for numbers, and even strings can be "added." *Monoid* adds the idea of a unit value. If you add zero to a number, you get the number back. If you prepend or append an empty string to another string, you get the second string back.

Here are the definitions for these types:[3]

```scala
// src/main/scala/progscala3/contexts/typeclass/MonoidTypeClass.scala
package progscala3.contexts.typeclass
import scala.annotation.targetName

trait Semigroup[T]:
  extension (t: T)
    infix def combine(other: T): T                                    ❶
    @targetName("plus") def <+>(other: T): T = t.combine(other)

trait Monoid[T] extends Semigroup[T]:
  def unit: T                                                         ❷

given StringMonoid: Monoid[String] with                               ❸
  def unit: String = ""
  extension (s: String) infix def combine(other: String): String = s + other

given IntMonoid: Monoid[Int] with
  def unit: Int = 0
  extension (i: Int) infix def combine(other: Int): Int = i + other
```

❶ Define an instance extension method `combine` and an alternative operator method `<+>` that calls `combine`. Note that combining one element with another of the same type returns a new element of the same type, like adding numbers. For given instances of the type class, we will only need to define `combine`, since `<+>` is already concrete.

❷ The definition for `unit`, such as zero for addition of numbers. It's not defined as an extension method but rather as a type-level or object method because we only need one instance of the value for all instances of a particular type `T`.

❸ Define `Monoid` instances as givens for `String` and `Int`.

> Even though the abstract `combine` extension method in `Semigroup` is declared `infix`, the concrete `combine` methods are not automatically `infix`. They must be declared `infix` too.

3 Adapted from the Dotty documentation.

The combine operation is associative. Here are examples for Strings and Ints:

```
// src/script/scala/progscala3/contexts/typeclass/MonoidTypeClass.scala
import progscala3.contexts.typeclass.{Monoid, given}         ❶

"2" <+> ("3" <+> "4")          // "234"
("2" <+> "3") <+> "4"          // "234"
("2" combine "3") combine "4"  // "234"
StringMonoid.unit <+> "2"      // "2"
"2" <+> StringMonoid.unit      // "2"

2 <+> (3 <+> 4)                // 9
(2 <+> 3) <+> 4                // 9
(2 combine 3) combine 4        // 9
IntMonoid.unit <+> 2           // 2
2 <+> IntMonoid.unit           // 2
```

❶ Import Monoid and the defined givens. This use of given in the import statement will be explained in "Givens and Imports" on page 159.

Notice how each unit is referenced. Easy to remember names for the given instances are convenient here. Alternatively, we could have kept them anonymous and used summon[Monoid[String]].unit, for example, as before.

Finally, we don't actually need to define separate instances for each Numeric type. Here is how to implement it once for a type T for which Numeric[T] exists:

```
given NumericMonoid[T : Numeric]: Monoid[T] with
  def unit: T = summon[Numeric[T]].zero
  extension (t: T)
    infix def combine(other: T): T = summon[Numeric[T]].plus(t, other)

2.2 <+> (3.3 <+> 4.4)          // 9.9
(2.2 <+> 3.3) <+> 4.4          // 9.9
(2.2 combine 3.3) combine 4.4  // 9.9

BigDecimal(3.14) <+> NumericMonoid.unit                  ❶
NumericMonoid[BigDecimal].unit <+> BigDecimal(3.14)
NumericMonoid[BigDecimal].unit combine BigDecimal(3.14)
```

❶ The righthand side could be written NumericMonoid[BigDecimal].unit, but BigDecimal can be inferred. This doesn't work for the next two lines because the Monoid.unit is the object on which the methods are called.

The type [T : Numeric] is a *context bound*, a shorthand way of writing the definition this way:

```
given NumericMonoid[T](using num: Numeric[T]): Monoid[T] with
  def unit: T = num.zero
  extension (t: T)
    infix def combine(other: T): T = num.plus(t, other)
```

Note the using clause. If a given Numeric is in scope for a particular type T, then this type class instance can be used. The bodies are slightly different too. In the previous version, we used summon to get the anonymous using parameter, so we can reference zero and plus. In this version, we have a name for the using parameter, num.

Finally, you can still make this Monoid anonymous. Take either version we just discussed and drop the name. Here are both variants:

```
given [T : Numeric]: Monoid[T] with
  def unit: T = summon[Numeric[T]].zero
  extension (t: T)
    infix def combine(other: T): T = summon[Numeric[T]].plus(t, other)
// or
given [T](using num: Numeric[T]): Monoid[T] with
  def unit: T = summon[Numeric[T]].zero
  extension (t: T)
    infix def combine(other: T): T = summon[Numeric[T]].plus(t, other)

BigDecimal(3.14) <+> summon[Monoid[BigDecimal]].unit
summon[Monoid[BigDecimal]].unit <+> BigDecimal(3.14)
summon[Monoid[BigDecimal]].unit combine BigDecimal(3.14)
```

In both cases, summon[Monoid[BigDecimal]].unit is now required, as shown.

Note the colon, :, before Monoid[T] with in both alternatives. It will be easy to forget that colon. Fortunately, the compiler error message will tell you it's missing.

We will return to context bounds in "Context Bounds" on page 167 and using clauses in Chapter 6.

Alias Givens

While we have our Monoid example, let's learn about another feature. Look again at what the REPL prints when we define one of the NumericMonoid instances. Compare it to a new ByteMonoid definition:

```
scala> given NumericMonoid[T : Numeric]: Monoid[T] with
     |   def unit: T = summon[Numeric[T]].zero
     |   extension (t: T) infix def combine(other: T): T =
     |     summon[Numeric[T]].plus(t, other)
// defined class NumericMonoid

scala> given ByteMonoid: Monoid[Byte] with
     |   def unit: Byte = 0
     |   extension (b: Byte) infix def combine(other: Byte): Byte =
     |     (b + other).toByte
// defined object ByteMonoid
```

With the type parameter T, NumericMonoid must be a class for which instances will be created by the compiler when T is specified. In contrast, an object is created for ByteMonoid (as it was for IntMonoid and StringMonoid).

Suppose you don't want a given instance constructed eagerly. Perhaps it is implemented with something expensive like a database connection that should only be initialized when it's actually used, *if* it is used.

An *alias given* declares a named or anonymous given instance in a way that superficially looks like a more verbose syntax than what we used previously, but it actually produces a different result. Consider the following definitions:

```
// src/script/scala/progscala3/contexts/typeclass/MonoidAliasGiven.scala
scala> import progscala3.contexts.typeclass.Monoid

scala> given NumericMonoid2[T : Numeric]: Monoid[T] = new Monoid[T]:
     |   println("Initializing NumericMonoid2")
     |   def unit: T = summon[Numeric[T]].zero
     |   extension (t: T) infix def combine(other: T): T =
     |     summon[Numeric[T]].plus(t, other)
def NumericMonoid2
  [T](using evidence$1: Numeric[T]): progscala3.contexts.typeclass.Monoid[T]

scala> given StringMonoid2: Monoid[String] = new Monoid[String]:
     |   println("Initializing StringMonoid2")
     |   def unit: String = ""
     |   extension (s: String)
     |     infix def combine(other: String): String = s + other
lazy val StringMonoid2: progscala3.contexts.typeclass.Monoid[String]
```

In both examples, the syntax new Monoid[…]: body creates an anonymous subtype of the Monoid trait. Those println statements are in the bodies of the subtypes, so they will be called each time an instance is created.

Note the returned types printed by the REPL. Now we have a *method* for NumericMonoid2 and a lazy val for StringMonoid2 (see "Lazy Values" on page 97). What are the implications of these details?

```
scala> 2.2 <+> (3.3 <+> 4.4)            // 9.9
Initializing NumericMonoid2
Initializing NumericMonoid2
val res0: Double = 9.9

scala> (2.2 <+> 3.3) <+> 4.4            // 9.9
Initializing NumericMonoid2
Initializing NumericMonoid2
val res1: Double = 9.9

scala> "2" <+> ("3" <+> "4")            // "234"
Initializing StringMonoid2
val res2: String = 234
```

```
scala> ("2" <+> "3") <+> "4"             // "234"
val res3: String = 234
```

The method NumericMonoid2 is called every single time the <+> extension method is used for a Numeric[T] value. The println output occurs twice for each example because we construct two instances, one for each <+> invocation. So for given instances with type parameters, be careful about using an alias given.

However, because StringMonoid2 is a lazy val, it will be initialized once and only once, and initialization will be delayed until the first time we use it. Hence, this is a good option when you need delayed initialization.

Scala 2 Type Classes

Scala 2 also has a syntax for implementing type classes and instances. For a while, it will still be supported, and you'll see it in Scala 2 code bases. Returning to the ToJSON type class, you write an implicit conversion that wraps the Point and Shape instances in new instances of a type that has the toJSON method, then call the method.

First, we need a slightly different ToJSON trait because the extension method code used previously won't work with Scala 2:

```
// src/main/scala/progscala3/contexts/typeclass/old/ToJSONTypeClasses.scala
package progscala3.contexts.typeclass.old

import progscala3.introscala.shapes.{Point, Shape, Circle, Rectangle, Triangle}

trait ToJSONOld[T]:
  def toJSON(name: String = "", level: Int = 0): String         ❶

  protected val indent = "  "
  protected def indentation(level: Int): (String,String) =
    (indent * level, indent * (level+1))
  protected def handleName(name: String): String =
    if name.length > 0 then s""""$name": """ else ""
```

❶ A regular instance method, not an extension method.

Here is implementations for Point and Circle of toJSON type class instances using Scala 2 syntax:

```
implicit final class PointToJSON(
    point: Point) extends ToJSONOld[Point]:
  def toJSON(name: String = "", level: Int = 0): String =
    val (outdent, indent) = indentation(level)
    s"""${handleName(name)}{
      |${indent}"x": "${point.x}",
      |${indent}"y": "${point.y}"
      |$outdent}""".stripMargin
```

```scala
implicit final class CircleToJSON(
    circle: Circle) extends ToJSONOld[Circle]:
  def toJSON(name: String = "", level: Int = 0): String =
    val (outdent, indent) = indentation(level)
    s"""${handleName(name)}{
      |${indent}${circle.center.toJSON("center", level + 1)},
      |${indent}"radius": ${circle.radius}
      |$outdent}""".stripMargin
```

Classes are declared to define the type class instances. Because Scala 3 type class instances are (usually) implemented as `objects`, they are more consistent with the type class instance terminology. Note that an instance of `PointToJSON` is created every time an instance of `Point` calls `toJSON`, for example.

Use sbt to run the following to try out the code:

```scala
@main def TryJSONTypeClasses() =
  val c = Circle(Point(1.0,2.0), 1.0)
  val r = Rectangle(Point(2.0,3.0), 2, 5)
  val t = Triangle(Point(0.0,0.0), Point(2.0,0.0), Point(1.0,2.0))
  println(c.toJSON("circle", 0))
  println(r.toJSON("rectangle", 0))
  println(t.toJSON("triangle", 0))
```

Because these classes are declared as implicit, when the compiler sees `circle.toJSON()`, for example, it will find the implicit conversion in scope that returns some wrapper type that has this method.

The output of `TryJSONOldTypeClasses` works as expected. However, we didn't solve the problem of iterating through some `Shapes` and calling `toJSON` polymorphically. You can try that yourself.

We didn't declare our implicit classes as cases classes. In fact, Scala doesn't allow an implicit class to also be a case class. It wouldn't make much sense anyway, because the extra, autogenerated code for the case class would never be used. Implicit classes have a very narrow purpose. Similarly, declaring them `final` is recommended to eliminate some potential surprises when the compiler resolves which type classes to use.

If you need to support Scala 2 code for a while, then using this type class pattern will work for a few versions of Scala 3. However, in most cases, it will be better to migrate to the new type class syntax because it is more concise and purpose-built, and it doesn't require the overhead of implicit conversions.

Scala 3 Implicit Conversions

We saw that an implicit conversion called `ArrowAssoc` was used in the Scala 2 library to implement the `"one" -> 1` idiom, whereas we can use an extension method in Scala 3. We also saw implicit conversions used for type classes in Scala 2, while Scala 3 combines extension methods and given instances to avoid doing conversions.

Hence, in Scala 3, the need to do implicit conversions is greatly reduced, but it hasn't disappeared completely. Sometimes you want to convert between types for other reasons. Consider the following example that defines types to represent `Dollars`, `Percentages`, and a person's `Salary`, where the gross salary and the percentage to deduct for taxes are encapsulated. When constructing a `Salary` instance, we want to allow users to enter `Doubles`, for convenience. First, let's define the types for the problem:

```
// src/main/scala/progscala3/contexts/accounting/NewImplicitConversions.scala
package progscala3.contexts.accounting

case class Dollars(amount: Double):
  override def toString = f"$$$amount%.2f"
  def +(d: Dollars): Dollars = Dollars(amount + d.amount)          ❶
  def -(d: Dollars): Dollars = Dollars(amount - d.amount)
  def /(d: Double): Dollars  = Dollars(amount / d)
  def *(p: Percentage): Dollars = Dollars(amount * p.toMultiplier)

object Dollars:
  val zero = Dollars(0.0)

/**
 * @param amount where 11.0 means 11%, so 11% of 100 == 11.0.
 */
case class Percentage(amount: Double):
  override def toString = f"${amount}%.2f%%"
  def toMultiplier: Double = amount/100.0
  def +(p: Percentage): Percentage = Percentage(amount + p.amount)
  def -(p: Percentage): Percentage = Percentage(amount - p.amount)
  def *(p: Percentage): Percentage = Percentage(toMultiplier * p.toMultiplier)
  def *(d: Dollars): Dollars = d * this

object Percentage:
  val hundredPercent = Percentage(100.0)
  val zero = Percentage(0.0)

case class Salary(gross: Dollars, taxes: Percentage):
  def net: Dollars = gross * (Percentage.hundredPercent - taxes)
```

❶ Math operations.[4]

The Dollars class encapsulates a Double for the amount, with toString overridden to return the familiar "$dollars.cents" output. Similarly, Percentage wraps a Double and overrides toString.

Implicit conversions is an optional language feature that we enable by importing scala.language.implicitConversions to enable this language feature. You can also set the global -language:implicitConversions compiler flag. The following entry point adds this import inside the method:

```
@main def TryImplicitConversions() =
  import scala.language.implicitConversions                    ❶

  given Conversion[Double,Dollars] = d => Dollars(d)           ❷
  given Conversion[Double,Percentage] = d => Percentage(d)

  val salary = Salary(100_000.0, 20.0)
  println(s"salary: $salary. Net pay: ${salary.net}")

  given Conversion[Int,Dollars] with                           ❸
    def apply(i:Int): Dollars= Dollars(i.toDouble)

  val dollars: Dollars = 10                                    ❹
  println(s"Dollars created from an Int: $dollars")
```

❶ An import to enable the implicit conversion language feature.

❷ The most concise syntax for declaring a given conversion from Double to Dollars, and a second conversion from Double to Percentage.

❸ A longer form for defining a conversion as an alias given.

❹ Conversions are invoked when doing assignments, not just method arguments.

Running this example prints the following:

```
salary: Salary($100000.00,20.00%). Net pay: $80000.00
Dollars created from an Int: $10.00
```

If you define a given conversions in the REPL, observe what the REPL prints for the following different forms:

```
...
scala> import progscala3.contexts.accounting.Dollars
```

4 These calculations are crude and insufficiently accurate for real accounting applications. For one thing, BigDecimal would be a safer representation.

```
       | import scala.language.implicitConversions

scala> given Conversion[Double,Dollars] = d => Dollars(d)
lazy val given_Conversion_Double_Dollars:
  Conversion[Double, progscala3.contexts.accounting.Dollars]

scala> given id: Conversion[Int,Dollars] = i => Dollars(i.toDouble)
lazy val id: Conversion[Int, progscala3.contexts.accounting.Dollars]

scala> given Conversion[Float,Dollars] with
     |  def apply(f: Float): Dollars = Dollars(f.toDouble)
// defined object given_Conversion_Float_Dollars

scala> given ld: Conversion[Long,Dollars] with
     |  def apply(l: Long): Dollars = Dollars(l.toDouble)
// defined object ld
```

For the anonymous instances, note the naming convention `given_Conversion_A_B`.

Why are the resulting types different? Both `given_Conversion_Double_Dollars` and `id` are alias givens (see "Alias Givens" on page 150), so they are implemented as lazy vals. Conversely, `given_Conversion_Float_Dollars` and `ld` use the `given…with` syntax and define `apply` explicitly. These two givens are `objects`. Either approach works fine for conversions, but the alias given syntax is more concise:

```
scala> val fromDouble: Dollars = 10.1   // invoke conversions in assignments
     | val fromInt: Dollars = 20
     | val fromFloat: Dollars = 30.3F
     | val fromLong: Dollars = 40L
val fromDouble: progscala3.contexts.accounting.Dollars = $10.10
val fromInt: progscala3.contexts.accounting.Dollars = $20.00
val fromFloat: progscala3.contexts.accounting.Dollars = $30.30
val fromLong: progscala3.contexts.accounting.Dollars = $40.00

scala> summon[Conversion[Double,Dollars]](10.1)    // summon them...
     | summon[Conversion[Int,Dollars]](20)
     | summon[Conversion[Float,Dollars]](30.3)
     | summon[Conversion[Long,Dollars]](40)
val res0: progscala3.contexts.accounting.Dollars = $10.10
val res1: progscala3.contexts.accounting.Dollars = $20.00
val res2: progscala3.contexts.accounting.Dollars = $30.30
val res3: progscala3.contexts.accounting.Dollars = $40.00

scala> given_Conversion_Double_Dollars(10.1)       // call them directly
     | id(20)
     | given_Conversion_Float_Dollars(30.3)
     | ld(40)
val res4: progscala3.contexts.accounting.Dollars = $10.10
val res5: progscala3.contexts.accounting.Dollars = $20.00
val res6: progscala3.contexts.accounting.Dollars = $30.30
val res7: progscala3.contexts.accounting.Dollars = $40.00
```

Scala 3 still supports the Scala 2 mechanism of using an implicit method for conversion:

```
scala> implicit def toDollars(s: String): Dollars = Dollars(s.toDouble)
def toDollars(s: String): progscala3.contexts.accounting.Dollars

scala> toDollars("3.14")
val res11: progscala3.contexts.accounting.Dollars = $3.14

scala> val fromString: Dollars = "3.14"
val fromString: progscala3.contexts.accounting.Dollars = $3.14

scala> summon[String => Dollars]                                    ❶
val res9: String => progscala3.contexts.accounting.Dollars = Lambda$8531/...

scala> summon[String => Dollars]("3.14")
val res10: progscala3.contexts.accounting.Dollars = $3.14
```

❶ Ask for a given function. The compiler lifts `toDollar` to a function.

Trying to define an equivalent implicit val function won't work. The compiler will ignore it when searching for implicit conversions:

```
scala> implicit val b2D: Byte => Dollars = (b: Byte) => Dollars(b.toDouble)
val b2D: Byte => progscala3.contexts.accounting.Dollars = Lambda$8534/...

scala> val fromByte: Dollars = 0x1.toByte
1 |val fromByte: Dollars = 0x1.toByte
  |                        ^^^^^^^^^^
  |                      Found:    Byte
  |                      Required: progscala3.contexts.accounting.Dollars
```

Why are we even allowed to define `b2D`? We might need an implicit value somewhere that happens to be a function `Byte => Dollars`, but it won't be considered for implicit conversions.

Rules for Implicit Conversion Resolution

Let's summarize the lookup rules used by the compiler when a method is called on a target instance and it is necessary to find and apply either a new or old conversion. I'll refer to given instances, but the rules apply to both old and new conversions.

1. No conversion will be attempted if the target instance and method combination type check successfully.
2. Only given instances for conversion are considered.
3. Only given instances in the current scope are considered, as well as givens defined in the companion object of the target type.

4. Given conversions aren't chained to get from the available type, through intermediate types, to the target type. Only one conversion will be considered.
5. No conversion is attempted if more than one possible conversion could be applied and have the same scope. There must be one, and only one, unambiguous possibility.

Type Class Derivation

Type class derivation is the idea that we should be able to automatically generate type class `given` instances as long as they obey a minimum set of requirements, further reducing boilerplate. A type uses the new keyword `derives`, which works like `extends` or `with`, to trigger derivation.

For example, Scala 3 introduces `scala.CanEqual`, which restricts use of the comparison operators `==` and `!=` for instances of arbitrary types. Normally, it is permitted to do these comparisons, but when the compiler flag `-language:strictEquality` or the import statement `import scala.language.strictEquality` is used, then the comparison operators are only allowed in certain specific contexts. Here is an example:

```
// src/main/scala/progscala3/contexts/Derivation.scala

package progscala3.contexts
import scala.language.strictEquality

enum Tree[T] derives CanEqual:
  case Branch(left: Tree[T], right: Tree[T])
  case Leaf(elem: T)

@main def TryDerived() =
  import Tree.*
  val l1 = Leaf("l1")
  val l2 = Leaf(2)
  val b = Branch(l1,Branch(Leaf("b1"),Leaf("b2")))
  assert(l1 == l1)
  // assert(l1 != l2)   // Compilation error!
  assert(l1 != b)
  assert(b  == b)
  println(s"For String, String: ${summon[CanEqual[Tree[String],Tree[String]]]}")
  println(s"For Int, Int: ${summon[CanEqual[Tree[Int],Tree[Int]]]}")
  // Compilation error:
  // println(s"For String, Int: ${summon[CanEqual[Tree[String],Tree[Int]]]}")
```

Because of the `derives CanEqual` clause in the `Tree` declaration, the equality checks in the assertions are allowed. The `derives CanEqual` clause has the effect of generating the following given instance:

```
given CanEqual[Tree[T], Tree[T]] = CanEqual.derived
```

CanEqual.derived functions as a universal CanEqual given instance. It is defined as follows:

```
object CanEqual:
  object derived extends CanEqual[Any, Any]
  ...
```

Furthermore, T will be constrained to types with given CanEqual[T, T] = CanEqual.derived. What all this effectively means is that we can only compare Tree[T] instances for the same T types.

The terminology used is Tree as the *deriving type* and the CanEqual instance is a *derived instance*.

In general, any type T defined with a companion object that has the derived instance or method can be used with derives T clauses. We'll discuss more implementation details in "Type Class Derivation: Implementation Details" on page 491. The reason CanEqual and the strictEquality language feature were introduced is discussed in "Multiversal Equality" on page 296.

> If you want to enforce stricter use of comparison operators, use -language:strictEquality, but expect to add derives CanEqual to many of your types.

Givens and Imports

In "A Taste of Futures" on page 42, we imported an implicit ExecutionContext, scala.concurrent.ExecutionContext.Implicits.global. The name of the enclosing object Implicits reflects a common convention in Scala 2 for making implicit definitions more explicit in code that uses them, at least if you pay attention to the import statements.

Scala 3 introduces a new way to control imports of givens and implicits, which provides an effective alternative form of visibility, as well as allowing developers to use wildcard imports frequently while retaining control over if and when givens and implicits are also imported.

Consider the following example adapted from the Dotty documentation:

```
// src/script/scala/progscala3/contexts/GivenImports.scala

object O1:
  val name = "O1"
  def m(s: String) = s"$s, hello from $name"
  class C1
  class C2
```

```
  given c1: C1 = C1()
  given c2: C2 = C2()
```

Now consider these import statements:

```
import O1.*              // Import everything EXCEPT the givens, c1 and c2
import O1.given          // Import ONLY the givens (of type C1 and C2)
import O1.{given, *}     // Import everything, givens and nongivens in O1
import O1.{given C1}     // Import just the given of type C1
import O1.c2             // Import just the given c2 of type C2
```

The `import foo.given` selector also imports Scala 2 implicits. Note that when you qualify what to import, a given import expects a type, not a name. You really shouldn't define more than one given of the same type in the same scope anyway, as this would be ambiguous if you imported all of them. Since `given` and `using` clauses support anonymous values (while implicits didn't), anonymous values are frequently sufficient. However, if you want to import a given by name, just use a regular import statement, as shown in the last example.

What if you have parameterized given instances and you want to import only those, not any others in the scope?

```
trait Marker[T]
object O2:
  class C1
  given C1 = C1()
  given Marker[Int] with {}                    ❶
  given Marker[List[?]] with {}                ❷

import O2.{given Marker[?]}     // Import all given Markers
import O2.{given Marker[Int]}   // Import just the Marker[Int]
```

❶ There is nothing to implement for `Marker`, but the `with` is required and `{}` provides an empty body that is needed.

❷ The `?` is the wildcard for the type parameter.

> The use of `?` as a *type wildcard* is new to Scala 3. In Scala 2 you use `_`, which is still allowed, but it will be deprecated in a future release. The reason for the change is to reserve `_` for *type lambdas* (see "Type Lambdas" on page 391), just like the character is used for anonymous function arguments.

The new rules for the behavior of wildcard imports are breaking changes. Hence, they are being phased in gradually:

- In Scala 3.0, an old-style implicit definition will still be brought into scope using `foo.*`, as well as when using `foo.given`.
- In Scala 3.1, an old-style implicit accessed through a `*` wildcard import will give a deprecation warning.
- In some version after 3.1, old-style implicits accessed through a `*` wildcard import will give a compiler error.

Givens Scoping and Pattern Matching

Givens can be scoped when you don't want them globally visible in a source file. They can also be used in pattern-matching expressions, which also scopes their visibility. Consider the following definitions of ordinary objects:

```
// src/script/scala/progscala3/contexts/MatchGivens.scala

trait Witness                                                        ❶
case object IntWitness extends Witness
case object StringWitness extends Witness

def useWitness(using Witness): String = summon[Witness].toString    ❷
```

❶ A simple hierarchy of objects, none of which is declared as given instances.

❷ A method with a `using Witness` clause, which will require a `given Witness` to be in scope when called.

Let's see how pattern matching can be used to treat the objects as givens dynamically and also to scope their visibility:

```
scala> useWitness                                                   ❶
1 |useWitness
  |          ^
  |no implicit argument of type Witness was found...

scala> for given Witness <- Seq(IntWitness, StringWitness)          ❷
     | do println(useWitness)
IntWitness
StringWitness

scala> useWitness                                                   ❸
  |...no implicit argument of type Witness was found...
```

❶ Trying `useWitness` here shows that no given `Witness` is in scope.

❷ A loop over the objects, where the pattern `given Witness` types each object as a given, but also scoped to the body of the `for` loop. We see that each pass through the loop has one and only one given `Witness`, which it prints.

❸ Still throws a `no implicit` error, confirming that the `givens` in the `for` loop were scoped within its body.

Resolution Rules for Givens and Extension Methods

Extension methods and `given` definitions obey the same scoping rules as other declarations (i.e., they must be visible to be considered). The previous examples scoped the extension methods to packages, such as the `new1` and `new2` packages. They were not visible unless the package contents were imported or we were already in the scope of that package.

Within a particular scope, there could be several candidate givens or extension methods that the compiler might use for a type extension. The Dotty documentation has the details for the Scala 3 resolution rules. I'll summarize the key points here. Givens are also used to resolve implicit parameters in method using clauses, which we'll explore in the next chapter. The same resolution rules apply.

Rules for Given Resolution

I'll use the term "given" in the following discussion to include given instances, extension methods, and Scala 2 implicits. Resolving to a particular given happens in the following order:

1. Any type-compatible given that doesn't require a *prefix path*, such as other packages.
2. A given that was imported into the current scope.
3. Imported givens take precedence over the givens already in scope.
4. In some cases, several possible matches are type compatible. The most specific match wins. Suppose a `Foo` given is needed and `Foo` and `AnyRef` givens are in scope, then the `Foo` given will be chosen over the `AnyRef` given.
5. If two or more candidate givens are ambiguous, for example, they have the same exact type, it triggers a compiler error.

The compiler always puts some library givens in scope, while other library givens require an import statement. For example, `Predef` extends a type called `LowPriorityImplicits`, which makes the givens defined in `Predef` lower priority when potential conflicts arise with other givens in scope. The rationale is that the other givens are likely to be user defined or imported from special libraries, and hence more "important" to the user.

The Expression Problem

We learned in this chapter some powerful tools for adding new functionality to existing types without editing their source code! This desire to extend modules without modifying their source code is called the *Expression Problem*, a term coined by Philip Wadler.

Object-oriented programming solves this problem with subtype polymorphism. We program to abstractions and use derived classes to customize behavior. The Expression Problem in OOP terms is the *Open/Closed Principle*, coined by Bertrand Meyer. Base types declare the behaviors as abstract that should be open for extension or variation in subtypes, while keeping invariant behaviors closed to modification.

Working through the `ToJSON` examples, we saw the pros and cons of using *ad hoc extension* with type classes versus the OOP way of subtype polymorphism. Scala easily supports both approaches. Mixin composition provides additional flexibility. We have to decide what's best in a given context. Is some functionality core to a type hierarchy's state and behavior or is it peripheral? Is it used pervasively or only in limited contexts? What's the burden on maintainers and users of that functionality, implemented one way or another?

Recap and What's Next

We started our exploration of context abstractions in Scala 2 and 3, beginning with tools to extend types with additional state and behavior, such as type classes, extension methods, and implicit conversions.

The next chapter explores using clauses, which work with given instances to address particular design scenarios and to simplify user code.

CHAPTER 6

Abstracting Over Context: Using Clauses

In Chapter 5, we began our discussion of the powerful tools and idioms in Scala 2 and 3 for *abstracting over context*. In particular, we discussed type classes, extension methods, and implicit conversions as tools for extending the behaviors of existing types.

This chapter explores *using clauses*, which work with given instances to address particular design scenarios and to simplify user code.

Using Clauses

The other major use of context abstractions is to provide method parameters implicitly rather than explicitly. When a method argument list begins with the keyword `using` (Scala 3) or `implicit` (Scala 2 and 3), the user does not have to provide values explicitly for the parameters, as long as given instances or implicit values are in scope that the compiler can use instead.

In Scala 2 terminology, those method parameters were called *implicit parameters*, and the whole list of parameters was an *implicit parameter list* or *implicit parameter clause*. Only one such list was allowed, and it held all the implicit parameters. In Scala 3, they are *context parameters* and the whole parameter list is a using clause. There can be more than one using clause.[1] Here is an example:

```
class BankAccount(...):
  def debit(amount: Money)(using transaction: Transaction)
  ...
```

1 A regular parameter list is also known as a *normal parameter clause*, but I have just used the more familiar term *parameter list* in this book. *Using clause* is more of a formal term in Scala 3 documentation than *implicit parameter clause* was, which is why I emphasize it here.

Here, the *using clause* starts with the `using` keyword and contains the context parameter `transaction`.

The values in scope that can be used to fill in these parameters are called *implicit values* in Scala 2. In Scala 3 they are the given instances, or *givens* for short.

I'll mostly use the Scala 3 terminology in this book, but when I use Scala 2 terminology, it will usually be when discussing a Scala 2 library that uses `implicit` definitions and parameters. Scala 3 more or less treats them interchangeably, although the Scala 2 implicits will be phased out eventually.

For each parameter in a using clause, a type-compatible given must exist in the enclosing scope. Using Scala 2–style implicits, an implicit value or an implicit method or class returning a compatible value must be in scope.

For comparison, recall you can also define default values for method parameters. While sufficient in many circumstances, they are statically scoped to the method definition at compile time and are defined by the implementer of the method. Using clauses, on the other hand, provide greater flexibility for users of a method.

As an example, suppose we implement a simple type that wraps sequences for convenient sorting (ignoring the fact that this capability is already provided by `Seq`). One way to do this is for the user to supply an implementation of `math.Ordering`, which knows how to sort elements of the particular type used in the sequence. That object could be passed as an argument to the `sort` method, but the user might also like the ability to specify the value once, as a given, and then have all sequences of the same element type use it automatically.

This first implementation uses syntax valid for both Scala 2 and 3:

```
// src/script/scala-2/progscala3/contexts/ImplicitClauses.scala

case class SortableSeq[A](seq: Seq[A]) {                              ❶
  def sortBy1[B](transform: A => B)(implicit o: Ordering[B]): SortableSeq[A]=
    SortableSeq(seq.sortBy(transform)(o))

  def sortBy2[B : Ordering](transform: A => B): SortableSeq[A] =
    SortableSeq(seq.sortBy(transform)(implicitly[Ordering[B]]))
}

val seq = SortableSeq(Seq(1,3,5,2,4))

def defaultOrdering() = {                                             ❷
  assert(seq.sortBy1(i => -i) == SortableSeq(Seq(5, 4, 3, 2, 1)))    ❸
  assert(seq.sortBy2(i => -i) == SortableSeq(Seq(5, 4, 3, 2, 1)))
}
defaultOrdering()

def oddEvenOrdering() = {
```

```scala
    implicit val oddEven: Ordering[Int] = new Ordering[Int]:            ❹
      def compare(i: Int, j: Int): Int = i%2 compare j%2 match
        case 0 => i compare j
        case c => c

    assert(seq.sortBy1(i => -i) == SortableSeq(Seq(5, 3, 1, 4, 2)))     ❺
    assert(seq.sortBy2(i => -i) == SortableSeq(Seq(5, 3, 1, 4, 2)))
  }
  oddEvenOrdering()
```

❶ Use braces because this is also valid Scala 2 code.

❷ Wrap examples in methods to scope the use of implicits.

❸ Use the default ordering provided by `math.Ordering` for `Int`s, which is already in scope.

❹ Define a custom `oddEven` ordering, which will be the implicit value that takes precedence in the method's scope for the following lines.

❺ Implicitly use the custom `oddEven` ordering.

Let's focus on `sortBy1` for now. All the implicit parameters must be declared in their own parameter list. Here we need two lists because we have a regular parameter, the function `transform`. If we only had implicit parameters, we would need only one parameter list.

The implementation of `sortBy1` just uses the existing `Seq.sortBy` method. It takes a function that transforms the values to affect the sorting, and an `Ordering` instance to sort the values after transformation.

There is already a default implicit implementation in scope for `math.Ordering[Int]`, so we don't need to supply one if we want the usual numeric ordering. The anonymous function `i => -1` transforms the integers to their negative values for the purposes of ordering, which effectively results in sorting from highest to lowest.

Next, let's discuss the other method, `sortBy2`, and also explore new Scala 3 syntax for this purpose.

Context Bounds

If you think about it, while `SortableSeq` is declared to support any element type `A`, the two `sortBy*` methods bind the allowed types to those for which an `Ordering` exists. Hence, the term *context bound* is used for the implicit value in this situation.

In `SortableSeq.sortBy1`, the implicit parameter `o` is a context bound. A major clue is the fact that it has type `Ordering[B]`, meaning it is parameterized by the output element type, B. So, while it doesn't bind A explicitly, the result of applying transform is to convert A to B and then B is context bound by Ordering[B].

Context bounds are so common that Scala 2 defined a more concise way of declaring them in the types, as shown in sortBy2, where the syntax B : Ordering appears. (Note that it's not B : Ordering[B], as the [B] is omitted.) Also, they are sometimes referred to as *view types* because they filter the allowed types for B.

In the generated byte code for Scala 2, this is just shorthand for the same code we wrote explicitly for sortBy1, with one difference. In sortBy1, we defined a name for the Ordering parameter, o, in the second argument list. We don't have a name for it in sortBy2, but we need it in the body of the method. The solution is to use the method Predef.implicitly, as shown in the method body. It binds the implicit Ordering that is in scope so it can be passed as an argument.

Let's rewrite this code in Scala 3:

```
// src/script/scala/progscala3/contexts/UsingClauses.scala

case class SortableSeq[A](seq: Seq[A]):
  def sortBy1a[B](transform: A => B)(using o: Ordering[B]): SortableSeq[A] =
    SortableSeq(seq.sortBy(transform)(o))

  def sortBy1b[B](transform: A => B)(using Ordering[B]): SortableSeq[A] =
    SortableSeq(seq.sortBy(transform)(summon[Ordering[B]]))

  def sortBy2[B : Ordering](transform: A => B): SortableSeq[A] =
    SortableSeq(seq.sortBy(transform)(summon[Ordering[B]]))
```

The sortBy1a method is identical to the previous sortBy1 method with a using clause instead of an implicit parameter list. In sortBy1b, the name is omitted, making the parameter anonymous, and a new Predef method, summon, is used to bind the value instead (summon is functionally identical to implicitly). The sortBy2 here is written identically to the previous one in ImplicitClauses, but in Scala 3 it is implemented with a using clause.

The previously defined test methods, defaultOrdering and oddEvenOrdering, are almost the same in this source file, but they are not shown here. There is an additional test method in this file that uses a given instance instead of an implicit value:

```
def evenOddGivenOrdering() =
  given evenOdd: Ordering[Int] with
    def compare(i: Int, j: Int): Int = i%2 compare j%2 match
      case 0 => i compare j
      case c => -c

  val seq = SortableSeq(Seq(1,3,5,2,4))
  val expected = SortableSeq(Seq(4, 2, 5, 3, 1))
  assert(seq.sortBy1a(i => -i) == expected)                    ❶
```

```
  assert(seq.sortBy1b(i => -i) == expected)
  assert(seq.sortBy2(i => -i)  == expected)

  assert(seq.sortBy1a(i => -i)(using evenOdd) == expected)      // 2
  assert(seq.sortBy1b(i => -i)(using evenOdd) == expected)
  assert(seq.sortBy2(i => -i)(using evenOdd)  == expected)

evenOddGivenOrdering()
```

1. Use the given `evenOdd` instance implicitly.

2. Use the given `evenOdd` instance explicitly with `using`.

The syntax `given foo: Type[T]` is used instead of `implicit val foo: Type[T]`, essentially the same way we used givens when discussing type classes.

If the using clause is provided explicitly, as marked with comment 2, the `using` keyword is *required* in Scala 3, whereas Scala 2 didn't require or even allow the `implicit` keyword here. The reason `using` is now required is twofold. First, it's better documentation for the reader that this second argument list is a using clause. Second, it removes an occasional ambiguity that is illustrated in the following contrived Scala 2 example:

```
// src/script/scala-2/progscala3/contexts/ImplicitGotcha.scala

trait Context                                              // 1
implicit object SomeContext extends Context

case class Worker[T](seed: T)(implicit c: Context) {       // 2
  def apply(value: T): String = s"$seed, $value"
}

val i = Worker(-5)(2)                                      // 3
```

1. Some simple type to use as an implicit parameter.

2. A simple class that takes a regular parameter and an implicit parameter. It also has an `apply` method, which is necessary for the ambiguity to occur.

3. Attempt to use the implicit `SomeContext` and `apply`, which doesn't compile.

In the Scala 2 REPL, the last line is ambiguous:

```
.../ImplicitGotcha.scala:10: error: type mismatch;
 found   : Int(2)
 required: this.Context
val i = Worker(-5)(2)
                   ^
```

The 2 was not expected, even though it's a valid argument for `apply`. Because a second argument list is provided, it's assumed to be for the implicit value of type `Context`, rather than what I meant, the argument to `apply` with the implicit value `SomeContext` used automatically. We could use one of the following to work around this ambiguity:

```
val w = Worker(-5)
val i1 = w(2)
val i2 = Worker(-5).apply(2)
val i3 = Worker(-5)(SomeContext)(2)
```

As written with an `implicit`, the same error occurs in Scala 3, but if you change `implicit c: Context` to `using c: Context`, then the 2 is no longer ambiguous. The compiler knows you want to use the in-scope implicit value `SomeContext` and pass `2` to `apply`. When you want to explicitly pass a `Context`, you must now write `Work(5)(using AnotherContext)(2)`.

> The intent of the new `given`… syntax and the `using`… syntax is to make their purpose more explicit, but they function almost identically to Scala 2 `implicit` definitions and parameters.

Context parameters can be by-name parameters, providing the benefit of delayed evaluation until it is actually used. Here is a sketch of an example using a by-name context parameter for an expensive database connection:

```
// src/script/scala/progscala3/contexts/ByNameContextParameters.scala

type Status = String                                                    ❶

case class Transaction(database: String):                               ❷
  def begin(query: String): Status = s"$database: Starting transaction: $query"
  def rollback(): Status = s"$database: Rolling back transaction"
  def commit(): Status = s"$database: Committing transaction"

case class ConnectionManager(database: String):                         ❸
  println(s"... expensive initialization for database $database")
  def createTransaction: Transaction = Transaction(database)

def doTransaction(query: => String)(                                    ❹
    using cm: => ConnectionManager): Seq[Status] =
  val trans = cm.createTransaction
  Seq(trans.begin(query), trans.commit())

def doPostgreSQL =                                                      ❺
  println("Start of doPostgreSQL.")
  given ConnectionManager = ConnectionManager("PostgreSQL")
```

```
  println("Start of doTransaction.")
  doTransaction("SELECT * FROM table")
```

❶ Simple representation of the status returned by database commands.

❷ Transactions. The methods just return strings.

❸ A connection manager, intended to be expensive to create, so it's better to delay construction until needed.

❹ A method that runs a transaction. Note the using clause has a by-name parameter for the connection manager.

❺ A method to try a *PostgreSQL* transaction.

You get this output from `doPostgreSQL`:

```
scala> doPostgreSQL
Start of doPostgreSQL.
Start of doTransaction.
... expensive initialization for database PostgreSQL
val res2: Seq[Status] = List(
  PostgreSQL: Starting transaction: SELECT * FROM table,
  PostgreSQL: Committing transaction)
```

Note that the given instance isn't constructed until `doTransaction` is called.

Other Context Parameters

In "A Taste of Futures" on page 42, we saw that `Future.apply` has a second, implicit argument list that is used to pass an `ExecutionContext`:

```
object Future:
  apply[T](body: => T)(implicit executor: ExecutionContext): Future[T]
  ...
```

It is not a context bound because the `ExecutionContext` is independent of `T`.

We didn't specify an `ExecutionContext` when we called these methods, but we imported a global default that the compiler used:

```
import scala.concurrent.ExecutionContext.Implicits.global
Future(...)                   // Use the value implicitly
Future(...)(using global)     // Pass the value explicitly with "using"
```

`Future` supports a lot of the operations like `filter` and `map`. Like `Future.apply`, all have two parameter lists, where the second is a using clause for the `ExecutionContext`. The using clauses make the code much less cluttered than it would be if we had to pass the arguments explicitly:

```
given customExecutionContext: ExecutionContext = ...

val f1 = Future(...)(using customExecutionContext)
  .map(...)(using customExecutionContext)
  .filter(...)(using customExecutionContext)
// versus:
val f2 = Future(...).map(...).filter(...)
```

Similar using clauses might include transaction or web session identifiers, database connections, etc.

> The example shows that using contexts can make code more concise, but they can be overused too. When you see the same `using FooContext` all over a code base, it feels more like a global variable than pure FP.

Context Functions

Context functions are functions with context parameters only. Scala 3 introduces a new context function type for them, indicated by `?=> T`. Distinguishing context functions from regular functions is useful because of how they are invoked.

Consider this alternative for handling the `ExecutionContext` passed to `Future.apply()`, using a wrapper `FutureCF` (for context function):

```
// src/script/scala/progscala3/contexts/ContextFunctions.scala

import scala.concurrent.{Await, ExecutionContext, Future}
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration.*

object FutureCF:
  type Executable[T] = ExecutionContext ?=> T                       ❶

  def apply[T](body: => T): Executable[Future[T]] = Future(body)    ❷

def sleepN(dur: Duration): Duration =                               ❸
  val start = System.currentTimeMillis()
  Thread.sleep(dur.toMillis)
  Duration(System.currentTimeMillis - start, MILLISECONDS)

val future1 = FutureCF(sleepN(1.second))                            ❹
val future2 = FutureCF(sleepN(1.second))(using global)
val duration1 = Await.result(future1, 2.seconds)                    ❺
val duration2 = Await.result(future2, 2.seconds)
```

❶ Type member alias for a context function with an `ExecutionContext`.

❷ Compare this definition of `apply()` to `Future.apply()`. I discuss it in more detail next.

❸ Define some work that will be passed to futures; sleep for some `scala.concurrent.duration.Duration` and return the actual elapsed time as a `Duration`.

❹ Two futures are created in these two lines, one with an implicit `ExecutionContext` and the second with an explicit one.

❺ Await the results of the futures. Wait no longer than two seconds.

The last two lines print the following (your actual numbers may vary slightly):

```
val duration1: concurrent.duration.Duration = 1004 milliseconds
val duration2: concurrent.duration.Duration = 1002 milliseconds
```

Let's understand what really happens when `FutureCF.apply` is called. First, I need to explain a concept called *partial application* of argument lists.

Applying some but not all arguments for the parameters of a function or method is *partial application*. For example, for a method `def m(a: String)(b: Int)`, if I call `m("hello")` without the second parameter list, a new function is returned that can be called with the remaining parameters. In Scala all the arguments for a particular parameter list have to be provided, but there is no limit to the number of parameter lists you can have. You can partially apply as many as you want at a time, working from the left. You can't skip over parameter lists. The same mechanism happens here, with slightly different details.

First, for `future1`, when `FutureCF.apply(sleepN(1.second))` is called, the following sequence happens:

1. `Executable(Future(sleepN(1.second)))` is supposed to be returned, which is the same as `(given ExecutionContext) ?=> Future(sleepN(1.second))` (from the type member alias for `Executable`).
2. The compiler converts `Executable(Future(sleepN(1.second)))` to `Future(sleepN(1.second))(given ExecutionContext)`.
3. Then, it invokes the converted term to return the `Future`.

The same given `ExecutionContext` is passed implicitly to `Future.apply()`, which I used to implement `FutureCF.apply()`.

The only difference for `future2` is that the `ExecutionContext` is provided explicitly, but the effect is the same: `(given ExecutionContext) ?=> Future(...)`.

Context functions can be used to replace a common Scala 2 idiom, where parameters to function literals are sometimes declared `implicit`. Consider the following example that provides a convenient way to run simple code blocks asynchronously:

```
// src/script/scala/progscala3/contexts/ImplicitParams2ContextFunctions.scala

import scala.concurrent.{Await, ExecutionContext, Future}
import scala.concurrent.duration.*

val sameThreadExecutionContext = new ExecutionContext:          ❶
  def execute(runnable: Runnable): Unit =
    printf("start > ")
    runnable.run()
    printf("finish > ")
  def reportFailure(cause: Throwable): Unit =
    println(s"sameThreadExecutionContext failure: $cause")

object AsyncRunner2:
  def apply[T](body: ExecutionContext => Future[T]): T =       ❷
    val future = body(sameThreadExecutionContext)
    Await.result(future, 2.seconds)

val result2 = AsyncRunner2 {                                     ❸
  implicit executionContext =>
    Future(1).map(_ * 2).filter(_ > 0)
}
```

❶ Create a simple `ExecutionContext` that just runs tasks in the same thread. It is used to demonstrate replacing the use of the `global` implicit value with one we control in scoped contexts.

❷ The Scala 2 way of writing this logic. The user passes a function that takes an `ExecutionContext` argument and returns a `Future`. `AsyncRunner2.apply()` calls the function, passing our custom `ExecutionContext`, then waits up to two seconds for the results (arbitrary).

❸ How it is used. The function takes a normal `ExecutionContext`, but if you add the `implicit` keyword, it becomes an implicit value that will be passed to all the `Future` methods called inside the function that take an implicit `ExecutionContext`.

The value for `result2` will be the integer `2`.

In Scala 3, this idiom can still be used (obviously, because I just did!), but you can't replace the `implicit` keyword in the function literal with `using`. Instead, context functions are the new way to implement this scenario:

```
object AsyncRunner3:
  type RunnerContext[T] = ExecutionContext ?=> Future[T]
```

```
  def apply[T](body: => RunnerContext[T]): T =                    ❶
    given ExecutionContext = sameThreadExecutionContext
    Await.result(body, 2.seconds)

val result3 = AsyncRunner3 {                                        ❷
  Future(1).map(_ * 2).filter(_ > 0)
}
```

❶ Now a by-name parameter of type `RunnerContext[T]`, aliased to `ExecutionContext ?=> Future[T]`, is passed as the body to execute. A given `ExecutionContext` is declared in this scope, *aliased* to `sameThreadExecutionContext` (recall the discussion in "Alias Givens" on page 150).

❷ Now the user code is more concise. The required context function argument for `AsyncRunner3.apply()` is passed implicitly, so all we need is the `Future` body.

So context functions can result in more concise code, both for library implementers and users of those libraries. However, it takes a bit more work at first to understand what's going on.

The code examples contain a more extensive example where context functions are used to build a mini-DSL for constructing JSON objects. See *src/main/scala/progscala3/contexts/json/JSONBuilder.scala*.

Constraining Allowed Instances

Sometimes a context bound is used as a *witness*, by which I mean that the mere existence of a context bound is all we care about, but the instance is not actually needed to do any work.

Let's see an example of context-bound instances that witness allowed argument types and are used to do work. Consider the following sketch of an API for data records with ad hoc schemas, like in some NoSQL databases. Each row is encapsulated in a `Map[String,Any]`, where the keys are the field names and the column values are unconstrained. However, the `add` and `get` methods, for adding column values to a row and retrieving them, do constrain the allowed instance types.

Here is the example:

```
// src/main/scala/progscala3/contexts/NoSQLRecords.scala
package progscala3.contexts.scaladb

import scala.language.implicitConversions
import scala.util.Try

case class InvalidFieldName(name: String)
  extends RuntimeException(s"Invalid field name $name")
```

```scala
object Record:                                                              ❶
  def make: Record = new Record(Map.empty)
  type Conv[T] = Conversion[Any,T]

case class Record private (contents: Map[String,Any]):                      ❷
  import Record.Conv
  def add[T : Conv](nameValue: (String, T)): Record =                       ❸
    Record(contents + nameValue)
  def get[T : Conv](colName: String): Try[T] =                              ❹
    Try {
      val conv = summon[Conv[T]]
      conv(col(colName))
    }
  private def col(colName: String): Any =
    contents.getOrElse(colName, throw InvalidFieldName(colName))

@main def TryScalaDB =
  import Record.Conv
  given Conv[Int] = _.asInstanceOf[Int]                                     ❺
  given Conv[Double] = _.asInstanceOf[Double]
  given Conv[String] = _.asInstanceOf[String]
  given ab[A : Conv, B : Conv]: Conv[(A, B)] = _.asInstanceOf[(A,B)]

  val rec = Record.make.add("one" -> 1).add("two" -> 2.2)
    .add("three" -> "THREE!").add("four" -> (4.4, "four"))
    .add("five" -> (5, ("five", 5.5)))

  val one   = rec.get[Int]("one")
  val two   = rec.get[Double]("two")
  val three = rec.get[String]("three")
  val four  = rec.get[(Double, String)]("four")
  val five  = rec.get[(Int, (String, Double))]("five")
  val bad1  = rec.get[String]("two")                                        ❻
  val bad2  = rec.get[String]("five")
  val bad3  = rec.get[Double]("five")
  // val error  = rec.get[Byte]("byte")

  println(
    s"one, two, three, four, five ->\n  $one, $two, $three, $four,\n  $five")
  println(
    s"bad1, bad2, bad3 ->\n  $bad1\n  $bad2\n  $bad3")
```

❶ The companion object defines `make` to start safe construction of a `Record`. It also defines a type member alias for `Conversion`, where we always use `Any` as the first type parameter. This alias is necessary when we define the given `ab` inside the method `TryScalaDB`.

❷ Define `Record` with a single field `Map[String,Any]` to hold the user-defined fields and values. Use of `private` after the type name declares the constructor private, forcing users to create records using `Record.make` followed by `add` calls. This prevents users from using an unconstrained `Map` to construct a `Record`!

❸ A method to add a field with a particular type and value. The context bound, `Conv[T]`, is used only used as a witness to constrain the allowed values for `T`. Its `apply` method won't be used. Since `Record`s are immutable, a new instance is returned.

❹ A method to retrieve a field value with the desired type `T`. Here the context bound both constrains the allowed `T` types and handles conversion from `Any` to `T`. On failure, an exception is returned in the `Try`. Hence, this example can't catch all type errors at compile time, as shown in the "bad" examples.

❺ Only `Int`, `Double`, `String`, and pairs of them are supported. These definitions work as witnesses for the allowed types in both the `add` and `get` methods, as well as function as implicit conversions from `Any` to specific types when used in `get`. Note that the given `ab` is for pairs, but the `A` and `B` types are themselves constrained by `Conv`, which could also be other pairs. Hence, nested pairs are allowed.

❻ `Failure[ClassCastException]`s are returned for `bad1`, `bad2`, and `bad3` because we attempt to return a `String` or `Double` when the underlying values have incompatible types.

Recall that the context bound can be written several ways. For `add`, the following are equivalent:

```
def add[T : Conv](nameValue: (String, T)): Record = ...
def add[T](nameValue: (String, T))(using Conv[T]): Record = ...
```

Attempting to retrieve an unsupported column, like `Byte`, would cause a compilation error.

Running this example with `runMain progscala3.contexts.scaladb.TryScalaDB`, you get the following output (abbreviated):

```
one, two, three, four, five ->
  Success(1), Success(2.2), Success(THREE!), Success((4.4,four)),
  Success((5,(five,5.5)))
bad1, bad2, bad3 ->
  Failure(... java.lang.Double cannot be cast to class java.lang.String ...)
  Failure(... scala.Tuple2 cannot be cast to class java.lang.String ...)
  Failure(... scala.Tuple2 cannot be cast to class java.lang.Double ...)
```

Hence, the only failures we can't prevent at compile time are attempts to retrieve a column using the incorrect type.

The type member `Conv[T]` is necessary for the context bounds on `A` and `B` in `ab` because context bounds always require one and only one type parameter, but `Conversion[A,B]` has two. Fortunately, the `A` is always `Any` in our case, so we were able to define a type alias that has just one type parameter, as required. It also makes the code more concise than using `Conversion[Any,T]`.

> `Conversion[A,B]` didn't meet our needs for a parameterized type with only one type parameter. We solved the problem with a type alias that fixed `A` (to `Any`), leaving a single type parameter. This trick is frequently useful.[2]

To recap, we limited (witnessed) the allowed types that can be passed to a parameterized method by passing a context bound and only defining given values for the types we wanted to allow.

Implicit Evidence

In the previous example, the `Record.add` method showed one flexible way to constrain the allowed types. The witnesses used also did some work for us. Now we'll discuss a special kind of witness historically called *implicit evidence*, which uses some convenient features in the standard library.

A nice example of this technique is the `toMap` method available for all iterable collections. Recall that the `Map` constructor wants key-value pairs (i.e., two-element tuples) as arguments. If we have a `Seq[(A,B)]`, a sequence of pairs, wouldn't it be nice to create a `Map` out of them in one step? That's what `toMap` does, but we have a dilemma. We can't allow the user to call `Seq[X].toMap` if the `X` is not `(A,B)`.

The implementation of `toMap` constrains the allowed types. It is defined in `IterableOnceOps`:

```
trait IterableOnceOps[+A]:
  def toMap[K, V](implicit ev: <:<[A, (K, V)]): immutable.Map[K, V]
  ...
```

The implicit parameter `ev` is the evidence we need to enforce our constraint. It uses a type defined in `Predef` called `<:<`, named to resemble the type parameter constraint `<:`. Effectively, it imposes the requirement that `A <: (K,V)`. In other words, `A` must be a subtype of `(K,V)`.

2 For a more general solution, see "Type Lambdas" on page 391.

Recall that types with two type parameters can be written with infix notation. So the following two expressions are equivalent:

```
<:<[A, (T,U)]
A <:< (T,U)
```

So when we have a traversable collection that we want to convert to a `Map` with `toMap`, the implicit evidence `ev` value will be synthesized by the compiler, but only if `A <: (T,U)`. If `A` is not a pair type, the code fails to compile. If successful, `toMap` passes the pairs to the `Map` constructor.

Hence, evidence only has to exist to enforce a type constraint, which the compiler generates for us. We don't have to define a `given` or `implicit` value ourselves.

There is also a related type in `Predef` for providing evidence that two types are equivalent, called `=:=`. Here are a few simple examples of how you can do some simple type checking in the REPL:

```
// src/script/scala/progscala3/contexts/ImplicitEvidence.scala

summon[Int <:< Int]
summon[Int <:< AnyVal]
summon[Int =:= Int]
summon[Int =:= AnyVal]                          // ERROR!

summon[(Int, String) <:< (Int, String)]
summon[(Int, String) <:< (AnyVal, AnyRef)]
summon[(Int, String) =:= (Int, String)]
summon[(Int, String) =:= (AnyVal, AnyRef)]      // ERROR!
```

The two examples marked with `ERROR!` trigger compilation errors: "Cannot prove that…"

Working Around Type Erasure with Using Clauses

We discussed type erasure in "Defining Operators" on page 71 and how to work around it using `@targetName`. We can also work around erasure with a using clause. This works with Scala 2 implicits, as well. We used the following example previously:

```
// src/script/scala/progscala3/rounding/TypeErasureProblem.scala

object O:
  def m(is: Seq[Int]): Int = is.sum
  def m(ss: Seq[String]): Int = ss.length
```

However, we can add an implicit parameter to disambiguate the methods:

```
// src/script/scala/progscala3/contexts/UsingTypeErasureWorkaround.scala

object O2:
  trait Marker[T]                                 ❶
```

```
given IntMarker: Marker[Int] with {}
given StringMarker: Marker[String] with {}

def m(is: Seq[Int])(using IntMarker.type): Int = is.sum        ❷
def m(ss: Seq[String])(using StringMarker.type): Int = ss.length
```

❶ Define a marker trait and two named givens that will be used to disambiguate the methods affected by type erasure, for `Int`s and `String`s.

❷ Redefine the two methods to add using clauses with the markers. Because `IntMarker` is an `object`, its type is `IntMarker.type`.

The `Marker`s are very similar to the witnesses we discussed earlier. Let's try it:

```
scala> import O2.{given, *}

scala> m(Seq(1,2,3))
     | m(Seq("one", "two", "three"))
val res0: Int = 6
val res1: Int = 3
```

Now the compiler considers the two `m` methods to be distinct after type erasure.

Why do the given instances have to be named? If the using clauses had `using Marker[Int]`, for example, I would still have the type erasure problem in the using clauses, and the compiler would reject the definitions! So the instances have to be used as shown.

You might also wonder why I didn't just use given `Int` and `String` values, rather than invent the `Marker` type. Using given values for very common types is not recommended. It would be too easy for one or more given `String` values, for example, to show up in a particular scope. If you don't expect a given `String` instance to be in scope, you will be surprised when it gets used. If you do expect one to be in scope, but there are several of them, you'll get a compiler error because of the ambiguous choices.

> Avoid given instances and using clauses for very common types like `Int` and `String`, as they are more likely to cause confusing behavior or compilation errors. Stick with types created specifically for this purpose.

Rules for Using Clauses

Here are the general rules for using clauses:

1. Zero or more argument lists can be using clauses.

2. The `implicit` or `using` keyword must appear first and only once in the parameter list, and all the parameters are context parameters.

Hence, any one parameter list can't mix context parameters with other parameters. Here are a few more examples:

```
// src/script/scala/progscala3/contexts/UsingClausesLists.scala

case class U1[+T](t: T)
case class U2[+T](t: T)

def f1[T1,T2](name: String)(using u1: U1[T1], u2: U2[T2]): String =        ❶
  s"f1: $name: $u1, $u2"
def f2[T1,T2](name: String)(using u1: U1[T1])(using u2: U2[T2]): String = ❷
  s"f2: $name: $u1, $u2"
def f3[T1,T2](name: String)(using u1: U1[T1])(u2: U2[T2]): String =        ❸
  s"f3: $name: $u1, $u2"

given u1i: U1[Int] = U1[Int](0)
given u2s: U2[String] = U2[String]("one")
```

❶ One using clause with two parameters.

❷ Two using clauses, each with one parameter.

❸ One using clause sandwiched between two regular parameter lists.

In `f3`, we have a regular parameter list following a using clause. This is allowed in Scala 3, but not in Scala 2. Let's try them:

```
scala> f1("f1a")                        // These two are essentially the same.
     | f1("f1b")(using u1i, u2s)
val res0: String = f1: f1a: U1(0), U2(one)
val res1: String = f1: f1b: U1(0), U2(one)

scala> f2("f2a")                        // These two are essentially the same.
     | f2("f2b")(using u1i)(using u2s)
val res2: String = f2: f2a: U1(0), U2(one)
val res3: String = f2: f2b: U1(0), U2(one)
```

The results for calling `f1` and `f2` should make sense. Recall that when passing values explicitly, the `using` keyword is required. Now try `f3`:

```
scala> f3("f3c")(using u1i)(u2s)        // These two are essentially the same.
     | f3("f3c")(u2s)
val res4: String = f3: f3c: U1(0), U2(one)
val res5: String = f3: f3c: U1(0), U2(one)
```

The first call passes all parameter lists explicitly, while the second one fills in the second parameter list implicitly and interprets `(u2s)` as the third parameter list.

I don't recommend putting a using clause between other parameter lists because it looks strange, but it is allowed.

Improving Error Messages

Finally, you can improve the errors reported by the compiler when a context parameter isn't found in scope. The compiler's default messages are usually sufficiently descriptive, but you can customize them with the `implicitNotFound` annotation,[3] as follows:

```
// src/script/scala/progscala3/contexts/ImplicitNotFound.scala

import scala.annotation.implicitNotFound

@implicitNotFound("No implicit found: Tagify[${T}]")
trait Tagify[T]:
  def toTag(t: T): String

case class Stringer[T : Tagify](t: T):
  override def toString: String =
    s"Stringer: ${summon[Tagify[T]].toTag(t)}"

object O:
  def makeXML[T](t: T)(
      using @implicitNotFound("makeXML: No Tagify[${T}] implicit found")
        tagger: Tagify[T]): String =
    s"<xml>${tagger.toTag(t)}</xml>"
```

Let's try it:

```
scala> given Tagify[Int]:
     |   def toTag(i: Int): String = s"<int>$i</int>"
     | given Tagify[String]:
     |   def toTag(s: String): String = s"<string>$s</string>"

scala> Stringer("Hello World!")
     | Stringer(100)
     | O.makeXML("Hello World!")
     | O.makeXML(100)
val res0: Stringer[String] = Stringer: <string>Hello World!</string>
val res1: Stringer[Int] = Stringer: <int>100</int>
val res2: String = <xml><string>Hello World!</string></xml>
val res3: String = <xml><int>100</int></xml>

scala> Stringer(3.14569)
     | O.makeXML(3.14569)
1 |Stringer(3.14569)
```

3 At the time of this writing, there is no `givenNotFound` or similar replacement annotation in Scala 3.

```
  |                   ^
  |                   No implicit found: Tagify[Double]
2 |O.makeXML(3.14569)
  |                    ^
  |                    makeXML: No Tagify[Double] implicit found
```

You can only annotate types intended for use as givens. This is another reason for creating custom types for context parameter uses, rather than reusing types with other purposes, like `Int`, `String`, and `Person`. You can't use this annotation with those types.

Recap and What's Next

We completed our exploration into the details of abstracting over context in Scala 2 and 3. I hope you can appreciate their power and utility, but also the need to use them wisely. Unfortunately, because the old implicit idioms are still supported for backward compatibility, at least for a while, it will be necessary to understand how to use both the old and new constructs, even though they are redundant.

Now we're ready to dive into the principles of FP. We'll start with a discussion of the core concepts and why they are important. Then we'll look at the powerful functions provided by most container types in the library. We'll see how we can use those functions to construct concise, yet powerful, programs.

CHAPTER 7

Functional Programming in Scala

> It is better to have 100 functions operate on 1 data structure than 10 functions on 10 data structures.
>
> —Alan J. Perlis

This chapter introduces functional programming (FP). Even if you have prior experience with FP in other languages, you should still skim the chapter for Scala-specific details. I'll start with an explanation of the origin and value of FP and then discuss in depth the many ways functions can be used and manipulated in Scala. I'll finish with a discussion of the power and flexibility of functional data structures and their composable operations. Chapter 18 discusses more advanced concepts in FP.

What Is Functional Programming?

Every decade or two, a major computing idea goes mainstream. The idea may have lurked in the background of academic computer science research or in obscure corners of industry, perhaps for decades. The transition to mainstream acceptance comes in response to a perceived problem for which the idea is well suited. Object-oriented programming (OOP), which was invented in the 1960s, went mainstream in the 1980s, arguably in response to the emergence of graphical user interfaces (GUIs), for which the OOP paradigm is a natural fit.

FP experienced a similar breakout over the last 15 years or so. FP is actually much older than OOP, going back to theoretical work in the 1930s! FP offers effective techniques for three major challenges that became pressing in the 2000s and remain pressing today:

- The need for pervasive concurrency, so we can scale our applications horizontally and improve their resiliency against service disruptions. Concurrent programming is now an essential skill for every developer to master.
- The need to write data-centric (e.g., big data) applications. Of course, at some level all programs are about data, but the growth of big data highlighted the importance of effective techniques for working with large data sets.
- The need to write bug-free applications. This old problem has grown more pressing as software has become more pervasive and bugs have become more potentially disruptive to society. FP gives us new tools from mathematics that move us further in the direction of *provably bug-free* programs.

Immutability eliminates the hardest problem in concurrency, coordinating access to shared, mutable state. We'll explore concurrency in Chapter 19.

The benefits of FP for data-centric applications will become apparent as we master the functional operations discussed in this and subsequent chapters. We'll explore the connection in depth in "Scala for Big Data: Apache Spark" on page 459.

Finally, embracing FP combines mathematical rigor with immutability to create programs with fewer flaws.

Scala is a mixed-paradigm language, supporting both FP and OOP. It encourages you to use both programming models to get the best of both of them.

All programming languages have functions of some sort. Functional programming is based on the rules of mathematics for the behavior of functions and values. This starting point has far-reaching implications for software.

Functions in Mathematics

In mathematics, functions have no side effects. Consider the classic function $y = sin(x)$. No matter how much work $sin(x)$ does, all the results are returned and assigned to y. No global state of any kind is modified internally by the $sin(x)$ algorithm. Also, all the data it needs to compute the value is passed in through x. Hence, we say that such a function is free of side effects, or pure.

Purity drastically simplifies the challenge of analyzing, testing, debugging, and reusing a function. You can do all these things without having to know anything about the context in which the function is invoked.

This obliviousness to the surrounding context provides *referential transparency*, which has two implications. First, you can call such a function anywhere and be confident that it will always behave the same way, independent of the calling context. Because no global state is modified, concurrent invocation of the function is also straightforward and reliable. No tricky thread-safe coding is required.

The second sense of the term is that you can substitute the value computed by an expression in place of invocations of the expression. Consider, for example, the equation $sin(pi/2) = 1.0$. A code analyzer could replace repeated calls to $sin(pi/2)$ with 1.0 with no loss of correctness, as long as *sin* is truly pure.

Conversely, a function that returns `Unit` can only perform side effects. It can only modify mutable states somewhere. A simple example is a function that just does input or output, which modifies "the world."

Note that there is a natural uniformity between values and functions, due to the way we can substitute one for the other. What about substituting functions for values, or treating functions as values?

In fact, functions are *first-class* values in FP, just like data values. You can compose functions from other functions (for example, $tan(x) = sin(x)/cos(x)$). You can assign functions to variables. You can pass functions to other functions as arguments. You can return functions as values from other functions.

A function that takes another function as a parameter or returns a function is called a *higher-order function*. In calculus, two examples of higher-order functions are derivation and integration. We pass an expression, like a function, to the derivation operation, which returns a new function, the derivative.

We've seen many examples of higher-order functions already, such as the `map` method on collections, which takes a single function parameter that is applied to each element.

Variables That Aren't

In most programming languages, variables are mutable. In FP, variables are immutable, as they are in mathematics.

This is another consequence of the mathematical orientation. In the expression $y = sin(x)$, once you pick x, then y is fixed. Similarly, values are immutable; if you increment the integer 3 by 1, you don't modify the 3 object, you create a new value to represent 4. I have been using the term *value* as a synonym for immutable instances.

Immutability is difficult to work with at first when you're not used to it. If you can't change a variable, then you can't have loop counters, you can't have objects that change their state when methods are called on them, and you can't do input and output, which changes the state of the world!

More practically, you limit the use of mutation, reserving it for specific situations and staying pure the rest of the time.

Why Aren't Input and Output Pure?

It's easy to grasp that idea that `sin(x)` is a pure function without side effects. Why are input and output considered side effects and therefore not pure? They modify the state of the world around us, such as the contents of files or what we see on the screen. They aren't referentially transparent either. Every time I call `readline` (defined in `Predef`), I get a different input. Every time I call `println` (also defined in `Predef`), I pass a different argument, but `Unit` is always returned.

This does not mean that FP is stateless. If so, it would also be useless. Instead of mutating in place, state changes are handled with new instances.

Recall this example from Chapter 2:

```scala
// src/script/scala/progscala3/typelessdomore/Factorial.scala

def factorial(i: Int): BigInt =
  def fact(i: Int, accumulator: BigInt): BigInt =
    if i <= 1 then accumulator
    else fact(i - 1, i * accumulator)

  fact(i, BigInt(1))

(0 to 5).foreach(i => println(s"$i: ${factorial(i)}"))
```

We calculate factorials using recursion. Updates to the `accumulator` are pushed on the stack. We don't modify a running value in place. At the end of the example, we mutate the world by printing the results.

Almost all the constructs we have invented in the history of programming have been attempts to manage complexity. Higher-order, pure functions are called *combinators* because they compose together very well as flexible, fine-grained building blocks for constructing larger, more complex programs.

Encapsulation is another tool for complexity management, but a mutable state often breaks it. When a mutable object is shared between modules, a change made in one of the modules is unexpected by the other modules, causing a phenomenon known as *spooky action at a distance*.

Purity simplifies designs by eliminating a lot of the *defensive* boilerplate required in object-oriented code where mutation is used freely. It's common to encapsulate access to mutable data structures because we can't risk sharing them with clients unprotected. Such accessors increase code size and the ad hoc quality of code. Copies of mutable objects are given to accessors to avoid the risk of uncontrolled mutation. All this boilerplate increases the testing and maintenance burden. It broadens the footprint of APIs, which increases the learning burden for users.

With immutable data structures, most of these problems simply vanish. We can make internal data public without fear of data loss or corruption. Encapsulation is still useful to minimize coupling and to expose coherent abstractions, but there is less fear about data access.

What about performance? If you can't mutate an object, then you must copy it when the state changes, right? Fortunately, *functional data structures* minimize the overhead of making copies by sharing the unmodified parts of the data structures between the two copies.

Another useful idea inspired by mathematics is *lazy evaluation*. We talk about the set of natural numbers, the set of prime numbers, etc., even though they are infinite. We only pay the cost of computing values when we need them. In Scala's `LazyList`, evaluation is delayed until an element is required, allowing infinite sets to be represented, like this definition of the natural numbers:

```
// src/script/scala/progscala3/fp/datastructs/LazyListNaturals.scala

scala> val natNums = LazyList.from(0)
val natNums: LazyList[Int] = LazyList(0, 1, 2, ... 999, <not computed>)

scala> natNums.take(100).toList        ❶
val res0: List[Int] = List(0, 1, 2, ..., 99)
```

❶ Take the first (100) elements, returning another `LazyList`, then convert to a regular `List`.

Scala uses eager or strict evaluation by default, but lazy evaluation avoids work that isn't necessary now and may never be necessary. A `LazyList` can be used for processing a very large stream of incoming data, yielding results as values become available, rather than waiting until all the data has been received.

So why isn't Scala lazy by default? There are many scenarios where lazy evaluation is less efficient and it is harder to predict the performance of lazy evaluation. Hence, most functional languages use eager evaluation, but most also provide lazy data structures for when laziness is needed.

The rest of this chapter covers the essentials that every new Scala programmer needs to know. Functional programming is a large and rich field. In Chapter 18, we'll cover some of the more advanced topics that are less essential for people new to FP.

Functional Programming in Scala

As a hybrid object-functional language, Scala does not require functions to be pure, nor does it require variables to be immutable. It does encourage you to write your code this way whenever possible.

Let's quickly recap a few things we've seen already.

Here are several higher-order functions that we compose together to iterate through a list of integers, filter for the even ones, map each one to its value multiplied by two, and finally multiply them together using `reduce`:

```
// src/script/scala/progscala3/fp/basics/HOFsComposition.scala

val result = (1 to 10).filter(_ % 2 == 0).map(_ * 2).reduce(_ * _)
assert(result == 122880)
```

Recall that `_ % 2 == 0`, `_ * 2`, and `_ * _` are function literals. The first two functions take a single parameter assigned to the placeholder `_`. The last function, which is passed to `reduce`, takes two parameters.

The `reduce` method is new for us. It's used here to multiply all the elements together, two pairs of numbers at a time. That is, it reduces the collection of integers to a single value. As `reduce` works through the collection, it could process the values from the left, from the right, or as parallel trees. This choice is unspecified for `reduce`. By implication, the function passed to `reduce` must be associative, like multiplication or addition of integers ($(a + b) + c = a + (b + c)$), because we are not guaranteed that the collection elements will be processed in a particular order.

Back to our example, we use the `_` placeholder for both parameters, `_ * _` is equivalent to `(x,y) => x * y` for the function passed to `reduce`.

So, with a single line of code, we successfully looped through the list without the use of a mutable counter to track iterations, nor did we require mutable accumulators for the reduction as it was performed.

Anonymous Functions, Lambdas, and Closures

Consider the following modifications of the previous example:

```
// src/script/scala/progscala3/fp/basics/HOFsClosures.scala

scala> var factor = 2
var factor: Int = 2

scala> val multiplier = (i: Int) => i * factor
val multiplier: Int => Int = Lambda$...

scala> val result1 =
     |   (1 to 10).filter(_ % 2 == 0).map(multiplier).reduce(_ * _)
val result1: Int = 122880

scala> factor = 3
     | val result2 =
     |   (1 to 10).filter(_ % 2 == 0).map(multiplier).reduce(_ * _)
```

```
factor: Int = 3
val result2: Int = 933120
```

We define a variable, `factor`, to use as the multiplication factor, and we extract the previous anonymous function `_ * 2` into a function `val` called `multiplier` that uses `factor`.

Running the same expression with different values for `factor` changes the results. Even though `multiplier` was an immutable function value, its behavior changes when `factor` changes.

Of the two variables in `multiplier`, `i` and `factor`, `i` is called a *formal parameter* to the function. It is bound to a new value each time `multiplier` is called.

Conversely, `factor` is not a formal parameter but a *free variable*, a reference to a variable in the enclosing scope. Hence, the compiler creates a *closure* that encompasses (or closes over) `multiplier` and the external context of the unbound variables that `multiplier` references, thereby binding those variables as well.

If a function has no external references, it is trivially closed over itself. No external context is required.

This works even if `factor` is a local variable in some scope, like a method, and we passed `multiplier` to another scope, like another method. The free variable `factor` would be carried along for the ride.

This is illustrated in the following refactoring of the example, where `mult` returns a function of type `Int => Int`. That function references the local variable `factor` value, which goes out of scope once `mult` returns. That's OK because the `2` is captured in the returned function:

```
scala> def mult: Int => Int =
     |   val factor = 2
     |   (i: Int) => i * factor
def mult: Int => Int

scala> val result3= (1 to 10).filter(_ % 2 == 0).map(mult).reduce(_ * _)
val result3: Int = 122880
```

There are a few partially overlapping terms that are used a lot:

Function
: An operation that is named or anonymous. Its code is not evaluated until the function is called. It may or may not have free (unbound) variables in its definition.

Lambda
: An anonymous (unnamed) function. It may or may not have free (unbound) variables in its definition.

Closure
: A function, anonymous or named, that closes over its environment to bind variables in scope to free variables within the function.

> ### Why the Term Lambda?
>
> The term *lambda* for anonymous functions originated in *lambda calculus*, where the Greek letter lambda (λ) is used to represent anonymous functions. First studied by Alonzo Church in lambda calculus, his research in the mathematics of computability theory formalizes the properties of functions as abstractions of calculations. Functions can be evaluated or applied when we bind values (or expressions) for the function parameters. (The term *applied* here is the origin of the default method name `apply` we've already seen.) Lambda calculus also defines rules for simplifying expressions, variable substitution, etc.

Different programming languages use these and other terms to mean slightly different things. In Scala, we typically just say *anonymous function* or *function literal* for lambdas. Java and Python use the term *lambda*. Also, we don't distinguish closures from other functions unless it's important for the discussion.

> I discussed the concept of free variables like `factor` in `multiplier`, so you'll understand this capability. However, `factor` is really an example of a shared mutable state, so avoid doing this unless you have a good reason for it! You won't see any more examples like this in the book.

Methods as Functions

While discussing variable capture in the preceding section, we defined the function `multiplier` as a value:

```
val multiplier = (i: Int) => i * factor
```

However, you can also use a method:

```scala
// src/script/scala/progscala3/fp/basics/HOFsClosures2.scala

var factor2 = 2
def multiplier2(i: Int) = i * factor2

val result3 =
  (1 to 10).filter(_ % 2 == 0).map(multiplier2).reduce(_ * _)
assert(result3 == 122880)

factor2 = 3
val result4 =
  (1 to 10).filter(_ % 2 == 0).map(multiplier2).reduce(_ * _)
assert(result4 == 933120)
```

Now `multiplier2` is a method like the function `multiplier`. However, we can use `multiplier2` just like a function because it doesn't reference `this`.[1] When a method is used where a function is required, Scala lifts the method to be a function. I've been using *function* to refer to methods or functions generically. This was not wrong!

Purity Inside Versus Outside

If we called *sin*(*x*) thousands of times with the same value of *x*, it would be wasteful if it performed the calculation every single time.[2] Even in pure functional libraries, it is acceptable to perform internal optimizations like caching previously computed values. This is called *memoization*.

Caching is a side effect, as the cache has to be modified, of course. If the performance benefits are worth it and caching is implemented in a thread-safe way, fully encapsulated from the user, then the function is effectively referentially transparent.

Recursion

Recursion plays a larger role in FP than in imperative programming. Recursion is the pure way to implement looping without mutable loop counters.

Calculating factorials provides a good example, which we saw in "Nesting Method Definitions and Recursion" on page 45:

```scala
// src/script/scala/progscala3/typelessdomore/FactorialTailrec.scala
import scala.annotation.tailrec

def factorial(i: Int): BigInt =
  @tailrec
  def fact(i: Int, accumulator: BigInt): BigInt =
```

1 Even in the REPL, the compiler has an internal object it uses to hold method and other member definitions, for JVM compatibility.

2 I'm ignoring the tricky fact that comparing floating points numbers for equality is fraught with peril.

```
      if i <= 1 then accumulator
      else fact(i - 1, i * accumulator)

    fact(i, BigInt(1))

  (0 to 5).foreach(i => println(s"$i: ${factorial(i)}"))
```

There are no mutable variables, and the implementation is tail recursive.

Tail Calls and Tail-Call Optimization

Tail-call self-recursion is the best kind of recursion because the compiler can optimize it into a loop (see "Nesting Method Definitions and Recursion" on page 45). This eliminates the function call for each iteration, thereby improving performance and eliminating the potential for a stack overflow, while still letting us use the purity of recursion in source code. You should use the `@tailrec` annotation to trigger a compilation error if the annotated method is not actually tail recursive.

> The tail-call optimization won't be applied when the method can be overridden in a derived type. Hence, the recursive method must be declared `private` or `final`, or it must be defined inside another method.

A *trampoline* is a loop that works through a list of functions, calling each one in turn. Its main purpose is to avoid stack overflow situations. The metaphor of bouncing the functions back and forth off a trampoline is the source of the name.

Consider a *mutual recursion* where a function `f1` doesn't call itself recursively, but instead it calls another function `f2`, which then calls `f1`, which calls `f2`, etc. This is obviously not self-recursion, but it can also be converted into a loop using a *trampoline* data structure. The Scala library has a `scala.util.control.TailCalls` object for this purpose.

The following example defines an inefficient way of determining if a number is even or odd (adapted from the `TailCalls` Scaladoc):

```
// src/script/scala/progscala3/fp/recursion/Trampoline.scala

scala> import scala.util.control.TailCalls.*
     |
     | def isEven(xs: Seq[Int]): TailRec[Boolean] =
     |   if xs.isEmpty then done(true) else tailcall(isOdd(xs.tail))
     |
     | def isOdd(xs: Seq[Int]): TailRec[Boolean] =
     |   if xs.isEmpty then done(false) else tailcall(isEven(xs.tail))
     |
     | val eo = (1 to 5).map(i => (i, isEven(1 to i).result))
```

```
...
val eo: IndexedSeq[(Int, Boolean)] =
  Vector((1,false), (2,true), (3,false), (4,true), (5,false))
```

The code bounces back and forth between `isOdd` and `isEven` for each list element until the end of the list. If it hits the end of the list while it's in `isEven`, it returns `true`. If it's in `isOdd`, it returns `false`.

Partially Applied Functions Versus Partial Functions

We learned in "Context Functions" on page 172 that applying some, but not all argument lists for a function is called *partial application*, where a new function is returned that can be called with the remaining parameter lists.

All the arguments for a single parameter list have to be provided, but there is no limit to the number of parameter lists you can have, and you can partially apply as many of the lists at a time as you want. You have to work from left to right, though. You can't "skip over" parameter lists. Consider the following session where we define and use different string concatenation methods:

```
// src/script/scala/progscala3/fp/basics/PartialApplication.scala

scala> def cat1(s1: String)(s2: String) = s1 + s2
     | def cat2(s1: String) = (s2: String) => s1 + s2
def cat1(s1: String)(s2: String): String
def cat2(s1: String): String => String

scala> cat1("hello")("world")      // Call with both parameter lists.
val res0: String = helloworld

scala> val fcat1 = cat1("hello")   // One applied argument list
val fcat1: String => String = Lambda$...

scala> fcat1("world")              // Second argument list applied
val res2: String = helloworld

scala> cat2("hello")("world")      // Same usage as cat1!
val res3: String = helloworld

// ... Same results using cat2 instead of cat1 ...
```

When used, it appears that `cat1` and `cat2` are the same, but while `cat1` has two parameter lists, `cat2` has one parameter list, but it returns a function that takes the equivalent of the second parameter list.

3 Note the function type for `fcat1`. This conversion from partially applied methods to functions is called *automatic eta expansion* (from *lambda calculus*). In Scala 2, you have to append an underscore, `val fcat1 = cat1("hello") _` to trigger eta expansion.

Let's look at a function definition that is equivalent to `cat2`:

```
scala> val cat2F = (s1: String) => (s2: String) => s1+s2
val cat2F: String => String => String = Lambda$...
```

It takes a bit of practice to read signatures like this. Here is the same function with the type signature added explicitly (which means now we don't need the types on the righthand side):

```
scala> val cat2F: String => String => String = s1 => s2 => s1+s2
     | (s1: String) => (s2: String) => s1+s2
val cat2F: String => String => String = Lambda$...
```

The type binding is right to left, which means that arguments are applied left first. To see this, note that the definition is equivalent to the following, which uses parentheses to emphasize that precedence:

```
scala> val cat2Fc: String => (String => String) = s1 => s2 => s1+s2
val cat2Fc: String => String => String = Lambda$...
```

Again, if we pass a string to `cat2Fc`, it returns a function that takes another string and returns a final string. The REPL prints all three function types as `String => String => String`.

We can use all three functions exactly the way we used `cat2` previously. I'll let you try that yourself.

A *partially applied function* is an expression with some but not all of a function's parameter lists applied, returning a new function that takes the remaining parameter lists.

In contrast, a *partial function* is a single-parameter function that is not defined for all values of the type of its parameter. The literal syntax for a partial function is one or more `case` clauses (see "Partial Functions" on page 36).

Currying and Uncurrying Functions

Methods and functions with multiple parameter lists have a fundamental property called *currying*, which is named after the mathematician Haskell Curry (for whom the Haskell language is named). Actually, Curry's work was based on an original idea of Moses Schönfinkel, but *Schönfinkeling* or maybe *Schönfinkelization* never caught on.

Currying is the transformation of a function that takes multiple parameters into a chain of functions, each taking a single parameter. Scala provides ways to convert between curried and uncurried functions:

```
scala> def mcat(s1: String, s2: String) = s1 + s2
     | val mcatCurried = mcat.curried
def mcat(s1: String, s2: String): String
val mcatCurried: String => String => String = scala.Function2$$Lambda$...

scala> val fcat = (s1: String, s2: String) => s1 + s2
     | val fcatCurried = fcat.curried
val fcat: (String, String) => String = Lambda$...
val fcatCurried: String => String => String = scala.Function2$$Lambda$...

scala> mcat("hello", "world")
     | fcat("hello", "world")
     | mcatCurried("hello")("world")
     | fcatCurried("hello")("world")
val res0: String = helloworld
...     // Same result for the other three.
```

Whether we start with a method or function, `curried` returns a function that is a subtype of a Scala trait `scala.Function2` (2 for the number of parameters).

We can also uncurry a function or method:

```
scala> def mcat2(s1: String) = (s2: String) => s1 + s2  // "Curried" method
def mcat2(s1: String): String => String

scala> val mcat2Uncurried = Function.uncurried(mcat2)
     | val mcatUncurried = Function.uncurried(mcatCurried)
val mcat2Uncurried: (String, String) => String = scala.Function$$$Lambda$...
val mcatUncurried: (String, String) => String = scala.Function$$$Lambda$...
```

A practical use for currying and partial application is to specialize functions for particular types of data. For example, suppose we always pass `"hello"` as the first argument to `mcat`:

```
scala> val hcat = mcat.curried("hello")
     | val uni = hcat("universe!")
val hcat: String => String = scala.Function2$$Lambda$...
val uni: String = hellouniverse!
```

Tupled and Untupled Functions

One scenario you'll encounter occasionally is when you have data in a tuple, let's say an *N*-element tuple, and you need to call an *N*-parameter function:

```
// src/script/scala/progscala3/fp/basics/Tupling.scala

scala> def mult(d1: Double, d2: Double) = d1 * d2
scala> val d23 = (2.2, 3.3)
```

```
     | val d = mult(d23._1, d23._2)
val d23: (Double, Double) = (2.2,3.3)
val d: Double = 7.26
```

It's tedious extracting the tuple elements like this.

Because of the literal syntax for tuples, like `(2.2, 3.3)`, there seems to be a natural symmetry between tuples and function parameter lists. We would love to have a new version of `mult` that takes the tuple itself as a single parameter. Fortunately, the `scala.Function` object provides `tupled` and `untupled` methods for us. There is also a `tupled` method available for methods like `mult`:

```
scala> val multTup1 = Function.tupled(mult) // Scala 2: Function.tuples(mult _)
     | val multTup2 = mult.tupled           // Scala 2:(mult _).tupled
val multTup1: ((Double, Double)) => Double = scala.Function...
val multTup2: ((Double, Double)) => Double = scala.Function2...

scala> val d2 = multTup1(d23)
     | val d3 = multTup2(d23)
val d2: Double = 7.26
val d3: Double = 7.26
```

3 The comments show that Scala 2 required the _ when using the two `tupled` methods.

There is a `Function.untupled`:

```
scala> val mult2 = Function.untupled(multTup2)  // Go back...
     | val d4 = mult2(d23._1, d23._2)
val mult2: (Double, Double) => Double = scala.Function$$$Lambda$...
val d4: Double = 7.26
```

However, there isn't a corresponding `multTup2.untupled` available. Also, `Function.tupled` and `Function.untupled` only work for *arities* between two and five, inclusive, an arbitrary limitation. Above arity five, you can call `myfunc.tupled` up to arity 22.

Partial Functions Versus Functions Returning Options

In "Partial Functions" on page 36, we discussed the synergy between partial functions and total functions that return an `Option`, either `Some(value)` or `None`, corresponding to the case where the partial function can return a value and when it can't, respectively. Scala provides transformations between partial functions and total functions returning options:

```
// src/script/scala/progscala3/fp/basics/PartialFuncOption.scala

scala> val finicky: PartialFunction[String,String] =
     |   case "finicky" => "FINICKY"
val finicky: PartialFunction[String, String] = <function1>
```

```
scala> finicky("finicky")
val res0: String = FINICKY

scala> finicky("other")
scala.MatchError: other (of class java.lang.String)
  at scala.PartialFunction$$anon$1.apply(PartialFunction.scala:344)
  ...
```

Now "lift" it to a total function returning `Option[String]`:

```
scala> val finickyOption = finicky.lift
val finickyOption: String => Option[String] = <function1>

scala> finickyOption("finicky")
     | finickyOption("other")
val res1: Option[String] = Some(FINICKY)
val res2: Option[String] = None
```

We can go from a total function returning an option using `unlift`:

```
scala> val finicky2 = Function.unlift(finickyOption)
val finicky2: PartialFunction[String, String] = <function1>

scala> finicky("finicky")
val res3: String = FINICKY

scala> finicky("other")
scala.MatchError: other (of class java.lang.String)
  at scala.PartialFunction$$anon$1.apply(PartialFunction.scala:344)
  ...
```

Note that `unlift` only works on single parameter functions returning an option.

Lifting a partial function is especially useful when we would prefer to handle optional values instead of dealing with thrown `MatchErrors`. Conversely, unlifting is useful when we want to use a regular function returning an option in a context where we need a partial function.

Functional Data Structures

Functional programming emphasizes the use of a core set of data structures and algorithms that are maintained separately from the data structures. This enables algorithms to be added, changed, or even removed without having to edit the data structure source code. As we'll see, this approach leads to flexible and composable libraries, leading to concise applications.

The minimum set of data structures includes sequential collections, like lists, vectors, and arrays, unordered maps and sets, and trees. Each collection supports a subset of the same higher-order, side effect–free functions, called *combinators*, such as `map`, `filter`, and `fold`. Once you learn these combinators, you can pick the appropriate collection to meet your requirements for data access and performance, then apply the same familiar combinators to manipulate that data. These collections are the most successful tools for code reuse and composition that we have in all of software development.

Let's look at a few of the most common data structures in Scala, focusing on their functional characteristics. Other details, like the organization of the library, will be discussed in Chapter 14. Unless otherwise noted, the particular collections we'll discuss are automatically in scope without requiring import statements.

Sequences

Many data structures are *sequential*, where the elements can be traversed in a predictable order, which might be the order of insertion or sorted in some way, like a priority queue. The `collection.Seq` trait is the abstraction for all mutable and immutable sequential types. Child traits `collection.mutable.Seq` and `collection.immutable.Seq` represent mutable and immutable sequences, respectively. The default `Seq` type is `immutable.Seq`. You have to import the other two if you want to use them.

When we call `Seq.apply()`, it constructs a linked `List`, the simplest concrete implementation of `Seq`. When adding an element to a list, it is prepended to the existing list, becoming the head of a new list that is returned. The existing tail list remains unchanged. `Lists` are immutable, so the tail list is unaffected by prepending elements to construct a new list.

Figure 7-1 shows two lists, `List(1,2,3,4,5)` and `List(2,3,4,5)`, where the latter is the tail of the former.

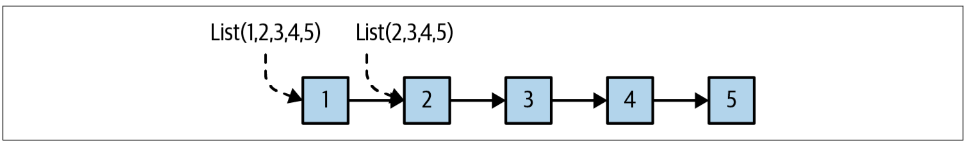


Figure 7-1. Two linked lists

Note that we created a new list from an existing list using an *O(1)* operation. (Accessing the head is also *O(1)*.) We shared the tail with the original list and just constructed a new link from the new head element to the old list. This is our first example of an important idea in functional data structures, sharing a structure to minimize the cost of making copies. To support immutability, we need the ability to make copies with minimal cost.

Any operation that requires list traversal, such as computing the size or accessing an arbitrary element—e.g., `mylist(5)`) is *O(N)*, where *N* is the `size`.

The following example demonstrates ways to construct `Lists`:

```
// src/script/scala/progscala3/fp/datastructs/Sequence.scala
scala> val seq1 = Seq("Programming", "Scala")                    ❶
     | val seq2 = "Programming" +: "Scala" +: Nil                ❷
val seq1: Seq[String] = List(Programming, Scala)
val seq2: List[String] = List(Programming, Scala)

scala> val seq3 = "People" +: "should" +: "read" +: seq1  ❸
val seq3: Seq[String] = List(People, should, read, Programming, Scala)

scala> seq3.head                                                  ❹
     | seq3.tail
val res0: String = People
val res1: Seq[String] = List(should, read, Programming, Scala)
```

❶ `Seq.apply()` takes a repeated parameters list and constructs a `List`.

❷ The `+:` method on `Seq` used in infix notation, with `Nil` as the empty list. Note the different inferred types for `seq1` and `seq2`. This is because `Nil` is of type `List[Nothing]`, so the whole sequence will be a `List` at each step.

❸ You can start with a nonempty sequence on the far righthand side.

❹ Getting the head and tail of the sequence.

We discussed the cons (for *construct*) method, `+:`, in "Operator Precedence Rules" on page 78. It binds to the right because the name ends with `:`. If we used regular method syntax, we would write `list.+:(element)`.

The case object `Nil` is a subtype of `List` that is a convenient object when an empty list is required. Note that the chain of cons operators requires a list on the far right, empty or nonempty. `Nil` is equivalent to `List.empty[Nothing]`, where `Nothing` is the subtype of *all* other types in Scala.

The construction of `seq2` is parsed as follows, with parentheses added to make the ordering of construction explicit. Each set of parentheses encloses an immutable list:

```
val seq2b = ("Programming" +: ("Scala" +: (Nil)))
```

How can we start with `List[Nothing]`, the type of `Nil`, and end up with `List[String]` after `"Scala"` is prepended to `Nil`? It is because `+:` is typed so that the new type parameter of the output `Seq` will be the least upper bound of the input elements. Since `Nothing` is a subtype of all types, including `String`, then `String` is the new least upper bound after `"Scala" +: Nil`.

There are many more methods available on `Seq` for concatenation of sequences, transforming them, etc.

What about other types that implement `Seq`? Let's consider `immutable.Vector`, which is important because random access operations are *O(log(N))*, and some operations like `head`, `tail`, `+:` (prepend), and `:+` (append) are *O(1)* to *O(log(N))*, worst case:

```
// src/script/scala/progscala3/fp/datastructs/Vector.scala
scala> val vect1 = Vector("Programming", "Scala")
     | val vect2 = "People" +: "should" +: "read" +: Vector.empty      ❶
     | val vect3 = "People" +: "should" +: "read" +: vect1
val vect1: Vector[String] = Vector(Programming, Scala)
val vect2: Vector[String] = Vector(People, should, read)
val vect3: Vector[String] = Vector(People, should, read, Programming, Scala)

scala> val vect4 = Vector.empty :+ "People" :+ "should" :+ "read"      ❷
val vect4: Vector[String] = Vector(People, should, read)

scala> vect3.head
     | vect3.tail
val res0: String = People
val res1: Vector[String] = Vector(should, read, Programming, Scala)

scala> val seq1 = Seq("Programming", "Scala")
     | val vect5 = seq1.toVector                                        ❸
val seq1: Seq[String] = List(Programming, Scala)
val vect5: Vector[String] = Vector(Programming, Scala)
```

❶ Use `Vector.empty` instead of `Nil` so that the whole sequence is constructed as a Vector from the beginning.

❷ Use the append method, `:+`. This is worst case *O(log(N))* for `Vector`, but *O(N)* for list.

❸ An alternative; take an existing sequence and convert it to a `Vector`. This is inefficient, if the `Seq` is not already a `Vector`, as a copy of the collection has to be constructed.

Maps

Another common data structure is the `Map`, used to hold pairs of keys and values, where the keys must be unique. `Map`s and the common `map` method reflect a similar concept, associating a key with a value and associating an input element with an output element, respectively:

```
// src/script/scala/progscala3/fp/datastructs/Map.scala

scala> val stateCapitals = Map(
     |   "Alabama" -> "Montgomery",
     |   "Alaska"  -> "Juneau",
     |   "Wyoming" -> "Cheyenne")
```

```scala
val stateCapitals: Map[String, String] =
  Map(Alabama -> Montgomery, Alaska -> Juneau, Wyoming -> Cheyenne)
```

The order of traversal is undefined for Map, but it may be defined by particular subtypes, such as SortedMap (import required).

You can add new key-value pairs, possibly overriding an existing definition, or add multiple key-value pairs. All of these operations return a new Map:

```scala
scala> val stateCapitals2a = stateCapitals + ("Virginia" -> "Richmond")
val stateCapitals2a: Map[String, String] = Map(..., Virginia -> Richmond)

scala> val stateCapitals2b = stateCapitals + ("Alabama" -> "MONTGOMERY")
val stateCapitals2b: Map[String, String] = Map(Alabama -> MONTGOMERY, ...)

scala> val stateCapitals2c = stateCapitals ++ Seq(
     |   "Virginia" -> "Richmond", "Illinois" -> "Springfield")
val stateCapitals2c: Map[String, String] = HashMap(
  Alaska -> Juneau, Illinois -> Springfield, Wyoming -> Cheyenne,
  Virginia -> Richmond, Alabama -> Montgomery)
```

Map is a trait, like Seq. We used the companion object method, Map.apply, to construct an instance of a concrete implementation class that is optimal for the data set, usually based on size. In fact, there are concrete map classes for one, two, three, and four key-value pairs! Why? For very small instances, it's more efficient to use custom implementations, like an array lookup, rather than use a hash-based implementation, HashMap, which is the default used for larger maps. (SortedMap uses a tree structure.)

Sets

Like Maps, Sets are unordered collections, so they aren't sequences. Like Map keys, they also enforce uniqueness among the elements they hold:

```scala
// src/script/scala/progscala3/fp/datastructs/Set.scala

scala> val states = Set("Alabama", "Alaska", "Wyoming")

scala> val states2 = states + "Virginia"
val states2: Set[String] = Set(Alabama, Alaska, Wyoming, Virginia)

scala> val states3 = states ++ Seq("New York", "Illinois", "Alaska")
val states3: Set[String] = HashSet(
  Alaska, Alabama, New York, Illinois, Wyoming)  // Alaska already present
```

Traversing, Mapping, Filtering, Folding, and Reducing

Traversing a collection is a universal operation for working with the contents. Most collections are traversable by design, like `List` and `Vector`, where the ordering is defined. `LazyList` is also traversable, but potentially infinite! Other traversable collections don't guarantee a particular order, such as hash-based `Set` and `Map` types. Other types may have a `toSeq` method, which returns a new collection that is traversable. `Option` is a collection with zero or one element, implemented by the `None` and `Some` subtypes. Even `Product`, an abstraction implemented by tuples and case classes, has the notion of iterating through its elements. Your own container types can and should be designed for traversability, when possible.

This protocol is one of the most reusable, flexible, and powerful concepts in all of programming. Scala programs use this capability extensively. In the sense that all programs boil down to "data in, data out," traversing is a superpower.

Traversing

The method for traversing a collection and performing only side effects is `foreach`. It is declared in `collection.IterableOnce`. The operations we'll discuss are defined in "mixin" traits like this. See Chapter 14 for more details.

This is the signature for `foreach`:

```
trait IterableOnce[A] {  // Some details omitted.
  ...
  def foreach[U](f: A => U): Unit
  ...
}
```

The return type of the function `U` is not important, as the output of `foreach` is always `Unit`. Hence, it can only perform side effects. This means `foreach` isn't consistent with the FP principle of writing pure, side effect–free code, but it is useful for writing output and other tasks. Because it takes a function parameter, `foreach` is a higher-order function, as are all the operations we'll discuss.

Performance is at best *O(N)* in the number of elements. Here are examples using it with our `stateCapitals` `Map`:

```
scala> var str1 = ""
     | stateCapitals.foreach { case (k, v) => str1 += s"${k}: ${v}, " }
scala> str1
val res0: String = "Alabama: Montgomery, Alaska: Juneau, Wyoming: Cheyenne, "
```

Since nothing is returned, you could say that `foreach` transforms each element into zero elements, or *one to zero* for short.

Mapping

One of the most useful operations, which we have used many times already, is `map`. Mapping is *one to one*, where for each input element an output element is returned. Hence, an invariant is the size of the input and output collections and must be equal. For any collection `C[A]` for elements of type `A`, `map` has the following logical signature:

```
class C[A]:
  def map[B](f: (A) => B): C[B]
```

The real signature is more involved, in part because of the need to construct the correct collection for the result, but the details don't concern us now. This will be true for all the combinators we explore later too. I'll simplify the signatures to focus on the concepts, then return to more of the implementation details in Chapter 14.

Note the signature for `f`. It must transform an `A` to a `B`. Usually the type `B` is inferred from this function, while `A` is known from the original collection:

```
scala> val in = Seq("one", "two", "three")
val in: Seq[String] = List(one, two, three)

scala> val out1 = in.map(_.length)
val out1: Seq[Int] = List(3, 3, 5)

scala> val out2 = in.map(s => (s, s.length))
val out2: Seq[(String, Int)] = List((one,3), (two,3), (three,5))
```

When calling `Map.map`, the anonymous function must accept a pair (two-element tuple) for each key and value:

```
// src/script/scala/progscala3/fp/datastructs/Map.scala

scala> val lengths = stateCapitals.map(kv => (kv._1, kv._2.length))
val lengths: Map[String, Int] = Map(Alabama -> 10, Alaska -> 6, Wyoming -> 8)
```

Sometimes it's more convenient to pattern match on the key and value:

```
scala> val caps = stateCapitals.map { case (k, v) => (k, v.toUpperCase) }
val caps: Map[String, String] = Map(Alabama -> MONTGOMERY, ...)
```

We can also use parameter untupling here (see "Parameter Untupling" on page 116):

```
scala> val caps = stateCapitals.map((k, v) => (k, v.toUpperCase))
val caps: Map[String, String] = Map(Alabama -> MONTGOMERY, ...)
```

Another way to think of `map` is that it transforms `Seq[A] => Seq[B]`. This fact is obscured by the object syntax—e.g., `myseq.map(f)`. If instead we had a separate module of functions that take `Seq` instances as parameters, it would look something like this:

```
// src/script/scala/progscala3/fp/combinators/MapF.scala
```

```
object MapF:                                                    ❶
  def map[A,B](f: (A) => B)(seq: Seq[A]): Seq[B] = seq.map(f)   ❷
```

❶ `MapF.map` used to avoid conflict with the built-in `Map`.

❷ A `map` that takes the transforming function as the first parameter list, then the collection. I'm cheating and using `Seq.map` to implement the transformation for simplicity.

Now try it. Note the type of the value returned from `MapF.map`:

```
scala> val intToString = (i:Int) => s"N=$i"
     | val input = Seq(1, 2, 3, 4)

scala> val ff = MapF.map(intToString)
val ff: Seq[Int] => Seq[String] = Lambda$...

scala> val seq = ff(input)
val seq: Seq[String] = List(N=1, N=2, N=3, N=4)
```

Partial application of `MapF.map` lifts a function `Int => String` into a new function `Seq[Int] => Seq[String]`, the type of `ff`. We call this new function with a collection argument and it returns a new collection.

Put another way, partial application of `MapF.map` is a transformer of functions. There is a symmetry between the idea of mapping over a collection with a function to create a new collection versus mapping a function to a new function that is able to transform one collection into another.

Flat Mapping

A generalization of the `Map` operation is `flatMap`, where we generate zero or more new elements for each element in the original collection. In other words, while `map` is *one to one*, `flatMap` is *one to many*.

Here the logical signature, with `map` for comparison, for some collection `C[A]`:

```
class C[A]:
  def flatMap[B](f: A => Seq[B]): C[B]
  def map[B](f: (A) => B): C[B]
```

We pass a function that returns a collection, instead of a single element, and `flatMap` flattens those collections into one collection, `C[B]`. It's not required for `f` to return collections of the same type as `C`.

Consider this example that compares `flatMap` and `map`. First we map over a `Range`. In the function passed to `map`, for each integer, we construct a new range from that value until 5:

```
// src/script/scala/progscala3/fp/datastructs/FlatMap.scala

scala> val seq = 0 until 5
val seq: Range = Range 0 until 5         // i.e., 0, 1, 2, 3, 4, but not 5

scala> val seq1 = seq.map(i => i until 5)
val seq1: IndexedSeq[Range] = Vector(Range 0 until 5, Range 1 until 5, ...)
```

`Vector` was used by Scala itself because of its efficient append operation. Now use another method, `flatten`:

```
scala> val seq2 = seq1.flatten
val seq2: IndexedSeq[Int] = Vector(0, 1, 2, 3, 4, 1, 2, 3, 4, 2, 3, 4, 3, 4, 4)
```

Finally, use `flatMap` instead. It effectively combines `map` followed by `flatten` (but it is more efficient):

```
scala> val seq3 = seq.flatMap(i => i until 5)
val seq3: IndexedSeq[Int] = Vector(0, 1, 2, 3, 4, 1, 2, 3, 4, 2, 3, 4, 3, 4, 4)
```

If we had nested sequences returned from the function passed to `flatMap`, they would not be flattened beyond one level.

So far, `flatMap` might not seem like a particularly useful operation, but it has far greater benefit than first appears. As a teaser for what's to come, consider the following example using `Options` where we simulate validating account information a user might provide in a form:

```
// src/script/scala/progscala3/fp/datastructs/FlatMapValidate.scala

scala> case class Account(name: String, password: String, age: Int)   ❶

scala> val validName: Account => Option[Account] =                     ❷
     |   a => if a.name.length > 0 then Some(a) else None
     |
     | val validPwd: Account => Option[Account] =
     |   a => if a.password.length > 0 then Some(a) else None
     |
     | val validAge: Account => Option[Account] =
     |   a => if a.age > 18 then Some(a) else None                     ❸

scala> val accounts = Seq(
     |   Account("bucktrends", "1234", 18),
     |   Account("", "1234", 29),
     |   Account("bucktrends", "", 29),
     |   Account("bucktrends", "1234", 29))

scala> val validated = accounts.map { account =>                       ❹
     |   Some(account).flatMap(validName).flatMap(validPwd).flatMap(validAge)
     | }
val validated: Seq[Option[Account]] =
  List(None, None, None, Some(Account(bucktrends,1234,29)))
```

❶ Define a case class for `Account` form data.

❷ Define separate validation functions for each field in an `Account` instance. Note that each has exactly the same signature, other than the name.

❸ Disallow minors.

❹ Map over the accounts, testing each one with the validators.

Note that the last account passes validation. `None` is returned for the rest.

How did this work? Take the third `Account` object, which has an invalid password. Let's assume we assigned it to a val named `acc3`. Now `Some(acc3).flatMap(validName)` succeeds, so it returns `Some(acc3)` again. Try checking this yourself in the REPL if you're not sure. Now, calling `acc3.flatMap(validPwd)` returns `None`, and all subsequent calls to `None.flatMap` will always just return `None`.

If we didn't care about the three bad ones, we could use `accounts.flatMap` instead to filter out the `None`s. Try it!

Using `Option`s and `flatMap` this way might seem like overkill. We could just call simpler validation methods that don't return `Option`. But if we made the list of validators more configurable, perhaps defined in an external library, then using this protocol lets us sequence together the separate tests without having to add logic to handle each success or failure ("if this passes, try that…"). As shown, the first three instances fail at one validator, while the last one passes all of them.

What's missing here is information about which validator failed and why for each case, which you would want to show to the user in a form. We'll see alternatives in Chapter 8 that fill in this gap but still leverage the same `flatMap` approach.

Using `flatMap` in this way is extremely common in Scala code. Any time a value is "inside a box," `flatMap` is the easy way to extract that value, do something with it, then put it back inside a new box, as long as `Box.flatMap` is defined. We'll see many more examples.

Filtering

It is common to traverse a collection and extract a new collection from it with elements that match certain criteria.

For any collection `C[A]`:

```
class C[A]:
  def filter(f: A => Boolean): C[A]
```

Hence, filtering is *one to zero or one*. For example:

```
scala> val numbers = Map("one" -> 1, "two" -> 2, "three" -> 3)

scala> val tnumbers = numbers filter { case (k, v) => k.startsWith("t") }
val tnumbers: Map[String, Int] = Map(two -> 2, three -> 3)
```

Most collections that support `filter` have a set of related methods to retrieve the subset of a collection. Note that some of these methods won't return for infinite collections, and some might return different results for different invocations unless the collection type is ordered. The descriptions are adapted from the Scaladoc:

`def drop(n: Int): C[A]`
: Return a new collection without the first `n` elements. The returned collection will be empty if this collection has less than `n` elements.

`def dropWhile (p: (A) => Boolean): C[A]`
: Drop the longest prefix of elements that satisfy the predicate `p`. The new collection returned starts with the first element that doesn't satisfy the predicate.

`def exists (p: (A) => Boolean): Boolean`
: Return true if the predicate holds for at least one of the elements of this collection. Return false, otherwise.

`def filter (p: (A) => Boolean): C[A]`
: Return a collection with all the elements that satisfy a predicate. The order of the elements is preserved.

`def filterNot (p: (A) => Boolean): C[A]`
: The negation of `filter`; select all elements that do not satisfy the predicate.

`def find (p: (A) => Boolean): Option[A]`
: Find the first element of the collection that satisfies the predicate, if any. Return an `Option` containing that first element, or `None` if no element exists satisfying the predicate.

`def forall (p: (A) => Boolean): Boolean`
: Return true if the predicate holds for all elements of the collection. Return false, otherwise.

`def partition (p: (A) => Boolean): (C[A], C[A])`
: Partition the collection into two new collections according to the predicate. Return the pair of new collections where the first one consists of all elements that satisfy the predicate and the second one consists of all elements that don't. The relative order of the elements in the resulting collections is the same as in the original collection.

`def take (n: Int): C[A]`
: Return a collection with the first `n` elements. If `n` is greater than the size of the collection, return the whole collection.

`def takeWhile (p: (A) => Boolean): C[A]`
: Take the longest prefix of elements that satisfy the predicate.

`def withFilter (p: (A) => Boolean): WithFilter[A]`
: Works just like `filter`, but it is used by `for` comprehensions to reduce the number of collection copies created (see Chapter 8).

Note that concatenating the results of `take` and `drop` yields the original collection. Same for `takeWhile` and `dropWhile`. Also, the same predicate used with `partition` would return the same two collections:

```
// src/script/scala/progscala3/fp/datastructs/FilterOthers.scala

val seq = 0 until 10
val f = (i: Int) => i < 5

for i <- 0 until 10 do
  val (l1,r1) = (seq.take(i), seq.drop(i))
  val (l2,r2) = (seq.takeWhile(f), seq.dropWhile(f))
  val (l3,r3) = seq.partition(f)
  assert(seq == l1++r1)
  assert(seq == l2++r2)
  assert(seq == l3++r3)
  assert(l2 == l3 && r3 == r3)
```

Folding and Reducing

Let's discuss folding and reducing together because they're similar. Both are operations for shrinking a collection down to a smaller collection or a single value, so they are many-to-one operations.

Folding starts with an initial seed value and processes each element in the context of that value. In contrast, reducing doesn't start with a user-supplied initial value. Rather, it uses one of the elements as the initial value, usually the first or last element:

```
// src/script/scala/progscala3/fp/datastructs/Reduce.scala

scala> val int1 = Seq(1,2,3,4,5,6).reduceLeft (_ + _)              ❶
     |
     | val int2 = Seq(1,2,3,4,5,6).foldLeft(0)(_ + _))             ❷
val int1: Int = 21
val int2: Int = 21

scala> val int3 = Seq.empty[Int].reduceLeft(_ + _))                ❸
java.lang.UnsupportedOperationException: empty.reduceLeft
...
```

```
scala> val int4 = Seq(1).reduceLeft(_ + _))              ❹
val int4: Int = 1

scala> val opt1 = Seq.empty[Int].reduceLeftOption(_ + _)  ❺
val opt1: Option[Int] = None

scala> val opt2 = Seq(1,2,3,4,5,6).reduceLeftOption(_ * _)
val opt2: Option[Int] = Some(720)
```

❶ Reduce a sequence of integers by adding them together, going left to right, returning 21.

❷ Do the same calculation with `foldLeft`, which takes a seed value, 0 in this case.

❸ An attempt to reduce an empty sequence causes an exception because there needs to be at least one element for the reduction.

❹ Having one element is sufficient. The returned value is just the single element.

❺ A safer way to reduce if you aren't sure if the collection is empty. `Some(value)` is returned if the collection is not empty. Otherwise, `None` is returned.

There are similar `foldRight` and `reduceRight` methods for traversing right to left, as well as `fold` and `reduce`, where traversal order is undefined.

If you think about it, reducing can only return the least upper bound, the closest common supertype of the elements. If the elements all have the same type, the final output will have that type. In contrast, because folding takes a seed value, it offers more options for the final result. Here are fold examples that implement mapping, flat mapping, and filtering:

```
// src/script/scala/progscala3/fp/datastructs/Fold.scala

scala> val vector = Vector(1, 2, 3, 4, 5, 6)

scala> val vector2 = vector.foldLeft(Vector.empty[String]) {     ❶
     |   (vector, x) => vector :+ ("[" + x + "]")
     | }
val vector2: Vector[String] = Vector([1], [2], [3], [4], [5], [6])

scala> val vector3 = vector.foldLeft(Vector.empty[Int]) {        ❷
     |   (vector, x) => if x % 2 == 0 then vector else vector :+ x
     | }
val vector3: Vector[Int] = Vector(1, 3, 5)

scala> val vector4a = vector.foldLeft(Vector.empty[Seq[Int]]) {  ❸
     |   (vector, x) =>  vector :+ (1 to x)
     | }
```

```
val vector4a: Vector[Seq[Int]] =
  Vector(Range 1 to 1, Range 1 to 2, Range 1 to 3, Range 1 to 4, ...)

scala> val vector4 = vector4a.flatten                          ❹
val vector4: Vector[Int] =
  Vector(1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6)

scala> val vector2b = vector.foldRight(Vector.empty[String]) {  ❺
     |   (x, vector) => ("[" + x + "]") +: vector
     | }
val vector2b: Vector[String] = Vector([1], [2], [3], [4], [5], [6])

scala> vector2 == vector2b
val res0: Boolean = true
```

❶ Map over a vector, creating strings `[1]`, `[2]`, etc. While `fold` doesn't guarantee a particular traversal order, `foldLeft` traverses left to right. Note the anonymous function. For `foldLeft`, the first parameter is the accumulator, the new vector we are building, and the second parameter is an element. We return a new vector with the element string appended to it, using `:+`. This returned vector will be passed in as the new accumulator on the next iteration or returned to `vector2`.

❷ Filter a vector, returning just the odd values. Note that for even values, the anonymous function just returns the current accumulator.

❸ A map that creates a vector of ranges.

❹ Flattening the previous output vector, thereby implementing `flatMap`.

❺ Traverse from the right. Note the parameters are reversed in the anonymous function and we prepend to the accumulator. Hence `vector2` and `vector2b` are equal.

Folding really is the *universal operator* because it can be used to implement all the others, where `map`, `filter`, and `flatMap` implementations are shown here. (Try doing `foreach` yourself.)

Here are the signatures and descriptions for the various fold and reduce operations available on the iterable collections. The descriptions are paraphrased from the Scaladoc. Where you see the type parameter `A1 >: A`, recall that it means that the final output type `A1` must be a supertype of `A`, although they will often be the same types:

`def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1`
: Fold the elements of this collection using the specified associative binary operator `op`. The order in which operations are performed on elements is unspecified

and may be nondeterministic. However, for most ordered collections like `Lists`, `fold` is equivalent to `foldLeft`.

`def foldLeft[B](z: B)(op: (B, A) => B): B`
: Apply `op` to the start value `z` and all elements of this collection, going left to right. For the function `op`, the first argument is the accumulator.

`def foldRight[B](z: B)(op: (A, B) => B): B`
: Apply `op` to all elements of this collection and a start value, going right to left. For the function `op`, the second argument is the accumulator.

`def reduce[A1 >: A](op: (A1, A1) => A1): A1`
: Reduce the elements of this collection using the specified associative binary operator `op`. The order in which operations are performed on elements is unspecified and may be nondeterministic. However, for most ordered collections like `Lists`, `reduce` is equivalent to `reduceLeft`. An exception is thrown if the collection is empty.

`def reduceLeft[A1 >: A](op: (A1, A1) => A1): A1`
: Apply `op` to all elements of this collection, going left to right. An exception is thrown if the collection is empty. For the function `op`, the first argument is the accumulator.

`def reduceRight[A1 >: A](op: (A1, A1) => A1): A1`
: Apply `op` to all elements of this collection going right to left. An exception is thrown if the collection is empty. For the function `op`, the second argument is the accumulator.

`def optionReduce[A1 >: A](op: (A1, A1) => A1): Option[A1]`
: Like `reduce`, but return `None` if the collection is empty or `Some(…)` if not.

`def reduceLeftOption[B >: A](op: (B, A) => B): Option[B]`
: Like `reduceLeft`, but return `None` if the collection is empty or `Some(…)` if not.

`def reduceRightOption[B >: A](op: (A, B) => B): Option[B]`
: Like `reduceRight`, but return `None` if the collection is empty or `Some(…)` if not.

`def scan[B >: A](z: B)(op: (B, B) => B): C[B]`
: Compute a prefix scan of the elements of the collection. Note that the neutral element `z` may be applied more than once. (I'll show a following example.)

`def scanLeft[B >: A](z: B)(op: (B, B) => B): C[B]`
: Produce a collection containing cumulative results of applying the operator `op` going left to right.

`def scanRight[B >: A](z: B)(op: (B, B) => B): C[B]`
: Produce a collection containing cumulative results of applying the operator `op` going right to left.

`def product[B >: A](implicit num: math.Numeric[B]): B`
: Multiply the elements of this collection together, as long as the elements have an implicit conversion to type `Numeric`, which effectively means `Int`, `Long`, `Float`, `Double`, `BigInt`, etc. We discussed such conversions in "Constraining Allowed Instances" on page 175.

`def sum[B >: A](implicit num: math.Numeric[B]): B`
: Similar to `product`; add the elements together.

`def mkString: String`
: Display all elements of this collection in a string. This is a custom implementation of `fold` used for conveniently generating a custom string from the collection. There will be no delimiter between elements in the string.

`def mkString(sep: String): String`
: Display all elements of this collection in a string using the specified separator (`sep`) string.

`def mkString(start: String, sep: String, end: String): String`
: Display all elements of this collection in a string using the specified `start` (prefix), `sep` (separator), and `end` (suffix) strings.

Pay careful attention to the parameters passed to the anonymous functions for various `reduce`, `fold`, and `scan` methods. For the `Left` methods (e.g., `foldLeft`), the first parameter is the accumulator and the collection is traversed left to right. For the `Right` functions (e.g., `foldRight`), the second parameter is the accumulator and the collection is traversed right to left. For the methods like `fold` and `reduce` that aren't left- or right-biased, the traversal order and which parameter is the accumulator are *undefined*, but usually they delegate to the left-biased methods.

The `fold` and `scan` methods can output a completely different type, based on the seed value, while the `reduce` methods always return the same element type or a supertype.

None of these functions will terminate for infinite collections. Also, they might return different results for different runs if the collection is not a sequence (i.e., the elements are not stored in a defined order) or the operation isn't associative.

The `scan` methods are useful for processing successive subsets of a collection. Consider the following example:

```
scala> val ints = Seq(1,2,3,4,5,6)

scala> val plus = ints.scan(0)(_ + _)
```

```
val plus: Seq[Int] = List(0, 1, 3, 6, 10, 15, 21)

scala> val mult = ints.scan(1)(_ * _)
val mult: Seq[Int] = List(1, 1, 2, 6, 24, 120, 720)
```

For the `plus` example, first the seed value `0` is emitted, followed by the first element plus the seed, `1 + 0 = 1`, followed by the second element plus the previous value, `1 + 2 = 3`, and so on. Try rewriting `scan` using `foldLeft`.

Finally, the three `mkString` methods are quite handy when the default `toString` for a collection isn't what you want.

Left Versus Right Folding

There are nonobvious differences going from left to right when folding or reducing. We'll focus on folding, but the same remarks apply to reducing too.

Recall that `fold` does not guarantee a particular traversal order and the function passed to it must be associative, while `foldLeft` and `foldRight` guarantee traversal order. Consider the following examples:

```
// src/script/scala/progscala3/fp/datastructs/FoldLeftRight.scala

scala> val seq6 = Seq(1,2,3,4,5,6)
     | val int1 = seq6.fold(0)(_ + _)
     | val int2 = seq6.foldLeft(0)(_ + _)
     | val int3 = seq6.foldRight(0)(_ + _)
val seq6: Seq[Int] = List(1, 2, 3, 4, 5, 6)
val int1: Int = 21
val int2: Int = 21
val int3: Int = 21
```

All yield the same result, which is hopefully not surprising. It doesn't matter in what order we traverse the sequence, as addition is associative and also *commutative*.

Let's explore examples where order matters, meaning *noncommutative*. First, recall that for many sequences, `fold` just calls `foldLeft`. So we'll focus on `foldLeft` and `foldRight`. Second, while we used the same anonymous function previously `_ + _`, recall that the parameters passed to this function are actually reversed for `foldLeft` versus `foldRight`. To spell it out:

```
val int4 = seq6.foldLeft(0)((accum: Int, element: Int) => accum + element)
val int5 = seq6.foldRight(0)((element: Int, accum: Int) => element + accum)
```

For addition, the names are meaningless. Suppose instead that we build up a string from the sequence by folding. We'll add parentheses to show the order of evaluation:

```
scala> val left = (accum: String, element: Int) => s"($accum $element)"
     | val right = (element: Int, accum: String) => s"($accum $element)"
     | val right2 = (element: Int, accum: String) => s"($element $accum)"
     |
```

```
     | val strLeft = seq6.foldLeft("(0)")(left)
     | val strRight = seq6.foldRight("(0)")(right)
     | val strRight2 = seq6.foldRight("(0)")(right2)
val strLeft: String = ((((((0) 1) 2) 3) 4) 5) 6)
val strRight: String = (((((((0) 6) 5) 4) 3) 2) 1)
val strRight2: String = (1 (2 (3 (4 (5 (6 (0)))))))
```

Note that the bodies of `left` and `right` are the same, while the parameter list is reversed. I wrote them this way so all the parentheses would line up the same way. Clearly the numbers are different. However, `right2` reverses the way the arguments are used in the body, so the parentheses come out very different, but the order of the numbers is almost the same in `strLeft` and `strRight2`. It's worth studying these examples to be sure you understand how we got the output.

It turns out that `foldLeft` has an important advantage over `foldRight`; left traversals are tail recursive, so they can benefit from Scala's tail-call optimization.

Recall that a tail call must be the last operation in a recursion. Looking at the output for `strRight2`, the outermost string construction `(1…)` can't be performed until all of the nested strings are constructed, shown as "…" Hence, right folding is not tail recursive and it can't be converted to a loop.

In contrast, for the `reduceLeft` example, we can construct the first substring `((0) 1)`, then the next outer string `(((0) 1) 2)`, etc. In other words, we can convert this process to a loop because it is tail recursive.

Another way to see this is to implement our own simplified `reduceLeft` and `reduceRight` for `Seq`s using recursion:

```
// src/main/scala/progscala3/fp/datastructs/FoldLeftRight.scala
package progscala3.fp.datastructs

import scala.annotation.tailrec

/**
 * Simplified implementations of foldLeft and foldRight.
 */
object FoldLeftRight:
  def foldLeft[A,B](s: Seq[A])(seed: B)(f: (B,A) => B): B =
    @tailrec
    def fl(accum: B, s2: Seq[A]): B = s2 match
      case head +: tail => fl(f(accum, head), tail)
      case _ => accum
    fl(seed, s)

  def foldRight[A,B](s: Seq[A])(seed: B)(f: (A,B) => B): B =
    s match
      case head +: tail => f(head, foldRight(tail)(seed)(f))
      case _ => seed
```

Using them, we should get the same results as before:

```
scala> import progscala3.fp.datastructs.FoldLeftRight.*

scala> val strLeft3 = foldLeft(seq6)("(0)")(left)
     | val strRight3 = foldRight(seq6)("(0)")(right)
     | val strRight4 = foldRight(seq6)("(0)")(right2)
val strLeft3: String = ((((((0) 1) 2) 3) 4) 5) 6)
val strRight3: String = ((((((0) 6) 5) 4) 3) 2) 1)
val strRight4: String = (1 (2 (3 (4 (5 (6 (0)))))))
```

These implementations are simplified in the sense that they don't attempt to construct the correct subtype of the input `Seq` for the output. For example, if you pass in a `Vector`, you'll get a `List` back instead. The Scala collections handle this correctly.

You should learn these two recursion patterns well enough to always remember the behaviors and trade-offs of left versus right recursion, even though in practice you'll almost always use Scala's built-in functions instead of writing your own.

Because we are processing a `Seq`, we should normally work with the elements left to right. It's true that `Seq.apply(index: Int)` returns the element at position `index` (counting from zero). However, for a linked list, this would require an $O(N)$ traversal for each call to `apply`, yielding an $O(N^2)$ algorithm rather than $O(N)$, which we want. So the implementation of `foldRight` "suspends" prefixing the value to the rest of the new `Seq` until the recursive invocation of `foldRight` returns. Hence, `foldRight` is not tail recursive.

For `foldLeft`, we use a nested function `rl` to implement the recursion. It carries along an `accum` parameter that accumulates the new `Seq[B]`. When we no longer match on `head +: tail`, we've hit the empty tail `Seq`, at which point we return `accum`, which has the completed `Seq[B]` we'll return. When we make a recursive call to `rl`, it is the last thing we do (the tail call) because we prepend the new element to `accum` before passing its updated value to `rl`. Hence, `foldLeft` is tail recursive.

In contrast, when we hit the end of the input `Seq` in `foldRight`, we return an empty `Seq[B]` and *then* the new elements are prefixed to it as we "pop the stack."

However, right recursion has one advantage over left recursion. Consider the case where you have a potentially infinite stream of data coming in. You can't conceivably put all that data into a collection in memory, but perhaps you only need to process the first *N* elements, for some *N*, and then discard the rest. The library's `LazyList` is designed for this purpose. `LazyList` only evaluates the head and tail on demand. We discussed it briefly near the beginning of this chapter.

This on-demand evaluation is the only way to define an infinite stream, and the assumption is that we'll never ask for all of it! That evaluation could be reading from an input channel, like a socket, a Kafka *topic*, or a social media "firehose." Or it could be a function that generates a sequence of numbers. For example, `LazyList.from(0)` can generate all the natural numbers.

How is it useful here? Let's develop an intuition for it by reviewing the output we just generated for `strLeft3` and `strRight4`:

```
val strLeft3: String = (((((((0) 1) 2) 3) 4) 5) 6)
val strRight4: String = (1 (2 (3 (4 (5 (6 (0)))))))
```

Suppose we only care about the first four numbers. Visually, we could grab the prefix string `(1 (2 (3 (4` from `strRight4` and then stop. (We might add right parentheses for aesthetics.) We're done! In contrast, assume we could generate a string like `strLeft3` with an infinite sequence. To get the first four numbers, we would have to traverse the infinite left parentheses to reach them.[3]

Let's consider an interesting example of using `LazyList` to define a famous infinite Fibonacci sequence.

Recall that a Fibonacci number `fib(n)` is defined as follows for natural numbers:

```
f(n) =
  0                  if n = 0
  1                  if n = 1
  f(n-1) + f(n-2)    otherwise
```

Like any good recursion, `n` equals 0 or 1 provides the termination condition we need, in which case `f(n) = n`. Otherwise, `f(n) = f(n-1) + f(n-2)`. We saw a tail recursive implementation in "Nesting Method Definitions and Recursion" on page 45.

Now consider this definition using `LazyList` and described in its documentation:

```
// src/main/scala/progscala3/fp/datastructs/LazyListFibonacci.scala
package progscala3.fp.datastructs

import scala.math.BigInt

object Fibonacci:
  val fibs: LazyList[BigInt] =
    BigInt(0) #:: BigInt(1) #:: fibs.zip(fibs.tail).map(n12 => n12._1 + n12._2)
```

Let's try it:

```
scala> import progscala3.fp.datastructs.Fibonacci

scala> Fibonacci.fibs.take(10).toList
val res0: List[BigInt] = List(0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
```

3 To be clear, our `foldRight`, as well as the standard library implementations, do not provide a way to terminate the recursion. You would have to write this recursion yourself, stopping after a desired level.

`LazyList` defines `#::`, its own lazy version of the cons operation.[4] We construct the first two values of the sequence eagerly for the special case of `n` equals 0 and 1, then we define the rest of the stream with a *recursive definition*. It is right recursive, but we'll only take the first `n` elements and discard the rest.

Note that we are both defining `fibs` and defining the tail portion using `fibs` itself: `fibs.zip(fibs.tail).map(...)`. This tail expression pairs up all elements of `fibs` with the successor elements because you always calculate a Fibonacci number by adding its two predecessors together. For example, we have tuple elements like `(f(2), f(3))`, `(f(3), f(4))`, etc., going on to infinity (at least lazily). Note that the tuples are then mapped to an integer, the sum of their values, since `f(4) = f(2) + f(3)`.

This is a clever and powerful recursive definition of the Fibonacci sequence. It helps to play with the pieces of the expression to understand what each one does and then work out the first several values by hand.

It's important to note that the structure of `fibs` is very similar to our implementation of `FoldLeftRight.foldRight`, `f(0) + f(1) + tail`. Because it is effectively a right recursion, we can stop evaluating `tail` when we have as many head elements as we want. In contrast, trying to construct a left recursion that is also lazy is not possible because it would look conceptually like this: `f(0 + f(1 + f(tail))`. (Compare our implementation of `FoldLeftRight.foldLeft`.) Hence, a right recursion lets us work with infinite, lazy streams, truncating them where we want, while a left recursion does not, but at least it is tail recursive!

> Left recursions are tail recursive. Right recursions let us use truncation to handle potentially infinite streams.

Combinators: Software's Best Component Abstractions

When OOP went mainstream in the late '80s and early '90s, there was great hope that it would usher in an era of reusable software components, even an industry of component libraries. It didn't work out that way, except in special cases, like various GUI libraries.

---

4 Two colons are used because `List` defines cons operator `::`, for historical reasons. I haven't mentioned it before because it is now the convention in Scala to always use `+:` for prepending elements to all sequences, including `List`s.

Why wasn't OOP more successful at promoting reuse? There are many factors, but the fundamental reason is that OOP provided the wrong abstractions to create reusable modules. It's a paradox that the richness of class hierarchies, polymorphic methods, etc., actually undermined modularization into reusable components because they were too open ended. They didn't constrain innovation in the right ways to cause the abstractions and protocols to emerge at the right level of abstraction.

In the larger world, component models that succeeded are all based on very simple foundations. Digital integrated circuits plug into buses with 2^n signaling wires, each of which is Boolean, either on or off. Upon the foundation of this extremely simple protocol, an industry was born with the most explosive growth of any industry in human history.

HTTP is another successful example of a component model. Services interact through a narrow, well-defined interface, involving a handful of message types, a naming standard (URLs), and simple standards for message content.

In both cases, these foundations were quite restrictive, but flexible enough for higher-level protocols to emerge from them, protocols that enabled composition to generate more complex structures. In digital circuits, some binary patterns are interpreted as CPU instructions, others as memory addresses, and others as data values. REST, data formats like JSON, and other higher-level standards are built upon the foundation elements of HTTP.

In this chapter, we discussed sets of collections, `Seq` (`List`), `Vector`, `Map`, etc. All share a set of uniform operations that work consistently across them, yet because they are higher-order functions, they provide the flexibility needed to do almost any data manipulation required. Except for `foreach`, all are pure and composable. Their composability can be described by *combinatory logic*, from which we get the term *combinators*.

We can chain these combinators together to build up nontrivial computations with relatively little code. This is why the chapter started with the Alan J. Perlis quote.

Let's finish this discussion with a final example using both OOP and FP features, a simplified payroll calculator:

```
// src/test/scala/progscala3/fp/combinators/PayrollSuite.scala
package progscala3.fp.combinators

import munit.*

class PayrollSuite extends FunSuite:

  case class Employee (name: String, title: String, annualSalary: Double,
    taxRate: Double, insurancePremiumsPerWeek: Double)

  val employees = List(
```

```
    Employee("Buck Trends", "CEO", 200000, 0.25, 100.0),
    Employee("Cindy Banks", "CFO", 170000, 0.22, 120.0),
    Employee("Joe Coder", "Developer", 130000, 0.20, 120.0))

  val weeklyPayroll = employees map { e =>
    val net = (1.0 - e.taxRate) * (e.annualSalary / 52.0) -
      e.insurancePremiumsPerWeek
    (e, net)
  }

  test("weeklyPayroll computes pay for each employee") {
    val results1 = weeklyPayroll map {
      case (e, net) => (e.name, f"${net}%.2f")
    }
    assert(results1 == List(
      ("Buck Trends", "2784.62"),
      ("Cindy Banks", "2430.00"),
      ("Joe Coder",   "1880.00")))
  }

  test("from weeklyPayroll, the totals can be calculated") {
    val report = weeklyPayroll.foldLeft( (0.0, 0.0, 0.0) ) {
      case ((totalSalary, totalNet, totalInsurance), (e, net)) =>
        (totalSalary + e.annualSalary/52.0,
          totalNet + net, totalInsurance + e.insurancePremiumsPerWeek)
    }
    assert(f"${report._1}%.2f" == "9615.38", "total salary")
    assert(f"${report._2}%.2f" == "7094.62", "total net")
    assert(f"${report._3}%.2f" == "340.00" , "total insurance")
  }
```

We could have implemented this logic in many ways, but let's consider a few of the design choices.

OOP encapsulation of some domain concepts, like `Employee`, is useful for code comprehension and concision. *Meaningful names* is an old principle of good software design. Although I've emphasized the virtues of fundamental collections, FP does not say that custom types are bad. As always, design trade-offs should be carefully considered.

However, `Employee` could be called *anemic*. It is a structure with minimal behavior—only the methods generated by the compiler for all case classes. In classic object-oriented design, we might add a lot of behavior to `Employee` to help with the payroll calculation or other domain logic. I believe the design chosen here provides optimal separation of concerns. It's also so concise that the maintenance burden is small if the structure of `Employee` changes and this code has to change.

Note also that the logic was implemented in small code snippets rather than defined in lots of classes spread over many files. Of course, it's a toy example, but hopefully you can appreciate that nontrivial applications don't always require large code bases.

There is a counterargument for using a dedicated type—the overhead of constructing instances. Here, this overhead is unimportant. What if we have billions of records? We'll return to this question when we explore big data in "Scala for Big Data: Apache Spark" on page 459.

What About Making Copies?

Let's finish this chapter by considering a practical problem. Making copies of functional collections is necessary to preserve immutability, but suppose I have a `Vector` of 100,000 items and I need a copy with the item at index 8 replaced. It would be terribly inefficient to construct a completely new, 100,000-element copy.

Fortunately, we don't have to pay this penalty, nor must we sacrifice immutability. The secret is to realize that 99,999 elements are not changing. If we can share the parts of the original `Vector` that aren't changing, while representing the change in some way, then creating the new vector can still be very efficient. This idea is called *structure sharing*.

If other code on a different thread is doing something different with the original vector, it is unaffected by the copy operation because the original vector is not modified. In this way, a history of vectors is preserved, as long as there are references to one or more older versions. No version will be garbage-collected until there are no more references to it.

Because this history is maintained, a data structure that uses structure sharing is called a *persistent data structure*.

Let's start with an easier example, prepending an element to a `List`, which is defined by its head element and tail `List`. All we need to do is return a new `List` with the new element as the head and the old `List` as the tail. No copying required!

`Vector` is more challenging. We have to select an implementation data structure that lets us expose `Vector` semantics, while providing efficient operations that exploit structure sharing. Let's sketch the underlying data structure and how the copy operation works. We won't cover all the details in depth. For more information, start with the Wikipedia page on persistent data structures.

The tree data structure with a branching factor of 32 is used. The branching factor is the maximum number of child nodes each parent node is allowed to have. We said earlier in this chapter that some `Vector` search and modification operations are $O(\log(N))$, but with 32 as the branching factor, that becomes $O(\log_{32}(N))$, effectively a constant for even large values of N!

Figure 7-2 shows an example for `Vector(1,2,3,4,5)`. For legibility, a branching factor of 2 or 3 is used instead of 32.

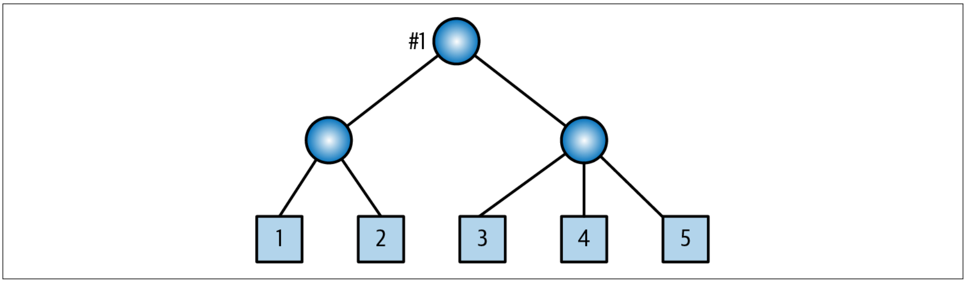


Figure 7-2. A `Vector` represented as a tree

When you reference the `Vector`, you're actually referencing the root of this tree, marked by `#1`. As an exercise, you might work through how operations, such as accessing a particular element by its index, `map`, `flatMap`, etc., would work on a tree implementation of a `Vector`.

Now suppose we want to insert 2.5 between 2 and 3. To create a new copy, we don't mutate the original tree, but instead create a new tree. Figure 7-3 shows one way to add 2.5 between 2 and 3.

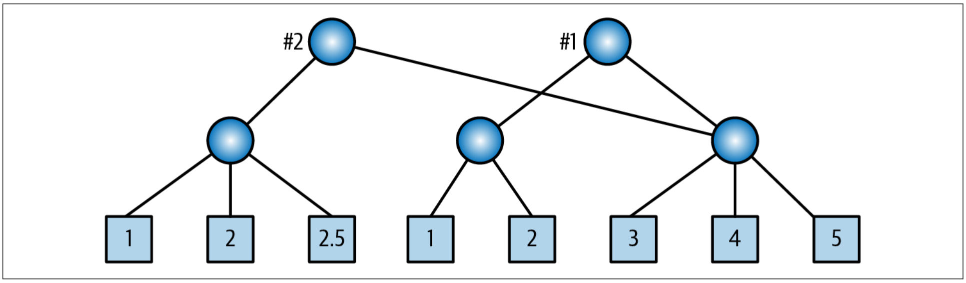


Figure 7-3. Two states of a `Vector`, before and after element insertion

Note that the original tree (`#1`) remains, but we have created a new root (`#2`), new nodes between it, and the child holding the new element. A new left subtree was created. With a branching factor of 32, we will have to copy up to 32 child references per level, but this number of copy operations is far less than the number required for all references in the original tree.

Deletion and other operations work similarly. A good textbook on data structures will describe the standard algorithms for tree operations.

Therefore, it is possible to use large, immutable data structures if their implementations support an efficient copy operation. There is extra overhead compared to a mutable vector, where you can simply modify an entry in place very quickly. Ironically, that doesn't mean that object-oriented and other procedural programs are necessarily simpler and faster.

Because of the dangers of mutability, it's common for OOP classes to wrap mutable collections they hold in accessor methods. This increases the code footprint, testing burden, etc. Worse, if the collection itself is exposed through a getter method, the defensive class author might make a copy of the collection to return, so that the internal copy can't be modified. Because collection implementations in nonfunctional languages often have inefficient copy operations and more complex surrounding code, the net effect can be less efficient and more complex programs, compared to equivalent functional programs. Immutable collections can be efficient and eliminate defensive programming.

There are other kinds of functional data structures that are optimized for efficient copying, optimized for modern hardware, such as minimizing cache misses. Many of these data structures were invented as alternatives to mutable data structures that are commonly discussed in classic textbooks on data structures and algorithms.

Recap and What's Next

We discussed the basic concepts of FP and argued for their importance for solving modern problems in software development. We saw how the fundamental collections and their common higher-order functions, *combinators*, yield concise, powerful, modular code.

Typical functional programs are built on this foundation. At the end of the day, all programs input data, perform transformations on it, then output the results. Much of the ceremony in typical programs just obscures this essential purpose.

Since FP is still relatively new for many people, let's finish this chapter with some references for more information (see the Bibliography for more details): [Alexander2017] is a gentle introduction to FP, while [Chiusano2013] and [Volpe2020] provide in-depth introductions. For more on functional data structures, see [Okasaki1998], [Bird2010], and [Rabhi1999]. See [Vector2020] for details on the standard library's `Vector` implementation. To practice using combinators, see "Ninety-Nine Scala Problems".

Next, we'll return to `for` comprehensions and use our new knowledge of FP to understand how `for` comprehensions are implemented, how we can implement our own data types to exploit them, and how the combination of `for` comprehensions and combinator methods yield concise, powerful code. Along the way, we'll deepen our understanding of functional concepts.

We'll dive into more of the implementation details of Scala collections in Chapter 14. We'll return to more advanced features of FP in Chapter 18.

CHAPTER 8

for Comprehensions in Depth

"for Comprehensions" on page 86 described the syntax for `for` comprehensions, including lots of examples. At this point, they look like a nice, more flexible version of the venerable `for` loop, but not much more. In fact, lots of sophistication lies below the surface. This chapter explores how `for` comprehension syntax is a more concise way to use `foreach`, `map`, `flatMap`, and `withFilter`, some of the *functional combinators* we discussed in the previous chapter. You can write concise code with elegant solutions to a number of design problems.

We'll finish with some practical design problems implemented using `for` comprehensions, such as error handling during the execution of a sequence of processing steps.

Recap: The Elements of for Comprehensions

A `for` comprehension contains one or more generator expressions, optional guard expressions for filtering, and optional value definitions. The output can be yielded to create a new collection, or a side-effecting block of code can be executed on each pass, such as printing output. The following example demonstrates all these features. It removes blank lines from a text file. This is a full program with an example of how to parse input arguments (although there are libraries available for this purpose), handle help messages, etc.:

```
// src/main/scala/progscala3/forcomps/RemoveBlanks.scala
package progscala3.forcomps

object RemoveBlanks:
  def apply(path: String, compress: Boolean, numbers: Boolean): Seq[String] =
    for                                                                    ❶
      (line, i) <- scala.io.Source.fromFile(path).getLines.toSeq.zipWithIndex
      if line.matches("""^\s*$""") == false                                ❷
      line2 = if compress then line.trim.replaceAll("\\s+", " ")           ❸
```

```
            else line
    numLine = if numbers then "%4d: %s".format(i+1, line2)        ❹
            else line2
  yield numLine

protected case class Args(                                        ❺
  compress: Boolean = false,
  numbers: Boolean = false,
  paths: Vector[String] = Vector.empty)

def main(params: Array[String]): Unit =                           ❻

  val Args(compress, numbers, paths) = parseParams(params.toSeq, Args())
  for                                                             ❼
    path <- paths
    seq = s"\n== File: $path\n" +: RemoveBlanks(path, compress, numbers)
    line <- seq
  do println(line)

protected val helpMessage = """
  |usage: RemoveBlanks [-h|--help] [-c|--compress] [-n|--numbers] file ...
  |where:
  | -h | --help      Print this message and quit.
  | -c | --compress  Compress whitespace.
  | -n | --numbers   Print original line numbers, meaning output numbers will
  |                  skip the removed blank lines.
  | file ...         One or more files to process.
  |""".stripMargin

protected def help(messages: Seq[String], exitCode: Int) =
  messages.foreach(println)
  println(helpMessage)
  sys.exit(exitCode)

protected def parseParams(params2: Seq[String], args: Args): Args =
  params2 match
    case ("-h" | "--help") +: tail =>
      println(helpMessage)
      sys.exit(0)
    case ("-c" | "--compress") +: tail =>
      parseParams(tail, args.copy(compress = true))
    case ("-n" | "--number") +: tail =>
      parseParams(tail, args.copy(numbers = true))
    case flag +: tail if flag.startsWith("-") =>
      println(s"ERROR: Unknown option $flag")
      println(helpMessage)
      sys.exit(1)
    case path +: tail =>
      parseParams(tail, args.copy(paths = args.paths :+ path))
    case Nil => args
```

❶ Start with a generator. Use `scala.io.Source` to open the file and get the lines, where `getLines` returns an `Iterator`, which we must convert to a sequence because we can't return an `Iterator` from the `for` comprehension and the return type is determined by the initial generator. Using `zipWithIndex` adds line numbers (zero based).

❷ A guard. Filters out blank lines using a regular expression. This will result in line number gaps.

❸ Value definition for the nonblank line, with or without whitespace compression.

❹ Another value definition for the line with the one-based line number, if enabled, or just the line.

❺ Convenience class to hold parsed command-line arguments, including the files to process and flags for whether or not to compress the whitespace in lines and whether or not to print line numbers.

❻ The `main` method to process the argument list.

❼ A second `for` comprehension to process the files. Note that we prepend a line to print with the filename.

Try running it at the `sbt` prompt:

```
> runMain progscala3.forcomps.RemoveBlanks --help
> runMain progscala3.forcomps.RemoveBlanks README.md build.sbt -n -c
```

Try different files and different command-line options.

for Comprehensions: Under the Hood

Having a second way to invoke `foreach`, `map`, `flatMap`, and `withFilter` aims for easier comprehension and concision, especially for nontrivial processing. After a while, you develop an intuition about when to use comprehensions and when to use the *combinator* methods directly.

The method `withFilter` is used for filtering elements instead of `filter`. The compiler uses it with neighboring combinators so that one less new collection is generated. Like `filter`, `withFilter` restricts the domain of the elements allowed to pass through subsequent combinators like `map`, `flatMap`, `foreach`, and other `withFilter` invocations.

To see what the `for` comprehension sugar encapsulates, let's walk through several informal comparisons first, then we'll discuss the details of the precise mapping. As you look at the examples that follow, ask yourself which syntax is easier to understand in each case, the `for` comprehension or the corresponding method calls.

Consider this example of a simple `for` comprehension and the equivalent use of `foreach` on a collection:

```
// src/script/scala/progscala3/forcomps/ForForeach.scala

scala> val states = Vector("Alabama", "Alaska", "Virginia", "Wyoming")

scala> var lower1 = Vector.empty[String]
scala> for
     |   s <- states
     | do lower1 = lower1 :+ s.toLowerCase
var lower1: Vector[String] = Vector(alabama, alaska, virginia, wyoming)

scala> var lower2 = Vector.empty[String]
     | for s <- states do lower2 = lower2 :+ s.toLowerCase    // same output

scala> var lower3  = Vector.empty[String]
     | states.foreach(s => lower3 = lower3 :+ s.toLowerCase)  // same output
```

When there is just one generator (the `s <- states`) in a `for` comprehension, it can be written on a single line, as shown for `lower1`. You can still put the `do` clause on the next line, if you prefer.

A single generator with a `do` statement corresponds to an invocation of `foreach` on the collection.

What happens if we use `yield` instead?

```
// src/script/scala/progscala3/forcomps/ForMap.scala

scala> val upper1 = for
     |   s <- states
     | yield s.toUpperCase
val upper1: Vector[String] = Vector(ALABAMA, ALASKA, VIRGINIA, WYOMING)

scala> val upper2 = for s <- states yield s.toUpperCase    // same output
scala> val upper3 = states.map(_.toUpperCase)              // same output
```

A single generator followed by a `yield` expression corresponds to an invocation of `map`. When `yield` is used to construct a new container, its type is determined by the first generator. This is consistent with how `map` works.

What if we have more than one generator?

```
// src/script/scala/progscala3/forcomps/ForFlatmap.scala

scala> val results1 = for
     |   s <- states
     |   c <- s
```

```
     | yield s"$c-${c.toUpper}"
val results1: Vector[String] = Vector(A-A, l-L, a-A, b-B, a-A, m-M, a-A, ...)

scala> val results2 = states.
     |   flatMap(s => s.toSeq).
     |   map(c => s"$c-${c.toUpper}")                    // same output
```

The second generator iterates through each character in the string `s`. The contrived `yield` statement returns the character and its uppercase equivalent, separated by a dash.

When there are multiple generators, all but the last are converted to `flatMap` invocations. The last is a `map` invocation. Already, you may find the `for` comprehension more concise and easier to understand.

What if we add a guard to remove the capital letters?

```
// src/script/scala/progscala3/forcomps/ForGuard.scala

scala> val results1 = for
     |   s <- states
     |   c <- s
     |   if c.isLower
     | yield s"$c-${c.toUpper}"
val results1: Vector[String] = Vector(l-L, a-A, b-B, a-A, m-M, a-A, l-L, ...)

scala> val results2 = states.
     |   flatMap(s => s.toSeq).
     |   withFilter(c => c.isLower).
     |   map(c => s"$c-${c.toUpper}")                    // same output
```

Note that the `withFilter` invocation is injected before the final `map` invocation.

Try rewriting this example using `do println(s"…")` instead of `yield…`

Finally, defining a variable works as follows:

```
// src/script/scala/progscala3/forcomps/ForVariable.scala

scala> val results1 = for
     |   s <- states
     |   c <- s
     |   if c.isLower
     |   c2 = s"$c-${c.toUpper}"
     | yield (c, c2)
val results1: Vector[(Char, String)] = Vector((l,l-L), (a,a-A), (b,b-B), ...)

scala> val results2 = states.
     |   flatMap(s => s.toSeq).
     |   withFilter(c => c.isLower).
     |   map(c => (c, s"$c-${c.toUpper}"))               // same output
```

This time I output tuples to illustrate how the variable definition is handled when translating the `for` comprehension to the corresponding sequence of methods. We only need `c` and `c2` here, so `s` isn't carried forward into the `map` call.

Translation Rules of for Comprehensions

Now that we have an intuitive understanding of how `for` comprehensions are translated to collection methods, let's define the details more precisely.

First, in a generator such as `pat <- expr`, `pat` is a pattern expression. For example, `(x, y) <- Seq((1,2),(3,4))`. Similarly, in a value definition like `pat2 = expr`, `pat2` is also interpreted as a pattern. For example, `(x, y) = aPair`.

Because these lefthand expressions are interpreted as patterns, the compiler translates them using partial functions. The first step in the translation is to convert a simple comprehension with a generator, `pat <- expr`. The translation is similar to the following example comprehensions (`yield`) and loops (`do`):

```
// src/script/scala/progscala3/forcomps/ForTranslated.scala

scala> val seq = Seq(1,2,3)

scala> for i <- seq yield 2*i
val res0: Seq[Int] = List(2, 4, 6)

scala> seq.map { case i => 2*i }
val res1: Seq[Int] = List(2, 4, 6)

scala> var sum1 = 0
scala> for i <- seq do sum1 += 1
var sum1: Int = 3

scala> var sum2 = 0
scala> seq.foreach { case i => sum2 += 1 }
var sum2: Int = 3
```

A conditional is translated to `withFilter` conceptually, as shown next:

```
scala> for
     |   i <- seq
     |   if i%2 != 0
     | yield 2*i
val res2: Seq[Int] = List(2, 6)

scala> for
     |   i <- seq if i%2 != 0                    ❶
     | yield 2*i
val res3: Seq[Int] = List(2, 6)

scala> seq.withFilter {
```

```
     |   case i if i%2 != 0 => true
     |   case _ => false
     | }.map { case i => 2*i }
val res4: Seq[Int] = List(2, 6)
```

❶ You can write the guard on the same line as the previous generator.

After this, the translations are applied repeatedly until all comprehension expressions have been replaced. Note that some steps generate new `for` comprehensions that subsequent iterations will translate.

First, a `for` comprehension with two generators and a `yield` expression:

```
scala> for
     |   i <- seq
     |   j <- (i to 3)
     | yield j
val res5: Seq[Int] = List(1, 2, 3, 2, 3, 3)

scala> seq.flatMap { case i => for j <- (i to 3) yield j }       ❶
val res6: Seq[Int] = List(1, 2, 3, 2, 3, 3)

scala> seq.flatMap { case i => (i to 3).map { case j => j } }    ❷
val res7: Seq[Int] = List(1, 2, 3, 2, 3, 3)
```

❶ One level of translation. Note the nested `for...yield`.

❷ Completed translation.

A `for` loop, with `do`, again translating in two steps:

```
scala> var sum3=0
scala> for
     |   i <- seq
     |   j <- (i to 3)
     | do sum3 += j
var sum3: Int = 14

scala> var sum4=0
scala> seq.foreach { case i => for j <- (i to 3) do sum4 += j }
var sum4: Int = 14

scala> var sum5=0
scala> seq.foreach { case i => (i to 3).foreach { case j => sum5 += j } }
var sum5: Int = 14
```

A generator followed by a value definition has a surprisingly complex translation. Here I show complete `for...yield...` expressions:

```
scala> for
     |   i <- seq
     |   i10 = i*10
```

```
     | yield i10
val res8: Seq[Int] = List(10, 20, 30)

scala> for
     |   (i, i10) <- for
     |     x1 @ i <- seq                    ❶
     |   yield
     |     val x2 @ i10 = x1*10             ❷
     |     (x1, x2)                         ❸
     | yield i10                            ❹
val seq9: Seq[Int] = List(10, 20, 30)
```

❶ Recall from Chapter 4 that `x1 @ i` means assign to variable `x1` the value corresponding to the whole expression on the righthand side of `@`, which is trivially `i` in this case, but it could be an arbitrary pattern with nested variable bindings to the constituent parts.

❷ Assign to `x2` the value of `i10`.

❸ Return the tuple.

❹ Yield `i10`, which will be equivalent to `x2`.

Here is another example of what `x @ pat = expr` does for us:

```
scala> val z @ (x, y) = (1 -> 2)
val z: (Int, Int) = (1,2)
val x: Int = 1
val y: Int = 2
```

This completes the translation rules. Whenever you encounter a `for` comprehension, you can apply these rules to translate it into method invocations on containers. You won't need to do this often, but sometimes it's a useful skill for debugging problems.

Options and Container Types

We used collections like `Lists`, `Arrays`, and `Maps` for our examples, but any types that implement `foreach`, `map`, `flatMap`, and `withFilter` (or `filter`) can be used in `for` comprehensions and not just the obvious collection types. In the general case, these are containers and eligible for use in `for` comprehensions.

Let's consider several other types that are similar to containers. We'll see how exploiting `for` comprehensions can transform your code in unexpected ways.

Option as a Container?

`Option` is like a container that has a single item or it doesn't. It implements the four methods we need.

Here is a simplified version of the Option abstract class in the Scala library; the full source is on GitHub:

```
sealed abstract class Option[+A] { self =>                    ❶
  ...
  def isEmpty: Boolean = this eq None                          ❷

  final def foreach[U](f: A => U): Unit =
    if (!isEmpty) f(this.get)

  final def map[B](f: A => B): Option[B] =
    if (isEmpty) None else Some(f(this.get))

  final def flatMap[B](f: A => Option[B]): Option[B] =
    if (isEmpty) None else f(this.get)

  final def filter(p: A => Boolean): Option[A] =
    if (isEmpty || p(this.get)) this else None

  final def withFilter(p: A => Boolean): WithFilter = new WithFilter(p)

  class WithFilter(p: A => Boolean) {                          ❸
    def map[B](f: A => B): Option[B] = self filter p map f     ❹
    def flatMap[B](f: A => Option[B]): Option[B] = self filter p flatMap f
    def foreach[U](f: A => U): Unit = self filter p foreach f
    def withFilter(q: A => Boolean): WithFilter =
      new WithFilter(x => p(x) && q(x))
  }
}
```

❶ `Option[+A]` means it is covariant in `A`, so `Option[String]` is a subtype of `Option[AnyRef]`. The `self =>` expression defines an alias for `this` for the `Option` instance. It is used inside `WithFilter` below to refer to the `Option` instance (see "Self-Type Declarations" on page 382).

❷ Test if `this` is actually the `None` instance, not value equality.

❸ The `WithFilter`, which is used by `withFilter` combined with the other operations to avoid creation of an intermediate collection when filtering.

❹ Here's where the `self` reference we defined earlier is used to operate on the enclosing `Option` instance. Using `this` would refer to the instance of `WithFilter` itself.

The `final` keyword prevents subtypes from overriding the implementation. It might be surprising to see the supertype refer to subtypes. Normally, in object-oriented design this would be considered bad. However, with `sealed` type hierarchies, this file knows all the possible subtypes. Referring to subtypes makes the implementation more concise and efficient overall, as well as safe.

The crucial feature about these `Option` methods shown is that the function arguments are only applied if the `Option` isn't empty. This feature allows us to address a common design problem in an elegant way.

Let's recap an idea we explored in "Pattern Matching as Filtering in for Comprehensions" on page 125. Say for example that we want to distribute some tasks around a cluster and then gather the results together. We want a simple way to ignore any returned results that are empty. Let's wrap each task return value in an `Option`, where `None` is used for an empty result and `Some` wraps a nonempty result. We want an easy way to filter out the `None` results. Here is an example, where we have the returned `Option`s in a `Vector`:

```
// src/script/scala/progscala3/forcomps/ForOptionsFilter.scala

scala> val options: Seq[Option[Int]] = Vector(Some(10), None, Some(20))
val options: Seq[Option[Int]] = Vector(Some(10), None, Some(20))

scala> val results = for
     |   case Some(i) <- options
     | yield (2 * i)
val results: Seq[Int] = Vector(20, 40)
```

`case Some(i) <- options` pattern matches on each element in `results` and extracts the integers inside the `Some` values. Since a `None` won't match, all of them are skipped. We then yield the final expression we want. The reason partial functions are used by Scala to implement `for` comprehensions is so we don't get `MatchError`s because we're not matching on `None`.

3 In Scala 2, you can omit the `case` keyword, but Scala 3 requires it to make it more explicit that pattern matching and filtering are being performed.

As an exercise, let's work through the translation rules. First, convert each `pat <- expr` expression to a `withFilter` expression:

```
scala> val results2 = for
     |   case Some(i) <- options.withFilter {
     |     case Some(i) => true
     |     case None => false
     |   }
     | yield (2 * i)
val results2: Seq[Int] = Vector(20, 40)
```

Finally, we convert the outer `for x <- y yield (z)` expression to a `map` call:

```
scala> val results3 = options.withFilter {
     |   case Some(i) => true
     |   case None => false
```

```
     | } map {
     |   case Some(i) => (2 * i)
     |   case None => -1             // hack
     | }
val results3: Seq[Int] = Vector(20, 40)
```

The hack is there because we don't actually need the `case None` clause, because the `withFilter` has already removed all `None`s. However, the compiler doesn't understand this, so it warns us we'll risk a `MatchError` without the clause. Try removing this clause and observe the warning you get.

Consider another design problem. Instead of independent tasks where we ignore the empty results and combine the nonempty results, consider the case where we run a sequence of dependent steps and want to stop the whole process as soon as we encounter a `None`.

Note that we have a limitation that using `None` means we receive no feedback about why the step returned nothing, such as a failure. We'll address this limitation when we discuss alternatives starting with `Either` in the next section.

We could write tedious conditional logic that tries each case, one at a time, and checks the results, but a `for` comprehension is more concise:

```
// src/script/scala/progscala3/forcomps/ForOptionsSeq.scala

scala> def positiveOption(i: Int): Option[Int] =
     |   if i > 0 then Some(i) else None

scala> val resultSuccess = for
     |   i1 <- positiveOption(5)
     |   i2 <- positiveOption(10 * i1)
     |   i3 <- positiveOption(25 * i2)
     |   i4 <- positiveOption(2  * i3)
     | yield (i1 + i2 + i3 + i4)
val resultSuccess: Option[Int] = Some(3805)

scala> val resultFail = for
     |   i1 <- positiveOption(5)
     |   i2 <- positiveOption(-1 * i1)          ❶
     |   i3 <- positiveOption(25 * i2)
     |   i4 <- positiveOption(-2 * i3)
     | yield (i1 + i2 + i3 + i4)
val resultFail: Option[Int] = None
```

❶ `None` is returned. The subsequent generators don't call `positiveOption`, they just pass the `None` through.

At each step, the integer in the `Some` returned by `positiveOption` is extracted and assigned to a variable. Subsequent generators use those values. It appears we assume the "happy path" always works, which is true for the first `for` comprehension. It also works fine for the second for comprehension because once a None is returned, the subsequent generators simply propagate the None and don't call positiveOption.

Let's look at three more container types with similar properties, Either and Try from the Scala library, and Validated from Typelevel Cats. Validated is a sophisticated tool for sequencing validation steps.

Either: An Alternative to Option

We noted that the use of Option has the disadvantage that None carries no information that could tell us why no value is available. Did an error occur? What kind? Using Either instead is one solution. As the name suggests, Either is a container that holds one and only one of two things. In other words, where Option handled the case of zero or one items, Either handles the case of one item or another.

Either is a parameterized type with two parameters, Either[+A, +B], where the A and B are the two possible types of the element contained in an Either instance.

Either is also a sealed abstract class with two subtypes defined, Left and Right. That's how we distinguish between the two possible elements.

The concept of Either predates Scala. It has been used for a long time as an alternative to throwing exceptions. By historical convention, the Left value is used to hold the error indicator, such as a message string or thrown exception, and the normal return value is returned in a Right.

Let's port our Option example. It's almost identical:

```
// src/script/scala/progscala3/forcomps/ForEithers.scala

scala> def positiveEither(i: Int): Either[String,Int] =
     |   if i > 0 then Right(i) else Left(s"nonpositive number $i")

scala> val result1 = for
     |   i1 <- positiveEither(5)
     |   i2 <- positiveEither(10 * i1)
     |   i3 <- positiveEither(25 * i2)
     |   i4 <- positiveEither(2  * i3)
     | yield (i1 + i2 + i3 + i4)
val result1: Either[String, Int] = Right(3805)

scala> val result2 = for
     |   i1 <- positiveEither(5)
     |   i2 <- positiveEither(-1 * i1)     ❶
     |   i3 <- positiveEither(25 * i2)
     |   i4 <- positiveEither(-2 * i3)
     | yield (i1 + i2 + i3 + i4)
val result2: Either[String, Int] = Left(nonpositive number -5)
```

❶ A `Left` is returned here, stopping the process.

Note how `Left` and `Right` objects are constructed in `positiveEither`. Note the types for `result1` and `result2`. In particular, `result2` now tells us where the first negative number was encountered, but not the second occurrence of one.

`Either` isn't limited to this error-handling idiom. It could be used for any scenario where you want to hold an object of one or another type. Recall we also have union types, like `String | Int`, which aren't limited to two types! However, union types don't have the combinators `map`, `flatMap`, etc., so they can't be used conveniently in `for` comprehensions.

That raises some questions, though. Why do `Lefts` stop the `for` comprehension and `Rights` don't? It's because `Either` isn't really symmetric in the types. Since it is almost always used for this error-handling idiom, the implementations of `Left` and `Right` bias toward the `right` as the "happy path."

Let's look at how the combinators and some other methods work for these two types, using `result1` and `result2`:

```
scala> result1    // Reminder of these values:
     | result2
val res6: Either[String, Int] = Right(3805)
val res7: Either[String, Int] = Left(nonpositive number -5)

scala> var r1  = 0
     | result1.foreach(i => r1 = i * 2)
     | var r2  = 0
     | result2.foreach(i => r2 = i * 2)
var r1: Int = 7610
var r2: Int = 0                                              ❶

scala> val r3  = result1.map(_ * 2)
     | val r4  = result2.map(_ * 2)
     |
val r3: Either[String, Int] = Right(7610)
val r4: Either[String, Int] = Left(nonpositive number -5)

scala> val r5a = result1.flatMap(i => Right(i * 2))
     | val r5b = result1.flatMap(i => Left("hello"))
     | val r5c = result1.flatMap(i => Left[String,Double]("hello"))
     | val r5d: Either[String,Double] = result1.flatMap(i => Left("hello"))
     | val r6  = result2.flatMap(i => Right(i * 2))
     |
val r5a: Either[String, Int] = Right(7610)
val r5b: Either[String, Nothing] = Left(hello)               ❷
val r5c: Either[String, Double] = Left(hello)
val r5d: Either[String, Double] = Left(hello)
val r6:  Either[String, Int] = Left(nonpositive number -5)
```

❶ No change is made to `r2` after initialization.

❷ Note the second type for `r5b` versus `r5c` and `r5d`. Using `Left("hello")` alone provides no information about the desired second type, so `Nothing` is used.

The `filter` and `withFilter` methods aren't supported. They are somewhat redundant in this case.

You can infer that the `Left` method implementations ignore the function and just return their value. `Right.map` extracts the value, applies the function, then constructs a new `Right`, while `Right.flatMap` simply returns the value that the function returns.

Finally, here is a `for` comprehension that uses `Either`s:

```
// src/script/scala/progscala3/forcomps/ForEithersSeq.scala

scala> val seq: Seq[Either[RuntimeException,Int]] =
     |   Vector(Right(10), Left(RuntimeException("boo!")), Right(20))
     |
     | val results3 = for
     |   case Right(i) <- seq
     | yield 2 * i
val results3: Seq[Int] = Vector(20, 40)
```

Throwing exceptions versus returning either values

Just as `Either` encourages handling of errors as normal return values, avoiding thrown exceptions is also valuable for uniform handling of errors and normal return types. Thrown exceptions violate referential transparency; you can't replace the function invocation with a "value"! To see this, consider the following contrived example:

```
// src/script/scala/progscala3/forcomps/RefTransparency.scala

scala> def addInts(s1: String, s2: String): Int = s1.toInt + s2.toInt

scala> def addInts2(s1: String, s2: String): Either[String,Int] =
     |   try
     |     Right(s1.toInt + s2.toInt)
     |   catch
     |     case nfe: NumberFormatException => Left("NFE: "+nfe.getMessage)

scala> val add12a = addInts("1", "2")
     | val add12b = addInts2("1", "2")
val add12a: Int = 3
val add12b: Either[String, Int] = Right(3)

scala> val add1x  = addInts2("1", "x")
     | val addx2  = addInts2("x", "2")
     | val addxy  = addInts2("x", "y")
val add1x: Either[String, Int] = Left(NFE: For input string: "x")
```

```
val addx2: Either[String, Int] = Left(NFE: For input string: "x")
val addxy: Either[String, Int] = Left(NFE: For input string: "x")
```

We would like to believe that addInts is referentially transparent, so we could replace calls to it with values from a cache of previous invocations, for example. However, addInts will throw an exception if we pass a String that can't be parsed as an Int. Hence, we can't replace the function call with values that can be returned for all parameter lists.

Also, the type signature of addInts provides no indication that trouble lurks.

Using Either as the return type of addInts2 restores referential transparency, and the type signature is explicit about potential errors. It is referentially transparent because we could replace all calls with a value, even using Lefts for bad string input.

Also, instead of grabbing control of the call stack by throwing the exception, we've *reified* the error by returning the exception as a Left value.

So Either lets us maintain control of calling the stack in the event of a wide class of failures. It also makes the behavior more explicit to users of your APIs, through type signatures.

However, look at the implementation of addInts2 again. Handling exceptions is quite common, so the try…catch… boilerplate shown appears a lot in code.

So for handling exceptions, we should encapsulate this boilerplate with types and use names for these types that express more clearly when we have either a failure or a success. While Either does that for the general case, the Try type does that for the special case where the error is an exception.

Try: When There Is No Do

When failure is caused by an exception, use scala.util.Try. It is structurally similar to Either. It is a sealed abstract class with two subtypes, Success and Failure.

Success is analogous to the conventional use of Right. It holds the normal return value. Failure is analogous to Left, but Failure always holds a Throwable, which is why Try has one type parameter, for the value held by Success.

Here are the signatures of the three Try types (omitting some unimportant details):

```
sealed abstract class Try[+T] extends AnyRef {...}
final case class Success[+T](value: T) extends Try[T] {...}
final case class Failure[+T](exception: Throwable) extends Try[T] {...}
```

Try is clearly asymmetric, unlike Either, where the asymmetry isn't clear from the type signature.

Let's see how Try is used, again porting our previous example. First, if you have a list of Try values and just want to discard the Failures, a simple for comprehension does the trick:

```
// src/script/scala/progscala3/forcomps/ForTries.scala

scala> import scala.util.{Try, Success, Failure}

scala> def positiveTries(i: Int): Try[Int] = Try {
     |   assert (i > 0, s"nonpositive number $i")
     |   i
     | }

scala> val result4 = for
     |   i1 <- positiveTries(5)
     |   i2 <- positiveTries(10 * i1)
     |   i3 <- positiveTries(25 * i2)
     |   i4 <- positiveTries(2  * i3)
     | yield (i1 + i2 + i3 + i4)
val result4: scala.util.Try[Int] = Success(3805)

scala> val result5 = for
     |   i1 <- positiveTries(5)
     |   i2 <- positiveTries(-1 * i1)      // FAIL!
     |   i3 <- positiveTries(25 * i2)
     |   i4 <- positiveTries(-2 * i3)
     | yield (i1 + i2 + i3 + i4)
     |
val result5: scala.util.Try[Int] =
  Failure(java.lang.AssertionError: assertion failed: nonpositive number -5)
```

Note the concise definition of positiveTries. If the assertion fails, the Try block will catch the thrown java.lang.AssertionError and return a Failure wrapping it. Otherwise, the result of the Try expression is wrapped in a Success.

The for comprehensions look exactly like those for the original Option example. With type inference, there is very little boilerplate here too. You can focus on the "happy path" logic and let Try capture errors.

When striving to write pure functions and methods, a thrown exception breaks referential transparency because you are no longer always returning something. Instead, the flow of control jumps one or more stack frames, until the exception is caught. Furthermore, the return-type signature doesn't cover all cases now! Try reifies exceptions. The normal return mechanism is always used, but the value could either be a successful result or a Throwable. Referential transparency is preserved, as you can even substitute the Throwable that is returned for an invocation with bad input. The return type is correct too.

Returning `Try` also forces you to think carefully about your design, rather than just give up and throw an exception, hoping someone will catch it and know what to do with it. Can you prevent possible errors in the first place? If not, can you handle the exception locally? If not, should you move the code somewhere else where better handling is possible?

> Avoid throwing exceptions. Return a `Try` instead.

Validated from the Cats Library

While using `Option`, `Either`, or `Try` meets most needs, there is one common scenario where using any of them remains tedious. Consider the case of form validation, where a user submits a form with several fields, all of which need to be validated. Ideally, we would validate all at once and report all errors, rather than doing one at a time, which is not a friendly user experience. Using `Option`, `Either`, or `Try` in a `for` comprehension doesn't support this need because processing is short-circuited as soon as a failure occurs. This is where `cats.datatypes.Validated` from the Cats library provides several useful approaches.

We'll consider one approach here. First, start with some domain-specific classes:

```scala
// src/main/scala/progscala3/forcomps/LoginFormValidation.scala

package progscala3.forcomps

case class ValidLoginForm(userName: String, password: String)          ❶

sealed trait LoginValidation:                                          ❷
  def error: String

case class Empty(name: String) extends LoginValidation:
  val error: String = s"The $name field can't be empty"

case class TooShort(name: String, n: Int) extends LoginValidation:
  val error: String = s"The $name field must have at least $n characters"

case class BadCharacters(name: String) extends LoginValidation:
  val error: String = s"The $name field has invalid characters"
```

❶ A case class with the form fields to validate.

❷ A sealed trait used by the case classes that encapsulate the possible errors.

Now we use them in the following code, where the acronym `Nec` stands for *nonempty chain*. In this context, that means that a failed validation will have a sequence (chain) of one or more error objects:

```scala
// src/main/scala/progscala3/forcomps/LoginFormValidatorNec.scala
package progscala3.forcomps

import cats.implicits.*
import cats.data.*
import cats.data.Validated.*

/**
 * Nec variant, where NEC stands for "non empty chain".
 * @see https://typelevel.org/cats/datatypes/validated.html
 */
object LoginFormValidatorNec:

  type V[T] = ValidatedNec[LoginValidation, T]                          ❶

  def nonEmpty(field: String, name: String): V[String] =                ❷
    if field.length > 0 then field.validNec
    else Empty(name).invalidNec

  def notTooShort(field: String, name: String, n: Int): V[String] =
    if field.length >= n then field.validNec
    else TooShort(name, n).invalidNec

  /** For simplicity, just disallow whitespace. */
  def goodCharacters(field: String, name: String): V[String] =
    val re = raw".*\s..*".r
    if re.matches(field) == false then field.validNec
    else BadCharacters(name).invalidNec

  def apply(                                                            ❸
      userName: String, password: String): V[ValidLoginForm] =
    (nonEmpty(userName, "user name"),
    notTooShort(userName, "user name", 5),
    goodCharacters(userName, "user name"),
    nonEmpty(password, "password"),
    notTooShort(password, "password", 5),
    goodCharacters(password, "password")).mapN {
      case (s1, _, _, s2, _, _) => ValidLoginForm(s1, s2)
    }
end LoginFormValidatorNec

@main def TryLoginFormValidatorNec =
  import LoginFormValidatorNec.*
  assert(LoginFormValidatorNec("", "") ==
    Invalid(Chain(
      Empty("user name"), TooShort("user name", 5),
      Empty("password"), TooShort("password", 5))))
```

```scala
  assert(LoginFormValidatorNec("1234", "6789") ==
    Invalid(Chain(
      TooShort("user name", 5),
      TooShort("password", 5))))

  assert(LoginFormValidatorNec("12345", "") ==
    Invalid(Chain(
      Empty("password"), TooShort("password", 5))))

  assert(LoginFormValidatorNec("123 45", "678 90") ==
    Invalid(Chain(
      BadCharacters("user name"), BadCharacters("password"))))

  assert(LoginFormValidatorNec("12345", "67890") ==
    Valid(ValidLoginForm("12345", "67890")))
end TryLoginFormValidatorNec
```

❶ Shorthand type alias. `ValidationNec` will encapsulate errors or successful results.

❷ Several functions to test that fields meet desired criteria. When successful, an appropriate `ValidationNec` is constructed by calling either of the extension methods on `String`, `validNec`, or `invalidNec`.

❸ The `apply` method uses a Cats function `mapN` for mapping over the `N` elements of a tuple. It returns a final `ValidationNec` instance with all the accumulated errors in an `Invalid(Chain(...))`, or if all validation criteria were met, a `Valid(ValidLoginForm(...))` holding the passed-in field values.

For comparison, see also in the example code *src/main/scala/progscala3/forcomps/LoginFormValidatorSingle.scala*, which handles single failures using `Either`, but following a similar implementation approach.

Without a tool like Cats `Validated`, we would have to manage the chain of errors ourselves.

Recap and What's Next

`Either`, `Try`, and `Validated` express through types a fuller picture of how the program actually behaves. All three say that a valid value or values will (hopefully) be returned, but if not, they also encapsulate the failure information needed. Similarly, `Option` encapsulates the presence or absence of a value explicitly in the type signature.

Using these types instead of thrown exceptions keeps control of the call stack, signals to the reader the kinds of errors that might occur, and allows error conditions to be less exceptional and more amenable to programmatic handling, just like the "happy path" scenarios.

Another benefit we haven't mentioned yet is a benefit for asynchronous (concurrent) code. Because asynchronous code isn't guaranteed to be running on the same thread as the caller, it might not be possible to catch and handle an exception. However, by returning errors the same way normal results are returned, the caller can more easily intercept and handle the problem. We'll explore the details in Chapter 19.

You probably expected this chapter to be a perfunctory explanation of Scala's fancy `for` loops. Instead, we broke through the facade to find a surprisingly powerful set of tools. We saw how a set of functions, `map`, `flatMap`, `foreach`, and `withFilter`, plug into `for` comprehensions to provide concise, flexible, yet powerful tools for building nontrivial application logic.

We saw how to use `for` comprehensions to work with collections, but we also saw how useful they are for other container types, specifically `Option`, `Either`, `Try`, and Cats `Validated`.

Now we have finished our exploration of the essential parts of FP and their support in Scala. We'll learn more concepts when we discuss the type system in Chapter 16 and Chapter 17 and explore advanced concepts in Chapter 18.

Let's now turn to Scala's support for OOP. We've already covered many of the details in passing. Now we'll complete the picture.

CHAPTER 9

Object-Oriented Programming in Scala

One reason Scala is a superb OOP language is because Martin Odersky and his collaborators have thought long and hard about how to make OOP best practices as concise as possible. While we already know many of Scala's features for OOP, now we will explore them more systematically. We'll see more examples of Scala's concise syntax and how it enables effective OOP in combination with FP.

I've waited until now to explore Scala as an OOP language for two reasons.

First, I wanted to emphasize that FP has become an essential skill set for modern problems, a skill set that may be new to you. When you start with Scala, it's easy to use it as a better OOP language, a "better Java," and neglect the power of its FP side.

Second, a common architectural approach with Scala has been to use FP for *programming in the small* and OOP for *programming in the large*. Using FP for implementing algorithms, manipulating data, and managing state in a principled way is our best way to minimize bugs, the amount of code we write, and the risk of schedule delays. On the other hand, Scala's OOP model provides tools for designing composable, reusable, and encapsulated modules, which are essential for building larger applications. Hence, Scala gives us the best of both worlds.

I've assumed you already know the basics of OOP from other languages, so many concepts were defined quickly and informally throughout the book. This chapter starts with a quick review of class and object basics, then fills in the details, such as the mechanics of creating type hierarchies, how constructors work for Scala classes, and runtime-efficient types using *opaque type aliases* (new to Scala 3) and value classes. The next chapter will dive into traits, and then we'll spend a few chapters filling in additional details on Scala's object model and the standard library.

Class and Object Basics: Review

Classes are declared with the `class` keyword, while singleton objects are declared with the `object` keyword. For this reason, I have used the term *instance* in this book to refer to objects generically, whether they are class instances or declared objects. In most OOP languages, *instance* and *object* are synonymous.

An instance can refer to itself using the `this` keyword, although it's somewhat rare in Scala code. One reason is that constructor boilerplate is absent in Scala. Consider the following Java code:

```
// src/main/java/progscala3/basicoop/JavaPerson.java
package progscala3.basicoop;

public class JavaPerson {
  private String name;
  private int    age;

  public JavaPerson(String name, int age) {
    this.name = name;
    this.age  = age;
  }

  public void   setName(String name) { this.name = name; }
  public String getName()            { return this.name; }

  public void setAge(int age) { this.age = age;  }
  public int  getAge()        { return this.age; }
}
```

Other OOP languages are similar. Now compare it with the following equivalent Scala declaration, in which all the boilerplate disappears:

```
class Person(var name: String, var age: Int)
```

Prefixing a constructor parameter with a `var` makes it a mutable *field* of the class, also called an *instance variable* or *attribute* in other languages. Prefixing a constructor parameter with a `val` makes it an immutable field. Using the `case` keyword infers the `val` keyword and also adds additional methods, as we've learned:

```
case class Person(name: String, age: Int)
```

This is just one example of how concise OOP can be in Scala. You can also declare other `val` and `var` fields inside the type body.

The term *member* refers to a field, method, or type in a generic way. The term *method* refers to a function that is tied to an instance. Its parameter list has an implied `this`. Method definitions start with the `def` keyword.

Scala allows overloaded methods. Two or more methods can have the same name as long as their full signatures are unique. The signature includes the enclosing type name, method name, and the types of all the parameters. The parameter names are not part of the signature for typing purposes, but they are significant because you can provide them when calling the method—e.g., `log(message = "Error!")`. Also, different return types alone are not sufficient to distinguish methods.

> In Scala 2, only the parameters in the first parameter list were considered when determining the method signature for the purposes of overloading. In Scala 3, all parameter lists are considered.

Member types are declared using the `type` keyword. They are used to provide shorter names for complex types and to provide a complementary mechanism to type parameterization (see "Parameterized Types Versus Abstract Type Members" on page 66).

A field and method can have the same name, but only if the method has a parameter list:

```
scala> trait Good:
     |   def x(suffix: String): String
     |   val x: String

scala>
     | trait Bad:
     |   def x: String
     |   val x: String
4 |  val x: String
  |      ^
  |      Double definition...
```

Open Versus Closed Types

Scala encourages us to think carefully about what types should be abstract versus concrete, what types should be singletons, what types should be mixins, and what types should be open versus closed for extension, meaning allowed to be subtyped or not. OOP languages also use the terms *subclassing* or *deriving* from a supertype. I've used subtyping to emphasize the more general type system in Scala's combination of FP and OOP.

Mixins promote *composition over inheritance*, discussed in "Good Object-Oriented Design: A Digression" on page 265. Traits are used to define mixins, while abstract classes or traits are used as base classes in a hierarchy. Here is a sketch of a hierarchy of services with logging mixed in. First, define a basic logging trait:

```scala
// src/main/scala/progscala3/basicoop/Abstract.scala
package progscala3.basicoop

enum LoggingLevel:                                                    ❶
  case Info, Warn, Error

trait Logging:
  import LoggingLevel.*
  final def info(message: String): Unit = log(Info, message)
  final def warn(message: String): Unit = log(Warn, message)
  final def error(message: String): Unit = log(Error, message)
  final def log(level: LoggingLevel, message: String): Unit =
    write(s"${level.toString.toUpperCase}: $message")

  protected val write: String => Unit                                 ❷

trait ConsoleLogging extends Logging:
  protected val write = println                                       ❸
```

❶ Define a simple logging abstraction with three levels.

❷ Do as much as possible in the `Logging` mixin trait. The protected abstract function value `write` is implemented by subtypes for actually writing to the log. Every other method is declared `final` to prevent overriding.

❸ `ConsoleLogger` just uses `println`.

Now, define an abstract base class for services that mixin logging:

```scala
abstract class Service(val name: String) extends Logging:             ❶
  import Service.*
  final def handle(request: Request): Response =
    info(s"($name) Starting handle for request: $request")
    val result = process(request)
    info(s"($name) Finishing handle with result $result")
    result

  protected def process(request: Request): Response

object Service:                                                       ❷
  type Request = Map[String,Any]
  type Response = Either[String,Map[String,Any]]

open class HelloService(override val name: String)                    ❸
    extends Service(name) with ConsoleLogging:
  import Service.*

  protected def process(request: Request): Response =
    request.get("user") match
      case Some(user) => Right(Map("message" -> s"Hello, $user"))
      case None => Left("No user field found!")
```

1. A service abstraction. It uses the `Logging` trait as the supertype because there isn't another supertype (other than `AnyRef`), but really this is mixin composition. Concrete subtypes of `Service` must implement `write` themselves or mixin a subtrait of `Logging` that does this. The way `handle` is implemented is discussed later on.

2. Define convenient `type` aliases for `Request` and `Response`. The choices are inspired by typical web service APIs, where maps of key-value pairs are often used. Note that errors are handled by returning an `Either`. The logging of service requests is handled in the final method `handle`, so users of this trait only need to worry about the specific logic of processing a request, by defining the protected method `process`.

3. A concrete class that extends `Service` and mixes in the implementation trait `ConsoleLogging`. The `process` method expects to find a key `user` in the map. The `open` keyword is discussed later on.

Finally, an entry point:

```
@main def HelloServiceMain(name: String, users: String*): Unit =
  val hs = HelloService("hello")
  for
    user <- users
    request = Map("user" -> user)
  do hs.handle(request) match
    case Left(error) => println(s"ERROR! $error")
    case Right(map) => println(map("message"))

  println("Try an empty map:")
  println(hs.handle(Map.empty))
```

Let's run `HelloServiceMain` in `sbt`:

```
> runMain progscala3.basicoop.HelloServiceMain hello Dean Buck
...
INFO: (hello) Starting handle for request: Map(user -> Dean)
INFO: (hello) Finishing handle with result Right(Map(message -> Hello, Dean))
Hello, Dean
INFO: (hello) Starting handle for request: Map(user -> Buck)
INFO: (hello) Finishing handle with result Right(Map(message -> Hello, Buck))
Hello, Buck
Try an empty map:
INFO: (hello) Starting handle for request: Map()
INFO: (hello) Finishing handle with result Left(No user field found!)
Left(No user field found!)
[success] Total time: 1 s, completed Aug 21, 2020, 1:37:05 PM
```

Let's explore the key ideas in this example.

Classes Open for Extension

3 Scala 2 constrained subtyping when a type was declared `final` or an abstract supertype was declared `sealed`, but otherwise you could create subtypes of concrete types. Scala 3 tightens the default rules by adding the `open` keyword. It says that it is permissible to define a subtype from this concrete type. Without this keyword, the compiler now issues a warning when a subtype is created from a concrete type.

In the preceding `HelloService`, we left open the possibility that someone might want to use `HelloService` as a supertype for another `Service`. As a practical matter, the Scaladoc for such a type should provide details about how to extend the type, including what *not* to do. Whether or not `open` is used is a deliberate design decision.

There are two exceptions to the rule that `open` is now required for extension:

1. Subtypes in the same source file, like how `sealed` hierarchies work.
2. Use of the `adhocExtensions` language feature.

The `adhocExtensions` language feature is enabled globally by adding the compiler flag, `-language:adhocExtensions`, or in single scopes using `import scala.language.adhocExtensions`.

Because this is a breaking change, it is being introduced gradually. In Scala 3.0, the feature warning is only emitted when you compile with `-source:future`. The warning will occur by default in a subsequent Scala 3 release.

A type that is neither `open` nor `final` now has similar subtyping behavior as a `sealed` type. The difference is that you can still subtype a closed type by enabling the language feature, while `sealed` type hierarchies can't be reopened. An important example of this advantage is when you need a test double in a unit test, where you create a subtype to stub out certain members. The test source file imports the language feature to enable this subtyping, but subtyping is disallowed in the rest of the code.

As a rule, I try to use only abstract types as supertypes and treat all concrete types as final, except for the testing scenario. The main reason for this rule is because it's difficult to get the semantics and implementations correct for `hashCode`, `equals`, and user-defined members. For example, if `Manager` is a subtype of `Employee` (assuming this is a good design), when should a `Manager` instance be considered equal to an `Employee` instance? If the common subset of fields is equal, is that good enough? The answer depends on the context and other factors. This is one reason why Scala simply prohibits case classes from being subtypes of other case classes.

A few other points. First, `open` is a soft modifier, meaning it can be used as a normal identifier when it's not in the modifier position. Hence, you don't need to rename all your `open` methods! Second, `open` can't be used with `final` or `sealed` because that would be a contradiction. Finally, traits and abstract classes are by definition already open, so the keyword is redundant for them.

> Because composition is usually more robust than inheritance, use `open` rarely.

Overriding Methods? The Template Method Pattern

Notice how I declared and implemented the methods in the `Logging` and `Service` types previously.

Just as you should avoid subtyping concrete types, you should avoid overriding concrete methods. It is a common source of subtle behavioral bugs. Should the subtype implementation call the supertype method? If so, when should it call it: at the beginning or end of the overriding implementation? The correct answers depend on the context. It is too easy to make mistakes. Unfortunately, we are so accustomed to overriding the concrete `toString` method that we consider it normal practice. It should not be normal.

The preceding example uses the *template method pattern* ([GOF1995]) to eliminate the need to override concrete methods. The supertypes `Logging` and `Service` define `final`, concrete methods that are publicly visible. `Service.handle` is a template that calls abstract methods, which are the points of allowed variance for subtypes to define. The `Logger` concrete methods are simple templates. They just call the `protected`, abstract function `Logger.write`. It's easy to implement this correctly because it does only one thing: write a formatted string somewhere. `ConsoleLogger.write` writes the string to the console.

Similarly, `Service.handle` was implemented carefully to add logging while correctly handling the result of the computation. The `protected`, abstract method `Service.process` is *agnostic* to logging. It focuses on processing the request.

However, we can't completely eliminate overriding concrete methods, like `toString`. Fortunately, Scala requires the `override` keyword, which you should treat as a reminder to be careful. When you need to call a supertype method `foo`, use `super().foo(…)`. See also "Self-Type Declarations" on page 382 for handling the special case when multiple supertypes implement the same method and you need a way to specify a particular one of them.

Avoid overriding concrete methods, except when replacing default implementations like `toString`, mixing in *orthogonal* behavior, or *stubbing* for tests. Be careful to preserve the contract of the method. Otherwise, use the template method pattern to provide extensibility without overrides. To prevent overrides, declare methods final.

Reference Versus Value Types

While Scala is now a cross-platform language, its roots as a JVM language are reflected in the top-level types.

For performance reasons, the JVM implements a set of special primitive types: `short`, `int`, `long`, `float`, `double`, `boolean`, `char`, and `byte`. When used by themselves, meaning not enclosed in other objects, they can be used without the overhead of allocating space for them on the heap and reading and writing the memory location. For example, the compiler-generated byte code or runtime processing could push these values directly onto stack frames or store them in CPU registers. Arrays of these values require only one heap allocation, for the array itself. The values can be inlined in the array. These primitives are called *value types* because the byte code works with these values directly.

All other types are called *reference types* because all instances of them are allocated on the heap, and variables for these instances refer to the corresponding heap locations.

Scala source code blurs this distinction between primitives and reference types to provide a more consistent programming model, but without sacrificing performance where possible.

In Scala, all reference types are subtypes of `scala.AnyRef` on the JVM and `js.Object` in Scala.js. `AnyRef` is a subtype of `Any`, the root of the Scala type hierarchy. For Scala.js, `js.Any` is the equivalent supertype of `js.Object`. Note that Java's root type, `Object`, is actually equivalent to `AnyRef`, not `Any`. You will sometimes see documentation refer to `Object` instead of `AnyRef`, but it can be confusing to see them used interchangeably. I've used `AnyRef` in this book, but keep in mind that you'll see both in documentation.

The Scala types `Short`, `Int`, `Long`, `Float`, `Double`, `Boolean`, `Char`, `Byte`, and `Unit` are value types. They correspond to the JVM primitives `short`, `int`, `long`, `float`, `double`, `boolean`, `char`, `byte`, and the `void` keyword. All value types are subtypes of `scala.AnyVal`, which is also a subtype of `Any`. For Scala.js, the JavaScript primitives are used, including `String`, with a rough correspondence to the `AnyVal` types. In the Java and JavaScript object models, primitives don't have a common supertype.

To avoid confusion, I have used `Any`, `AnyRef`, and `AnyVal` consistently with a bias toward the JVM implementations. See the Scala.js Type Correspondence guide for more details about Scala.js types. The Scala Native documentation discusses its handling of Scala types.

`Unit` is an `AnyVal` type, but it involves no storage at all. Loosely speaking, `Unit` is analogous to the `void` keyword in many languages. This is only true in the sense that a method returning `Unit` or `void` doesn't return anything you can use. Otherwise, `Unit` or `void` are quite different. While `void` is a keyword, `Unit` is a real type with one literal value, `()`, and we rarely use that value explicitly. This means that all functions and methods in Scala return a value, whereas languages with `void` have a separate idea of functions that return a value and procedures that don't.

Why Is Unit's Literal Value ()?

3 Unit really behaves like a tuple with zero elements, written as `()`.[1] If there are no elements, it contains no useful information. The name *unit* comes from algebra, where adding the unit to any value returns the original value, such as 0 for integers. For multiplication, 1 is the unit value. So if I add `()` to `(1, 2.2)`, I get back `(1, 2.2)`, but if I add `(3L)` to `(1, 2.2)`, I get back `(3L, 1, 2.2)`, or `(1, 2.2, 3L)`. We'll explore this idea more precisely in "Algebraic Data Types" on page 397.

As an aside, consider a sequence of `AnyVal`s. What is the least upper bound? For the special case where a sequence of numbers contains `Float`s and `Int`s, like `Seq(1, 2.2F, 3)`, the inferred type is `Seq[Float]`. The `Int`s are converted to `Float`s. Similarly if `Int`s and `Double`s are mixed, you get `Seq[Double]`. For all other combinations of `AnyVal`s, the inferred type is `Seq[AnyVal]`, even when all the values are `Double`s and `Float`s.

Opaque Types and Value Classes

In "Scala 3 Implicit Conversions" on page 154, we defined some wrapper types for `Dollars` and `Percentage`s:

```
// src/main/scala/progscala3/contexts/accounting/NewImplicitConversions.scala
package progscala3.contexts.accounting
import scala.language.implicitConversions

case class Dollars(amount: Double):
  ...
case class Percentage(amount: Double):
  ...
```

1 Scala 3 adds an actual `EmptyTuple` type, which is different than `Unit`.

Now imagine you are writing a big-data application that creates millions or more instances of these types. The extra overhead of heap allocations and memory accesses for these wrapper types becomes very expensive. They can be quite stressful for the garbage collector. Fundamentally, these types just wrap `Doubles`, so we would prefer to keep the efficiency of primitive `doubles`, without giving up the convenience of object orientation and custom types.

Let's consider three potential solutions to this issue: regular member types, opaque types, and value classes. First, we could define member type aliases for `Dollars` and `Percentage`:

```
// src/script/scala/progscala3/basicoop/DollarsPercentagesTypes.scala

object Accounting:
  type Dollars = Double
  type Percentage = Double

import Accounting.*
case class Salary(gross: Dollars, taxes: Percentage):
  def net: Dollars = gross * (1.0 - (taxes/100.0))
  override def toString =
    f"Salary(gross = $$$gross%.2f, taxes = $taxes%.2f%%)"
```

Now let's try it:

```
scala> import Accounting.*
     | val gross: Dollars = 10000.0
     | val taxes: Percentage = 0.1
val gross: Accounting.Dollars = 10000.0
val taxes: Accounting.Percentage = 0.1

scala> val salary1 = Salary(gross, taxes)
     | val net1 = salary1.net
val salary1: Salary = Salary(gross = $10000.00, taxes = 10.00%)
val net1: Accounting.Dollars = 9000.0

scala> val salary2 = Salary(taxes, gross)   // Error, but it compiles!!
     | val net2 = salary2.net
val salary2: Salary = Salary(gross = $0.10, taxes = 1000000.00%)
val net2: Accounting.Dollars = -999.9000000000001
```

This is a simple solution, but it has some problems that impact users of the API. The type aliases are just new names for the same type, so the compiler doesn't catch the mistake of mixing up the arguments. Hence, type aliases provide no additional type safety. Furthermore, we can't define custom methods for `Dollars` and `Percentage`. Attempting to use extension methods will add them to `Double`, not separately for the two types.

Opaque Type Aliases

Scala 3 introduces *opaque type aliases*, which are declared like regular member type aliases, but with the `opaque` keyword. They preserve type safety and have zero runtime overhead beyond the value they wrap, but they provide some of the benefits of using richer types. Here is the same example rewritten with opaque type aliases:

```
// src/script/scala/progscala3/basicoop/DollarsPercentagesOpaque.scala

object Accounting:
  opaque type Dollars = Double                                    ❶
  opaque type Percentage = Double

  object Dollars:                                                 ❷
    def apply(amount: Double): Dollars = amount

    extension (d: Dollars)
      def +(d2: Dollars): Dollars = d + d2
      def -(d2: Dollars): Dollars = d - d2
      def *(p: Percentage): Dollars = d*p
      def toDouble: Double = d
      // override def toString = f"$$$d%.2f"                        ❸
      // override def equals(other: AnyRef): Boolean = ???

  object Percentage:                                              ❹
    def apply(amount: Double): Percentage = amount

    extension (p: Percentage)
      def +(p2: Percentage): Percentage = p + p2
      def -(p2: Percentage): Percentage = p - p2
      def *(d: Dollars): Dollars = d*p
      def toDouble: Double = p
      // override def toString = f"${(p*100.0)}%.2f%%"
      // override def equals(other: AnyRef): Boolean = ???

import Accounting.*
case class Salary(gross: Dollars, taxes: Percentage):             ❺
  def net: Dollars = gross - (gross * taxes)
```

❶ Like regular member type aliases, but prefixed with the `opaque` keyword.

❷ For each opaque type alias, define an object that looks like a companion object for factory methods like `apply`, so they behave similar to user expectations for other types. All instance methods for an opaque type are defined as extension methods.

❸ Our original wrapper types for `Dollars` and `Percentage` had nice `toString` methods, and we could have implemented `equals` methods that incorporate accounting rules. We don't have the option to override concrete methods for opaque type aliases.

❹ The companion for `Percentage`.

❺ A case class that uses these types.

The `opaque` keyword is soft. It is only treated as a keyword in a declaration as shown. Otherwise, it can be used as a regular identifier.

Compared to case classes, opaque type aliases have most of the limitations of regular member type aliases. You can't override concrete methods like `equals` and `toString` for opaque type aliases, nor pattern match on them. You can only pattern match on the underlying type.

You can define an object with the same name for factory methods like `apply`, but they aren't generated automatically like they are for case classes.

Note that regular `Double` methods, like those for arithmetic and comparisons, are not automatically available for users of these types. We have to define extension methods for the operations we want or call `toDouble` first.

However, this only applies for users of an opaque type outside the scope where the type is defined. This is why they are called *opaque*. Inside the scope, the type looks like a `Double`. Note how the bodies are implemented using `Double` methods.

Outside the defining scope, an opaque type is considered abstract, even though the definition inside the scope is concrete. Note how `Dollars` and `Percentage` are used in the next code snippet. When we construct `Dollars`, we call the `Dollars` object method `apply`. Everywhere else, like arguments to `Salary`, there's nothing to require `Dollars` to be concrete, just like using `Seq` everywhere, even though it is not a concrete type:

```
scala> import Accounting.*
scala> val gross = Dollars(10000.0)
     | val taxes = Percentage(0.1)
val gross: Accounting.Dollars = 10000.0
val taxes: Accounting.Percentage = 0.1

scala> val salary1 = Salary(gross, taxes)
     | val net1 = salary1.net
val salary1: Salary = Salary(10000.0,0.1)
val net1: Accounting.Dollars = 9000.0

scala> val salary2 = Salary(taxes, gross)   // Won't compile!
5 |val salary2 = Salary(taxes, gross)   // Won't compile!
  |                     ^^^^^
  |                     Found:    (taxes : Accounting.Percentage)
  |                     Required: Accounting.Dollars
```

```
5 |val salary2 = Salary(taxes, gross)   // Won't compile!
  |                              ^^^^^
  |                              Found:    (gross : Accounting.Dollars)
  |                              Required: Accounting.Percentage
```

When printing the values, we no longer have the dollar and percentage signs. If we still want them, we would have to implement an ad hoc `print` method of some kind and use that instead of relying on `toString`.

However, as desired, we get the type checking for `Dollars` versus `Percentages` that we want, and we don't pay a runtime penalty for wrapping these types.

The inability to override the equality methods–`equals`, `==`, and `!=`–means the underlying type's methods are used. This can cause surprises:

```
scala> val dollars = Dollars(100.0)
     | val percentage = Percentage(100.0)

scala> dollars == percentage
val res0: Boolean = true
```

However, you can define extension methods for `<=`, `>=`, etc.

> When comparing instances of different opaque types that alias the same underlying type, the underlying type's equality operations are used, even if the instances should not be considered comparable! Avoid using these methods. Define other, ad hoc extension methods to fine-tune the behavior and use them instead.

At compile time, opaque types work like regular types, but the byte code generated only uses the overhead of the aliased type, `Double` in this case. For all three approaches we are discussing here, you aren't limited to aliasing `AnyVal` types either. Even wrapping `Strings`, for example, means one less heap allocation and fewer memory accesses.

Opaque types can also have type parameters. The code examples contain two variations of an example that I won't discuss here. They are adapted from an example in the *Scala Improvement Process* proposal for opaque types, SIP-35, which shows a no-overhead way to *tag* values with metadata. In this case, units like meters versus feet are the tags. The implementations are a bit advanced, but worth studying to appreciate both the idea of tagging data with metadata in a type-safe way and how it is implemented with no runtime overhead. See *src/main/scala/progscala3/basicoop/tagging/Tags.scala* and *Tags2.scala*. SIP-35 contains other nontrivial examples too.[2]

2 Some of the SIP-35 details are obsolete. The Scala 3 documentation is correct.

Opaque type aliases and matchable

3 In “Safer Pattern Matching with Matchable” on page 105, we saw that pattern matching is now restricted to subtypes of the trait `Matchable`, which fixes a “hole” when pattern matching on certain type aliases like the new `IArray`. I explained that pattern matching on abstract types requires them to be bound by `Matchable`, which solves the issue with `IArray` discussed there. In fact, the library’s `IArray` is an opaque type alias for `Array`, so now I can fill in a few more details. Consider the following example with our own `Array` aliases:

```
scala> // src/script/scala/progscala3/basicoop/MatchableOpaque.scala
     |
     | object Obj:
     |   type Arr[T] = Array[T]
     |   opaque type OArr[T] = Array[T]

scala> summon[Obj.Arr[Int] <:< Matchable]     // Okay
val res2: Array[Int] =:= Array[Int] = generalized constraint

scala> summon[Obj.OArr[Int] <:< Matchable]    // Doesn't work
1 |summon[Obj.OArr[Int] <:< Matchable]    // Doesn't work
  |                                   ^
  |                       Cannot prove that Obj.OArr[Int] <:< Matchable.
```

Recall that we used `summon` like this in “Implicit Evidence” on page 178 to check type relationships. So why is it that our type alias `Arr` is considered a `Matchable`, but not our `opaque` alias `OArr`? It’s because `OArr` is considered abstract outside of `Obj`, as we discussed earlier. Recall from the discussion of `Matchable` that abstract types must be declared bounded by `Matchable`. In contrast, `Arr` is not abstract and it aliases the concrete type `Array`, which subtypes `Matchable`.

If you change the definition of `OArr` to `opaque type OArr[T] <: Matchable = Array[T]`, the last `summon` will succeed. Try it!

Value Classes

Scala 2 and 3 offer *value classes*, our third and final mechanism to eliminate the runtime overhead of wrapper types. They have some advantages and drawbacks compared to opaque type aliases.

Here is an example value class for North American phone numbers (excluding the country code):

```
// src/main/scala/progscala3/basicoop/ValueClassPhoneNumber.scala
package progscala3.basicoop

class NAPhoneNumber(val s: String) extends AnyVal:        ❶
  override def toString =
    val digs = digits(s)
```

```scala
    val areaCode  = digs.substring(0,3)
    val exchange  = digs.substring(3,6)
    val subnumber = digs.substring(6,10)  // "subscriber number"
    s"($areaCode) $exchange-$subnumber"

  private def digits(str: String): String = str.replaceAll("""\D""", "")
```

❶ Note that it extends `AnyVal`. For simplicity, validation of the input string is not shown.

Now we have the convenience of a domain-specific type, with customized methods, but instances don't require additional memory management beyond what the `String` requires.

To be a valid value class, the following rules must be followed:

- The value class has one and only one `val` parameter.
- The type of the parameter must not be a value class itself.
- The value class doesn't define secondary constructors (see "Constructors in Scala" on page 262).
- The value class defines only methods, but no other `val`s and no `var`s.
- The value class defines no nested `trait`s, `class`es, or `object`s.
- The value class can't be subtyped.
- The value class can only inherit from universal traits (more on that in a moment).
- The value class must be a top-level type or a member of an object that can be referenced.[3]

The compiler provides good error messages when we break the rules.

At compile time, the type is the outer type, `NAPhoneNumber`. The runtime type is the wrapped type, `String`. The wrapped type can be any other type, as long as the rules are followed.

A value class can be a `case` class, but the many extra methods and the companion object generated are less likely to be used and hence more likely to waste space in the output class file.

A *universal trait* has the following properties:

3 Because of Scala's richer type system, not all types can be referenced in normal variable and method declarations. (However, all the examples we've seen so far work fine.) In Chapter 16, we'll explore new kinds of types and learn the rules for what it means to say that a type can or can't be referenced.

- It subtypes Any (but not from other universal traits).
- It defines only methods.
- It does no initialization of its own.

Here is a refined version of NAPhoneNumber that mixes in two universal traits:

```
// src/main/scala/progscala3/basicoop/ValueClassUniversalTraits.scala
package progscala3.basicoop

trait Digitizer extends Any:
  def digits(s: String): String = s.replaceAll("""\D""", "")          ❶

trait Formatter extends Any:                                           ❷
  def format(
      areaCode: String, exchange: String, subnumber: String): String =
    s"($areaCode) $exchange-$subnumber"

case class NAPhoneNumberUT(s: String)
    extends AnyVal with Digitizer with Formatter:
  override def toString =
    val digs = digits(s)
    val areaCode = digs.substring(0,3)
    val exchange = digs.substring(3,6)
    val subnumber  = digs.substring(6,10)
    format(areaCode, exchange, subnumber)                              ❸
```

❶ Digitizer is a trait that contains the digits method we originally had in NAPhoneNumber.

❷ Formatter formats the phone number the way we want it.

❸ Use Formatter.format.

Formatter actually solves a design problem. We might like to specify a second parameter to NAPhoneNumber for a format string to use in toString because there are many popular format conventions for phone numbers. However, we're only allowed to pass one argument to the NAPhoneNumber constructor and it can't have any other fields. We can solve this problem by mixing in a universal trait to do the configuration we want. We could define a different Formatter trait and build a different PhoneNumber value class that uses it.

The biggest drawback of value classes is that some nonobvious circumstances can trigger instantiation of the wrapper type, defeating the purpose of using value classes. One situation involves universal traits. Here's a summary of the circumstances requiring instantiation:

- When a function expects a universal trait instance and it is passed an instance of a value class that implements the trait. However, if a function expects an instance of the value class itself, instantiation isn't required.
- An `Array` or another collection of value class instances.
- The type of a value class is used as a type parameter.

For example, when the following method is called with a `NAPhoneNumber`, an instance of it will be allocated on the heap:

```
def toDigits(d: Digitizer, str: String) = d.digits(str)
...
val digs = toDigits(NAPhoneNumber("987-654-3210"), "123-Hello!-456")
// Result: digs: String = 123456
```

Similarly, when the following parameterized method is passed a `NAPhoneNumber`:

```
def print[T](t: T) = println(t.toString)
print(NAPhoneNumber("987-654-3210"))
// Result: (987) 654-3210
```

Opaque type aliases work around these scenarios, although they bring their own limitations, like inability to override `toString` and `equals` for them.

> To clarify terminology, *value type* refers to the `Short`, `Int`, `Long`, `Float`, `Double`, `Boolean`, `Char`, `Byte`, and `Unit` types. *Value class* refers to user-defined classes that subtype `AnyVal`.

To summarize the three approaches for avoiding wrapper type overhead, regular member type aliases are a very simple approach, but don't provide any extra type safety. Both value classes and opaque type aliases provide better type safety and semantics. Use opaque types when performance is the highest priority, when avoiding all extra heap allocation and memory accesses is important. Use value classes if you want types that behave more like real types, such as the ability to override `toString` and `equals`, and you can tolerate some situations where heap allocation is necessary.

Supertypes

Throughout the book, I have mostly used the terms *supertype* and *subtype*, both as nouns and verbs. Subtyping creates a subtype from a supertype. Other common OOP terms for subtyping include *derivation*, *extension*, and *inheritance*. Scala documentation describes *type class derivation*, so I used that terminology in "Type Class Derivation" on page 158. The keywords `open` and *extension* are often used together, as I used them in "Classes Open for Extension" on page 250.

Supertypes are also called *parent* or *base* types. Subtypes are also called *child* or *derived* types.

Scala only supports *single inheritance*, but most of the benefits of *multiple inheritance* are achieved using mixins. All types have a supertype, except for the root of the Scala class hierarchy, `Any`. When declaring a type and when a parent type is omitted, the type implicitly subtypes `AnyRef`. In other words, it is automatically a reference type and instances will be heap allocated.

The keyword `extends` indicates the supertype class or trait. If other traits are mixed in, the `with` keyword is used.

Constructors in Scala

Scala distinguishes between the *primary constructor* and zero or more *auxiliary constructors*, also called *secondary constructors*. In Scala, the primary constructor is the entire body of the type. Any parameters that the constructor requires are listed after the type name.

Here is an example with auxiliary constructors. The type represents US zip codes, which have five digits and an optional extension of four digits:

```
// src/script/scala/progscala3/basicoop/people/ZipCodeAuxConstructors.scala

case class ZipCodeAuxCtor(zip: Int, extension: Int = 0):
  override def toString =
    if extension != 0 then s"$zip-$extension" else zip.toString

  def this(zip: String, extension: String) =
    this(zip.toInt, if extension.length == 0 then 0 else extension.toInt)
  def this(zip: String) = this(zip, "")
```

The two auxiliary constructors, named `this`, allow users to provide string arguments. They are converted to integers, where `0` is interpreted as "no extension." All auxiliary constructors are required to call the primary constructor or another auxiliary constructor as the first expression. The compiler also requires that a constructor called is one that appears earlier in the source code. So we must order secondary constructors carefully in our code.

Forcing all construction to go through the primary constructor eliminates duplication of constructor logic and the risk of inconsistent initialization of instances.

We haven't discussed auxiliary constructors before now because it's rare to use them. It's far more common to overload object `apply` methods instead when multiple invocation options are desired:

```
// src/script/scala/progscala3/basicoop/people/ZipCodeApply.scala
```

```
case class ZipCodeApply(zip: Int, extension: Int = 0):
  override def toString =
    if extension != 0 then s"$zip-$extension" else zip.toString

object ZipCodeApply:
  def apply(zip: String, extension: String): ZipCodeApply =
    apply(zip.toInt, if extension.length == 0 then 0 else extension.toInt)
  def apply(zip: String): ZipCodeApply = apply(zip, "")
```

This has the nice benefit of keeping the case class very simple. The complexity of invocation is moved to the companion object. There are actually three `apply` methods in the companion object. The compiler generates the `apply` method with two `Int` parameters, matching the constructor. The source code adds two more.

The following invocations all work:

```
ZipCodeApply(12345)
ZipCodeApply(12345, 6789)
ZipCodeApply("12345")
ZipCodeApply("12345", "6789")
```

Calling Supertype Constructors

The primary constructor in a subtype must invoke one of the supertype constructors:

```
class Person(name: String, age: Int)
class Employee(name: String, age: Int, salary: Float) extends Person(name, age)
class Manager(name: String, age: Int, salary: Float, minions: Seq[Employee])
  extends Employee(name, age, salary)
```

Export Clauses

In the next section, we'll discuss the benefits of composition over inheritance. There are times when you want a type to be composed of other types, but you also want members of those other types to be part of the public interface of the composed type. This happens automatically with public members of inherited types, including mixins. However, if dependencies are passed in as constructor arguments (also known as *dependency injection*) or local instances are created, and you would like some of the members of those instances to be part of the composed type's interface, you have to write *forwarding* methods. To see this, consider these types for authentication that some service might want to provide:

```
// src/script/scala/progscala3/basicoop/Exports.scala

import java.net.URL

case class UserName(value: String):                                    ❶
  assert(value.length > 0)
case class Password(value: String):
  assert(value.length > 0)
```

```scala
trait Authenticate:                                                    ❷
  final def apply(
      username: UserName, password: Password): Boolean =
    authenticated = auth(username, password)
    authenticated
  def isAuthenticated: Boolean = authenticated

  private var authenticated = false
  protected def auth(username: UserName, password: Password): Boolean

class DirectoryAuthenticate(location: URL) extends Authenticate:
  protected def auth(username: UserName, password: Password): Boolean = true
```

❶ Types to encapsulate valid usernames and passwords (encryption needed!).

❷ An abstraction for authentication with a stub implementation that uses a directory service available at some URL.

Now let's compose a service that declares a private field for an instance of `DirectoryAuthenticate`, then defines boilerplate methods to expose the two public methods provided by `dirAuthenticate`:

```scala
object ServiceWithoutExports:
  private val dirAuthenticate =
    DirectoryAuthenticate(URL("https://directory.wtf"))

  def authenticate(username: UserName, password: Password): Boolean =
    dirAuthenticate(username, password)
  def isAuthenticated: Boolean = dirAuthenticate.isAuthenticated
```

Scala 3 gives us a way to avoid the boilerplate methods using a new `export` clause:

```scala
object Service:
  private val dirAuthenticate =
    DirectoryAuthenticate(URL("https://directory.wtf"))

  export dirAuthenticate.{isAuthenticated, apply as authenticate}
```

Before explaining the `export` clause, let's see `Service` in action:

```scala
scala> Service.isAuthenticated
val res0: Boolean = false

scala> Service.authenticate(UserName("Buck Trends"), Password("1234"))
val res1: Boolean = true

scala> Service.isAuthenticated
val res1: Boolean = true
```

Export clauses are similar to import clauses and use the same syntax. We gave the `apply` method an alias; using `Service.apply` to authenticate would be weird. We can export any members, not just methods. We can exclude items. For example, `export dirAuthenticate.{*, apply => _}` would export all public members except for `apply`.

The following rules describe what members are eligible for exporting:

- The member can't be owned by a supertype of the type with the export clause.
- The member can't override a concrete definition in a supertype, but it can be used to implement an abstract member in a supertype.
- The member is accessible at the export clause.
- The member is not a constructor.
- The member is not the synthetic (compiler-generated) class part of an `object`.
- If the member is a given instance or `implicit` value, then the export must be tagged with `given`.

All exports are `final`. They can't be overridden in subtypes.

Export clauses can also appear outside types, meaning they are defined at the package level. Hence, one way to provide a very controlled view of what's visible in a package is to declare everything *package private*, then use export clauses to expose only those items you want publicly visible. See Chapter 15 for more details on public, protected, private, and more fine-grained visibility controls.

Good Object-Oriented Design: A Digression

Consider the preceding example where `Person` was a supertype of `Employee`, which was a supertype of `Manager`. It has several *code smells*.

First, there's a lot of boilerplate in the constructor argument lists, like `name: String, age: Int` repeated three times. Second, it seems like all three should be case classes, right?

We can create a regular subtype from a case class or the other way around, but we can't subtype one case class to create another. This is because the autogenerated implementations of `toString`, `equals`, and `hashCode` do not work properly for subtypes, meaning they ignore the possibility that an instance could actually be a subtype of the case-class type.

This limitation is by design. It reflects the problematic aspects of subtyping. For example, should `Manager`, `Employee`, and `Person` instances be considered equal if they all have the same name and age? A more flexible interpretation of object equality might say yes, while a more restrictive version would say no. Also, the mathematical definition of equality requires commutative behavior: `somePerson == someEmployee` should return the same result as `someEmployee == somePerson`, but you would never expect the `Employee.equals` method to return true in this case.

The real problem is that we are subtyping the state of these instances. We are using subtyping to add additional fields that contribute to the instance state. In contrast, subtyping behavior (methods) with the same state is easier to implement robustly. It avoids the problems with `equals` and `hashCode` just described, for example.

Of course, these problems with inheritance have been known for a long time. Today, good object-oriented design favors composition over inheritance, where we compose units of functionality rather than build class hierarchies, when possible.

Mixin composition with `traits` makes composition straightforward. The code examples mostly use type hierarchies with few levels and mixins to enhance them. When bits of cleanly separated state and behavior are combined, mixin composition is robust.

When subtyping is used, I recommend the following rules:

- Use only one level of subtyping from a supertype, if at all possible.
- Concrete classes are never subtyped, except for two cases:
  - Classes that mix in other behaviors defined in `traits` (see Chapter 10). Ideally, those behaviors should be orthogonal (i.e., not overlapping).
  - Test-only versions to promote automated unit testing.
- When subtyping seems like the right approach, consider partitioning behaviors into traits, and mixin those traits instead. Recall our `NAPhoneNumber` design earlier in this chapter.
- Never build up logical state across supertype-subtype boundaries.
- Only use case classes for leaf nodes in a type hierarchy. That is, don't subtype case classes.
- Make your intentions explicit by marking types `open`, `final`, or `sealed`, as appropriate.

By *logical* state in the fourth bullet, I mean the fields and methods, which together define a *state machine* for the logical behavior. There might have some private, implementation-specific state that doesn't affect this external behavior, but be very careful that the internals don't leak through the type's abstraction. For example, a library might include private subtypes for special cases. Types might have private fields to implement caching, auditing, or other concerns that aren't part of the public abstraction.

So what about our `Person` hierarchy? What should we do instead? The answer really depends on the context of use. If we're implementing a Human Resources application, do we need a separate concept of `Person` or can `Employee` just be the only type, declared as a case class? Do we even need any types for this at all? If we're processing a result set from a database query, is it sufficient to use tuples or maps to hold the values returned from the query for each unique use case? Can we dispense with the ceremony of declaring a type altogether?

Here is an alternative with just a single `Employee` case class that embeds the assumption that nonmanagers will have an empty set of subordinates:

```
// src/script/scala/progscala3/basicoop/people/Employee.scala

case class Employee(
  name:    String,
  age:     Int,
  title:   String,
  manages: Set[Employee] = Set.empty)

val john = Employee("John Smith", 35, "Accountant")
val jane = Employee("Jane Doe", 28, "Full Stack Developer")
val tom  = Employee("Tom Tired", 22, "Junior Minion")
val minions = Set(john, jane, tom)
val ceo = Employee("John Smith", 60, "CEO", minions)
```

Fields in Types

We started the chapter with a reminder that the primary constructor parameters become instance fields if they are prefixed with the `val` or `var` keyword. For case classes, `val` is implied. This convention greatly reduces source-code boilerplate, but how does it translate to byte code?

Actually, Scala just does implicitly what Java code does explicitly. There is a private field created internal to the class, and the equivalents of getter and optional setter accessor methods are generated. Consider this simple Scala class:

```
class Name(var value: String)
```

Conceptually, it is equivalent to this code:

```
class Name(s: String):
  private var _value: String = s                        ❶

  def value: String = _value                            ❷
  def value_=(newValue: String): Unit = _value = newValue
```

❶ Invisible field, declared mutable in this case.

❷ The getter (reader) and the setter (writer) methods.

Note the convention used for the `value_=` method name. When the compiler sees a method named like this, it will allow client code to drop the _, effectively enabling infix notation as if we were setting a bare field in the object:

```
scala> val n = Name("Dean")
val n: Name = Name@333f6b2d

scala> n.value
val res0: String = Dean

scala> n.value = "Buck"

scala> n.value
val res1: String = Buck
```

If we declare a field immutable with the `val` keyword, the field is declared with `val` and the writer method is not synthesized, only the reader method.

You can use these conventions yourself if you want to expose a field implemented with custom logic for reading and writing.

Constructor parameters in noncase classes without the `val` or `var` don't become fields. It's important to note that the value is still in the scope of the entire class body. Let's fill in a bit more detail that might be required for a real `DirectoryAuthenticate` implementation:

```
class DirectoryAuthenticate(location: URL) extends Authenticate:
  protected def auth(username: UserName, password: Password): Boolean =
    directory.auth(username, password)

  protected val directory = connect(location)  // we can use location!
  protected def connect(location:URL) = ???
```

While `connect` refers to `location`, it is not a field. It is just in scope!

Why not always make these parameters fields with `val`? Unless these parameters are really part of the logical state exposed to users, they shouldn't be exposed as fields. Instead, they are effectively private to the class body.

The Uniform Access Principle

In the `Name` example, it appears that users can read and write the bare `value` field without going through accessor methods, but in fact they are calling methods. On the other hand, we could just declare a field in the class body with the default public visibility and then access it as a bare field:

```
class Name2(s: String):
  var value: String = s
```

This uniformity is called the *uniform access principle*. The user experience is identical. We are free to switch between bare field access and accessor methods as needed. For example, if we want to add some sort of validation on writes or lazily construct the field value on reads, then methods are better. Conversely, bare field access is faster than a method call, although some simple method invocations will be inlined by the compiler or runtime environment anyway.

Therefore, the uniform access principle has an important benefit in that it minimizes coupling between user code and implementation details. We can change that implementation without forcing client code changes, although a recompilation is required.

Because of the flexibility provided by uniform access, a common convention is to declare abstract, constant fields as methods instead:

```
// src/main/scala/progscala3/basicoop/AbstractFields.scala
package progscala3.basicoop

trait Logger:
  def loggingLevel: Int                                        ❶
  def log(message: String): Unit

case class ConsoleLogger(loggingLevel: Int) extends Logger:    ❷
  def log(message: String): Unit = println(s"$loggingLevel: $message")
```

❶ Declare the logging level as a method with no parentheses, rather than a `val`.

❷ Implement `loggingLevel` with a `val` constructor parameter instead of a concrete method.

Implementers have the choice of using a concrete method or using a `val`. Using a `val` is legal here because the contract of `loggingLevel` is that it returns some `Int`. If the same value is always returned, that satisfies the contract. Conversely, if we declared `loggingLevel` to be a field in `Logger`, then using a concrete method implementation would not be allowed because the compiler can't confirm that the method would always return a single value consistently.

I have used this convention in earlier examples. Did you notice it?

> When declaring an abstract field, consider declaring an abstract method instead. That gives implementers the freedom to use a method or a field.

Unary Methods

We saw earlier how to define an assignment method `foo_=` for field `foo`, which enables the intuitive syntax `myinstance.foo = bar`. How can we implement *unary* operators?

An example is negation. If we implement a complex number class, we want to support the negation of some instance `c` with `-c`:

```
// src/main/scala/progscala3/basicoop/Complex.scala
package progscala3.basicoop

import scala.annotation.targetName

case class Complex(real: Double, imag: Double):
  @targetName("negate") def unary_- : Complex =                  ❶
    Complex(-real, imag)
  @targetName("minus")  def -(other: Complex) =                  ❷
    Complex(real - other.real, imag - other.imag)
```

❶ The method name is `unary_X`, where `X` is the prefix operator character we want to use, `-` in this case. Note that the space between the `-` and the `:` is necessary to tell the compiler that the method name ends with `-` and not with `:`!

❷ For comparison, we also implement the minus infix operator for subtraction. Here is an example:

```
scala> import progscala3.basicoop.Complex
scala> val c = Complex(1.1, 2.2)
scala> assert(-c == Complex(-1.1, 2.2))
```

Recap and What's Next

We filled in more details for OOP in Scala, including constructors versus object apply methods, inheritance, and some tips on good object-oriented design.

We also set the stage for diving into `traits`, Scala's powerful tool for composing behaviors from constituent parts. In the next chapter we'll complete our understanding of traits and how to use them to solve various design problems.

CHAPTER 10

Traits

Scala traits function as *interfaces*, abstract declarations of members (methods, fields, and types) that together express state and behavior. Traits can also provide concrete definitions (implementations) of some or all of those declarations.

A trait with concrete definitions works best when those definitions provide state and behavior that are well encapsulated and loosely coupled or even orthogonal to the rest of the state and behavior in the types that use the trait. The term *mixin* is used for such traits because we should be able to "mix together" such traits to compose different concrete types.

Traits as Mixins

We first discussed traits as mixins in "Traits: Interfaces and Mixins in Scala" on page 99, where we explored an example that mixes logging into a service. Logging is a good example of mixin behavior that can be well encapsulated and orthogonal to the state and behavior of the rest of a service.

Let's revisit and expand on what we learned with a new example. First, consider the following code for a button in a GUI toolkit, which uses callbacks to notify clients when clicks occur:

```
// src/main/scala/progscala3/traits/ui/ButtonCallbacks.scala
package progscala3.traits.ui

abstract class ButtonWithCallbacks(val label: String,
    val callbacks: Seq[() => Unit] = Nil) extends Widget:           ❶

  def click(): Unit =                                                ❷
    updateUI()
    callbacks.foreach(f => f())
```

```
  protected def updateUI(): Unit                                        ❸
```

❶ The class is `abstract` because `updateUI` is abstract. For now, `Widget` is defined as `abstract class Widget` in the same package.

❷ When the button is clicked, invoke the list of callback functions of type `() => Unit`, which can only perform side effects, like sending a message to a backend service.

❸ Update the user interface (UI).

This class has two responsibilities: updating the visual appearance and handling callback behavior, including the management of a list of callbacks and calling them whenever the button is clicked. Separation of concerns would be better, achieved through composition.

So let's separate the button-specific logic from the callback logic, such that each part becomes simpler, more modular, easier to test and modify, and more reusable. The callback logic is a good candidate for a mixin.

Callbacks are a special case of the *Observer Design Pattern* [GOF1995]. So let's create two traits that declare and partially implement the `Subject` and `Observer` logic in this pattern, then use them to handle callback behavior. To simplify things, we'll start with a single callback that counts the number of button clicks:

```
// src/main/scala/progscala3/traits/observer/Observer.scala
package progscala3.traits.observer

trait Observer[State]:                                                  ❶
  def receiveUpdate(state: State): Unit

trait Subject[State]:                                                   ❷
  private var observers: Vector[Observer[State]] = Vector.empty         ❸

  def addObserver(observer: Observer[State]): Unit =                    ❹
    observers.synchronized { observers :+= observer }                   ❺

  def notifyObservers(state: State): Unit =                             ❻
    observers foreach (_.receiveUpdate(state))
```

❶ The trait for clients who want to be notified of state changes. They must implement the `receiveUpdate` message.

❷ The trait for subjects who will send notifications to observers.

❸ A mutable vector of observers to notify.

❹ A method to add observers.

❺ Since `observers` is mutable, we use `observers.synchronized` to ensure thread-safe updates. The update expression is equivalent to `observers = observer +: observers`.

❻ A method to notify observers of state changes.

Often, the most convenient choice for the `State` type parameter is just the type of the class mixing in `Subject`. Hence, when the `notifyObservers` method is called, the instance just passes itself (i.e., `this`).

> Traits with abstract members don't have to be declared abstract by adding the `abstract` keyword before the `trait` keyword. However, if a class has abstract members, it must be declared abstract.

Now, we can define a simple `Button` type:

```
// src/main/scala/progscala3/traits/ui/Button.scala
package progscala3.traits.ui

abstract class Button(val label: String) extends Widget:
  def click(): Unit = updateUI()
  protected def updateUI(): Unit
```

`Button` is considerably simpler. It has only one concern, handling clicks. At the moment, it seems trivial now to have a public method `click` that does nothing except delegate to a protected method `updateUI`, but we'll see next why this is still useful.

Now we construct `ObservableButton`, which subtypes `Button` and mixes in `Subject`:

```
// src/main/scala/progscala3/traits/ui/ObservableButton.scala
package progscala3.traits.ui
import progscala3.traits.observer.*

abstract class ObservableButton(name: String)                  ❶
    extends Button(name) with Subject[Button]:                 ❷

  override def click(): Unit =                                 ❸
    super.click()                                              ❹
    notifyObservers(this)                                      ❺
```

❶ A subtype of `Button` that mixes in observability.

❷ Extends `Button`, mixes in `Subject`, and uses `Button` as the `Subject` type parameter, named `State` in the declaration of `Subject`.

❸ In order to notify observers, we have to override the `click` method.

❹ First, call the supertype `click` to perform the normal GUI update logic.

❺ Notify the observers, passing `this` as the `State`. In this case, there isn't any state other than the event that a button click occurred.

So we modified `click`, but kept it simple for future subtypes to correctly implement `updateUI`, without having to worry about correct handling of the observer logic. This is why we kept both `click` and `updateUI` in `Button`. Note that `ObservableButton` is still abstract.

Let's try it. First, let's define an observer to count button clicks:

```
// src/main/scala/progscala3/traits/ui/ButtonCountObserver.scala
package progscala3.traits.ui
import progscala3.traits.observer.*

class ButtonCountObserver extends Observer[Button]:
  var count = 0
  def receiveUpdate(state: Button): Unit =
    count.synchronized { count += 1 }
```

Now try it:

```
// src/script/scala/progscala3/traits/ui/ButtonCountObserver1.scala
import progscala3.traits.ui.*
import progscala3.traits.observer.*

val button = new ObservableButton("Click Me!"):
  def updateUI(): Unit = println(s"$label clicked")

val bco1 = ButtonCountObserver()
val bco2 = ButtonCountObserver()

button.addObserver(bco1)
button.addObserver(bco2)

(1 to 5) foreach (_ => button.click())

assert(bco1.count == 5)
assert(bco2.count == 5)
```

The script declares an observer type, `ButtonCountObserver`, that counts clicks. Then it creates an anonymous subtype of `ObservableButton` with a definition of `updateUI`. Next it creates and registers two observers with the button, clicks the button five times, and then verifies that the counts for each observer equals five. You'll also see the string `Click Me! clicked` printed five times.

Suppose we only need one instance of an `ObservableButton`. We don't need to declare a class that mixes in the traits we want. Instead, we can declare a `Button` and mix in the `Subject` trait in one step (the rest of the example is unchanged):

```
// src/script/scala/progscala3/traits/ui/ButtonCountObserver2.scala
val button = new Button("Click Me!") with Subject[Button]:
  override def click(): Unit =
    super.click()
    notifyObservers(this)
  def updateUI(): Unit = println(s"$label clicked")
```

> When declaring a class that only mixes in traits and doesn't extend another class, you must use the `extends` keyword anyway for the first trait listed and the `with` keyword for the rest of the traits. However, when instantiating a class and mixing in traits with the declaration, use the `with` keyword with all the traits.

In "Good Object-Oriented Design: A Digression" on page 265, I recommended that you avoid method overrides. We didn't have a choice for `click`, but we proceeded carefully. In the `click` override, we added invocation of the subject-observer logic but didn't modify the UI update logic. Instead, we left in place the protected method `updateUI` to keep that logic separated from observation logic. Our click logic follows the examples in "Overriding Methods? The Template Method Pattern" on page 251.

Stackable Traits

There are several further refinements we can do to improve the reusability of our code and to make it easier to use more than one trait at a time (i.e., to stack traits).

First, clicking is not limited to buttons in a GUI. We should make that logic abstract too. We could put it in `Widget`, the so-far empty supertype of `Button`, but it may not be true that all GUI widgets accept clicks. Instead, let's introduce another trait, `Clickable`:

```
// src/main/scala/progscala3/traits/ui2/Clickable.scala
package progscala3.traits.ui2                                      ❶

trait Clickable:                                                   ❷
  def click(): String = updateUI()
  protected def updateUI(): String
```

❶ Use a new package because we're reimplementing types in `traits.ui`.

❷ Essentially just like the previous `Button` definition, except now we return a `String` from `click`.

Having `click` return a `String` is useful for discussing the stacking of method calls.

Here is the refactored button, which uses the trait:

```
// src/main/scala/progscala3/traits/ui2/Button.scala
package progscala3.traits.ui2
import progscala3.traits.ui.Widget

abstract class Button(val label: String) extends Widget with Clickable
```

It is still abstract, little more than a name for a convenient GUI concept that is implemented with a composition of reusable types!

Observation should now be tied to `Clickable` and not `Button`, as it was before. When we refactor the code this way, it becomes clear that we don't really care about observing buttons. We really care about observing events, such as clicks. Here is a trait that focuses solely on observing `Clickable`:

```
// src/main/scala/progscala3/traits/ui2/ObservableClicks.scala
package progscala3.traits.ui2
import progscala3.traits.observer.*

trait ObservableClicks extends Clickable with Subject[Clickable]:
  abstract override def click(): String =                          ❶
    val result = super.click()
    notifyObservers(this)
    result
```

❶ Note the `abstract override` keyword combination, discussed next.

The implementation is very similar to the previous `ObservableButton` example. The important difference is the `abstract` keyword. We had just `override` before.

Look closely at this method. It calls `super.click()`, but what is `super` in this case? At this point, it could only appear to be `Clickable`, which declares but does not define the `click` method, or it could be `Subject`, which doesn't have a `click` method. So `super` can't be bound to a real instance, at least not yet. This is why `abstract` is required here.

In fact, `super` will be resolved when this trait is mixed into a concrete instance that defines the `click` method, such as `Button`. The `abstract` keyword tells the compiler (and the reader) that `click` is not yet fully implemented, even though `ObservableClicks.click` has a body.

> The `abstract` keyword is only required on a method in a trait when the method has a body, but it invokes another method in `super` that doesn't have a concrete implementation in supertypes of the trait.

Let's use this trait with `Button` and its concrete `click` method. First we'll define an observer that does counting, whether from clicks or anything else:

```
// src/main/scala/progscala3/traits/ui2/CountObserver.scala
package progscala3.traits.ui2
import progscala3.traits.observer.*

trait CountObserver[State] extends Observer[State]:
  var count = 0
  def receiveUpdate(state: State): Unit = count.synchronized { count += 1 }
```

Now use it:

```
// src/script/scala/progscala3/traits/ui2/ClickCountObserver.scala
import progscala3.traits.ui2.*
import progscala3.traits.observer.*

// No override of "click" in Button required.
val button = new Button("Button") with ObservableClicks:
  def updateUI(): String = s"$label clicked"

val cco = new CountObserver[Clickable] {}                    ❶
button.addObserver(cco)

(1 to 5) foreach (_ => assert("Button clicked" == button.click()))
assert(cco.count == 5)
```

❶ `CountObserver` is a trait, so we have to provide a body. However, the body is empty because all the members are concrete. I used a trait so that this type could also be used as a mixin when needed.

Note that we can now declare a `Button` instance and mix in `ObservableClicks` without having to override the `click` method ourselves. We have also gained a reusable mixin for click handling, `Clickable`.

Let's finish our example by adding a second trait, where an observer can veto a click after a certain number has been received:

```
// src/main/scala/progscala3/traits/ui2/VetoableClicks.scala
package progscala3.traits.ui2

trait VetoableClicks(val maxAllowed: Int = 1) extends Clickable:   ❶
  private var count = 0                                            ❷

  abstract override def click(): String =
    count.synchronized { count += 1 }
    if count <= maxAllowed then                                    ❸
      super.click()
    else
      s"Max allowed clicks $maxAllowed exceeded. Received $count clicks!"

  def resetCount(): Unit = count.synchronized { count = 0 }        ❹
```

❶ Also extends `Clickable`.

❷ Use a private variable to avoid collision with any other fields named `count` from other mixins.

❸ The maximum number of allowed clicks. Once the number of clicks exceeds the allowed value (counting from zero), no further clicks are sent to `super`.

❹ A method to reset the count.

Note that this `count` should be different from the `count` used in `ClickCountObserver`. Keeping this one private prevents confusion. The compiler will flag collisions.

In this use of both traits, we limit the number of handled clicks to 2:

```
// src/script/scala/progscala3/traits/ui2/VetoableClickCountObserver.scala
import progscala3.traits.ui2.*
import progscala3.traits.observer.*

val button = new Button("Button!")
    with ObservableClicks with VetoableClicks(maxAllowed = 2):
  def updateUI(): String = s"$label clicked"

val cco = new CountObserver[Clickable] {}
button.addObserver(cco)

(1 to 5) map (_ => button.click())
assert(cco.count == 2)
```

The `map` with calls to `click` returns the following sequence of strings:

```
Vector("Button! clicked", "Button! clicked",
  "Max allowed clicks 2 exceeded. Received 3 clicks!",
  "Max allowed clicks 2 exceeded. Received 4 clicks!",
  "Max allowed clicks 2 exceeded. Received 5 clicks!")
```

Try this experiment. Switch the order of the traits in the declaration of `button` to this:

```
val button = new Button("Click Me!")
    with VetoableClicks(maxAllowed = 2) with ObservableClicks:
  def updateUI(): String = s"$label clicked"
```

What happens when you run this code now? The strings returned will be the same, but the assertion that `cco.count == 2` will now fail. The count is actually 5, so the extra clicks are not actually vetoed!

We have three versions of `click` wrapped like an onion. The question is which version of `click` gets called first when we mix in `VetoableClicks` and `ObservableClicks`? The answer is determined by the declaration order, from right to left.

This means that `ObservableClicks` will be notified before `VetoableClicks` has the chance to prevent calling up the chain, `super.click()`. Hence, declaration order matters.

An algorithm called *linearization* is used to resolve the priority of traits and classes in the inheritance tree when resolving which overridden method to call for `super.method()`. However, it can get confusing quickly, so avoid complicated mixin structures! We'll cover the full details of how linearization works in "Linearization of a Type Hierarchy" on page 301.

This fine-grained composition through mixin traits is quite powerful, but it can be overused:

- It can be difficult to understand and debug code if many mixin traits are used.
- When method overrides are required, as in this sequence of examples, resolving the correct order can get confusing quickly.
- Lots of traits can slow down compile times.

3 Union and Intersection Types

Now is a good time to revisit union types and introduce *intersection types*, both of which are new in Scala 3.

We encountered union types in "When You Really Can't Avoid Nulls" on page 61, where we discussed using `T | Null` as a type declaration for the value returned by calling a method that will return either a value of type `T` or `null`.

Here's another example, where `Int | String` is used as an alternative to `Either[String,Int]` (note the different order) as a conventional way to return either a success (`Int`) or a failure described by a `String`:

```
// src/script/scala/progscala3/traits/UnionTypes.scala
scala> def process(i: Int): Int | String =
     |   if (i < 0) then "Negative integer!" else i

scala> val result1a: Int | String = process(-1)
     | val result2a: Int | String = process(1)
val result1a: Int | String = Negative integer!
val result2a: Int | String = 1

scala> val result1b = process(-1)
     | val result2b = process(1)
val result1b: Int | String = Negative integer!
val result2b: Int | String = 1

scala> Seq(process(-1), process(1)).map {
     |   case i: Int => "integer"
```

```
         |   case s: String => "string"
         | }
    val res0: Seq[String] = List(string, integer)
```

Note the types for the values. These types are unions in the sense that any `Int` *or* `String` value can be used. Hence, the type is the union of the set of all `Int` and `String` values.

I didn't mention it in the previous sections, but perhaps you noticed the type printed for this declaration (removing the package prefixes for clarity):

```
scala> val button = new Button("Button!")
     |     with ObservableClicks with VetoableClicks(maxAllowed = 2):
     |   def updateUI(): String = s"$label clicked"
val button: Button & ObservableClicks & VetoableClicks = ...
```

The type `Button & ObservableClicks & VetoableClicks` is an intersection type, which results when we construct instances with mixins. In Scala 2, the returned type signature would use the same `with` and `extends` keywords as the definition, `Button with ObservableClicks with VetoableClicks`. It's perhaps a little confusing that we get back something different. Unfortunately, you can't use `&` instead of `with` or `extends` in the definition. However, we can use `&` in a type declaration:

```
scala> val button2: Button & ObservableClicks & VetoableClicks =
     |   new Button("Button!")
     |     with ObservableClicks with VetoableClicks(maxAllowed = 2):
     |   def updateUI(): String = s"$label clicked"
val button2: Button & ObservableClicks & VetoableClicks = ...
```

These types are intersections in the sense that the only allowed values that you can assign to `button2` are those that belong to `Button` and `ObservableClicks` and `VetoableClicks`, which won't include values of each of those types separately. Note this compilation error (some output omitted again):

```
scala> val button3: Button & ObservableClicks & VetoableClicks =
     |   new Button("Button!"):
     |     def updateUI(): String = s"$label clicked"
2 |  def updateUI(): String = s"$label clicked"
  |                                         ^
  |Found:    Button {...}
  |Required: Button & ObservableClicks & VetoableClicks
  |)
```

Most of the time, this won't be an issue, but imagine a scenario where you have a `var` for a button that you want to observe now but eventually replace with an instance without observation:

```
scala> var button4 = new Button("Button!")
     |   with ObservableClicks with VetoableClicks(maxAllowed = 2):
     |     def updateUI(): String = s"$label clicked"
```

```
scala> // later...
scala> button4 = new Button("New Button!"):
     |   def updateUI(): String = s"$label clicked"
2 |  def updateUI(): String = s"$label clicked"
  |                                          ^
  |Found:    Button {...}
  |Required: Button & ObservableClicks & VetoableClicks
```

Not convenient. One easy workaround is to use a type declaration, `var button4: Button = ...`. Then the reassignment will work.

But wait, isn't that a type error, based on how I just described intersection types? It's not, because the type restriction declared for `button4` allows any `Button`, but the initial assignment happens to be a more restricted instance, a `Button` that also mixes in other types.

Scala 3 offers a new option for handling this scenario, discussed next.

3 Transparent Traits

Note that `ObservableClicks` and `VetoableClicks` are implementation details and not really part of the core domain types of a UI. A familiar example from Scala 2 is the way that several common traits are automatically added as mixins for case classes and objects. Here's an example:

```
// Use the Scala 2 REPL to try this example!
scala> trait Base
scala> case object Obj1 extends Base
scala> case object Obj2 extends Base
scala> val condition = true
scala> val x = if (condition) Obj1 else Obj2
x: Product with Serializable with Base = Obj1
```

The inferred type is the supertype `Base`, but with `scala.Product` and `java.lang.Serializable`. Usually, we only care about `Base`.

Scala 3 lets you add a keyword `transparent` to the declaration of a trait so that it isn't part of the inferred type:

```
scala>
     | open class Text(val label: String, value: String) extends Widget
     |
     | trait Flag
     | val t1 = new Text("foo", "bar") with Flag
val t1: Text & Flag = anon$1@28f8a295

scala> transparent trait TFlag
     | val t2 = new Text("foo", "bar") with TFlag
val t2: Text = anon$2@4a2219d
```

Note the different inferred types for `t1` and `t2`.

Common traits, like `scala.Product`, `java.lang.Serializable`, and `java.lang.Comparable`, are now treated as transparent. In the Scala 3 REPL, the preceding `Base` example will have the inferred type `Base` without `Product` and `Serializable` for `x`.

> If you need to cross-compile code for Scala 2.13 and 3 for a while, use the new annotation `scala.annotation.transparentTrait` instead of the `transparent` keyword.

Using Commas Instead of with

Scala 3 also allows you to substitute a comma (,) instead of `with`, but only when declaring a type:

```
scala> class B extends Button("Button!"),
     |     ObservableClicks, VetoableClicks(maxAllowed = 2):
     |   def updateUI(): String = s"$label clicked"

scala> var button4b: Button = new Button("Button!"),
     |     ObservableClicks, VetoableClicks(maxAllowed = 2):
     |   def updateUI(): String = s"$label clicked"
     |
1 |var button4b: Button = new Button("Button!"), ObservableClicks, ...
  |                                            ^
  |                               end of statement expected but ',' found
1 | ...
```

If this substitution worked for all situations where `with` is used, it would be more useful, but it is confusing to remember when it is allowed and when it isn't.

Trait Parameters

Scala 3 allows traits to have parameters, just like class constructor parameters, whereas in Scala 2, you had to declare fields in the trait body. We used this feature in `VetoableClicks`.

Here is a logging abstraction that uses a trait parameter for the `level`:

```
// src/main/scala/progscala3/traits/Logging.scala
package progscala3.traits.logging

enum LoggingLevel:
  case Debug, Info, Warn, Error, Fatal                       ❶

trait Logger(val level: LoggingLevel):                       ❷
  def log(message: String): Unit

trait ConsoleLogger extends Logger:
```

```scala
  def log(message: String): Unit =
    println(s"${level.toString.toUpperCase}: $message")

class Service(val name: String, level: LoggingLevel)                ❸
  extends ConsoleLogger with Logger(level)
```

❶ Define logging levels.

❷ The `level` is a trait parameter. This will also be a field in concrete types that use this trait.

❸ A concrete type or one of its supertypes must pass a `LoggingLevel` value to `Logger`.

Note that we have to mix in both `Logging` and `ConsoleLogger`, even though the latter extends the former, because we must specify the `level` parameter for `Logger` explicitly. `ConsoleLogger` can't be declared as follows, like we might see in a hierarchy of classes:

```scala
trait ConsoleLogger(level: LoggingLevel) extends Logger(level)  // ERROR
```

Note that the `name` argument for `Service` is declared to be a field (with `val`), but if we try declaring `level` as a field, it will conflict with the definition already provided by `Logger`. (Try it!) However, as shown, we can use the same name for a nonfield parameter.

Here is the Scala 2–compatible approach without trait parameters:

```scala
// src/main/scala/progscala3/traits/LoggingNoParameters.scala
package progscala3.traits.logging

trait LoggerNP:                                                     ❶
  def level: LoggingLevel
  def log(message: String): Unit

trait ConsoleLoggerNP extends LoggerNP:
  def log(message: String): Unit = println(s"$level: $message")

class ServiceNP(val name: String, val level: LoggingLevel)          ❷
  extends ConsoleLoggerNP
```

❶ `NP` for "no parameters." Note that the abstract `level` is now a method. It could be an abstract field, but using a method provides more flexibility for the implementer (see "The Uniform Access Principle" on page 268).

❷ The implementation of `level` is a `ServiceNP` constructor parameter.

The same `LoggingLevel` enumeration is used here.

It's convenient that we can now declare parameters for traits, but clearly it has limitations. If a hierarchy of classes mix in a parameterized trait, only one can pass arguments to it. Also, if `Service` were declared as a case class, the `level` argument would now be a field, which would conflict with `level` defined by `Logging`. (Try adding `case` and recompile.) We would need to use a different name, adding a redundant field!

However, to be fair, types like this that are composed of other types aren't good candidates for case classes. For example, if two service instances differ only by the logging level, should they still be considered equivalent?

Also, `LoggingNP` uses the technique I recommended earlier that abstract fields should be declared as methods, so implementers have the freedom to use a field or a method. This option isn't possible if the field is a parameter for the trait.

3 Trait parameters fix some scenarios with the order of initialization. Scala 2 had a feature to handle these cases called *early initializers*. This feature was dropped in Scala 3 because trait parameters now address these scenarios. Chapter 12 discusses construction of types with mixins and a hierarchy of supertypes.

On balance, I think parameterized traits will be best in some scenarios, while embedding the fields inside the traits, either as abstract methods or fields, will be best in other scenarios. Parameterized traits won't completely replace the older idioms.

Should That Type Be a Class or Trait?

When considering whether a type should be a trait or a class, keep in mind that traits are best for pure interfaces and when used as mixins for complementary state and behavior. If you find that a particular trait is used most often as a supertype of other types, then consider defining the type as a class instead to make this logical relationship more clear.

Recap and What's Next

In this chapter, we learned how to use traits to encapsulate cross-cutting concerns between classes with reusable mixins. We covered when and how to use traits, how to correctly stack multiple traits, and the rules for initializing values within traits versus using trait parameters.

In the next few chapters, we explore Scala's object system and class hierarchy, with particular attention to the collections. We also revisit construction of complex types, like our stacked traits example.

CHAPTER 11

Variance Behavior and Equality

An important concept in object-oriented type systems goes by the name *variance under inheritance*. More specifically, we need well-defined rules for when an instance of one type can be substituted for an instance of another type. This chapter begins with an exploration of these concepts.

A logical follow-on is the subject of instance equality, which is trickier than it might seem in object-oriented languages.

Parameterized Types: Variance Under Inheritance

Suppose a `val` is declared of type `Seq[AnyRef]`. Are you allowed to assign a `Seq[String]` to it? In other words, is `Seq[String]` considered substitutable for `Seq[AnyRef]`? The *Liskov substitution principle* (LSP) was the first to define formally what this means. In OOP, LSP is defined using type hierarchies. Instances of one type `A` are substitutable for instances of another type `B` if `A` is a subtype of `B`. Since `AnyRef` is a supertype of all reference types, like `String`, instances of `String` are substitutable where instances of `AnyRef` are required. (We'll discuss the Scala type hierarchy in depth in Chapter 13.)

So what about parameterized types, such as collections like `Seq[AnyRef]` and `Seq[String]`? Let's look at immutable parameterized types first.

The type parameter `A` is declared like this, `Seq[+A]`, where `+A` means that `Seq` is covariant in the `A` parameter. Since `String` is substitutable for `AnyRef`, then `Seq[String]` is substitutable for a `Seq[AnyRef]`. *Covariance* means the supertype-subtype relationship of the container (the parameterized type) goes in the same direction as the relationship between the type parameters.

We can also have types that are contravariant. A declaration X[-A] means that X[String] is a supertype of X[AnyRef]. The substitutability goes in the opposite direction of the type parameter values. This behavior is less intuitive, but we'll study an important example shortly.

If a parameterized type is neither covariant nor contravariant, it is called *invariant*. Any type parameter without a + or - is therefore invariant.

We first encountered these concepts in "Parameterized Types Versus Abstract Type Members" on page 66. Now we'll explore them in more depth.

The three kinds of variance notations and their meanings are summarized in Table 11-1. T^{sup} is a supertype of T and T_{sub} is a subtype of T.

Table 11-1. Type variance annotations and their meanings

Type	Description
+T	Covariant (e.g., Seq[T_{sub}] is a subtype of Seq[T])
-T	Contravariant (e.g., X[T^{sup}] is a subtype of X[T])
T	Invariant (e.g., can't substitute Y[T^{sup}] or Y[T_{sub}] for Y[T])

When a type like Seq[+A] has only one covariant type parameter, you'll often hear the shorthand expression "Seqs are covariant," for example.

Covariant and invariant types are reasonably easy to understand. What about contravariant types?

Functions Under the Hood

Let's dive into functions a bit more and then explore how they combine contravariant and covariant behavior.

A function literal with two arguments implements the trait Function2[-T1,-T2,+R], where the two type parameters for the function inputs, T1 and T2, are *contravariant* and the return type R is covariant. Therefore, functions have mixed variance behavior. We'll explore why shortly.

There are corresponding FunctionN types for arity N between 0 and 22, inclusive. New for Scala 3, for arity greater than 22, functions are instantiated with scala.FunctionXXL, which encodes the parameters in an array.[1]

1 Similar implementation techniques are used for tuples. See "Products, Case Classes, Tuples, and Functions" on page 316.

We've been using anonymous functions, also known as function literals, throughout the book. For example:

```
scala> Seq(1, 2, 3, 4).foldLeft(0)((result,i) => result+i)
val res0: Int = 10
```

The function expression `(result,i) => result+i` is actually syntactic sugar that the compiler converts to the following instantiation of an anonymous subtype of `Function2`:

```
scala> val f: (Int,Int) => Int = new Function2[Int,Int,Int]:
     |   def apply(i: Int, j: Int): Int = i + j
     |
val f: (Int, Int) => Int = <function2>

scala> Seq(1, 2, 3, 4).foldLeft(0)(f)
val res1: Int = 10
```

Note that I declared `f` with the literal type signature syntax `(Int,Int) => Int` between the colon and equal sign.

The function instance has an `apply` method that the compiler uses when the function is invoked. Therefore `f(1,2)` is actually `f.apply(1,2)`.

Now let's return to contravariance. The best example of it is the types of the parameters passed to a function's `apply` method. We'll use `scala.Function1[-T,+R]`. The same arguments apply for the other functions with more parameters.

Let's look at an example to understand what the variance behavior really means. It will also help us think about what *substitutability* really means, which is the key term to remember when you try to sort out these behaviors:

```
// src/script/scala/progscala3/objectsystem/variance/FunctionVariance.scala

class CSuper                                                    ❶
class C      extends CSuper
class CSub   extends C

val f1: C => C = (c: C)      => C()                             ❷
val f2: C => C = (c: CSuper) => CSub()
val f3: C => C = (c: CSuper) => C()
val f4: C => C = (c: C)      => CSub()
// Compilation errors!
// val f5: C => C = (c: CSub)   => CSuper()                     ❸
// val f6: C => C = (c: CSub)   => C()
// val f7: C => C = (c: C)      => CSuper()
```

❶ Define a three-type inheritance hierarchy, `CSub <: C <: CSuper`.

❷ Four different valid assignments for functions of the same type `C => C`.

❸ Three invalid assignments, which would trigger compilation errors. (Try them!)

All valid function instances must be substitutable or type compatible with `C => C` (i.e., `Function1[C,C]`). The values we assign must satisfy the constraints of variance under inheritance for functions. Let's work through each example. First, we'll confirm that the covariant and contravariant requirements are satisfied in each case, then we'll develop the intuition for why these requirements exist in the first place.

The assignment for `f1` is `(c: C) => C()`. It matches the types exactly so this one is easy. All the examples ignore the parameter `c`; it's the type of `c` that matters. All of them return a constructed instance. For `f1`, a `C` instance is returned.

The assignment for `f2` is `(c: CSuper) => CSub()`. It is valid, because the function parameter `C` is contravariant, so `CSuper` is a valid substitution. The return value is covariant, so `CSub` is a valid replacement for `C`.

The assignments for `f3` and `f4` are like `f2`, but we just use `C` for the return and the parameter, respectively. In other words, `f2` is really the most important test of the rules.

Similarly, `f5` through `f7` are analogous to `f2` through `f4`, but using invalid substitutions. So let's just discuss `f5`, where both type substitutions fail. Using `CSub` for the parameter type is invalid because the parameter must be `C` or a supertype. Using `CSuper` for the return type is invalid because `C` or a subtype is required.

Let's try to understand intuitively why these variance behaviors are required.

The key insight is to know that the function type signature `C => C` is a contract. Any function proposed must promise to accept any valid `C` instance and it must promise to return any valid `C` instance.

So consider the return type covariance first. If the function is actually of type `C => CSub`, which always returns an instance of the subtype `CSub`, that satisfies the contract because a `CSub` instance is always substitutable for a `C` instance. As the user of the function, my part of the contract is that I have to accept any `C` instance the function returns, so I can easily accept the fact I only ever receive instances of type `CSub`. In this sense, the function is more restrictive than it needs to be for return values, but that's OK with me.

In contrast, if the function is `CSuper => C`, it is more permissive about what arguments it will accept. Here my part of the contract is that I will only pass `C` instances to the function because that's what I said I will do with the `C => C` type. However, the actual function is able to handle the wider set of all `CSuper` instances, including instances of `C` and even `CSub`. So a `CSuper => C` function is more permissive than it needs to be, but that's also OK with me.

I said that `f5`, of type `CSub => CSuper`, breaks the contract for both types. Let's consider what would happen if this substitution were allowed.

For the inputs, the function only knows how to handle `CSub` instances passed to it, but the contract `C => C` says I'm allowed to pass `C` instances to it. Hence the function would be "surprised" when it has to handle a `C` instance. Similarly, because the function returns `CSuper` instances, I will be "surprised" because I only expect to receive `C` return values.

This is why function parameters must be contravariant, while the return values must be covariant.

> When thinking about variance under inheritance, ask yourself what substitutions are valid given the contract defined by the types. This is true for all types, not just functions.

When defining your own parameterized types, the compiler checks the use of variance annotations. Here's what happens if you attempt to define your own function with the wrong annotations:

```
scala> class MyFunction2[+T1, +T2, -R]:
     |   def apply(v1:T1, v2:T2): R = ???
2 |  def apply(v1:T1, v2:T2): R = ???
  |  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |contravariant type R occurs in covariant position in type
  |  (v1: T1, v2: T2): R of method apply
```

If we change `-R` to `+R`, we see the other errors:

```
scala> class MyFunction2[+T1, +T2, +R]:
     |   def apply(v1:T1, v2:T2): R = ???
2 |  def apply(v1:T1, v2:T2): R = ???
  |            ^^^^^
  | covariant type T1 occurs in contravariant position in type T1 of value v1
...
```

Finally, variance annotations only make sense on the type parameters for parameterized types, not for methods with type parameters. The annotations affect the behavior of subtyping. Methods aren't subtyped themselves, only their enclosing types. For example, the signature for the `Seq.map` method looks conceptually like this:

```
abstract class Seq[+A](...):
  ...
  def map[B](f: A => B): Seq[B] = ...
```

There is no variance annotation on `B`, and if you tried to add one, the compiler would throw an error. However, you will sometimes see *type bounds* on a method parameter. Here's what `fold` looks like:

```
abstract class Seq[+A](...):
  ...
  def reduce[B >: A](op: (B,B) => B): B = ...
```

This has nothing to do with substitutability for subtyping. It just says that when reducing a collection with elements of type A, you might end up with a result of supertype B.

Variance of Mutable Types

What about the variance behavior of type parameters for mutable types? The short answer is that only invariance is allowed when fields in instances of the type are mutable. Consider the following class definitions:

```
// src/script/scala/progscala3/objectsystem/variance/MutableVariance.scala

scala> class Invariant[A](var mut: A)
// defined class Invariant

scala> class Covariant[+A](var mut: A)
1 |class Covariant[+A](var mut: A)
  |                        ^
  |covariant type A occurs in contravariant position in type A of value mut_=

scala> class Contravariant[-A](var mut: A)
1 |class Contravariant[-A](var mut: A)
  |                            ^^^^^^^^^^
  |contravariant type A occurs in covariant position in type A of variable mut
```

Only the invariant definition compiles. When mut is declared as a mutable field, it behaves like a private field with public read and write methods, each of which limits the allowed variance. Here are logically equivalent definitions of the second and third classes:

```
scala> class Covariant[+A](val mutInit: A):
     |   private var _mut: A = mutInit
     |   def mut_=(a: A): Unit = _mut = a
     |   def mut: A = _mut
     |
3 |  def mut_=(a: A): Unit = _mut = a
  |            ^^^^
  |    covariant type A occurs in contravariant position in type A of value a

scala> class Contravariant[-A](val mutInit: A):
     |   private var _mut: A = mutInit
     |   def mut_=(a: A): Unit = _mut = a
     |   def mut: A = _mut
     |
1 |class Contravariant[-A](val mutInit: A):
  |                            ^^^^^^^^^^^^^^
  |contravariant type A occurs in covariant position in type A of value mutInit
```

```
4 |  def mut: A = _mut
  |  ^^^^^^^^^^^^^^^^^
  |contravariant type A occurs in covariant position in type => A of method mut
```

Recall from our earlier discussion about function variance that the types for function parameters have to be contravariant. That's why we get an error for `Covariant.mut_=`. We are attempting to use a covariant type `A` in a parameter list, which always requires a contravariant type.

Similarly, we learned that function return types need to be covariant, yet `Contravariant.mut_` attempts to return a value of a contravariant type `A`.

The mutable field type is used both as a return value type and a parameter type, each of which has conflicting variance requirements. The only thing that doesn't break is invariance, meaning no variance for `A` is allowed.

As for functions, valid substitution drives the requirement. Let's pretend these classes compile and reason about what would happen if we used them. We'll work with the same `CSuper`, `C`, and `CSub` used previously. First, let's check `Invariant`:

```
val obj: Invariant[C] = Invariant(C())
val c: C = obj.mut            // Type checks correctly
obj.mut = C()                 // Type checks correctly
```

Now try `Covariant`. Recall that we are attempting to prove that `Covariant[CSub]` is substitutable for `Covariant[C]`, since `CSub` is substitutable for `C`:

```
val obj: Covariant[C] = Covariant(CSub())  // Okay??
val c:C = obj.mut            // Type checks correctly, because CSub <: C
obj.mut = C()                // ERROR, because obj.mut requires CSub, not C!!
```

The actual type of `obj` is `Covariant[CSub]`. When we read `obj.mut`, we get a `CSub` instance, but that's OK because `CSub` is a subtype of `C`. However, if we attempt to assign a `C` instance to `obj.mut`, we get an error, because a `C` is not a subtype of `CSub`, it is a supertype. Hence, a `C` value isn't substitutable for a `CSub`. This is consistent with the error message we got when we attempted to compile `Covariant`.

Finally, let's try `Contravariant`. Recall that we are attempting to prove that `Contravariant[CSuper]` is substitutable for `Covariant[C]`, if `CSuper` is substitutable for `C`:

```
val obj: Contravariant[C] = Contravariant(CSuper())  // Okay??
val c:C = obj.mut            // ERROR, because c:C, but we return a CSuper!
obj.mut = C()                // Type checks correctly, because C <: CSuper
```

Now `obj` is of type `Covariant[CSuper]`. We can assign a `C` to `obj.mut`, but if we read `obj.mut`, we get an instance of `CSuper`, a superclass of `C`, and therefore not substitutable for a `C`, the type of the variable `c`.

Hence, if you think of a mutable field's type in terms of the corresponding getter and setter methods, it appears in both covariant position when read and contravariant position when written. There is no such thing as a type parameter that is both contravariant and covariant, so invariance is the only option for the type.

Improper Variance of Java Arrays

Scala `Arrays` are Java `Arrays`. Unfortunately, Java declares `Arrays` to be covariant in the type `T`, but `Arrays` are also mutable. We just learned this shouldn't be allowed. Instead, the Java compiler compiles code like the following example without error:

```
// src/main/java/progscala3/objectsystem/JavaArrays.java
package progscala3.objectsystem;

public class JavaArrays {
  public static void main(String[] args) {
    Integer[] array1 = new Integer[] {
      Integer.valueOf(1), Integer.valueOf(2), Integer.valueOf(3) };
    Number[] array2 = array1;          // Compiles fine, but shouldn't!!
    array2[2] = Double.valueOf(3.14);  // Compiles, but throws a runtime error!
  }
}
```

However, when you run it with `sbt`, hilarity ensues:

```
scala> runMain progscala3.objectsystem.JavaArrays
[info] Running progscala3.objectsystem.JavaArrays
[error] (runMain-4) java.lang.ArrayStoreException: java.lang.Double
java.lang.ArrayStoreException: java.lang.Double
  at progscala3.objectsystem.JavaArrays.main(JavaArrays.java:10)
  ...
```

Because Java arrays are covariant, we're allowed by the compiler to assign a Java `Double` to an `Array[Integer]` location. The compiler thinks this is OK, but in fact the array can only accept `Integer` values, so an exception is thrown.

3 Even though Scala wraps Java `Arrays`, the Scala class `scala.Array` is invariant in the type parameter, so the equivalent Scala program would not compile. Furthermore, Scala 3 introduced an immutable wrapper around `Arrays` called `scala.IArray` that we discussed in "Safer Pattern Matching with Matchable" on page 105.

See [Naftalin2006] for more details of Java's generics and arrays, from which this example was adapted.

Equality of Instances

Implementing a reliable equality test for instances is difficult to do correctly. [Bloch2008] and the Scaladoc page for `AnyRef.eq` describe the requirements for a good equality test. [Odersky2009] is a very good article on writing `equals` and `hashCode` methods correctly.

Since these methods are created automatically for case classes, tuples, and enumerations, I never write my own `equals` and `hashCode` methods anymore. I recommend the following practice:

> Any types you write that might be tested for equality or used as keys in a `Set` or a `Map` (where `hashCode` is used) should be case classes or enumerations. Tuples work well too.

Let's explore equality of instances in Scala, which can be tricky because of inheritance.

> Some of the equality methods have the same names as equality methods in other languages, but the semantics are sometimes different!

The equals Method

We'll use a case class to demonstrate how the different equality methods work:

```
// src/script/scala/progscala3/objectsystem/equality/Equality.scala

case class Person(firstName: String, lastName: String, age: Int)

val p1a = Person("Dean", "Wampler", 29)
val p1b = Person("Dean", "Wampler", 29)
val p2  = Person("Buck", "Trends",  30)
```

The `equals` method tests for value equality. That is, `obj1 equals obj2` is true if both `obj1` and `obj2` have the same value. Usually this is implemented by comparing their field values. They do not need to refer to the same instance:

```
assert((p1a.equals(p1a))  == true)
assert((p1a.equals(p1b))  == true)
assert((p1a.equals(p2))   == false)
assert((p1a.equals(null)) == false)
```

The == and != Methods

Whereas == is an operator in many languages, it is a method in Scala that delegates to `equals`. Similarly, `!=` calls == and returns the opposite value:

```
assert((p1a == p1a)   == true)
assert((p1a == p1b)   == true)
assert((p1a == p2)    == false)
assert((p1a == null)  == false)
```

```
assert((p1a != p1a)   == false)
assert((p1a != p1b)   == false)
assert((p1a != p2)    == true)
assert((p1a != null)  == true)
```

Comparing to `null` behaves as you might expect:

```
assert((null == p1a) == false)
assert((p1a == null) == false)
assert((null != p1a) == true)
assert((p1a != null) == true)
```

The eq and ne Methods

The `eq` method tests for reference equality. That is, `obj1 eq obj2` is true if and only if both `obj1` and `obj2` point to the same location in memory. These methods are only defined for `AnyRef`:

```
assert((p1a eq p1a)    == true)
assert((p1a eq p1b)    == false) // But p1a == p1b
assert((p1a eq p2)     == false)
assert((p1a eq null)   == false)
assert((null eq p1a)   == false)
assert((null eq null)  == true)
```

> In many languages, such as Java, C++, and C#, `==` behaves like `eq` instead of `equals`.

The `ne` method is the negation of `eq`. It is equivalent to `!(obj1 eq obj2)`.

Array Equality and the sameElements Method

Because `Arrays` are defined by Java, `equals` does reference comparison, like `eq`, rather than value comparison:

```
val a1 = Array(1, 2)
val a2 = Array(1, 2)
assert((a1.equals(a1)) == true)
assert((a1.equals(a2)) == false)
```

Instead, you have to use the `sameElements` method:

```
assert((a1.sameElements(a1)) == true)
assert((a1.sameElements(a2)) == true)
```

Because `Arrays` are mutable and have this behavior when comparing them, consider when it's better to use another collection instead. However, arrays do have performance benefits over most other collections. For example, an array of `Doubles` is a single block of memory, rather than having the elements scattered over the heap, as they would be for most other collections. This can greatly improve performance when fetching data into the CPU cache.

In contrast, Seqs, Maps, and most other collections work as you would expect:

```
val s1 = Seq(1, 2)
val s2 = Seq(1, 2)
assert((s1 == s1) == true)
assert((s1 == s2) == true)
assert((s1.sameElements(s2)) == true)

val m1 = Map("one" -> 1, "two" -> 2)
val m2 = Map("one" -> 1, "two" -> 2)
assert((m1 == m1) == true)
assert((m1 == m2) == true)
assert((m1.toSeq.sameElements(m2.toSeq)) == true)
```

Equality and Inheritance

We learned previously that case classes can't subclass other case classes. This would break `equals` and `hashCode`. The generated versions don't account for subtyping, where additional fields might be added. Consider this questionable example:

```
// src/script/scala/progscala3/objectsystem/equality/InheritanceEquality.scala

class Employee(val name: String, val annualSalary: Double)
class Manager(name: String, annualSalary: Double, val minions: Seq[Employee])
  extends Employee(name, annualSalary)

val e1  = new Employee("Buck Trends", 50000.0)
val e1b = new Employee("Buck Trends", 50000.0)
val e2  = new Employee("Jane Doe", 50000.0)
val m1  = new Manager("Jane Doe", 50000.0, Seq(e1, e2))
val all = Seq(e1, e1b, e2, m1)
```

Note that `e2` and `m1` have the same `name` and `annualSalary`.

This is a questionable use of inheritance, as we discussed in Chapter 9. I didn't use case classes to avoid the decision of which one of the two would be a good case class. I didn't define equality methods either, which means that reference equality, the default for objects on the JVM, will be used:

```
assert((e1 == e1)  == true)
assert((e1 == e1b) == false)   // Different references, so == returns false.
assert((e1 == e2)  == false)
assert((e2 == m1)  == false)
```

Suppose we decided to add `equals` methods? In particular, what should happen with `e2 == m1`, since they have the same `name` and `annualSalary` fields? For example, if I filter the list of all employees for those that are equal to `e2`:

```scala
val same = all.filter(e => e2 == e)
```

I might be tempted to argue that I'm just comparing `e2` to each `Employee`, so I only care about the `name` and `annualSalary`. When two objects have the same values for these two fields, I consider them equal, so I should consider `e2 == m1` equal. Certainly the `Employee.equals` method I might implement will only compare those two fields.

What if I write `m1 == e2` instead? Am I now thinking about whether two managers are equal? Now the `Manager.equals` method I might write will compare all three fields and return false for this comparison. In mathematics, equality is usually considered symmetric or commutative, where `a == b` means `b == a`.

The point is that equality might be contextual. If we really only care about employee fields, then maybe it's OK if `e2 == m1` returns `true`, but most of the time this creates ambiguities we should avoid. It reflects a poor design.

The resources mentioned at the beginning of this section all point out that a good `equals` method obeys the following rules for an *equivalence relation* (using ==):

- It is *reflexive*: For any `x` of type `Any`, `x == x`.
- It is *symmetric*: For any `x` and `y` of type `Any`, `x == y` returns the same value as `y == x`.
- It is *transitive*: For any `x`, `y`, and `z` of type `Any`, if `x == y` and `y == z`, then `x == z`.

Never use equality with instances in a type hierarchy where fields are spread between supertypes and subtypes. Only allow equality when comparing instances of the same concrete type. Obey the rules of an *equivalence relation*.

3 Multiversal Equality

The only reason I could even discuss the possibilities in the previous section is because Scala has historically followed Java's model of equality checking, which is called *universal equality*. This means the compiler allows us to compare any values to any other values, no matter what their types might be. In the previous section, I used `Any` as the type in the list of rules for equivalence relations, following the documentation for `Any.equals`.[2]

In Scala, universal equality translates to the following declarations for `equals`:

2 These methods may be moved to `Matchable` in a future release of Scala.

```
abstract class Any:
  def equals(other: Any): Boolean
  ...
abstract class AnyRef:
  def equals(other: AnyRef): Boolean
  ...
```

For implementation reasons, `AnyRef` can't use `Any` as the type for `other`, but otherwise, `Any.equals` defines how all `equals` methods work.

Scala 3 introduces *multiversal equality*, a mechanism for limiting what types can be compared to provide more rigorous type safety while also supporting backward compatibility. Instead of a "universe," we have a "multiverse," where equality checking is only allowed within each part of the multiverse.

Multiversal equality uses the `scala.CanEqual` type class first encountered in "Type Class Derivation" on page 158. There, we used an example `enum Tree[T] derives CanEqual` to only allow comparisons between trees with the same type `T`.

The `derives CanEqual` clause generates the following given instance:

```
given CanEqual[Tree[T], Tree[T]] = CanEqual.derived
```

The allowed types `T` must have their own given instance, `given CanEqual[T, T] = CanEqual.derived`.

For backward compatibility, this language feature must be turned on with the compiler flag `-language:strictEquality` or import statement `import scala.language.strictEquality`.

Scala 3 automatically derives `CanEqual` for the primitive types `Byte`, `Short`, `Char`, `Int`, `Long`, `Float`, `Double`, `Boolean`, `Unit`, `java.lang.Number`, `java.lang.Boolean`, `java.lang.Character`, `scala.collection.Seq`, and `scala.collection.Set`.

Additional given instances are defined as necessary to permit the following comparisons:

- Primitive numeric types can be compared with each other and with subtypes of `java.lang.Number`.
- `Boolean` can be compared with `java.lang.Boolean`.
- `Char` can be compared with `java.lang.Character`.
- Two arbitrary subtypes of `Seq` can be compared with each other if their element types can be compared. The two sequence types need not be the same. The same applies for `Set`. (This is a compromise with pragmatism.)
- Any subtype of `AnyRef` can be compared with `null`.

All these comparisons are symmetric, of course.

How does this feature improve type safety? Consider this contrived example:

```
// src/script/scala/progscala3/objectsystem/equality/CanEqualBug.scala

case class X(name: String)

def findMarkers[T](seq: Seq[T]): Seq[T] =                          ❶
  seq.filter(_ == X("marker"))

findMarkers(Seq(X("one"), X("two"), X("marker"), X("three")))      ❷

case class Y(name: String)                                         ❸
findMarkers(Seq(Y("one"), Y("two"), Y("marker"), Y("three")))      ❹
```

❶ Define a method to locate all the special instances of the class `X`.

❷ Use it, returning `List(X(marker))`.

❸ Introduce a new class `Y` as part of some enhancement.

❹ Reuse the same `findMarkers` code as before, but now it returns `Nil`.

The flaw is that `findMarkers` still uses the old value for the "marker." Because of universal equality, the compiler happily allows us to do the comparison `X("marker") == Y(…)`, which always returns false, and hence the filtering will always return `Nil`!

Enabling multiversal equality forces us to use a better design:

```
// src/script/scala/progscala3/objectsystem/equality/CanEqualBugFix.scala

import scala.language.strictEquality                               ❶

case class X(name: String) derives CanEqual                        ❷

def findMarkers[T](marker: T, seq: Seq[T])(                        ❸
    using CanEqual[T,T]): Seq[T] = seq.filter(_ == marker)

findMarkers(X("marker"), Seq(X("one"), X("two"), X("marker"), X("three")))

// Refactoring
case class Y(name: String) derives CanEqual
findMarkers(Y("marker"), Seq(Y("one"), Y("two"), Y("marker"), Y("three")))
```

❶ Enable the language feature.

❷ Derive from `CanEqual`.

❸ Pass in a type-compatible marker for filtering. The using clause limits allowed `T` values to those that derive `CanEqual`. Without this clause, you'll get a compilation error for `_ == marker` in `findMarkers`.

Now the last line will return the desired `List(Y(marker))`.

For advanced details and examples, see the multiversal equality documentation.

Case Objects and hashCode

The notion of equality goes hand in hand with the results of hashing an object, as done by `hashCode`, which is used in hash-based data structures, like the default `Map` and `Set` implementations. There is one gotcha with the implementation of `hashCode` for `case objects`, as demonstrated here:

```
// src/script/scala/progscala3/objectsystem/hashcode/CaseObjectHashCode.scala

scala> case object O1          // case object with no members
     |
     | case object O2:         // case object with two members
     |   val f = "O2"
     |   def m(i:Int): String = i.toString
     |
     | object O3:
     |   case object O4        // nested in another type

scala> println(s"O1:           ${O1.hashCode} == ${"O1".hashCode}")
     | println(s"O2:           ${O2.hashCode} == ${"O2".hashCode}")
     | println(s"O3:           ${O3.hashCode} != ${"O3".hashCode}")
     | println(s"O3.O4:        ${O3.O4.hashCode} != ${"O3.O4".hashCode}")
     | println(s"O3.O4 vs. O4: ${O3.O4.hashCode} == ${"O4".hashCode}")
     | println(s"O3.O4 vs. O3: ${O3.O4.hashCode} == ${"O3".hashCode}")
O1:           2498 == 2498
O2:           2499 == 2499
O3:           1595193154 != 2500
O3.O4:        2501 != 74524207
O3.O4 vs. O4: 2501 == 2501
O3.O4 vs. O3: 2501 == 2500
```

The compiler-generated `hashCode` for a `case object` simply hashes the object's name, without considering its members or its nesting inside other objects or packages. `O3` behaves better, but it's not a case object.

> Avoid using `case objects` as keys in maps and sets or other contexts where `hashCode` is used.

Recap and What's Next

We discussed covariance and contravariance for parameterized types. We explored comparison methods for Scala types and how careful design is required to correctly implement equivalence relations. We explored a new Scala 3 feature, multiversal equality, which makes equality checking more type safe. Finally, we noted a limitation of `hashCode` for case objects.

Next we'll continue our discussion of the object system by examining the behavior of field initialization during construction and member overriding and resolution in a type with multiple supertypes.

CHAPTER 12

Instance Initialization and Method Resolution

A type has a *directed acyclic graph* (DAG) of dependencies with its supertypes. When the fields in a type are spread over this DAG, initialization order can become important to prevent accessing a field before it is initialized. When one or more types in the DAG define and override the same method, then method resolution rules need to be understood. This chapter explores these concepts. As we'll see, they are closely related, governed by a concept called *linearization*.

Linearization of a Type Hierarchy

Because of single inheritance, if we ignored mixed-in traits, the inheritance hierarchy would be a simple linear relationship, one ancestor after another. When traits are considered, each of which may be a subtype of other traits and classes, the inheritance hierarchy forms a DAG.

The term *linearization* refers to the algorithm used to flatten this graph to determine the order of construction of the types in the graph, as well as the ordering of method invocations, including binding of `super` to invoke a supertype method. Think of the linearized traversal path flowing left to right with the supertypes to the left and the subtypes to the right. Construction follows this left-to-right ordering, meaning supertypes are constructed before subtypes. In contrast, when resolving which overloaded method to call in a hierarchy, the traversal goes right to left, where nearest super class definitions of the method take precedence over definitions farther away in the graph.

We saw an example of these behaviors in "Stackable Traits" on page 275. The `VetoableClickCountObserver` example required us to declare the two mixin traits in the order `Button(...) with ObservableClicks with VetoableClicks(maxAllowed =`

2). All three types defined a `click` method. We wanted to invoke the `VetoableClicks.click()` method first, which would only call the supertype `click` methods if we had not already exceeded the maximum number of allowed clicks. If not, it would call `ObservableClicks.click()`, which would first call `Button.click()` and then notify observers. Switching the order of declaration to `VetoableClicks with ObservableClicks` would mean that `Button.click()` is still only called up to an allowed number of invocations, but all invocations would trigger observation because `ObservableClicks.click()` would be called before `Vetoable.clicks()`.

Figure 12-1 diagrams a complicated hierarchy for a class `C3A`.

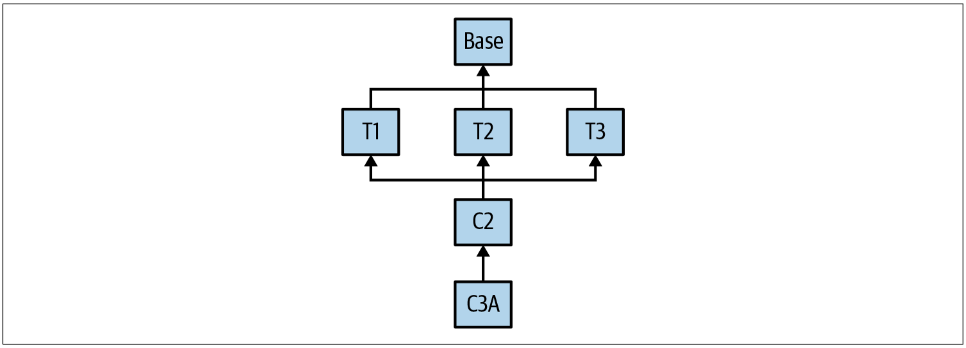


Figure 12-1. Type hierarchy example

Here is the corresponding code:

```
// src/script/scala/progscala3/objectsystem/linearization/Linearization.scala
trait Base:
  var str = "Base"                                          ❶
  def m(): String = "Base"                                  ❷

trait T1 extends Base:
  str = str + " T1"
  override def m(): String = "T1 " +  super.m()

trait T2 extends Base:
  str = str + " T2"
  override def m(): String = "T2 " +  super.m()

trait T3 extends Base:
  str = str + " T3"
  override def m(): String = "T3 " +  super.m()

class C2 extends T2:
  str = str + " C2"
  override def m(): String = "C2 " + super.m()

class C3A extends C2 with T1 with T2 with T3:
```

```
    str = str + " C3A"
    override def m(): String = "C3A " + super.m()

  class C3B extends C2 with T3 with T2 with T1:       ❸
    str = str + " C3B"
    override def m(): String = "C3B " + super.m()
```

❶ Use `str` to track construction ordering.

❷ Use `m` to track method invocation ordering.

❸ The only difference between `C3A` and `C3B` is the order of `T1`, `T2`, and `T3`.

The diagram doesn't draw a line from `C3A` to `T2`, even though it explicitly mixes it in, because it also gets mixed in through `C2`. `C3B` would have the same diagram, which doesn't capture composition ordering.

Let's see what happens:

```
scala> val c3a = new C3A
     | val c3b = new C3B
val c3a: C3A = C3A@4f0208a7
val c3b: C3B = C3B@7c6e67a6

scala> val c3c = new C2 with T1 with T2 with T3
scala> val c3d = new C2 with T3 with T2 with T1
val c3c: C2 & T1 & T3 = anon$1@75fcb651
val c3d: C2 & T3 & T1 = anon$2@2ae4d49a

scala> c3a.str     // Look at construction precedence.
     | c3b.str
val res0: String = Base T2 C2 T1 T3 C3A
val res1: String = Base T2 C2 T3 T1 C3B

scala> c3c.str
     | c3d.str
val res2: String = Base T2 C2 T1 T3
val res3: String = Base T2 C2 T3 T1

scala> c3a.m()     // Look at method invocation precedence.
     | c3b.m()
val res4: String = C3A T3 T1 C2 T2 Base
val res5: String = C3B T1 T3 C2 T2 Base

scala> c3c.m()
     | c3d.m()
val res6: String = T3 T1 C2 T2 Base
val res7: String = T1 T3 C2 T2 Base
```

Note the types reported for c3c and c3d. Only the order of T1 and T3 differ. Neither shows T2 because it is a parent of C2. Was it redundant for us to include T2 explicitly in the declaration of these anonymous objects?

The first four assertions show how the constructors are invoked, essentially left to right.

Consider the case of c3a. Before we can invoke the C3A constructor body, we have to construct the supertypes. The first one is C2, but it is a subtype of T2, which is a subtype of Base, so the construction order starts with Base, then T2, then C2. Next we move to T1, which depends on Base. We've already constructed Base, so we can immediately construct T1. Next is T2, but we've already constructed it. So it was redundant to include T2 in the declaration explicitly. Next is T3, which we can construct immediately. Finally, the C3A constructor body is invoked.

C3B just flips the order of the three traits, but note that T2 is always the first trait constructed because C2 goes first, but it requires Base and T2.

The output for c3c is very similar to c3a because it just constructs an anonymous type instead of the named type C3A, but with the exact same supertypes. The same argument applies for c3d and c3b.

Construction of Any and AnyRef happen before Base, but this isn't shown in Base.str.

The invocations of m show how method invocation precedence is implemented for overridden methods. It traverses the linearization in the opposite direction, right to left. You can see this when you compare c3a.str to c3a.m(), for example.

Working through c3a.m(), the C3A.m() method is called first. Then, to determine what to call for super.m() inside C3A.m(), we go right to left. T3 is a right-most type, so T3.m() is called next. You might think that the call to super.m() inside T3.m() should invoke Base.m(), but the precedence rules are evaluated globally for the instance, not by the static declaration of the trait. Hence, we go back to the declaration of C3A and see that T2 is next, but T2 is also a supertype dependency of C2, so we have to wait until C2.m() can be called. We keep going and arrive at T1; we can call its m() immediately, then we reach C2. C2 overrides m(), so we call it, then the super.m() call in C2.m() resolves to T2.m(). The super.m() call in T2.m() resolves to Base.m().

Linearization Algorithm

Here is the algorithm for calculating the linearization:

1. Put the actual type of the instance as the first element.
2. Starting with the rightmost supertype, work left, computing the linearization of each type, appending its linearization to the cumulative linearization. (Ignore `AnyRef` and `Any` for now.)
3. Working from left to right, remove any type if it appears again to the right of the current position.
4. Append `AnyRef` (or `AnyVal` for value classes), followed by `Any`.

Overly complex type hierarchies can result in method lookup surprises. If you have to work through this algorithm to figure out what's going on in your code, simplify it instead.

Initializing Abstract Fields

Initializing abstract fields in supertypes requires attention to initialization order. Consider this example that uses an undefined field before it is properly initialized:

```
// src/script/scala/progscala3/objectsystem/init/BadFieldInitOrder.scala

trait T1:
  val denominator: Int
  val inverse = 1.0/denominator                                   ❶

val obj1 = new T1:                                                ❷
  val denominator = 10

println(s"obj1: denominator = ${obj1.denominator}, inverse = ${obj1.inverse}")
```

❶ What is `denominator` when `inverse` is initialized?

❷ Construct an instance of an anonymous class where `T1` is the supertype.

This script prints `obj1: denominator = 10, inverse = Infinity`.

So `denominator` was 0, the default value for an `Int`, when `inverse` was calculated in `T1`. Afterward, `denominator` was initialized. Specifically, the trait body (constructor) was executed before the anonymous class body. A divide-by-zero exception wasn't thrown, but the compiler recognized the value as infinite.

3 Scala 3 adds a new experimental compiler flag `-Ysafe-init` that will issue a warning for common initialization problems like this. Pasting the previous example in a REPL with `-Ysafe-init` enabled prints the following:

```
...
9 |  val denominator = 10
  |      ^
  |Access non-initialized field denominator. Calling trace:
  | -> val inverse = 1.0/denominator                  [ rs$line$1:6 ]
```

The code examples don't use this flag because it rejects some safe code, such as the `LazyList` examples in "Left Versus Right Folding" on page 215. This may work once the feature is no longer experimental. Also, checking can be disabled for specific sections of code using the `@unchecked` annotation.

Scala provides three solutions to this ordering problem, the first two of which work for Scala 2.

The first solution is lazy values, which we discussed in "Lazy Values" on page 97:

```
// src/script/scala/progscala3/objectsystem/init/LazyValInit.scala

trait T2:
  val denominator: Int
  lazy val inverse = 1.0/denominator                              ❶

val obj2 = new T2:
  val denominator = 10

println(s"obj2: denominator = ${obj2.denominator}, inverse = ${obj2.inverse}")
```

❶ Added the keyword `lazy`.

This time, the print statement is `obj2: denominator = 10, inverse = 0.1`. Hence, `inverse` is initialized to a valid value, `0.1`, after `denominator` is initialized.

However, `lazy` only helps if the `inverse` isn't used too soon. For example, if you put a print statement at the beginning of the body of `T2` that references `inverse`, you'll force evaluation too soon!

The second solution is to define `inverse` as a method, which also delays evaluation, but only as long as someone doesn't ask for it:

```
// src/script/scala/progscala3/objectsystem/init/DefValInit.scala

trait T3:
  val denominator: Int
  def inverse = 1.0/denominator      // Use a method
...
```

While a `lazy val` is only evaluated once, a method is evaluated every time you invoke it. Recall that a `lazy val` also has some internal logic to check if initialization has already happened, which adds a bit of overhead on each invocation.

3 Finally, Scala 3 introduced parameters for traits, which is the most robust solution to initialization ordering issues and avoids the drawbacks of using lazy fields and methods:

```
// src/script/scala/progscala3/objectsystem/init/TraitParamValInit.scala

trait T4(val denominator: Int):                                    ❶
  val inverse = 1.0/denominator                                    ❷

val obj4 = new T4(10) {}                                           ❸

println(s"obj4: denominator = ${obj4.denominator}, inverse = ${obj4.inverse}")
```

❶ Pass a parameter to the trait, just as you do for classes.

❷ Use a regular, eager value for `inverse`.

❸ We have to provide a body, even though it is empty, when instantiating a trait.

To recap, the bad initialization showed us that the supertype's constructors are evaluated before the subtype, so `inverse` prematurely referenced `denominator`. We can either ensure that `denominator` is initialized first using a trait parameter or we can delay evaluation of `inverse` by making it either lazy or a method.

Overriding Concrete Fields

The same order of construction rules apply for concrete fields and accessing them. Here is an example with both a `val` that is overridden in a subtype and a `var` that is reassigned in the subtype:

```
// src/script/scala/progscala3/objectsystem/overrides/ClassFields.scala

trait T5:
  val name = "T5"
  var count = 0

class ClassT5 extends T5:
  override val name = "ClassT5"
  count = 1
```

Trying it:

```
scala> val c = ClassT5()
     | c.name
     | c.count
```

```
val res0: String = ClassT5
val res1: Int = 1
```

Just as for methods, the `override` keyword is required for the concrete `val` field `name` when it is overridden. `ClassT5` doesn't override the definition of the `var` field `count`. It just changes the assignment.

> Use caution when overriding concrete fields, for the same reasons you should use caution when overriding concrete methods (see "Overriding Methods? The Template Method Pattern" on page 251).

When designing supertypes, resist the urge the define a default field value unless it is likely to be used most of the time. Consider using the following idiom, where the use of `DEFAULT` indicates to the reader that overriding the value is sometimes expected:

```
scala> trait Alarm:
     |   val panicLevel: String = Alarm.DEFAULT_PANIC_LEVEL
     |
     | object Alarm:
     |   val DEFAULT_PANIC_LEVEL = "PANIC!!"

scala> val a = new Alarm:
     |   override val panicLevel: String = "Chillax"
val a: Alarm = anon$1@28361771

scala> a.panicLevel
val res2: String = Chillax
```

This is a contrived example. Use a trait parameter for `panicLevel` instead.

Abstract Type Members and Concrete Type Aliases

Recall that abstract type members become type aliases when they are given a concrete definition. Do they have similar initialization behavior compared to fields?

```
// src/script/scala/progscala3/objectsystem/init/TypeInitOrder.scala

trait TT1:
  type TA
  type TB = Seq[TA]                                    ❶
  val seed: TA
  val seq: TB = Seq.fill(5)(seed)                      ❷

class TT2 extends TT1:
  type TA = Int                                        ❸
  val seed: TA = 1
```

❶ This will work fine, even though `TA` is abstract at this point!

❷ Use `TB` to create a value.

❸ `TA` is initialized here, but `TB` will still be correctly initialized to `Seq[Int]`.

Does it work?

```
scala> val obj = TT2()
     | obj.seq
val res3: obj.TB = List(0, 0, 0, 0, 0)    // Should be 1s!
```

Not quite, but not because of the abstract type member `TA`. In this case, the `seed` is prematurely initialized to `0`, so the sequence has five zeros instead of five ones. Changing `seq` to a method or `lazy val` would fix this bug. Otherwise, the types work fine. You can even move the definition of `seed` before `TA` inside `TT2` without changing the behavior.

Unlike fields and methods, it is not possible to override a concrete type alias in a subtype.

Recap and What's Next

We walked through the details of Scala's linearization algorithm for type construction and method lookup resolution. We explored a few fine points of defining abstract members and overriding concrete members in subtypes.

In the next chapter, we'll learn about Scala's type hierarchy.

CHAPTER 13

The Scala Type Hierarchy

We've already seen many of the types available in Scala's library. Now, we'll fill in the details about the hierarchy of types. Chapter 14 will discuss the collections. Figure 13-1 shows the large-scale structure of the hierarchy for Scala types.

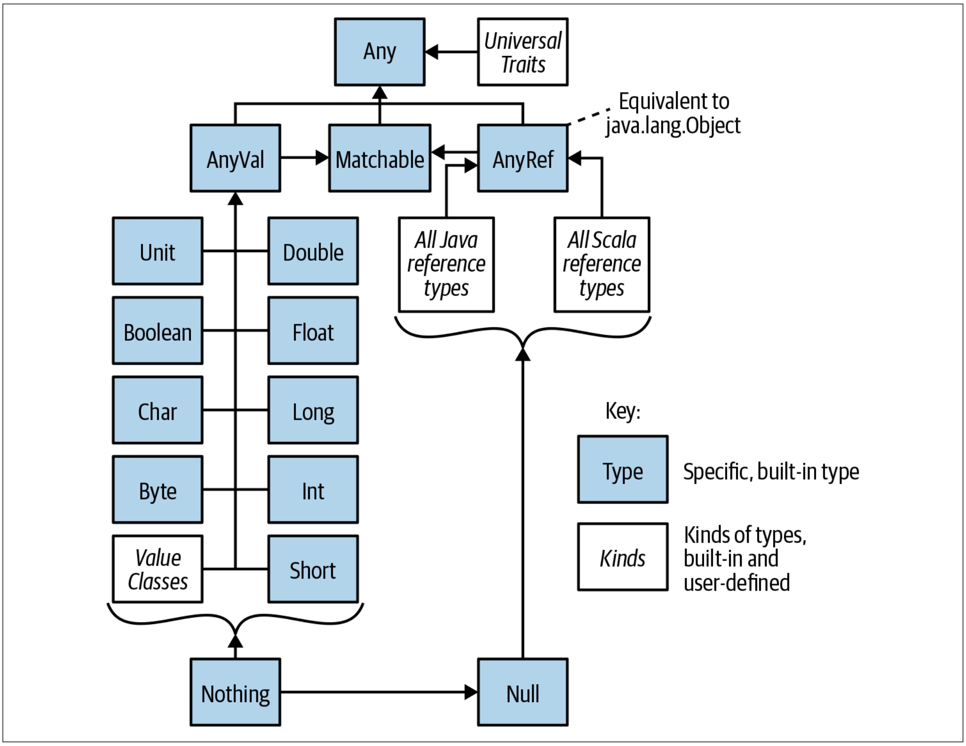


Figure 13-1. Scala's type hierarchy

3 At the root of the type hierarchy is `Any`. It has no supertypes and four subtypes:

- `Matchable`, the supertype of all types that support pattern matching, which we discussed in "Safer Pattern Matching with Matchable" on page 105. `Matchable` is also a supertype of `AnyVal` and `AnyRef`.
- `AnyVal`, the supertype of value types and value classes.
- `AnyRef`, the supertype of all reference types.
- Universal traits, which we discussed in "Value Classes" on page 258.

`AnyVal` has nine concrete subtypes, called the *value types*. They don't require heap allocation of instances. Seven of them are numeric value types: `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, and `Double`. The remaining two are nonnumeric: `Unit` and `Boolean`.

Value classes also extend `AnyVal` (see "Value Classes" on page 258).

In contrast, all the other types are reference types, because instances of them are allocated in the heap and managed by reference. They are subtypes of `AnyRef`.

3 Because `AnyVal` and `AnyRef` subtype `Matchable`, all their subtypes can be used in pattern matching. The only types for which values can't be used in pattern matching are `Any`, method type parameters and abstract types that are unbounded (i.e., don't have a `<: Foo` constraint), and type parameters and abstract types that are bounded only by universal traits, as discussed in "Safer Pattern Matching with Matchable" on page 105.

Let's discuss a few of the widely used types.

Much Ado About Nothing (and Null)

`Nothing` and `Null` are two unusual types at the bottom of the type system. Specifically, `Nothing` is a subtype of all other types, while `Null` is a subtype of all reference types.

`Null` is the familiar concept from most programming languages, although other languages may not define a `Null` *type*. It exists in Scala's type hierarchy because of the need to interoperate with Java and JavaScript libraries that support `null` values. However, Scala provides ways to avoid using `null`s, discussed in "Option, Some, and None: Avoiding Nulls" on page 60.

`Null` has the following definition:

```
abstract final class Null extends AnyRef
```

How can it be both `final` and `abstract`? This declaration disallows subtyping and creating your own instances, but the runtime environment provides one instance, `null`, our old nemesis.

Null is defined as a subtype of AnyRef, but it is also treated by the compiler as a subtype of all AnyRef types. This is the type system's formal way of allowing you to assign null to references for any reference type. On the other hand, because Null is not a subtype of AnyVal, it is not possible to assign null in place of an Int, for example.

Nothing has the following definition:

```
abstract final class Nothing extends Any
```

In contrast to Null, Nothing is a subtype of all other types, value types as well as reference types.

Nothing and Null are called *bottom types* because they reside at the bottom of the type hierarchy, so they are subtypes of all or a subset of the rest of the types.

Unlike Null, Nothing has no instances. We say the type is *uninhabited*. It is still useful because it provides two capabilities in the type system that contribute to robust, type-safe design.

The first capability is best illustrated with the List[+A] class. We now understand that List is covariant in A, so List[Nothing] must be a subtype of List[X] for any X. This is useful for the special case of an empty list, Nil, which is declared like this (omitting a few details):

```
package scala.collection.immutable
case object Nil extends List[Nothing]
```

Covariance of List and Nothing allows us to define just one object that works for all List[X] when we need an empty list. We don't need separate Nil[A] instances for each A. Just one will do.

The other use for Nothing is to represent expressions that terminate the program, such as by throwing an exception. An example is the special ??? method. It can be called in a temporary method definition so the method is concrete, allowing an enclosing, concrete type to compile:

```
object Foo:
  def bar(str: String): String = ???
```

However, if Foo.bar is called, an exception is thrown. Here is the definition of ??? inside scala.Predef:

```
def ??? : Nothing = throw NotImplementedError()
```

Because ??? "returns" Nothing, it can be called by any other method, no matter what type it returns.

Use Nothing as the return type of any method that always throws an exception, like ??? and sys.error, or terminates the application by calling sys.exit.

This means that a method can declare that it returns a "normal" type, yet choose to call sys.exit if necessary, and still type check. Consider this example that processes command-line arguments but exits if an unrecognized argument is provided:[1]

```
// src/main/scala/progscala3/objectsystem/CommandArgs.scala
package progscala3.objectsystem

object CommandArgs:

  val help = """
  |Usage: progscala3.objectsystem.CommandArgs arguments
  |Where the allowed arguments are:
  |  -h | --help                Show help
  |  -i | --in | --input path   Path for input (required)
  |  -o | --on | --output path  Path for output (required)
  |""".stripMargin

  def quit(status: Int = 0, message: String = ""): Nothing =         // 1
    if message.length > 0 then println(s"ERROR: $message")
    println(help)
    sys.exit(status)

  case class Args(inputPath: Option[String], outputPath: Option[String])

  def parseArgList(params: Array[String]): Args =                     // 2
    def pa(params2: Seq[String], args: Args): Args = params2 match   // 3
      case Nil => args                                                // 4
      case ("-h" | "--help") +: Nil => quit()                         // 5
      case ("-i" | "--in" | "--input") +: path +: tail =>
        pa(tail, args.copy(inputPath = Some(path)))
      case ("-o" | "--out" | "--output") +: path +: tail =>
        pa(tail, args.copy(outputPath = Some(path)))
      case _ => quit(1, s"Unrecognized argument ${params2.head}")     // 6

    val argz = pa(params.toList, Args(None, None))                    // 7
    if argz.inputPath == None || argz.outputPath == None then         // 8
      quit(1, "Must specify input and output paths.")
    argz

  def main(params: Array[String]): Unit =
    val argz = parseArgList(params)
    println(argz)
```

1. Print an optional message, followed by the help message, then exit with the specified error status. Following Unix conventions, 0 is used for normal exits and nonzero values are used for abnormal termination. Note that quit returns Nothing.

[1] Or use the scopt or mainargs libraries.

❷ Parse the argument list, returning an instance of `Args`, which holds the specified input and output paths determined from the argument list.

❸ A nested, recursively invoked function to process the argument list. We use the idiom of passing an `Args` instance to accumulate new settings by making a copy of it.

❹ End of input, so return the accumulated `Args`.

❺ Process options for help (where `quit` is invoked), input, and output. There are three ways the input and output flags can be specified.

❻ Handle the error of an unrecognized argument, calling `quit`.

❼ Call `pa` to process the arguments with a seed value for `Args`.

❽ Verify that the input and output arguments were provided.

I find this example of pattern matching particularly elegant and concise. It is fully type safe, even though an error or help request triggers termination.

Try it in `sbt` with various combinations of arguments: `runMain progscala3.objectsystem.CommandArgs…`

The scala Package

The top-level `scala` package defines many of the types we've discussed in this book, including most of the ones in Figure 13-1. This package is automatically in scope when compiling, so it's unnecessary to import it.

Table 13-1 describes the most interesting packages under `scala`.

Table 13-1. Interesting packages under `scala`

Name	Description
`annotation`	Where most annotations are defined. (Others are in `scala`.)
`collection`	All the collection types. See Chapter 14.
`concurrent`	Tools for concurrent programming. See Chapter 19.
`io`	A limited set of tools for input and output, such as `io.Source`.
`math`	Mathematical operations and definitions, including numeric comparisons.
`reflect`	Introspection on types. See Chapter 24.
`sys`	Access to underlying system resources, like properties and environment variables, along with operations like terminating the program.
`util`	Miscellaneous useful types, like `Either`, `Left`, `Right`, `Try`, `Success`, `Failure`, and `Using`.

The scala package defines aliases to popular types in other packages, so they are easily accessible. Examples include scala.math types like Numeric, Ordering, and Ordered; java.lang exceptions like Throwable and NullPointerException; and some of the collections types, like Iterable, List, Seq, and Vector.

Some annotations are also defined here, mostly those that signal information to the compiler, such as @main for marking a method as an entry point ("main" routine).

Finally, there are several types for case classes, tuples, and functions, discussed next.

Products, Case Classes, Tuples, and Functions

Case classes mix in the scala.Product trait, which provides a few generic methods for working with the fields of type and case-class instances:

```
scala> case class Person(name: String, age: Int)
scala> val p: Product = Person("Dean", 29)

scala> p.productArity     // Number of fields. For (1,2.2,3L), it would be 3.
val res0: Int = 2

scala> p.productIterator.foreach(println)
Dean
29

scala> (p.productElement(0), p.productElementName(0))
val res1: (Any, String) = (Dean,name)

scala> (p.productElement(1), p.productElementName(1))
val res2: (Any, String) = (29,age)

scala> (p.productElement(0), p.productElement(1))
val res3: (Any, Any) = (Dean,29)   // Note Any types.

scala> (p.productElementName(0), p.productElementName(1))
val res4: (String, String) = (name,age)

scala> val tup = ("Wampler", 39, "hello")
val tup: (String, Int, String) = (Wampler,39,hello)

scala> (tup.productArity, tup.productElement(2), tup.productElementName(2))
val res5: (Int, Any, String) = (3,hello,_3)
```

The tuple element name is the same as the method used to extract the value, _3 here.

While having generic ways of accessing fields can be useful, its value is limited by the fact that Any is used for the fields' types, not their actual types.

There are also subtypes of Product for specific arities, up to 22. They are supertypes of the corresponding TupleN types. For example, Product3[+T1,+T2,+T3] is defined for three-element products. These types add methods for selecting particular elements with the correct type information preserved. For example, Product3[+T1,+T2,+T3] adds these methods:

```
package scala
trait Product3[+T1, +T2, +T3] extends Product {
  abstract def _1: T1
  abstract def _2: T2
  abstract def _3: T3
  ...
}
```

For tuples and functions with 22 elements or less, there are corresponding TupleN and FunctionN types, for example Tuple3[+T1,+T2,+T3] and Function3[-T1,-T2,-T3,+R], which were discussed in "Functions Under the Hood" on page 286.

3 In Scala 2, tuples and functions were limited to 22 elements, but Scala 3 removes this limitation. For more than 22 elements, the compiler generates an instance of scala.TupleXXL and scala.FunctionXXL, respectively. In this case, an immutable array alias, scala.IArray, is used to store the elements for tuples and the arguments for functions.

While it may seem that 22 elements should be more than enough, especially when considering human comprehension, think about the case of a SQL query. It's common to want to put each record returned into a tuple or case class, but the number of columns can easily exceed 22.

3 Tuples and the Tuple Trait

Tuples are also subtypes of a new Scala 3 trait called Tuple, which adds several useful operations that make it easy to use tuples like collections or convert to collections. First, you can construct new tuples by prepending an element or concatenation:

```
// src/script/scala/progscala3/basicoop/Tuple.scala
scala> val t1 = (1, "two", 3.3)
     | val t2 = (4.4F, (5L, 6.6))
val t1: (Int, String, Double) = (1,two,3.3)
val t2: (Float, (Long, Double)) = (4.4,(5,6.6))

scala> val t01 = 0L *: t1                // Prepend an element
     | val t12 = t1 ++ t2                // Concatenate tuples
val t01: (Long, Int, String, Double) = (0,1,two,3.3)
val t12: Int *: String *: Double *: t2.type = (1,two,3.3,4.4,(5,6.6))
```

Note the last type for t12, t2.type. This is the *singleton type* for the particular instance (4.4,(5,6.6)).[2]

2 See "More on Singleton Types" on page 386 for more details.

You can pattern match with `*:`, like we did with sequences and `+:` before:

```
scala> val one *: two *: three *: four *: EmptyTuple = t01
val one: Long = 0
val two: Int = 1
val three: String = two
val four: Double = 3.3
```

You can drop or take elements or split a tuple:

```
scala> val t12d3 = t12.drop(3)           // Drop leading 3 elements
     | val t12d4 = t12.drop(4)           // Drop leading 4 elements
     | val t12t3 = t12.take(3)
     | val t12s3 = t12.splitAt(3)        // Like take and drop combined
val t12d3: (Float, (Long, Double)) = (4.4,(5,6.6))
val t12d4: (Long, Double) *: scala.Tuple$package.EmptyTuple.type = ((5,6.6),)
val t12t3: Int *: String *: Double *: EmptyTuple = (1,two,3.3)
val t12s3: (scala.Tuple.Take[Int *: String *: Double *: t2.type, 3],
  scala.Tuple.Drop[Int *: String *: Double *: t2.type, 3]
) = ((1,two,3.3),(4.4,(5,6.6)))
```

Look carefully at the type signature for `t12s3`. The `Take` and `Drop` types have a *dependent type* value `3`, based on the argument to `splitAt(3)`. We'll discuss dependent types in "Dependent Typing" on page 374.

You can convert to a few collection types and use the `Tuple` companion object to convert them to a tuple, but note that the element type returned is the least upper bound:

```
scala> val a  = t1.toArray               // Convert to collections
     | val ia = t1.toIArray
     | val l  = t1.toList
val a: Array[Object] = Array(1, two, 3.3)   // You may see Array[AnyRef]
val ia: opaques.IArray[Object] = Array(1, two, 3.3)
val l: List[Tuple.Union[t1.type]] = List(1, two, 3.3)

scala> val ta  = Tuple.fromArray(a)      // Convert to collections
     | val tia = Tuple.fromIArray(ia)
     | // val tl  = Tuple.fromList(l)    // Doesn't exist
val ta: Tuple = (1,two,3.3)
val tia: Tuple = (1,two,3.3)

scala> case class Person(name: String, age: Int)
     | val tp = Tuple.fromProduct(Person("Dean", 29))
val tp: Tuple = (Dean,29)
```

The last example extracted the elements of a case-class instance, an example of a `Product`, into a tuple. Finally, you can zip tuples:

```
scala> val z1 = t1.zip(t2)               // Note that t1's "3.3" is dropped
     | val z2 = t1.zip((4.4,5,6.6))      // Two, three-element tuples zipped
val z1: (Int, Float) *: (String, (Long, Double)) *: EmptyTuple =
  ((1,4.4),(two,(5,6.6)))
```

```
val z2: (Int, Double) *: (String, Int) *: (Double, Double) *: EmptyTuple =
  ((1,4.4),(two,5),(3.3,6.6))
```

Many other definitions can be found in the `scala` package, but we'll spend the rest of this chapter discussing the definitions in `Predef`.

The Predef Object

Some of the widely used definitions are in `scala.Predef`. Like the `scala` package, it is automatically in scope when you compile code, so you don't need to import it.

Let's summarize the contents of `Predef`. We'll wait to discuss some of them until Chapter 24.

Some useful definitions we've discussed before, like `summon` and `implicit` for working with anonymous given and implicit instances, and `<:<` and `=:=` for type comparisons, and `???`, discussed previously.

Implicit Conversions

First, `Predef` adds extension methods to some common JVM types by wrapping them with implicit conversions. The conversions have existed in Scala for a long time, so Scala 2 implicit conversions are used instead of Scala 3 extension methods, since Scala 3 uses the Scala 2 library. I provide examples here, but I won't list all the methods. See the `Predef` documentation for all the details.

First are conversions from `scala.Array` for specific `AnyVal` types and all `AnyRef` types. They can be converted to corresponding wrappers of type `scala.collection.mutable.ArraySeq`, providing all the methods from sequential collections for arrays:

```
implicit def wrapBooleanArray(
  xs: Array[Boolean]): scala.collection.mutable.ArraySeq.ofBoolean
...
implicit def wrapRefArray[T <: AnyRef](
  xs: Array[T]): scala.collection.mutable.ArraySeq.ofRef[T]
```

Having separate types for each of the `AnyVal` types exploits the fact that Java arrays of primitives are more efficient than arrays of boxed elements.

There are similar conversions to `scala.collection.ArrayOps`, which are similar to `ArraySeq`s, but the methods return `Array` instances instead of `ArraySeq` instances. Using `ArraySeq` is better when calling a chain of `ArraySeq` transformations.

Similarly, for `String`s, which are effectively *character arrays*, there are `WrappedString` and `StringOps` conversions:

```scala
implicit def wrapString(s: String): WrappedString
implicit def augmentString(x: String): StringOps
```

Having pairs of similar wrapper types, like ArraySeq/ArrayOps and WrappedString/StringOps, is confusing, but fortunately the implicit conversions are invoked automatically, selecting the correct wrapper type for the method you need.

Several conversions add methods to AnyVal types. For example:

```scala
implicit def booleanWrapper(b: Boolean): RichBoolean
implicit def byteWrapper(b: Byte): RichByte
...
```

The Rich* types add methods like comparison methods, such as <= and compare.

Why have two separate types for bytes, for example? Why not put all the methods in Byte itself? The reason is that the extra methods would force boxing of the value and allocation on the heap, due to implementation requirements for byte code. Recall that AnyVal instances are not heap allocated but are represented as the corresponding JVM primitives. So having separate Rich* types avoids the heap allocation except for those times when the extra methods are needed.

There are methods for converting between JVM boxed types for primitives and Scala AnyVal types, which make JVM interoperability easier:[3]

```scala
implicit def boolean2Boolean(x: Boolean): java.lang.Boolean
...
implicit def Boolean2boolean(x: java.lang.Boolean): Boolean
...
```

Finally, recall ArrowAssoc from "Extension Methods" on page 139. It is defined in Predef. For pattern matching on tuples, there is a definition in Predef val ->: Tuple2.type that supports using the same syntax:

```scala
scala> val x -> y = (1, 2)
val x: Int = 1
val y: Int = 2

scala> (1, 2) match:
     |   case x -> y => println(s"$x and $y")
     |   case _ => println("Error!!")
1 and 2
```

3 We'll see corresponding conversions between collections in Chapter 14.

Type Definitions

Predef defines several types and type aliases for convenience.

To encourage the use of immutable collections, Predef defines aliases for the most popular, immutable collection types:

```
type Map[A, +B]       = collection.immutable.Map[A, B]
type Set[A]           = collection.immutable.Set[A]
type Function[-A, +B] = (A) => B
```

Several convenient aliases point to JVM types:

```
type Class[T] = java.lang.Class[T]
type String = java.lang.String
```

Condition Checking Methods

Sometimes you want to assert a condition is true, perhaps to "fail fast" and especially during testing. Predef defines a number of methods that assist in this goal. The following methods come in pairs. One method takes a Boolean value. If false, an exception is thrown. The second version of the method takes the Boolean value and an error string to include in the exception's message. All the methods behave similarly, but the names convey different meanings, as shown in this example for a factorial method:

```
// src/script/scala/progscala3/hierarchy/Asserts.scala

scala> val n = 5
scala> assert(n > 0, s"Must assign a positive number. $n given.")
...
scala> def factorial(n: Long): Long = {  // Should really return BigDecimal.
     |   require(n >= 0, s"factorial($n): Must pass a positive number!")
     |   if n == 1 then n
     |   else n*factorial(n-1)
     | } ensuring(_ > 0, "BUG!!")
def factorial(n: Long): Long

scala> factorial(-1)
java.lang.IllegalArgumentException: requirement failed: factorial(-1):
  Must pass a positive number!
  ...

scala> factorial(5)
val res1: Long = 120
```

Many languages provide some form of assert. The require and assume (not shown) methods behave identically, but require is meant to convey that an input failed to meet the requirements, while assume verifies that assumptions are true. There is also assertFail(), which behaves like assert(false), and a corresponding assertFail(message).

Notice how ensuring is used to perform a final verification on the result and then return it if the assertion passes. Another Predef implicit conversion class, Ensuring[A], is invoked on the value returned from the block. It provides four variants of an ensuring method:

```
def ensuring(cond: (A) => Boolean, msg: => Any): A
def ensuring(cond: (A) => Boolean): A
def ensuring(cond: Boolean, msg: => Any): A
def ensuring(cond: Boolean): A
```

If the condition cond is true, the A value is returned. Otherwise an exception is thrown.

I'll discuss an approach to using these methods for writing robust code in "Better Design with Design by Contract" on page 474.

In Scala 2, if you wanted to turn the assertions off in production, passing -Xelide-below 2001 to the compiler would suppress compilation of them, except for the two require methods, because their definitions are annotated with @elidable(ASSERTION), where @elidable.ASSERTION is 2000. However, at the time of this writing, this feature is not implemented in Scala 3.

Input and Output Methods

We've enjoyed the convenience of writing println("foo"). Predef gives us four variants for writing strings to stdout:

```
def print(x: Any): Unit     // Print x as a String to stdout; no line feed
def printf(format: String, xs: Any*): Unit   // Printf-formatted string
def println(x: Any): Unit   // Print x as a String to stdout, with a line feed
def println(): Unit         // Print a blank line
```

All delegate to the corresponding scala.Console methods. For the printf format syntax, see java.util.Formatter.

Miscellaneous Methods

Finally, there are a few more methods defined in Predef you'll find useful that we haven't discussed before.

First, identity simply returns the argument. When using some of the combinator methods we discussed in Chapter 7, sometimes you'll apply a combinator, but no actual change is required for the inputs. Pass identity to the combinator for the function, which simply returns the input value.

In any context, if there is one unique value for a type that you want to get, use `valueOf[T]` to fetch it. For example:

```
scala> valueOf[EmptyTuple]
val res0: EmptyTuple.type = ()

scala> object Foo
     | valueOf[Foo.type]
val res1: Foo.type = Foo$@7210586e
```

Finally, `locally` works around some rare parsing ambiguities. See its documentation for details.

Recap and What's Next

We introduced the Scala type hierarchy and explored several definitions in the top-level package `scala` and the object `scala.Predef`. I encourage you to browse the *scala-lang.org/api* library documentation to learn more.

In the next chapter, we'll learn about Scala collections.

CHAPTER 14

The Scala Collections Library

This chapter finishes our discussion of the standard library with a discussion of the collections. They are organized in an object-oriented hierarchy with extensive use of mixins, yet their abstractions emphasize FP.

3 The whole Scala 2.13 standard library is reused in Scala 3 with some additions, but with no changes to the 2.13 content. The collections documentation provides a comprehensive discussion of the Scala collections. This chapter provides a succinct summary.

The collections were significantly redesigned for Scala 2.13. Nonetheless, most code from Scala 2 before 2.13 recompiles without change when upgrading to Scala 2.13. If deprecated features were used, most are now removed. See the collections migration documentation for details on moving from Scala 2.12 to 2.13 collections. Here, I'll focus on collections as they exist in Scala 2.13 and 3.

> To migrate from Scala 2 to Scala 3, I recommend first upgrading to Scala 2.13 to fix any issues with your use of collections and a few other changes, then upgrade to Scala 3.

We'll discuss how the collections are organized and some of the key types. We have used many of them in earlier chapters. In particular, Chapter 7 discussed most of the combinator methods like `map`, `flatMap`, and `filter`.

Different Groups of Collections

Table 14-1 lists the collection-related packages and their purposes.

Table 14-1. The collection-related packages and objects

Name	Description
`scala.collection`	Defines the base traits and objects needed to use and extend Scala collections.
`scala.collection.concurrent`	Defines a `Map` trait and `TrieMap` class with atomic, concurrent, and lock-free access operations.
`scala.collection.convert`	Defines types for wrapping Scala collections with Java collection abstractions and wrapping Java collections with Scala collection abstractions.
`scala.collection.generic`	Defines reusable components used to build collections in other packages.
`scala.collection.immutable`	Defines the immutable collections, the ones you'll use most frequently.
`scala.collection.mutable`	Defines mutable collections. Most of the specific collection types are available in mutable and immutable forms, but not all.
`scala.collection.parallel`	Parallelized versions of some collections.
`scala.jdk.CollectionConverters`	Implicit conversions for converting between Scala and Java collections.

The parallel collections were distributed in the Scala library before Scala 2.13, but they are now provided as a separate, community-maintained library. I won't discuss them further here.

There is also a separate "contrib" project available on GitHub. It has experimental extensions to the collections library, some of which may be added to the Scala library in a future release. These extensions include new types and new operations for existing types.

The rest of this chapter will focus on the most important types and idioms, but we'll also peek into the design of the library for useful tips you can apply in your own code. I encourage you to browse all these packages yourself. Look at the collections source code too. It is advanced but contains lots of useful design ideas.

Abstractions with Multiple Implementations

If you search the Scaladoc for `Map`, you get four types named `Map` and a page full of related types. This is typical for the most common abstractions. Fortunately, these four `Map`s are traits that declare or implement the same core abstractions but have some additional behaviors or different implementation details. For example, `scala.collection.Map` defines the common read-only operations, which are implemented by `scala.collection.immutable.Map`, while `scala.collection.mutable.Map` adds self-modification methods that don't exist in the other two types. Other types are handled similarly.

To emphasize using immutable collections, the immutable versions of `Seq`, `IndexedSeq`, `List`, `Map`, and `Set` are in scope without explicit import statements. When you want to use mutable collections or other immutable ones, you have to import them explicitly.

All the collections have companion objects with `apply` methods and other methods for creating instances. For the abstract types, like `Map`, `Seq`, and `Set`, instances of concrete subtypes are created.

For example, calling `Map("one" -> 1,...)` will return a `HashMap` if more than four key-value pairs are specified. However, for less than four key-value pairs, it's more efficient to do linear search of the keys when retrieving values! Hence, there are final classes called `EmptyMap` (for no elements) and `Map1` through `Map4` that `Map.apply` uses when given zero through four key-value pairs, respectively. When five or more pairs are specified to the `apply` method, a `HashMap` is returned. Immutable sets are implemented the same way.

Similarly, if a key-value pair is added to a four-element `Map4` instance, a new `HashMap` instance is returned. If a key-value pair is added to a `Map3` instance, a `Map4` is returned, and so forth.

This is a good pattern for your code. All the subtypes of `immutable.Map` behave exactly the same (ignoring performance differences). They differ in the implementation, which is optimized for the data they hold.

> Define an abstraction with implementations that optimize for different contexts, but with identical user-visible behavior. In the abstraction's companion object, let the `apply` methods choose the best implementation to instantiate for a given context.

Hence, most of the time you'll only think about `immutable.Map` or maybe `mutable.Map`. You'll concern yourself with concrete implementations when performance or other considerations requires a more careful choice.

Let's discuss each package, starting with the most important, `immutable` and `mutable`.

The scala.collection.immutable Package

You'll work with collections in the `scala.collection.immutable` package most of the time. Because they are immutable, they are thread-safe. Table 14-2 provides an alphabetical list of the most commonly used types.

Table 14-2. Most commonly used immutable collections

Name	Description
`ArraySeq[+A]`	Wrap arrays with `Seq` operations and covariance in the type parameter `A`. It is effectively immutable, even though the underlying array is mutable.
`BitSet`	Memory-efficient sets of nonnegative integers. The entries are represented as variable-size arrays of bits packed into 64-bit words. The largest entry determines the memory footprint of the set.
`HashMap[K, +V]`	Maps implemented with a compressed hash-array mapped prefix tree.
`HashSet[A]`	Sets implemented with a compressed hash-array mapped prefix tree.
`IndexedSeq[+A]`	Indexed sequences with efficient (*O(1)*) `apply` (indexing) and `length`.
`Iterable[+A]`	General abstraction for iterating through the collection with different operations.
`LazyList[+A]`	Final class for a linked list whose elements are evaluated in order and only when needed, thereby supporting potentially infinite sequences. It replaces the deprecated `Stream` type.
`List[+A]`	Sealed abstract class for linked lists, with *O(1)* head and tail access, and *O(N)* access to interior elements.
`ListMap[K, +V]`	A map backed by a list that preserves the insertion order.
`ListSet[+A]`	A set backed by a list that preserves the insertion order.
`Map[K, +V]`	Unordered, iterable collection of key-value pairs, with *O(1)* random access. The companion object factory methods construct instances depending on the input key-value pairs, as discussed previously.
`Nil`	An object for empty lists. Subtype of `List`.
`NumericRange[+A]`	A more generic version of the `Range` class that works with arbitrary numeric types.
`Queue[+A]`	A FIFO (first-in, first-out) queue.
`Range`	Integer values in a range between a start and end point with nonzero step size.
`Seq[+A]`	Immutable sequences. The companion object `apply` methods construct `Lists`.
`SeqMap[K, +V]`	Abstraction for maps that preserve the insertion order.
`Set[A]`	Unordered, iterable collection of unique elements, with *O(1)* random access. The companion object factory methods construct instances depending on the input key-value pairs, as discussed previously.
`SortedMap[K, +V]`	Maps with an iterator that traverses the elements in sorted order according to `math.Ordering` on the keys.
`SortedSet[A]`	Sets with an iterator that traverses the elements in sorted order according to `math.Ordering` on the keys.
`TreeMap[K, +V]`	A map with underlying red-black tree storage with *O(log(N))* operations.
`TreeSet[A]`	A set with underlying red-black tree storage with *O(log(N))* operations.
`Vector[+A]`	An indexed sequence with *O(1)* operations.
`VectorMap[K, +V]`	A map with an underlying vector implementation that preserves insertion order.

`LazyList` was introduced in Scala 2.13, replacing the now-deprecated `Stream` type. `LazyList` is fully lazy, while `Stream` is lazy in the tail, but not the head element. We discussed an example of `LazyList` in "Left Versus Right Folding" on page 215.

`Vector` is implemented using a tree-based, persistent data structure, as discussed in "What About Making Copies?" on page 222. It provides excellent performance, with amortized, nearly *O(1)* operations.[1]

The scala.collection.mutable Package

There are times when you'll need a mutable collection. The mutation operations on these collections are not thread-safe. However, careful use of mutable data can be appropriate for performance and other reasons.

Table 14-3 lists the most commonly used collections unique to the `mutable` package. Many of the collections in `scala.collection.immutable` have mutable alternatives, but they aren't shown here, for brevity.

Table 14-3. Most commonly used mutable collections

Name	Description
`AnyRefMap[K <: AnyRef, V]`	Map for `AnyRef` keys, implemented with a hash table and *open addressing*. Most operations are faster than for `HashMap`.
`ArrayBuffer[A]`	A buffer class that uses an array for internal storage. Append, update, and random access take *O(1)* (amortized) time. Prepends and removes are *O(N)*.
`ArrayBuilder[A]`	A builder class for arrays.
`ArrayDeque[A]`	A double-ended queue. It uses a resizable circular buffer. Operations like `append`, `prepend`, `removeFirst`, `removeLast`, and random-access lookup and replacement take amortized *O(1)* time.
`Buffer[A]`	Sequences that can be expanded and contracted.
`Clearable`	Collections that can be cleared with a `clear()` method.
`Cloneable[+C <: AnyRef]`	Collections that can be cloned.
`LinkedHashMap[K, V]`	A hash-table based map where the elements can be traversed in their insertion order.
`LinkedHashSet[A]`	A hash-table based set where the elements can be traversed in their insertion order.
`ListBuffer[A]`	A `Buffer` implementation backed by a list.
`MultiMap[K, V]`	A map where multiple values can be assigned to the same key.
`PriorityQueue[A]`	A heap-based, mutable priority queue. For the elements of type `A`, there must be an implicit `Ordering[A]` instance.
`Stack[A]`	A LIFO (last-in, first-out) stack.
`WeakHashMap[K, V]`	A mutable hash map with references to entries that are weakly reachable. Entries are removed from this map when the key is no longer (strongly) referenced. This class wraps `java.util.WeakHashMap`.

1 Technically, $O(log_{32}(N))$ which is very close to constant.

Whereas immutable collections are usually covariant in their element types, these collections are invariant because the elements are written as well as read. See "Variance of Mutable Types" on page 290 for a discussion of why this is necessary.

`MultiMap` is useful when you want an easy way to add more values for a given key, whereas the normal map operations like `+` will overwrite an older value. Here is an example:

```
// src/script/scala/progscala3/collections/MultiMap.scala
scala> import collection.mutable.{HashMap, MultiMap, Set}                  ❶

scala> val mm = HashMap[Int, Set[String]] with MultiMap[Int, String] ❷

scala> mm.addBinding(1, "a")                                               ❸
     | mm.addBinding(2, "b")
     | mm.addBinding(1, "c")                                               ❹
...
val res3: collection.mutable.HashMap[Int, collection.mutable.Set[String]] &
  collection.mutable.MultiMap[Int, String] =
    HashMap(1 -> HashSet(a, c), 2 -> HashSet(b))

scala> mm.entryExists(1, _ == "a") == true                                 ❺
     | mm.entryExists(1, _ == "b") == false
     | mm.entryExists(2, _ == "b") == true

scala> mm.removeBinding(1, "a")                                            ❻
     | mm.entryExists(1, _ == "a") == false
     | mm.entryExists(1, _ == "c") == true
```

❶ Import the types we need.

❷ Create a mutable `HashMap` that mixes in `MultiMap`.

❸ Use `addBinding` to add all key-value pairs. The values are actually stored in a `Set`.

❹ Add a second binding for key `1`.

❺ Use `entryExists` to test if a binding is defined.

❻ Use `removeBinding` to remove a value. A binding to `c` remains for `1`.

A few other types of collections are found here, but don't have `immutable` equivalents, including `PriorityQueue` and `Stack`.

The mutable collections have methods for adding and removing elements. Consider the following example (with some detailed elided) that uses `scala.collection.mutable.ArrayBuffer`, which is also the concrete class that `scala.collection.mutable.Seq(…)` instantiates.

```scala
// src/script/scala/progscala3/collections/MutableCollections.scala
import collection.mutable.ArrayBuffer

val seq = ArrayBuffer(0)

seq ++=  Seq(1, 2)              // Alias for appendAll
seq.appendAll(Seq(3, 4))        // Append a sequence
seq += 5                        // Alias for addOne
seq.addOne(6)                   // Append one element
seq.append(7)                   // Append one element
assert(seq == ArrayBuffer(0, 1, 2, 3, 4, 5, 6, 7))

Seq(-2, -1) ++=: seq            // Alias for prependAll
seq.prependAll(Seq(-4, -3))     // Prepend a sequence
-5 +=: seq                      // Alias for prepend
seq.prepend(-6)                 // Prepend one element
assert(seq == ArrayBuffer(-6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7))

seq -= -6                       // Alias for subtractOne
seq.subtractOne(7)              // Remove the element
seq --= Seq(-2, -4)             // Alias for subtractAll
seq.subtractAll(Seq(2, 4))      // Remove a sequence
assert(seq == ArrayBuffer(-5, -3, -1, 0, 1, 3, 5, 6))
```

The assertions show the state after each block has finished. Note that the operator forms have equals signs in their names, which is a clue that they modify the collection in place.

When subtracting, if an element doesn't exist in the collection, no change is made and no error is raised.

It can be confusing which methods modify a mutable collection versus return a new collection. Read the Scaladoc carefully to determine what happens. For example, `append` and `addOne` modify the collection, while `appended` returns a new collection.

Also, most of the mutable collections have in-place alternative implementations for many of the common methods, like `mapInPlace` instead of `map` and `takeInPlace` instead of `take`. They are designed to be more efficient by modifying the collection rather than returning a new copy. Note that supertype traits like `scala.collection.mutable.Seq` don't declare these methods, but you can search the Scaladoc for `InPlace` to find all of them.

The scala.collection Package

Let's briefly discuss the other packages, all of which you will use rarely. They mostly support implementing collections.

The types defined in `scala.collection` declare the abstractions shared in common by the immutable, mutable, concurrent, and parallel collections. A few types in this package are concrete, as are many methods in the abstract classes and traits. Table 14-4 lists several of the core types in this package.

Table 14-4. Most important core scala.collection types

Name	Description
`ArrayOps[A]`	Wrap arrays to add many of the indexed sequence operations. Discussed in "Implicit Conversions" on page 319.
`Factory[-A,+C]`	Build a collection of type `C` with elements of type `A`.
`IndexedSeq[+A]`	Indexed sequences with efficient (*O(1)*) `apply` (indexing) and `length` operations.
`Iterable[+A]`	General abstraction for iterating through the collection with different operations.
`Iterator[+A]`	Data structure for iterating over a sequence of elements.
`LinearSeq[+A]`	Sequences with efficient (*O(1)*) `head` and `tail` operations.
`Map[K, +V]`	Unordered, iterable collection of key-value pairs, with *O(1)* random access.
`Seq[+A]`	Ordered, iterable sequences of elements, with *O(N)* random access.
`Set[A]`	Unordered, iterable collection of unique elements, with *O(1)* random access.
`SortedMap[K, +V]`	Maps sorted by the keys according to `math.Ordering`.
`SortedSet[A]`	Sorted sets.
`StringOps`	Wraps strings to add many of the indexed sequence operations.
`StringView`	Similar to `StringOps`, but most of the methods return `View[Char]` instead of `Strings` or `IndexedSeqs`.
`View[+A]`	Collections whose transformation operations are nonstrict, meaning the elements are evaluated only when the view is traversed or when the view is converted to a strict collection type using the `to` operation.

For traits like `Iterable`, you'll find `*Ops` traits that implement many of the methods.

`Iterator` provides two methods, `hasNext`, which returns true if more elements exist to visit or false otherwise, and `next`, which returns the next available element. Hence, it is lazy. It only does work when you ask for the next element.

`Map` and `SortedMap` are invariant in their key types, but covariant in the value types. The element types for `Set`, `SortedSet`, and `SortedOps` are also invariant. Why the invariance? It's because `hashCode` is used to test for uniqueness of these keys. We learned in "Equality and Inheritance" on page 295 that you don't want to mix `equals`/`hashCode` and subtypes!

The purpose of `Factory` is discussed in "Polymorphic Methods" on page 336.

The scala.collection.concurrent Package

This package defines only two types, a `scala.collection.concurrent.Map` trait and one implementation of it—a hash-trie `scala.collection.concurrent.TrieMap`.

Map extends scala.collection.mutable.Map, but it makes the operations atomic, so they support thread-safe, concurrent access.

TrieMap is a concurrent, *lock-free* implementation of a hash-array mapped *trie* data structure. It aims for scalable concurrent insert and remove operations and memory efficiency.

The scala.collection.convert Package

The types defined in the scala.collection.convert package are used to implement converters wrappers of Scala collections as Java collections and vice versa. There have been several iterations of converters over various releases of Scala. All but the latest are deprecated.

Don't use this package directly. Instead, access the conversions using jdk.CollectionConverters. The jdk package also provides conversions between other types, as well as other utilities to support JDK interoperability. The conversions are usually wrappers, to avoid copying. Because most Java collections are mutable, the returned Scala collection will usually be a scala.collection type. The types in this package don't have mutation methods, to encourage immutable programming, which is why a scala.collection.mutable type is not returned. See "Conversions Between Scala and Java Collections" on page 463 for more details.

The scala.collection.generic Package

Whereas scala.collection declares abstractions for all collections, scala.collection.generic provides reusable components for implementing the specific mutable, immutable, parallel, and concurrent collections. Most of the types are only of interest to implementers of collections, so I won't discuss them further.

Construction of Instances

Let's now explore some of the key concepts and idioms in the collections.

Companion object apply methods are used as factories for all the collections. Even the abstract types, like Seq and IndexedSeq, have companion objects and apply methods that construct concrete subtypes:

```
scala> val seq = Seq(1,2,3)
val seq: Seq[Int] = List(1, 2, 3)

scala> val iseq = IndexedSeq(1,2,3)
val iseq: IndexedSeq[Int] = Vector(1, 2, 3)
```

Note that a `Vector` is created for `IndexedSeq`, rather than a `List`, because `IndexedSeqs` require efficient fetching of elements by index, for which `apply` is used, and `length` methods. For `Vector`, both are *O(1)*, while they are *O(N)* for `List`.

The companion objects may also have special-purpose factory methods. Here are a few examples for `scala.collection.immutable.Seq`. Other collection types have similar methods:

```
scala> Seq.empty[Int]                                               ❶
val res0: Seq[Int] = List()

scala> Seq.concat(0 until 3, 3 until 6, 6 until 9)                  ❷
val res1: Seq[Int] = List(0, 1, 2, 3, 4, 5, 6, 7, 8)

scala> Seq.fill(5)(math.random)                                     ❸
val res2: Seq[Double] = List(0.6292941497613453, ..., 0.3902341382377057)

scala> Seq.tabulate(5)(index => index*2)                            ❹
val res47: Seq[Int] = List(0, 2, 4, 6, 8)

scala> Seq.range(10, 15)                                            ❺
val res48: Seq[Int] = NumericRange 10 until 15

scala> Seq.iterate(2, 5)(index => index*2)                          ❻
val res49: Seq[Int] = List(2, 4, 8, 16, 32)
```

❶ Return an empty sequence of integers. Note that `res0 == Nil` is true.

❷ Concatenate zero or more sequences, three in this case.

❸ Fill an *N*-element sequence with values returned from the expression.

❹ Fill an *N*-element sequence with the values returned when the index is passed to the function.

❺ Return a numeric range.

❻ For each index `i`, `f` is called `i` times; for example, for `i=3`, `f(f(f(seed)))`. `2` is the seed in this case.

The Iterable Abstraction

The trait `scala.collection.Iterable` is a supertype of all the collections. Almost all the methods that are common to immutable and mutable collections, including the functional operators (see Chapter 7), are defined here and implemented by `scala.collection.IterableOps`.

For some methods that suggest an ordering, the results have undefined behaviors when called for unordered collections, like `HashSets`:

```
scala> val seq2 = Seq(1,2,3,4,5,6).drop(2)
val seq2: Seq[Int] = List(3,4,5,6)

scala> val set = Set(1,2,3,4,5,6)
val set: Set[Int] = HashSet(5, 1, 6, 2, 3, 4)

scala> val set2 = set.drop(2)
val set2: Set[Int] = HashSet(6, 2, 3, 4)

scala> val orderedSet = collection.immutable.TreeSet(1,2,3,4,5,6)
val orderedSet: ...immutable.TreeSet[Int] = TreeSet(1, 2, 3, 4, 5, 6)

scala> val orderedSet2 = orderedSet.drop(2)
val orderedSet2: ...immutable.TreeSet[Int] = TreeSet(3, 4, 5, 6)
```

While the default `Set` is unordered, `TreeSet` is one of the ordered implementations of `Set`.

Note that the element order shown by `set2.toString` is at least consistent with the result of dropping the first two elements.

All the `Iterable` members are concrete except for a single abstract method `def iterator: Iterator[A]`, where `Iterator` is a data structure supporting traversal through the collection. Concrete subtypes define this method for their particular data `structure`.

The signature of `Iterable.map` is the following:

```
trait Iterable[+A] extends ... :
  def map[B](f: A => B): Iterable[B]
  ...
```

However, if we use it, notice what happens:

```
scala> val doubles = Seq(1,2,3).map(2 * _)
val doubles: Seq[Int] = List(2, 4, 6)

scala> val doubles2 = Map("one" -> 1.1, "two" -> 2.2).map((k,v) => (k,2*v))
val doubles2: Map[String, Double] = Map(one -> 2.2, two -> 4.4)
```

The return types are `List` and `Map`, respectively, not `Iterable`. We prefer getting back a new `List` and `Map`, but how is `map` implemented so that the correct subtype is returned, rather than a more generic `Iterable`?

Polymorphic Methods

A challenge with object-oriented type hierarchies is defining reusable methods in supertypes, like `map`, yet returning a more appropriate concrete subtype. Let's explore how the collections solve this design problem.

> The implementation details here are quite technical, but you can skip the rest of this section without missing any essential information about using collections.

The implementation of `map` is defined by `scala.collection.IterableOps`, which `Iterable` mixes in. Here is the signature and implementation (simplified slightly):

```
trait IterableOps[+A, +CC[_], +C]:
  def map[B](f: A => B): CC[B] =
    iterableFactory.from(new View.Map(filtered, f))
  ...
```

The `CC[B]` type is the new type to instantiate. For `List`, it will be `List`, for `HashMap` it will be `HashMap`, etc., although it could vary from the input type. `CC` is technically called the *type constructor* of the collection in this context. The underscore wildcard is needed in the first line because the specific element type won't be known until a method like `map` is invoked, where the element type will be some `B`, whatever the function `f: A => B` returns.[2] The `C` type is the current concrete type of the collection, which is fully known.

For example, for a `Map[K,V]`, we have effectively `IterableOps[(K, V), Map, Map[K, V]]`. The element type is the tuple `(K, V)`.

The method `iterableFactory` is abstract in `IterableOps`. Concrete collections corresponding to `C` define this method to return a suitable `scala.collection.IterableFactory` instance that can construct a new instance of the appropriate output collection type. For example, if `CC` is `List`, then the factory will construct a new `List`. The new instance might be a different type, though. Recall that a `Map` could be an instance of `EmptyMap`, `Map1` through `Map4`, or `HashMap`. So if I have a `Map4` and I call a `flatMap(…)` that returns a new `Map` with five or more elements, I'll actually get a `HashMap` back.

2 Recall that use of _ as the wildcard is being replaced with ? in Scala 3, but we're still using the Scala 2.13 library.

Sometimes you may see a type returned by the REPL annotated with `@uncheckedVariance`. The mechanism we just described requires `CC` and `C` to have the same variance behavior (recall "Parameterized Types: Variance Under Inheritance" on page 285) to preserve sound behavior of the types. However, Scala's type system has no mechanism to enforce this constraint, so the annotation is added to the element types to indicate that this constraint was unchecked by the compiler. It is perfectly safe; the standard library implements the correct behavior for all its collection types.

Equality for Collections

When are two collections considered equal? First, the library divides the collections into groups for this purpose: maps, sequences, and sets. Two instances from different groups are never considered equal, even if both contain the same elements.

Within the same group, two instances are considered equal if they have the same elements, even if the specific collection type is different. This is a pragmatic choice, but rarely problematic, because when comparing collections, you are less likely to be interested in their implementations and more likely to be interested in their contents. Sequences have the additional requirement that the elements must be in the same order:

```
scala> List(1,2,3) == Vector(1,2,3)
val res0: Boolean = true

scala> List(1,2,3) == Vector(3,2,1)
val res1: Boolean = false

scala> Set(1,2,3) == Set(3,2,1)
val res2: Boolean = true

scala> import collection.immutable.VectorMap
scala> Map("one" -> 1, "two" -> 2) == Map("two" -> 2, "one" -> 1)
     | Map("one" -> 1, "two" -> 2) == VectorMap("two" -> 2, "one" -> 1)
     | VectorMap("one" -> 1, "two" -> 2) == Map("two" -> 2, "one" -> 1)
     | VectorMap("one" -> 1, "two" -> 2) == VectorMap("two" -> 2, "one" -> 1)
     |
val res36: Boolean = true
val res37: Boolean = true
val res38: Boolean = true
val res39: Boolean = true
```

It doesn't matter that `VectorMaps` are ordered because the contract of a map represents key-value pairs where the order is not guaranteed. However, an ordered map can be useful in contexts other than equality checking.

Nonstrict Collections: Views

In general, many of the collection operations are called *transformers* because they transform the input collection to something new. By default, these operations are *strict*, meaning they run when called and create output collections.

If we plan to perform a sequence of transformations, it's inefficient to create all those intermediate collections. Lazy evaluation could help us avoid this overhead. We saw `LazyList` before, a sequence where all operations are performed lazily, including evaluation of the elements.

For other collections, calling the `view` method returns a special kind of wrapper collection, called a `scala.collection.View`, which makes all the transformer operations lazy by default, rather than strict. Some of the other operations will still be strict when it's unavoidable. This means that instead of returning a new collection immediately, a new `View` is returned by a transformer operation. When these operations are sequenced together, the net result is that no intermediate collections are created. Instantiation only happens when a step requires a concrete collection as output.

Consider the following example that uses `groupBy` methods:

```
// src/script/scala/progscala3/collections/GroupBy.scala

scala> val ints = (1 until 100).toVector
val ints: Vector[Int] = Vector(1, 2, 3, ..., 97, 98, 99)

scala> val thirds = ints.groupBy(_%3)                                   ❶
val thirds: Map[Int, Vector[Int]] = HashMap(
  0 -> Vector(3, 6, ..., 99), ... 2 -> Vector(2, 5, ..., 98))

scala> val thirds1a = thirds.view.mapValues(ns => ns.map(n => (n,2*n))) ❷
val thirds1a: scala.collection.MapView[Int, Vector[(Int, Int)]] =
  MapView(<not computed>)

scala> val thirds1b = thirds1a.toMap                                    ❸
val thirds1b: Map[Int, Vector[(Int, Int)]] = Map(
  0 -> Vector((3,6), (6,12), ...), ..., 2 -> Vector((2,4), (5, 10), ...))

scala> val thirds2 = ints.groupMap(_%3)(n => (n,2*n))                   ❹
val thirds2: Map[Int, Vector[(Int, Int)]] = Map(
  0 -> Vector((3,6), (6,12), ...), ..., 2 -> Vector((2,4), (5, 10), ...))
```

❶ Split the integers into groups, modulo 3. Note the three keys.

❷ Map over the integer values, converting them into tuples. The keys are unchanged. We first convert to a `scala.collection.MapView`.

❸ Convert to a strict map, forcing evaluation.

❹ For reference, `groupMap` combines `groupBy` and the element-wise `mapValue` into one operation.

The performance advantages of nonstrict evaluation become important when working with large collections, for example, data processing applications.

Be careful when strict behavior is actually required. Don't be surprised when the following doesn't actually start any work!

```
(0 until 100).view.map { i => doAsynchronousWork(i) }
```

Recap and What's Next

We rounded out our understanding of the Scala collections, including the distinctions between the mutable and immutable variants. We also discussed common idioms used in the collections, such as views for nonstrict evaluation and how instances of the correct collection subtypes are output by methods implemented in supertype mixin traits. You can apply those techniques to your own type hierarchies.

The next chapter covers a topic with practical benefits for encapsulation and modularity: Scala's rich support for fine-grained control over visibility. Scala goes well beyond Java's `public`, `protected`, `private`, and default package scoping capabilities.

CHAPTER 15

Visibility Rules

Encapsulation of information in modules is a hallmark of good software design. Used well, it exposes logically consistent abstractions to users and hides implementation details. The latter is necessary to prevent unwanted access to internal details and to enable easier evolution of those details without impacting users. This chapter discusses Scala's visibility rules for specifying encapsulation.

Many languages provide a few scopes of visibility for definitions, often tied to subtype relationships (object-oriented inheritance). Java is typical, with four visibility scopes:

Public
: Visible everywhere

Protected
: Visible only inside the type and subtypes

Private
: Visible only inside the type where declared or defined

Default package
: Visible to any other type in the same package

Scala changes Java's visibility rules in two major ways. First, public is the default and there is no `public` keyword. Second, `protected` and `private` can be qualified by a scope, either a type or a package; for example, `protected[mypackage]` for scoping to some package `mypackage`.

The qualified scopes are underutilized in Scala code, in my opinion. They give library developers careful control over visibility inside and outside a library's API.

3 Scala 3 adds another tool for fine-grained visibility control, export clauses, which we explored in "Export Clauses" on page 263.

Public Visibility: The Default

Scala uses public visibility by default. It is common to declare members of types private when you want to limit their visibility to the type only, or to declare them protected to limit visibility to subtypes only.

Visibility Keywords

When used, `private` and `protected` appear at the beginning of a declaration. An exception is when you want nonpublic visibility for the primary constructor of a class. Here it is put after the type name and the optional type parameter list, but before the constructor's parameter list. Here is an example:

```
class Restricted[+A] private (name: String) {...}
```

Here, `Restricted` is still a public class, but the constructor is private.

Why do this? It forces users to call a factory instead of instantiating types directly, such as companion object `apply` methods. (Those factories must have access to the constructor, which is true by default for companion objects.) This idiom can be useful for separating concerns, the class itself stays focused on implementation logic, while the factory imposes controls like input parameter validation.

Table 15-1 summarizes the visibility scopes.

Table 15-1. Visibility scopes

Name	Keyword	Description
public	*none*	Public members and types are visible everywhere, across all boundaries.
protected	`protected`	Protected members are visible to the defining type, to subtypes, and to nested types. Protected types are visible only within the same package and subpackages.
private	`private`	Private members are visible only within the defining type and nested types. Private types are visible only within the same package.
scoped protected	`protected[scope]`	Visibility is limited to `scope`, which can be a package or type.
scoped private	`private[scope]`	Synonymous with scoped protected visibility, except under inheritance.

In Scala 2, `this` could also be used as a scope for `protected` and `private`, meaning visibility is restricted to the same instance of the type, but not other instances. Support for `this` is dropped in Scala 3, as the compiler can infer cases where it is useful.

> You can't apply any of the visibility scope modifiers to packages. Therefore, a package is always public, even when it contains no publicly visible types.

The rest of this chapter discusses some of the high-level details for visibility. In the book's code examples, see the directory *src/main/scala/progscala3/visibility/* where there are many examples that demonstrate the rules. We'll show one of these examples ahead.

Let's summarize the rules of private and protected access, starting with no scope qualifications.

Protected Visibility

Protected visibility is for the benefit of implementers of subtypes, who need a little more access to the details of their supertypes. Any member declared with the `protected` keyword is visible only to the defining type, including other instances of the same type, and any subtypes.

When a type is declared `protected`, visibility is limited to the enclosing package. This also means that type can't be subtyped outside the package.

Private Visibility

Private visibility completely hides implementation details, even from the implementers of subtypes. Any member declared with the `private` keyword is visible only to the defining type, including other instances of the same type. When applied to a type, `private` limits visibility to the enclosing package.

Just as for `protected` type declarations, the `private` types can't be subtyped outside the same package.

Scoped Private and Protected Visibility

Unique to Scala is the ability to specify a scope for private or protected visibility. Because a scope qualifier effectively overrides the default boundaries for `private` and `protected`, these declarations become interchangeable because they behave identically, except under inheritance when they are applied to members.

The following example shows the effects on visibility for different scoped declarations:

```
// src/main/scala/progscala3/visibility/ScopeInheritance.scala

package progscala3.visibility.scopeinheritance:

  package scopeA:
    class Class1:
      private[scopeA]   val scopeA_privateField = 1
      protected[scopeA] val scopeA_protectedField = 2
      private[Class1]   val class1_privateField = 3
      protected[Class1] val class1_protectedField = 4
      private           val class1_privateField2 = 5
      protected         val class1_protectedField2 = 6

    class Class2 extends Class1:
      val field1 = scopeA_privateField
      val field2 = scopeA_protectedField
      // Scope error:
      // val field3 = class1_privateField
      // val field4 = class1_privateField2
      val field5 = class1_protectedField
      val field6 = class1_protectedField2

  package scopeB:
    class Class2B extends scopeA.Class1:
      // Scope error:
      // val field1 = scopeA_privateField
      val field2 = scopeA_protectedField
      // Scope error:
      // val field3 = class1_privateField
      // val field4 = class1_privateField2
      val field5 = class1_protectedField
      val field6 = class1_protectedField2
```

Some possible declarations are commented out because they would fail to compile, indicating the differences between `private[scope]` and `protected[scope]`.

To summarize the differences in this example, all fields declared `protected[scopeA]` are visible to clients of `scopeA`, such as `scopeB`. That's why it's OK for `Class2B` inside `scopeB` to reference the field `scopeA_protectedField`, a field in the class it subtypes, `scopeA.Class1`.

However, the fields declared `private[scopeA]` and `private[Class1]` can't be seen inside `scopeB`.

The declarations `private[Class1] class1_privateField` and `protected[Class1] class1_protectedField` are actually equivalent to unscoped `private` and `protected` declarations, as illustrated by the `class1_privateField2` and `class1_protectedField2` declarations, which behave the same in terms of visibility.

Recap and What's Next

Scala visibility declarations are very flexible, and they behave consistently. They provide fine-grained control over visibility at a wide variety of possible scopes, allowing you to design APIs with optimal abstractions and minimal exposure of implementation details.

Now we turn to a tour of Scala's type system. We already know quite a lot about it, but to really exploit the type system's power, we need a systematic understanding of it.

CHAPTER 16

Scala's Type System, Part I

By now, you know quite a lot about Scala's type system. This chapter and the next fill in some details and introduce more advanced constructs.

Scala is a statically typed language. Its sophisticated type system combines FP and OOP. The type system tries to be logically comprehensive, complete, and consistent.

Ideally, a type system is expressive enough to prevent your applications from ever *inhabiting* an invalid state. It lets you enforce these constraints at compile time, so runtime failures never occur. In practice, we're far from that goal, but Scala's type system pushes the boundaries of what's possible.

However, Scala's type system can be intimidating. When people claim that Scala is complex, they usually have the type system in mind.

Fortunately, type inference hides many of the details. Mastery of the more arcane aspects of the type system is not required to use Scala effectively, although you'll eventually need to be familiar with most constructs.

Now let's begin by revisiting familiar ground, parameterized types.

Parameterized Types

In "Parameterized Types: Variance Under Inheritance" on page 285, we explored variance under subtyping. Recapping, a declaration like `Seq[+A]` means that `Seq` is parameterized by a single type, represented by `A`. The `+` is a variance annotation, indicating that `Seq` is covariant in the type parameter. This means that `Seq[String]` is considered a subtype of `Seq[AnyRef]` because `String` is a subtype of `AnyRef`.

Similarly, the `-` variance annotation indicates that the type is contravariant in the type parameter. One example is the types for the `N` parameters for the `FunctionN` types.

Consider Function2, which has the type signature Function2[-T1, -T2, +R]. We saw in "Functions Under the Hood" on page 286 why the types for the function parameters must be contravariant.

Abstract Type Members and Concrete Type Aliases

Parameterized types are common in statically typed, object-oriented languages, while type members are common in many functional languages. Scala supports both. We first discussed abstract and concrete type members in "Parameterized Types Versus Abstract Type Members" on page 66, where we also discussed the advantages and disadvantages of each approach. We explored the following example (some details not repeated):

```
abstract class BulkReader:
  type In
  def source: In
  def read: Seq[String]

case class StringBulkReader(source: String) extends BulkReader:
  type In = String
  def read: Seq[String] = Seq(source)

import scala.io.Source
case class FileBulkReader(source: Source) extends BulkReader:
  type In = Source
  def read: Seq[String] = source.getLines.toVector
```

BulkReader declares the abstract type member In, which is made concrete in the subtypes StringBulkReader and FileBulkReader, where it becomes an alias for String and Source, respectively. Note that the user no longer specifies a type through a type parameter. Instead, as implementers rather than users of the readers, we have total control over the type member In and its enclosing class, so the implementation keeps them consistent.

Comparing Abstract Type Members Versus Parameterized Types

Many abstraction idioms can be implemented using parameterized types or abstract type members. In practice, each feature is a natural fit for different design problems.

Parameterized types work nicely for containers, like collections, where there is little connection between the element types, which are represented by the type parameter and the container itself. For example, an Option works the same for a String, a Double, etc. Option is agnostic about the element's type.

What about using a type member instead? Consider the declaration of Some from the standard library:

```
case final class Some[+A](val value : A) { ... }
```

If we try to convert this to use abstract types, we might start with the following:

```
case final class Some(val value : ???) {
  type A
  ...
}
```

What should be the type of the parameter `value`? We can't use `A` because it's not in scope at the point of the constructor parameter. We could use `Any`, but that defeats the purpose of type safety.

Compare this attempt to the `BulkReader` example, where all the subtypes we defined for `BulkReader` were able to provide a concrete type for the `In` and `source` members. Abstract type members are most useful for type families like this, where the outer and inner types are closely linked.

Type Bounds

When defining a parameterized type or method, it may be necessary to specify bounds on the type parameter. For example, a container might assume that certain methods exist on all types used for the type parameter.

Upper Type Bounds

Upper type bounds specify that a type must be a subtype of another type. For a motivating example, we saw in "Implicit Conversions" on page 319 that `Predef` defines implicit conversions to wrap arrays in `collection.mutable.ArraySeq` instances, where the latter provides the sequence operations we know and love.

Several of these conversions are defined. Most are for the specific `AnyVal` types, like `Long`, but one handles conversions of `Array[AnyRef]` instances:

```
implicit def wrapRefArray[T <: AnyRef](xs: Array[T]): ArraySeq.ofRef[T] = ...
implicit def wrapBooleanArray(xs: Array[Boolean]): Array.ofBoolean = ...
... // Methods for the other AnyVal types.
```

The `ofRef[T]` and `ofBoolean` types are convenient subtypes of `ArraySeq`. The type parameter `T <: AnyRef` means "any type `T` that is a subtype of `AnyRef`." Hence, `wrapRefArray` won't be called for an array of `Int`s, for example, removing any potential ambiguity with the other `AnyVal`-related methods.

These bounds are called *upper type bounds*, following the de facto convention that type hierarchies are drawn with subtypes below their supertypes. We followed this convention in Figure 13-1 in Chapter 13.

> Type bounds and variance annotations cover unrelated issues. A type bound specifies constraints on allowed types that can be used for a type parameter. A variance annotation specifies when an instance of a subtype of a parameterized type can be substituted where a supertype instance is expected.

Lower Type Bounds

Similarly, a *lower type bound* expresses that one type must be a supertype (or the same type) as another. An example is the `getOrElse` method in `Option`:

```
sealed abstract class Option[+A] extends ... {
  ...
  final def getOrElse[B >: A](default: => B): B = {...}
  ...
}
```

If the `Option` instance is `Some[A]`, the value it contains is returned. Otherwise, the by-name parameter `default` is evaluated and returned. It is allowed to be a supertype of `A` (meaning it could also be `A`). Let's consider an example that illustrates why this requirement is necessary:

```
// src/script/scala/progscala3/typesystem/bounds/LowerBounds.scala
scala> class Super(val value: Int):                                    ❶
     |   override def toString = s"${this.getClass.getSimpleName}($value)"
     | class Sub(value: Int) extends Super(value)

scala> val optSub: Option[Sub] = Some(Sub(1))
val optSub: Option[Sub] = Some(Sub(1))

scala> var optSuper: Option[Super] = optSub                            ❷
var optSuper: Option[Super] = Some(Sub(1))

scala> val super1: Super = optSuper.getOrElse(Sub(0))                  ❸
val super1: Super = Sub(1)

scala> optSuper = None                                                 ❹
optSuper: Option[Super] = None

scala> val super2: Super = optSuper.getOrElse(Super(0))                ❺
val super2: Super = Super(0)
```

❶ A simple type hierarchy for demonstration purposes.

❷ The reference `optParent` only knows it's an `Option[Super]`, but it actually references a subtype, `Option[Sub]`.

❸ Calling `getOrElse` on `optSuper` returns a `Super` instance. In this case, it happens to be `Sub(1)`.

❹ Set the reference to None.

❺ This time the default value Super(0) is returned. This is fine, since our reference super2 expects a Super.

The last line illustrates the crucial point. Because Option is covariant in the parameter type, it's possible for an Option[Super] reference to point to an Option[Sub] instance, so getOrElse needs to support the case where the user provides a default value that is a supertype of Sub. Put another way, inside optChild, it doesn't know that references to it are actually of type Option[Super]. Outside, the user only cares about getting a Super instance out of the Option. The user provides a default value of type Super, if it is needed.

> When attempting to understand why variance annotations and type bounds work the way they do, remember to study what happens with instances of types from the perspective of code that uses them, where that code might have a reference to a supertype, but the actual instance is a subtype.

The fact that Option[+T] is covariant leads to the requirement that getOrElse must accept a default value that might be a supertype of T.

This behavior is true for any parameterized type that is covariant in the type parameter. It must have *contravariant* behavior in methods that provide new elements to add to a collection or default values for getOrElse and similar methods.

Consider the Seq.+: method for prepending an element to a sequence, creating a new sequence. Its signature is similar to Option.getOrElse:

```
final def +:[B >: A](elem: B): Seq[B]
```

The B parameter needs to be contravariant. Consider this example:

```
scala> val seq = 1 +: Seq(2.2, 3.3)
val seq: Seq[AnyVal] = List(1, 2.2, 3.3)
```

The type parameter inferred is the least upper bound, meaning the closest supertype of the original type A (Double) and the type of the new element (Int). Hence, B in the resulting Seq[B] is inferred to be AnyVal.

To recap, there is an intimate relationship between parameterized types that are covariant in their parameters and lower type bounds in method parameters, which are often contravariant in the collection's type parameter.

Finally, you can combine upper and lower type bounds:

```scala
class Upper
class Middle1 extends Upper
class Middle2 extends Middle1
class Lower extends Middle2
case class C[A >: Lower <: Upper](a: A)         ❶
// case class C1[A <: Upper >: Lower](a: A)     ❷
// case class C2[A >: Upper <: Lower](a: A)     ❸
```

❶ The type parameter, `A`, must appear first.

❷ Does not compile because the compiler requires the lower bound to be specified before the upper bound.

❸ Does not compile because there are no types in existence that satisfy `A >: Upper` *and* `A <: Lower`. In other words, the specified range is outside the range of `Lower <: Upper`.

Everything we've said about type bounds applies to abstract type members too. What's not allowed on type members are variance indicators, `+` and `-`, because type members are inside the outer type, where any variance behavior is defined via type parameters.

Context Bounds

We discussed context bounds in "Context Bounds" on page 167, another way to constrain allowed types. We saw four functionally equivalent ways to declare them:

```scala
trait SortableSeq[+A]:
  def sortBy1[B : Ordering](transform: A => B): SortableSeq[A]
  def sortBy2[B](transform: A => B)(using o: Ordering[B]): SortableSeq[A]
  def sortBy3[B](transform: A => B)(using Ordering[B]): SortableSeq[A]
  def sortBy4[B](transform: A => B)(implicit o: Ordering[B]): SortableSeq[A]
```

The only allowed types for `B` are those for which a given `Ordering[B]` exists in scope. The type expression `B : Ordering` is equivalent to just using `B` with an explicit *using* or *implicit* parameter list. In `sortBy3`, the using instance is anonymous, while it's named in `sortBy2` and `sortBy4`.

View Bounds

In older Scala code, you may see *view bounds*, which are a special case of context bounds. They can be declared in either of the following ways:

```scala
class C[A]:
  def m1[B](x: Int)(given view: A => B): C[B] = ???
  def m2[A <% B](x: Int): C[B] = ???
```

While a context bound `A : B` corresponds to the type `B[A]`, a view bound `A <% B` corresponds to a function that converts an `A` to a `B`. The idea is that "B is a view onto A." Also, compared to an upper bound expression `A <: B`, which says that `A` is a subtype of `B`, a view bound is a looser requirement. It says that `A` must be convertible to `B`.

> View bounds are deprecated. Instead, use a context bound, a given instance of a function from `A => B`, or use an implicit conversion.

3 Intersection and Union Types

Intersection and union types are new to Scala 3, introduced by the dependent object typing calculus that Scala 3's type system is based on. They make the type system more robust, based on the mathematical properties of intersection and union of sets. If we think of the possible instances for a type as the members of a set, what happens when we apply set operations like intersection and union to sets for different types?

Intersection Types

Intersection types replace compound types in Scala 2. These are *anonymous types* created by composing (or compounding) types while creating instances, instead of declaring a named type first that subtypes and mixes in other types. In Scala 2 and 3, the `with` keyword is used in definitions, the righthand side of the equals sign. In Scala 2, the resulting compound type also uses `with`. However, for Scala 3, the type is an intersection type, which uses `&` between the input types instead of `with`. This is also the syntax used in explicit type declarations, meaning the type on the lefthand side of an assignment.

```
scala> trait T1
     | trait T2
     | class C
     | class CC extends C with T1 with T2

scala> val c12a = new C with T1 with T2
val c12a: C & T1 & T2 = anon$1@6261a04c   // Scala 2: C with T1 with T2

scala> val c12b: C & T1 & T2 = new C with T1 with T2
val c12b: C & T1 & T2 = anon$1@bb0c81c

scala> val c12c: C with T1 with T2 = new C with T1 with T2
val c12c: C & (T1 & T2) = anon$1@49d149a7

scala> val bad = new C & T1 & T2
1 |val bad = new C & T1 & T2
  |          ^^^^^^^
```

```
  |      value & is not a member of C
1 |val bad = new C & T1 & T2
  |                  ^^
  |                  Not found: T1
```

The declarations shown for `c12a` and `c12c` would be written the same in Scala 2, but the resulting type of both would be `C with T1 with T2`.

In Scala 3, `with` and `extends` are still used when declaring types, like `CC`, and instantiating anonymous instances, as shown. However, the resulting type is now the intersection type, `C & T1 & T2`. Precedence rules are left to right by default, so `C & T1 & T2` is equivalent to `(C & T1) & T2`.

Notice what happens with `c12c`. Because we specify the type declaration explicitly using `with`, the precedence grouping is right to left, reflecting the convention of right-to-left binding of `with` from Scala 2. Hence `val c12c: C with T1 with T2` is equivalent to the grouping `val c12c: (C & (T1 & T2))`. Future versions of Scala 3 will deprecate and remove the use of the `with` keyword in type declarations, making the `c12c` declaration invalid.

Finally, note that we can't construct an anonymous instance using `&`, as shown for `bad`.

When resolving overridden method calls, the precedence rules specified by linearization ("Linearization of a Type Hierarchy" on page 301) apply for both Scala 2 and 3. Consider this expanded example:

```scala
// src/script/scala/progscala3/typesystem/intersectionunion/Intersection.scala
trait M:
  def m(s: String): String = s
trait T1 extends M:
  override def m(s: String): String = s"[ ${super.m(s)} ]"
trait T2 extends M:
  override def m(s: String): String = s"( ${super.m(s)} )"
open class C extends M:
  override def m(s: String): String = s"{ ${super.m(s)} }"
```

In which order are the `m` methods invoked?

```scala
val c12 = new C with T1 with T2
val c21 = new C with T2 with T1

assert(c12.m("hello") == "( [ { hello } ] )")
assert(c21.m("hello") == "[ ( { hello } ) ]")
```

For `c12.m()`, the order is `T2.m()`, `T1.m()`, `C.m()`, then `M.m()`. For `c21.m()`, the order of `T2.m()` and `T1.m()` are reversed. Hence the precedence order is right to left.

However, the change from compound types using `with` to intersection types using `&` is more than just a trivial renaming. Intersection types support reasoning about types as sets and instances as members of a particular (type) set. Crucially, set intersection *commutes*: `A & B == B & A`. Scala 2 compound types did not commute this way. Hence, all six of the type declaration permutations in the following declarations are valid:

```
val c12a: C & T1 & T2 = c12
val c12b: C & T2 & T1 = c12
val c12c: T1 & C & T2 = c12
val c12d: T2 & C & T1 = c12
val c12e: T1 & T2 & C = c12
val c12f: T2 & T1 & C = c12
```

Finally, recall from set theory that if an item is a member of set 1 and set 2, then it is also a member of the intersection of set 1 and set 2. Hence, the following declarations are all valid:

```
val t1a: T1 = c12
val t2a: T2 = c12
val c2a: C  = c12

val t12: T1 & T2 = c12
val ct1: C  & T1 = c12
val ct2: C  & T2 = c12
```

You can specify intersection types as type parameters for function parameters:

```
def f(t12: T1 & T2): String = t12.m("hello!")
val list12: Seq[T1 & T2] = Seq(c12, c21)
assert(list12.map(f) == List("( [ { hello! } ] )", "[ ( { hello! } ) ]"))
```

Rules for intersection types

Here is a summary of the rules for intersection types.

For subtyping, if `T <: A` and `T <: B`, then `T <: A & B`. For example, `C` is a subtype of `T1` in our example and it's also a subtype of `T2`. Hence, it is a subtype of `T1 & T2`.

Similarly, if `T <: A`, then `T & T2 <: A`. For example, `C` is a subtype of `T1`, so if we create a new subtype of C that mixes in a new trait, say `T3`, then `C & T3` is also a subtype of `T1`.

A formal way of writing the commutativity is `A & B <: B & A` and vice versa.

Intersection types are also associative. `A & (B & C)` is equivalent to `(A & B) & C`.

Let's consider variance under subtyping for parameterized types. Suppose that `C[A]` is covariant in `A`. If so, then `C[A & B]` is substitutable for `C[A] & C[B]`. Here is an example:

```
val listt1t2: Seq[T1 & T2] = Seq(c12, c21)
val list1: Seq[T1] = listt1t2
```

```
val list2: Seq[T2] = listt1t2
val list3: Seq[T1] & Seq[T2] = listt1t2
```

We declare `list1` to `Seq[T1]` but assign the subtype `listt1t2`, which is of type `Seq[T1 & T2]`, and similarly for `list2`. The type declarations for `listt1t2` and `list3` might be hard to understand at first, but remember that a type declaration is specifying a constraint; what values are allowed to be used here? `Seq[T1] & Seq[T2]` says (1) only sequences are allowed, (2) only elements of type `A` are allowed, and (3) only elements of type `B` are allowed. We can use `listt1t2` because it is a `Seq` and its elements, `c12` and `c21`, are of both type `T1` and `T2`.

Union Types

Union types follow the rules for unions of sets. If an element `a` is in set 1 and `b` is in set 2, then the union of those sets contains both `a` and `b`.

A value of type `A | B` is an instance of type `A` or an instance of type `B`. One useful example is to use a union type as an alternative to `Either[A,B]` for error handling (see "Either: An Alternative to Option" on page 236):

```
// src/script/scala/progscala3/typesystem/intersectionunion/Union.scala
scala> case class Bad(message: String)
     | case class Good(i: Int)
     |
     | val error = Bad("Failed!")
     | val result = Good(0)

scala> val seq1 = Seq(error, result)
val seq1: Seq[Object] = List(Bad(Failed!), Good(0))

scala> val seq: Seq[Good | Bad] = Seq(error, result)
val seq: Seq[Good | Bad] = List(Bad(Failed!), Good(0))
```

The type declaration for the definition of `seq` says it is sequence holding instances of either `Good` or `Bad`. However, for `seq1`, the least upper bound is inferred for the type parameter, yielding `Seq[Object]`. The union type isn't inferred here; you have to provide it explicitly for `seq`.

Pattern matching is required to determine the type of instance you have and process it:

```
scala> def work(i: Int): Good | Bad =
     |   if i > 0 then Bad(s"$i must be <= 0") else Good(i)
     |
     | def process(result: Good | Bad): String = result match
     |   case Bad(message) => message
     |   case Good(value) => s"Success! value = $value"

scala> val results = Seq(0, 1).map(work)
val results: Seq[Good | Bad] = List(Good(0), Bad(1 must be <= 0))
```

```
scala> val strings = results.map(process)
val strings: Seq[String] = List(Success! value = 0, 1 must be <= 0)
```

For `results`, the union type is inferred from the output of `map`. The value computed for `strings` is `List("Success! value = 0", "1 must be <= 0")`.

Rules for union types

Scala 2 pattern-matching syntax supported expressions like `A | B`, which does not mean a union type expression. The following case clauses are equivalent and work the same way in Scala 2 and 3, for backward compatibility, meaning match on a value is either a `Good` or a `Bad` instance:

```
case _: Good | Bad => ...
case (_: Good) | Bad => ...
```

In Scala 3, if you want to match a value of union type `A | B`, you must use explicit parentheses:

```
case _: (Good | Bad) => ...
```

Concerning subtyping, `A` is a subtype of `A | B` for all `A` and all `B`. Similarly, if `A <: C` and `B <: C`, then `A | B <: C`.

Like intersection types, union types are commutative and associative: `A | B` is equivalent to `B | A`, and `A | (B | C)` is equivalent to `(A | B) | C`.

Rules for union and intersection types together

Union and intersection types are *distributive*: `A & (B | C)` is equivalent to `(A & B) | (A & C)`, while `A | (B & C)` is equivalent to `(A | B) & (A | C)`:

```
trait A; trait B; trait C

summon[(A & (B | C)) =:= ((A & B) | (A & C))]
summon[(A | (B & C)) =:= ((A | B) & (A | C))]

val x1:  A & (B | C)       = new A with B {}
val x2:  A & (B | C)       = new A with C {}
val x3:  A & (B | C)       = new A with B with C {}
val x4:  (A & B) | (A & C) = new A with B {}
val x5:  (A & B) | (A & C) = new A with C {}
val x6:  (A & B) | (A & C) = new A with B with C {}

val x7:  A | (B & C)       = new A {}
val x8:  A | (B & C)       = new B with C {}
val x9:  A | (B & C)       = new A with B with C {}
val x10: (A | B) & (A | C) = new A {}
val x11: (A | B) & (A | C) = new B with C {}
val x12: (A | B) & (A | C) = new A with B with C {}
```

The two `summon` expressions show that the compiler considers the types equivalent under the distributive law. The six example declarations that follow, `x1` through `x6`, have equivalent types with valid example instances on the righthand side. The same applies for `x7` through `x12`. While studying these examples, remember that type declarations are constraints. What righthand-side values satisfy the constraints?

What about covariance and contravariance of parameterized types? We saw earlier how intersection types work for covariant parameterized types, like `Seq[T]`. This isn't the same for union types:

```
val tABCs: Seq[A | B | C] = Seq(new A {}, new B {}, new C {})
val tAs: Seq[A] = tABCs        // ERROR
val tBs: Seq[B] = tABCs        // ERROR
val tCs: Seq[C] = tABCs        // ERROR
```

We can't assign `tABCs` to a `Seq[A]` value, for example. This makes sense because the declaration `val tAs: Seq[A]` is a constraint that the only elements found in the sequence will be `A`s, but `tABCs` contains an `A`, a `B`, and a `C`:

However, the following works:

```
val seqAs: Seq[A] = Seq(new A {})
val seqBs: Seq[B] = Seq(new B {})
val seqCs: Seq[C] = Seq(new C {})
val seqABCs1: Seq[A | B | C] = seqAs
val seqABCs2: Seq[A | B | C] = seqBs
val seqABCs3: Seq[A | B | C] = seqCs
```

For Scala 3, the union type `A | B | C` is the true least upper bound for the types `A`, `B`, and `C`, even when the compiler infers `AnyRef`. When we define `val seqABCs1: Seq[A | B | C]`, for example, we are saying the sequence can have instances of any or all of these types. Convince yourself that the last three assignments are valid.

In the discussion of intersection types, I didn't mention what happens with parameterized types that have contravariant type parameters, like the types for function parameters. There is a relationship between intersection and union types here. For a contravariant type `C[-A]`, `C[A | B]` is substitutable for `C[A] & C[B]`.

To understand this, recall the type signature for `A => B`, `Function1[-A, +R]`. Let's now see that `Function1[A | B, R]` is substitutable for `Function1[A, R] & Function1[B, R]`. That is, `(A | B) => R <: (A => R) & (B => R)`.

```
val fABC1: (A | B | C) => String = _ match
  case t1: A => "A"
  case t2: B => "B"
  case t3: C => "C"
val fABC2: (A => String) & (B => String) & (C => String) = fABC1

val seqABCs: Seq[A | B | C] = Seq(new A {}, new B {}, new C {})
seqABCs.map(fABC1)
```

```
seqABCs.map(fABC2)
seqABCs.map((x: AnyRef) => s"<$x>")
```

The functions `fABC1` and `fABC2` have equivalent types. When mapping over `seqABCs` with both functions, `List("A", "B", "C")` is returned.

A type signature like `(A => String) & (B => String) & (C => String)` is hard to grasp, but once again, it is a constraint on set membership. This signature says that the only allowed function values we can use are those that can take an `A` and return a `String`, and take a `B` and return a `String`, and take a `C` and return a `String`. The only kind of function we can write like that is one that takes an argument of type `A | B | C`, meaning it can handle instances of any of these three types.

However, since `AnyRef` is the parent of `A`, `B`, and `C`, we can also use a function of type `AnyRef => String`, as shown in the last example.

Phantom Types

A *phantom type* is useful in situations where the mere existence of a type is all that's required. No actual instances are needed. Contrast this scenario to some of the idioms we explored that use given instances, where an instance must exist.

For example, phantom types are useful for defining workflows that must proceed in a particular order. Consider a simplified payroll calculator. In US tax law, payroll deductions for insurance premiums and contributions to certain retirement savings (401k) accounts can be subtracted before calculating taxes on the remaining pay amount. So a payroll calculator must process these pre-tax deductions first, then calculate the tax deductions, then calculate post-tax deductions, if any, to determine the net pay.

Here is one possible implementation, where a lot of details are elided to focus on the key elements. The full implementation is in the code examples:

```
// src/main/scala/progscala3/typesystem/payroll/PhantomTypesPayroll.scala
package progscala3.typesystem.payroll
import progscala3.contexts.accounting.*                    ❶

sealed trait Step                                          ❷
trait PreTaxDeductions extends Step
trait PostTaxDeductions extends Step
trait Final extends Step

case class Employee(
    name: String,
    annualSalary: Dollars,
    taxRate: Percentage,          // Assume one rate covers all taxes
    insurancePremiums: Dollars,
    _401kDeductionRate: Percentage,  // Pre-tax retirement plans in the US
```

```scala
    postTaxDeductions: Dollars):     // Other "after-tax" deductions
  override def toString: String = ...

case class Pay[S <: Step](                                        ❸
    employee: Employee,
    grossPay: Dollars,          // This pay period's gross, before taxes
    netPay:   Dollars,          // This pay period's net, after taxes
    taxes:    Dollars = Dollars(0.0),
    preTaxDeductions: Dollars = Dollars(0.0),
    postTaxDeductions: Dollars = Dollars(0.0)):
  override def toString: String = ...

object Payroll:
  def start(employee: Employee): Pay[PreTaxDeductions] =          ❹
    val gross = employee.annualSalary / 12          // Compute monthly
    Pay[PreTaxDeductions](employee, gross, gross)  // net == gross to start

  def deductInsurance(pay: Pay[PreTaxDeductions]): Pay[PreTaxDeductions] = ...
  def deduct401k(pay: Pay[PreTaxDeductions]): Pay[PreTaxDeductions] = ...
  def deductTax(pay: Pay[PreTaxDeductions]): Pay[PostTaxDeductions] = ...
  def deductFinalDeductions(pay: Pay[PostTaxDeductions]): Pay[Final] = ...

@main def TryPhantomTypes  =
  import Payroll.*
  val e = Employee("Buck Trends", 100000.0, 0.25, 200, 0.10, 100.0)
  val pay1 = start(e)
  val pay2 = deduct401k(pay1)                                     ❺
  val pay3 = deductInsurance(pay2)
  val pay4 = deductTax(pay3)
  val pay  = deductFinalDeductions(pay4)
  println(e); println(pay)    // Nice +toString+ formatting not shown above
```

❶ Use the `Dollars` and `Percentage` types from "Scala 3 Implicit Conversions" on page 154.

❷ Closed hierarchy of phantom types for the steps in the workflow. They have no members and aren't even concrete types.

❸ Hold the computed data for this pay period. Note the type parameter.

❹ Note how the return value is created. The type parameter is used to indicate the correct state in the workflow. Compare the type parameter used for `Pay` in all these methods.

❺ We can call `deduct401K` and `deductInsurance` in either order.

You can run it at the `sbt` prompt:

```
> runMain progscala3.typesystem.payroll.TryPhantomTypes
...
```

```
Employee: Buck Trends
  annual salary:         $100000.00
  tax rate:              25.00%
  per pay period deductions:
    insurance premiums:  $200.00
    401K deductions:     10.00%
    post tax deductions: $100.00

Pay for employee: Buck Trends
  gross pay:            $8333.33
  net pay:              $5375.00
  taxes:                $1825.00
  pre-tax deductions:   $1033.33
  post-tax deductions:  $100.00
```

The `Step` traits are used as type parameters for the `Pay` type, which is passed through the `Payroll` methods that implement each step. Each method in `Payroll` takes a `Pay[S <: Step]` object with a particular type for the `S` parameter. This constrains when we can call each method. The `TryPhantomTypes` method demonstrates the use of the API. We can't call steps out of order, like calling `deduct401k` with a `Pay[PostTaxDeductions]` object. You can try it, but it will be easier to try with the next example.

Hence, the tax rules are enforced by the API and user errors are avoided. Instances of the `Step` traits are never created, hence the term *phantom type*.

Actually, `TryPhantomTypes` is not very elegant. Let's fix that by borrowing a pipelining operator from the `F#` language:[1]

```
// src/main/scala/progscala3/typesystem/payroll/PhantomTypesPayrollPipes.scala
package progscala3.typesystem.payroll
import progscala3.contexts.accounting.*
import scala.annotation.targetName

object Pipeline:
  extension [V,R](value: V)
    @targetName("pipe") def |> (f : V => R) = f(value)

@main def TryPhantomTypesPipeline =
  import Pipeline.*
  import Payroll.*

  val e = Employee("Buck Trends", Dollars(100000.0), Percentage(25.0),
    Dollars(200), Percentage(10.0), Dollars(100.0))
  val pay = start(e)  |>
    deduct401k        |>
```

1 This example is adapted from James Iry, "Phantom Types in Haskell and Scala". See also the standard library's `util.chaining` for an implicit conversion to add a `pipe` method.

```
    deductInsurance    |>
    deductTax          |>
    deductFinalDeductions
  println(e); println(pay)
```

Now, `TryPhantomTypesPipeline` contains a more elegant sequencing of steps. The pipeline operator `|>` may look fancy, but all it really does is reorder expressions.

> Write APIs that do as much as possible to prevent users from making mistakes! Phantom types can be used to enforce proper sequencing of steps.

3 Structural Types

Occasionally, we miss the benefits of dynamic typing. Consider a SQL query that returns a result set of `Records` that have an ad hoc set of columns corresponding to the query. Suppose the `Records` returned by a user query of an `Employees` table include `name` (`String`) and `age` (`Int`) columns. The user would like to write type-safe code like `val name: String = record.name` and `val age: Int = record.age` rather than the more typical `val name: String = record.get[String]("name")`, for example. (However, the internals of the query API might have to do mapping like this.) The user would like the convenience of type-safe field access, without the need to define ad hoc case classes for all the possible query results.

The Scala 3 trait `scala.reflect.Selectable` balances type safety with many of the benefits of dynamic typing. Consider this example:

```
// src/script/scala/progscala3/typesystem/selectable/Selectable.scala

trait Record extends reflect.Selectable:                              ❶
  def id: Long      // Id of the record in the database

val persons = Seq("Dean" -> 29, "Dean" -> 29, "Dean" -> 30, "Fred" -> 30)
  .map { case (name1, age1) =>                                        ❷
    new Record:                                                       ❸
      def id: Long     = 0L
      val name: String = name1                                        ❹
      def age: Int     = age1
}

persons.map(p => s"<${p.id}, ${p.name}, ${p.age}>")
assert(persons(0) == persons(0))
assert(persons(0) != persons(1))                                      ❺
assert(persons(0) != persons(2))
assert(persons(0) != persons(3))
```

❶ Subclass (or mix in) the `Selectable` trait for query results.

❷ Simulate an actual query returning tuples.

❸ Return the `Records` using an anonymous type. The same `id` value is used, so we can test comparisons without the `id` values being trivially different.

❹ Arbitrarily use either a field or method for `name` and `age`.

❺ Equality checks don't compare member fields, like they would for case classes. Instead, `equals` returns true only for the same instance.

The type of `persons` shown in the REPL is `Seq[Record{name: String; age: => Int}]`, showing the additional members in the returned `Record` subtype. We can access `name` and `age` like regular type members.

However, using `Selectable` is not a good approach if you want to compare instances of the returned records. The default `equals` only returns true if the two instances are the same; it does not otherwise compare their fields. Use a case class instead when field comparisons are needed. You'll also want to make `id` and `age` fields, so the compiler uses them in the `equals` implementation, but if you don't want comparisons to use `id`, then leave it implemented with a method.

You could do something similar in Scala 2, without a parent trait for `Record`, but you had to enable the reflectiveCalls language feature, either by importing `scala.language.reflectiveCalls` or globally by using the compiler flag `-language:reflectiveCalls`. Enabling the language feature is a reminder that reflection is expensive and less type safe. See the Scala 2 version of the example: *src/script/scala-2/progscala3/typesystem/selectable/Reflection.scala*.

In Chapter 20, we'll explore another fully dynamic mechanism that is available in both Scala 2 and 3, the trait `scala.Dynamic`.

The `Selectable` companion object defines an implicit conversion called `reflectiveSelectable`. It supports runtime reflection over types that have a particular structure, meaning their member types, fields, and methods, independent of the types' names. Normally we work with types by name, called nominal typing (like `Shape` and `Seq`), and when we require types to have certain members, we require them to implement the same trait or abstract class.

Let's use structural types to implement the *Observer Design Pattern*. Compare what follows with the implementation in "Traits as Mixins" on page 271.

The minimum requirement for an observer is that it implements a method we can call with updates. We'll require a method named `update` but not impose any requirements on the enclosing types used as observers. Specifically, we won't require these observers to implement a particular trait. Here is an implementation using a structural type for the observer:

```
// src/main/scala/progscala3/typesystem/structuraltypes/Subject.scala
package progscala3.typesystem.structuraltypes
import reflect.Selectable.reflectiveSelectable                          ❶

private type Observer = {                                                ❷
  def update(): Unit
}

trait Subject:                                                           ❸
  protected var observers: Vector[Observer] = Vector.empty
  def addObserver(observer: Observer): Unit =
    observers :+= observer
  def notifyObservers(): Unit =                                          ❹
    observers foreach (_.update())
```

❶ Required import of the implicit conversion. Use reflection wisely because of the runtime overhead. Also, named types are easier to work with in IDEs, for example.

❷ A type alias that references an anonymous structure for observation. To emphasize that this alias is only used in this file for clarity and not used in client code, the alias is declared `private`.

❸ A normal mixin trait for subjects that manages the list of observers and updates them when changes occur.

❹ A method to notify all observers of a state change.

Any instance with an `update` method of this signature, no matter the type of the instance, can be used as an observer. `Observer` is just a private, convenient alias, not a trait or abstract class that observers must subtype. For simplicity, I'm assuming that concrete observers will keep a reference to their subjects and rely on `update` to know when to query the subjects for actual changes.

Let's try it:

```
// src/script/scala/progscala3/typesystem/structuraltypes/Observer.scala
import progscala3.typesystem.structuraltypes.Subject
import scala.reflect.Selectable.reflectiveSelectable

case class Counter(start: Int = 0) extends Subject:                ❶
  var count = start
  def increment(): Unit =
    count += 1
    notifyObservers()
```

```scala
case class CounterObserver(counter: Counter):                    // 2
  var updateCount = 0
  def update(): Unit = updateCount += 1

val c = Counter()
c.increment()
val observer1 = CounterObserver(c)
c.addObserver(observer1)
c.increment()
val observer2 = CounterObserver(c)
c.addObserver(observer2)
c.increment()
assert(c.count == 3)
assert(observer1.updateCount == 2)
assert(observer2.updateCount == 1)                               // 3
```

1. A type that increments an internal counter and mixes in `Subject` for the benefit of observation.

2. A concrete observer for `Counter`. Note that it doesn't implement some sort of observer trait, but it does provide the `update` method required. The constructor argument gives the observer the subject to watch, a `Counter`, although it isn't used in the `update` implementation. Instead, the observer tracks how many times `update` has been called.

3. The second observer was added after the `Counter` had already been incremented once.

Despite the disadvantages of reflection, structural types have the virtue of minimizing the coupling between things. In this case, the coupling consists of only a single method signature, rather than a type, such as a trait.

We still couple to a particular name—the method `update`. In a sense, we've only moved the problem of coupling from a type name to a method name. The name is still arbitrary, so let's push the decoupling to the next level: omit any definition of `Observer` and just use a function. It turns out that this change also makes it easier to use a custom `State` type for information passed to observers. Here is the final form of the example:

```scala
// src/main/scala/progscala3/typesystem/structuraltypes/SubjectFunc.scala
package progscala3.typesystem.structuraltypes

trait SubjectFunc:
  type State                                                          // 1

  private var observers: Vector[State => Unit] = Vector.empty         // 2
  def addObserver(observer: State => Unit): Unit =
```

```
    observers :+= observer
  def notifyObservers(state: State): Seq[Unit] =                    ❸
    observers map (o => o(state))
```

❶ An abstract type member for the state sent with updates.

❷ No more `Observer` definition. Now it's just a function `State => Unit`.

❸ Notifying each observer now means calling the function.

Here is the test script:

```
// src/script/scala/progscala3/typesystem/structuraltypes/ObserverFunc.scala
import progscala3.typesystem.structuraltypes.SubjectFunc          ❶

case class Counter(start: Int = 0) extends SubjectFunc:           ❷
  type State = Int
  var count = start
  def increment(): Unit =
    count += 1
    notifyObservers(count)

case class CounterObserver(var updateCalledCount: Int = 0) {      ❸
  def apply(count: Int): Unit = updateCalledCount += 1
}
val observer1 = CounterObserver()
val observer2 = CounterObserver()

val c = Counter()
c.increment()
c.addObserver(observer1.apply)                                    ❹
c.increment()
c.addObserver(observer2.apply)
c.increment()
assert(c.count == 3)
assert(observer1.updateCalledCount == 2)
assert(observer2.updateCalledCount == 1)
```

❶ No need to import `reflectiveSelectable` because there's no reflection used now!

❷ Nearly identical to the previous `Counter`, but now we define the `State` type as `Int` and pass the current `count` as the state to `notifyObservers`.

❸ Because we track how many updates we've received, we define a case class to hold this state information and use `CounterObserver.apply` as the function registered with `Counter` instances.

❹ Pass the apply method as the function expected by the subject c when registering an observer.

This has several advantages. All structure-based coupling is gone. Hence, we eliminate the overhead of reflection calls. It's also easier to use specific types for the State passed to observers. Hence, while structural typing can be useful, most of the time there are alternatives.

Refined Types

In the Selectable example in the previous section, we saw that our actual Record instances had the type Record{name: String; age: => Int}, not just Record. This is an example of a *refined type* because it has more specific details than Record alone.

Similarly, a refined type is created when we use mixin traits to create an anonymous instance:

```
scala> trait Logging:
     |   def log(message: String): Unit = println(s"Log: $message")

scala> abstract class Service:
     |   def start(): Unit

scala> val subject = new Service with Logging:
     |   def start(): Unit = log("starting!")
val subject: Service & Logging = anon$1@6429095e

scala> subject.start()
Log: starting!
```

Note the intersection type of subject, Service & Logging. Put another way, refinements are subtyping without naming the subtype.

3 Existential Types (Obsolete)

Scala 2 supported *existential types*, a way of abstracting over types. They let you assert that some type exists without specifying exactly what it is, usually because you don't know what it is and you don't need to know it in the current context.

For example, Seq[_ <: A] was really shorthand for Seq[T forSome {type T <: A}] in Scala 2 syntax.

However, existential types are incompatible with the stronger soundness principles of the type system in Scala 3. In Scala 2, they also interacted in negative ways with some other language features.

An expression like `Seq[? <: A]` is still supported (where `?` replaces `_` for type wildcards), but now this is considered a *refined type*. The `forSome` construct is no longer supported.

Recap and What's Next

This chapter filled in some details of type system features that we encountered before and it introduced new concepts. I focused on topics I think you'll encounter sooner rather than later. The next chapter continues the exploration of type system features, covering those that you are less likely to encounter except in more advanced Scala code.

CHAPTER 17

Scala's Type System, Part II

This chapter continues the type system survey that we started in the previous chapter, covering more advanced constructs. You can skip this chapter until you need to understand the concepts discussed here.

Let's begin with *match types* and the broad subject of *dependent typing*.

3 Match Types

Scala 3 extends pattern matching to work at the type level for match types. Let's look at an example adapted from the Scala documentation. The following match type definition returns a type that is the type parameter of another type with one type parameter. For example, for `Seq[Int]` it will return `Int`:

```
// src/script/scala/progscala3/typesystem/matchtypes/MatchTypes.scala

type Elem[X] = X match                                    ❶
  case String => Char                                     ❷
  case IterableOnce[t] => t                               ❸
  case Array[t] => t
  case ? => X                                             ❹
```

❶ Define a match type. It uses a `match` expression with case clauses to resolve the type.

❷ Special case handling of `String`s, which are actually `Array[Char]`.

❸ `scala.collection.IterableOnce` is a supertype of all collection types (and `Option[T]`), except for `Array`. Hence, we also need a clause for `Array`s.

❹ Use ? as a wildcard for all other types, including primitives and nonparameterized types. In this case, just return the type.

Let's try it:

```
val char: Elem[String] = 'c'
val doub: Elem[List[Double]] = 1.0
val tupl: Elem[Option[(Int,Double)]] = (1, 2.0)

val bad1: Elem[List[Double]] = "1.0"        // ERROR
val bad2: Elem[List[Double]] = (1.0, 2.0)   // ERROR
```

The last two examples fail to compile because the righthand sides are not `Doubles`.

There is another way to check our work:

```
summon[Elem[String] =:= Char]  // ...: Char =:= Char = generalized constraint
summon[Elem[List[Int]] =:= Int]
summon[Elem[Nil.type] =:= Nothing]                                    ❶
summon[Elem[Array[Float]] =:= Float]
summon[Elem[Option[String]] =:= String]
summon[Elem[Some[String]] =:= String]
summon[Elem[None.type] =:= Nothing]
summon[Elem[Float] =:= Float]

summon[Elem[Option[List[Long]]] =:= Long]                             ❷
summon[Elem[Option[List[Long]]] =:= List[Long]]
```

❶ Use `Nil.type` when type matching on the type for an `object`. Same for `None.type` ahead.

❷ This one fails because our `match` expression doesn't recurse into nested types. The correct result is `List[Long]`.

The type `=:=` functions like a type equality test. It is defined in `Predef` along with the `<:<` type we saw in "Implicit Evidence" on page 178. It is normally used with infix notation, as shown here. The following two expressions are equivalent:

```
scala> summon[Elem[String] =:= Char]
scala> summon[=:=[Elem[String], Char]]
```

This type is actually a sealed abstract class that the compiler alone can instantiate if it is possible to construct an instance implicitly, which can only happen if the lefthand-side type is equivalent to the righthand-side type. This is what we see in the REPL:

```
scala> summon[Elem[String] =:= Char]
val res0: Char =:= Char = generalized constraint

...
scala> summon[Elem[Option[List[Long]]] =:= Long]
         |
```

```
1 |summon[Elem[Option[List[Long]]] =:= Long]
  |                                       ^
  |                                  Cannot prove that List[Long] =:= Long.

scala> summon[Elem[Option[List[Long]]] =:= List[Long]]
val res8: List[Long] =:= List[Long] = generalized constraint
```

`match` type expressions can't have guard clauses because they are evaluated at compile time. Also, only types are allowed in the lefthand and righthand sides of the `case` clauses.

Dependently Typed Methods

Match types can be used to implement *dependently typed methods*, meaning the return type of a method depends on the arguments or the enclosing type. In what follows, you'll notice a structural similarity between the match type definition and the `match` expression used in the corresponding method.

The following example defines a recursive version of the previous match type example and uses it as the return type of a method. This method returns the first element in the parameterized type instance. For other types, it just returns the input value:

```
// src/script/scala/progscala3/typesystem/matchtypes/DepTypedMethods.scala

type ElemR[X] = X match          // "R" for "recursive"
  case String => Char
  case Array[t] => ElemR[t]                                        ❶
  case Iterable[t] => ElemR[t]                                     ❷
  case Option[t] => ElemR[t]
  case AnyVal => X

import compiletime.asMatchable                                     ❸
def first[X](x: X): ElemR[X] = x.asMatchable match                 ❹
  case s: String      => s.charAt(0)
  case a: Array[t]    => first(a(0))
  case i: Iterable[t] => first(i.head)
  case o: Option[t]   => first(o.get)
  case x: AnyVal      => x
```

❶ Recursively evaluate `ElemR[X]` on the nested type.

❷ In `Elem[X]`, we matched on `IterableOnce` to support both `Option` and `Iterable` collections. Here, we need different handling of these types in the `first` method, so we match separately on `Iterable` and `Option` in the match type.

❸ An import required for the next `match` expression.

❹ The `first` method. Notice the structure is very similar to the definition of `ElemR`, but why is `asMatchable` required?

The `asMatchable` method works around a limitation involving pattern matching for dependently typed methods and `Matchable`. If you remove the `asMatchable`, that is you just use `x match`, you get errors like this in each of the case clauses:

```
10 |   case s: String      => s.charAt(0)
   |           ^^^^^^
   |           pattern selector should be an instance of Matchable,
   |           but it has unmatchable type X instead
```

A future release of Scala 3 may remove the need for this workaround.

Let's try `first`. Notice the return type in each case:

```
scala> case class C(name: String)     // definitions used below
     | object O

scala> first("one")
val res1: Char = o

scala> first(Array(2.2, 3.3))
val res2: Double = 2.2

scala> first(Seq("4", "five"))
val res3: Char = 4

scala> first(6)
val res4: Int = 6

scala> first(true)
val res5: Boolean = true

scala> first(O)
val res6: O.type = O$@46a55811

scala> first(C("Dean"))
val res7: ElemR[C] = C(Dean)
```

The definitions of `ElemR` and `first` look structurally similar for a reason. Besides the requirements discussed previously for match types, dependently typed methods must have the same number of case clauses as the match type, the lefthand sides of these clauses must be *typed patterns*, meaning of the form `x: X`, and each case clause must be type equivalent (satisfying `=:=`) with its corresponding case clause in the match type.

Dependent Method and Dependent Function Types

A related concept is called *dependent method types* (which is confusing). In this case the return type of a method depends exclusively on one or more of its arguments. New for Scala 3 is support for *dependent function types*. Previously it was not possible to lift a method with a dependent type to a corresponding function. Now it is.

Consider another linked-list implementation that uses an abstract type member for the elements of the list:

```
// src/script/scala/progscala3/typesystem/deptypes/DepMethodFunc.scala
trait LinkedList:
  type Item                                                        ❶
  def head: Item                                                   ❷
  def tail: Option[LinkedList]

def head(ll: LinkedList): ll.Item = ll.head                        ❸
val h: (ll: LinkedList) => ll.Item = _.head
def tail(ll: LinkedList): Option[LinkedList] = ll.tail
val t: (ll: LinkedList) => Option[LinkedList] = _.tail
```

❶ Element `type`, analogous to a parameterized type `List[Item]`.

❷ Methods to return the `head` and `tail`. In this implementation, I use an optional tail to signal the end of the list.

❸ A `head` method and corresponding `h` function defined outside `LinkedList`. Both more clearly show that the dependent return type, `ll.item`, depends on the input type of the `LinkedList`, `ll`. Corresponding `tail` and `t` are also defined, but they don't return dependent types, just `Option[LinkedList]` instances.

Let's try it for `Int`s. Note that we implement `head` and `tail` methods with `val`s:

```
scala> case class IntLinkedList(head: Int, tail: Option[IntLinkedList])
     |     extends LinkedList:
     |   type Item = Int
     |
     | val ill = IntLinkedList(0,
     |   Some(IntLinkedList(1, Some(IntLinkedList(2, None)))))
val ill: IntLinkedList =
  IntLinkedList(0,Some(IntLinkedList(1,Some(IntLinkedList(2,None)))))

scala> head(ill)
     | tail(ill)
     | head(tail(ill).get)     // get retrieves the list from the option
     | head(tail(tail(ill).get).get)
val res0: ill.Item = 0
val res1: Option[LinkedList] =
  Some(IntLinkedList(1,Some(IntLinkedList(2,None))))
val res2: LinkedList#Item = 1
```

```
val res3: LinkedList#Item = 2

scala> h(ill)
     | t(ill)
     | h(t(ill).get)
     | h(t(t(ill).get).get)
... same output ...
```

The `head` method and `h` function have dependent return types, specifically the `Item` member of `LinkedList`, which will depend on the actual type used for `Item`. The `LinkedList#Item` types shown for the returned values from `head` are actually `Int`s.

Dependent Typing

Another sense of dependent typing are types that depend on values. It is a powerful concept for more precise type checking and enforcement of desired behaviors. Consider the following examples. First, we can be more specific than `Int`, `Double`, etc.:

```
// src/script/scala/progscala3/typesystem/deptypes/DependentTypesSimple.scala

scala> val one: 1 = 1
     | val two: 2.2 = 2.2
val one: 1 = 1
val two: 2.2 = 2.2

scala> val two: 2.2 = 2.21
1 |val two: 2.2 = 2.21
  |               ^^^^
  |               Found:    (2.21 : Double)
  |               Required: (2.2 : Double)
  | ...
```

Note the types printed. For example, the type of `one` is `1`. These are singleton types because only a single value is possible, just as `object`s are singleton types. The types `1` and `2.2` are considered subtypes of `Int` and `Double`, respectively:

```
scala> summon[1 <:< Int]
     | summon[2.2 <:< Double]
val res1: Int =:= Int = generalized constraint
val res2: Double =:= Double = generalized constraint

scala> summon[2 <:< Double]
1 |summon[2 <:< Double]
  |                    ^
  |                    Cannot prove that (2 : Int) <:< Double.
```

Comparisons at the type level can be used to determine true or false. This is computed at compile time. I'll just show the expressions, not the REPL results. For details on the operations shown, see the packages under `scala.compiletime.ops`, which are imported as shown:

```scala
def opsAny =                                          // ❶
  import scala.compiletime.ops.any.*

  val any1: 2 == 2       = true                       // ❷
  val any2: 1 == 2       = false
  val any3: 1 != 2       = true
  val any4: "" == ""     = true
  val any5: "" != ""     = false
  val any6: "" != "boo" = true

  valueOf[2 == 2]  == true                            // ❸
  valueOf[1 == 2]  == false
  valueOf[1 != 2]  == true

opsAny                                                // ❹
```

❶ Use a method to scope the `import` statement. In this case type-level comparisons are enabled.

❷ The type is computed from `2 == 2`, which is `true`, the only allowed value for the assignment.

❸ We can also play with these examples using `valueOf`, which returns the value corresponding to a singleton type.

❹ Try it!

Integer arithmetic is possible on types. Only some of the possibilities are shown here. See the `DependentTypesSimple.scala` file for more examples:

```scala
def opsInt =
  import scala.compiletime.ops.int.*                  // ❶

  val i1: 0 + 1     = 0 + 1
  val i2: 1 + 1     = 1 + 1
  val i3: 1 + 2     = 1 + 2
  val i4: 3 * 2 - 1 = 3 * 2 - 1
  val i5: 12 / 3    = 4
  val i6: 11 % 4    = 3

  val lshift:  1 <<  2 = 4                            // ❷
  val rshift:  8 >>  2 = 2
  val rshift2: 8 >>> 2 = 2                            // ❸

  val b2: 1 <  2 = true                               // ❹
  val b3: 1 <= 2 = true
  val b4: 2 <  1 = false
  val b5: 1 >  2 = false
  val b6: 1 >= 2 = false
  val b7: 2 >  1 = true
```

```
  val xor: 14 ^ 5          = 11  // 14 xor 5 => 1110 ^ 0101 => 1011 => 11
  val and: BitwiseAnd[5, 4] = 4  // 5 & 4 => 101 & 100 == 100 => 4
  val or:  BitwiseOr[5, 3]  = 7  // 5 | 3 => 101 | 011 == 111 => 7

  val abs: Abs[-1] = 1
  val neg: Negate[2] = -2
  val min: Min[3, 5] = 3
  val max: Max[3, 5] = 5
  val s:   ToString[123] = "123"

opsInt
```

❶ Import to enable integer type arithmetic.

❷ Left-shift 1 by 2 bits, yielding 4.

❸ Right-shift, filling with zeros on the left.

❹ More ways to compute Booleans, but only using integers. If you want to use == and !=, then import `scala.compiletime.ops.any.*`

One way to encode nonnegative integers is *Peano numbers*, which define a zero and a successor function used to compute all other values. Let's use the type-level `scala.compiletime.ops.int.S`, an implementation of the successor function:

```
def tryS =
  import scala.compiletime.ops.int.S

  val s1:  S[0] = 1                                          ❶
  val s2a: S[S[0]] = 2                                       ❷
  val s2b: S[1] = 2                                          ❸
  val s3a: S[S[S[0]]] = 3
  val s3b: S[2] = 3

tryS
```

❶ The successor of `0` is `1`.

❷ The successor of the successor of `0` is `2`.

❸ However, you don't have to start at `0`.

Boolean singletons and Boolean logic are supported:

```
def opsBoolean =
  import scala.compiletime.ops.boolean.*

  val t1: true = true
  val f1: ![true] = false                                    ❶
```

```
  val tt1: true  && true  = true
  val tf1: true  && false = false
  val ft1: false && true  = false
  val ff1: false && false = false
  val tt2: true  || true  = true
  val tf2: true  || false = true
  val ft2: false || true  = true
  val ff2: false || false = false
  val tt3: true   ^ true  = false                                 ❷
  val tf3: true   ^ false = true
  val ft3: false  ^ true  = true
  val ff3: false  ^ false = false

opsBoolean
```

❶ Negation. The brackets are required.

❷ Exclusive or (*xor*).

String singleton types and string concatenation are supported:

```
def opsString =
  import scala.compiletime.ops.string.*

  val s1: "ab" + "cd" = "abcd"
  val bad2: "ab" + "cd" = "abcdef"   // ERROR

opsString
```

OK, but how is all this useful? One use is to ensure that allowed state transitions are checked at compile time. For example, if I add an element to a collection the size must increase by one (zero or one for `Sets`!). Similarly, if I remove an element, the size must decrease by one. If I concatenate two sequences, the length of the resulting sequence is the sum of the two original sequences.

Let's consider a more detailed example. Here is yet another implementation of linked lists. It has a few advantages over the library's `List` type. It remembers the type of each element and carries its size as part of the type. On the other hand, it doesn't implement all the useful methods like `map` and `flatMap`. Also, for big-list literals, it will be somewhat expensive at compile time. The implementation uses some advanced constructs, but I'll walk you through it. I call it `DTList`, for *dependently typed list*, where the dependency is the value of the size of the list:

```
// src/script/scala/progscala3/typesystem/deptypes/DependentTypes.scala

import scala.compiletime.ops.int.*

sealed trait DTList[N <: Int]:                                    ❶
  inline def size: N = valueOf[N]                                 ❷
```

```
  def +:[H <: Matchable](h: H): DTNonEmptyList[N, H, this.type] =     ❸
    DTNonEmptyList(h, this)

case object DTNil extends DTList[0]                                   ❹

case class DTNonEmptyList[N <: Int, H <: Matchable, T <: DTList[N]]( ❺
    head: H, tail: T) extends DTList[S[N]]
```

❶ The base trait for empty and nonempty lists. While Scala's `List` uses a type parameter for the least upper bound (closest supertype) of the elements' types, `DTList` will retain each element's type. Instead, the type parameter here is the size of the list.

❷ Return the size of the list. The `inline` modifier tells the compiler to inline the method implementation (see Chapter 24 for more details). This method is dependently typed because the return type will be `0`, `1`, etc., depending on the particular list and the value of *N*. The value is obtained from the type using `valueOf`. For example, `valueOf[2]` returns `2`.

❸ A method like `Seq.+:` to construct a new `DTList` by prepending an element to this list. By definition, the result is nonempty list.

❹ The analog of `Nil` for empty lists with the size type parameter of `0`.

❺ The type for nonempty lists. Its size is actually `N + 1`; it passes `S[N]` as the parameter to `DTList`. It also has a type parameter for the new head element, `H`, and a type `T` for the tail that must be one of our `DTList` types.

The parameter `N` for `DTNonEmptyList` is actually one less than its actual size, which is why `S[N]` is passed to `DTList`. Hence, the size will be `N + 1`.

Let's try it:

```
scala> val list = 1 +: "two" +: DTNil
val list: DTNonEmptyList[1, Int, ? <: DTNonEmptyList[0, String, DTNil.type]] =
  DTNonEmptyList(1,DTNonEmptyList(two,DTNil))

scala> list.size
     | list.head
     | list.tail
val res0: Int = 2
val res1: Int = 1            // head element correctly typed as Int
val res2: list.T = DTNonEmptyList(two,DTNil)

scala> list.tail.size
     | list.tail.head
     | list.tail.tail
     | list.tail.tail.size
```

```
val res3: Int = 1
val res4: String = two        // head element correctly typed as String
val res5: DTNil.type = DTNil
val res6: Int = 0

scala> list.tail.tail.head         // list.tail.tail is res5 == DTNil
     | list.tail.tail.tail
1 |list.tail.tail.head
  |^^^^^^^^^^^^^^^^^^^
  |value head is not a member of object DTNil
2 |list.tail.tail.tail
  |^^^^^^^^^^^^^^^^^^^
  |value tail is not a member of object DTNil
```

Note the type returned for `list`. It retains the types of each element, unlike Scala's `List`, which only knows the least upper bound. When we call `head`, we get the correct type for the value returned.

3 Because `DTList` retains the type information for each element, it is similar to tuples. In fact, Scala 3 expands what you can do with tuples, as we saw in "Tuples and the Tuple Trait" on page 317, which makes them work more like lists with full typing of the elements. Scala 3 tuples now support many of the features of the advanced library `HList` in *Shapeless*.

You can verify that each of the types and values returned are what we expect, although `list.T` for `res2` reflects how we constructed it.

It's also notable that the `head` and `tail` accessors don't exist on the `DTList` trait nor on `DTNil`. Attempts to call them are caught at compile time. In contrast, Scala's `List` implementation has to declare these methods on the base trait because you don't always know the type of a `List`, whether you have a `Nil` or a nonempty `List`. This means these methods have to throw runtime exceptions if called on `Nil`. Instead, we can catch such errors at compile time.

The example source file also has a similar `SList` (for "simple") definition that is a stripped-down version of Scala's `List`, so it's easier to compare the two list implementations. To be honest, `SList` is easier to understand than `DTList`, and it's also very challenging to implement most of the `Seq[T]` combinator methods on `DTList`.

The same directory in the code examples has a few other examples using dependent types that I won't discuss here.

In practical terms, I think we'll see the use of dependent typing grow, squeezing out potential bugs and limitations of more conventional APIs, but the challenges of using dependent typing mean that growth will be slow and deliberate.

Path-Dependent Types

You can access nested types using a *path* expression and those path contexts differentiate between similar types. Consider this example:

```
// src/script/scala/progscala3/typesystem/typepaths/TypePath.scala

open class Service:                                                    ❶
  class Logger:
    def log(message: String): Unit = println(s"log: $message")

  val logger: Logger = Logger()

val s1 = new Service
val s2 = new Service:
  override val logger: Logger = s1.logger                              ❷
```

❶ Define a class `Service` with a nested class `Logger`.

❷ Attempt to override `logger` in `s2`, reusing `s1.logger`, but this causes a compilation error.

```
scala> val s2 = new Service:
     |   override val logger: Logger = s1.logger
2 |  override val logger: Logger = s1.logger
  |                                ^^^^^^^^^
  |                                Found:    (s1.logger : s1.Logger)
  |                                Required: Logger
```

The `s1.Logger` and `s2.Logger` types are considered different because they are *path dependent*, starting from different paths, `s1` and `s2`, respectively. Let's discuss the kinds of type paths.

Using this

For a class `C1`, you can use the familiar `this` inside the body to refer to the current instance, but `this` is actually a shorthand for `C1.this` in Scala:

```
// src/main/scala/progscala3/typesystem/typepaths/PathExpressions.scala
package progscala3.typesystem.typepaths

open class C1:
  var x = "1"
  def setX1(x:String): Unit = this.x = x
  def setX2(x:String): Unit = C1.this.x = x
```

Inside a type body, `this` can refer to the type itself when referencing a nested type definition:

```
trait T1:
  class C
```

```
    val c1: C = C()
    val c2: C = this.C()
```

Here, `this` in the expression `this.C` refers to the trait `T1`.

Using super

You can refer to the supertype of a type with `super`:

```
trait X:
  var xx = "xx"
  def setXX(x:String): Unit = xx = x

open class C2 extends C1
open class C3 extends C2 with X:
  def setX3(x:String): Unit = super.setX1(x)
  def setX4(x:String): Unit = C3.super.setX1(x)
  def setX5(x:String): Unit = C3.super[C2].setX1(x)
  def setX6(x:String): Unit = C3.super[X].setXX(x)
  // def setX7(x:String): Unit = C3.super[C1].setX1(x)    // ERROR
  // def setX8(x:String): Unit = C3.super.super.setX1(x)  // ERROR
```

`C3.super` is equivalent to `super` in this example. You can qualify which supertype using `super[T]`, as shown for `setX5` and `setX6`. However, you can't refer to `super` supertypes (`setX7`). You can't chain `super`, either (`setX8`). I'll discuss a workaround in "Self-Type Declarations" on page 382.

If you call `super` without qualification in a type with several ancestors, to which type does `super` bind? The rules of linearization determine the target of `super` (see "Linearization of a Type Hierarchy" on page 301).

Just as for `this`, you can use `super` to refer to the supertype to access a nested type:

```
open class C4:
  class C5

open class C6 extends C4:
  val c5a: C5 = C5()
  val c5b: C5 = super.C5()
```

Stable Paths

You can reach a nested type with a period-delimited path expression. All but the last elements of a type path must be *stable*, which roughly means they must be packages, singleton objects, or type declarations that alias the same. The last element in the path can be unstable, including classes, traits, and type members. Consider this example:

```
package P1:
  object O1:
    object O2:
      val name = "name"
```

```
    class C1:
      val name = "name"

open class C7:
  val  name1 = P1.O1.O2.name      // Okay  - a reference to a field
  type C1    = P1.O1.C1           // Okay  - a reference to a "leaf" class
  val  c1    = P1.O1.C1()         // Okay  - same reason
  // val name2 = P1.O1.C1.name    // ERROR - P1.O1.C1 isn't stable.
```

The `C7` members `name1`, `C1`, and `c1` all use stable elements until the last position, while `name2` has an unstable element (`C1`) before the last position. You can see this if you uncomment the `name2` declaration, leading to the following compilation error:

```
[error] 55 |  val name2 = P1.O1.C1.name    // ERROR - P1.O1.C1 isn't stable.
[error]    |              ^^^^^^^^
[error]    |value C1 is not a member of object ...typepaths.P1.O1
```

Of course, avoiding complex paths in your code is a good idea for clarity and comprehension.

Self-Type Declarations

You can use `this` in a method to refer to the enclosing instance, which is useful for referencing another member of the instance. Explicitly using `this` is not usually necessary for this purpose, but it's occasionally useful for disambiguating a reference when several items are in scope with the same name.

Self-type declarations (also called *self-type annotations*) support two objectives. First, they let you specify additional type expectations for `this`. Second, they can be used to create aliases for `this`, which solves the limitation we saw earlier that you can't use `super` to refer to types beyond the parent types.

To illustrate specifying additional type expectations, let's implement a `SubjectObserver` class to combine the concepts of `Subject` and `Observer` we've seen before:

```
// src/main/scala/progscala3/typesystem/selftype/SubjectObserver.scala
package progscala3.typesystem.selftype

abstract class SubjectObserver:
  type S <: Subject                                                    ❶
  type O <: Observer

  trait Subject:
    self: S =>                                                         ❷
    private var observers = List[O]()

    def addObserver(observer: O) = observers ::= observer

    def notifyObservers() = observers.foreach(_.receiveUpdate(self)) ❸
```

```
trait Observer:
  def receiveUpdate(subject: S): Unit
```

❶ Use abstract type members for the specific `Subject` and `Observer` types, subtypes of the traits defined next in `SubjectObserver`.

❷ Declare a self-type declaration for `Subject`, which is `self: S`. This means that we can now assume that a `Subject` will really be an instance of the subtype `S`, which will be whatever concrete types we define that mix in `Subject`. The name `self` is completely arbitrary.

❸ Pass `self` rather than `this` to `receiveUpdate`.

It's not obvious why the self-type declaration is necessary, but if you remove it and try passing `this` to `receiveUpdate` instead of `self`, you'll get a type error. This is because `this` is of type `Subject`, but it needs to be of the more specific type, `SubjectObserver.this.S`. Note that `S` is declared to be a subtype of `Subject`, so the more specific type is required when passing an instance to `receiveUpdate`.

Let's see how the types might be used to observe button clicks:

```
// src/main/scala/progscala3/typesystem/selftype/ButtonSubjectObserver.scala
package progscala3.typesystem.selftype

case class Button(label: String):                                        ❶
  def click(): Unit = {}

object ButtonSubjectObserver extends SubjectObserver:                   ❷
  type S = ObservableButton
  type O = Observer

  class ObservableButton(label: String) extends Button(label) with Subject:
    override def click() =                                               ❸
      super.click()
      notifyObservers()

  class ButtonClickObserver extends Observer:                           ❹
   val clicks = scala.collection.mutable.HashMap[String,Int]()

    def receiveUpdate(button: ObservableButton): Unit =
      val count = clicks.getOrElse(button.label, 0) + 1
      clicks.update(button.label, count)

@main def TryButtonSubjectObserver() =
  import ButtonSubjectObserver.*

  val button1 = ObservableButton("one")
  val button2 = ObservableButton("two")
  val observer = ButtonClickObserver()
```

```
  button1.addObserver(observer)
  button2.addObserver(observer)
  button1.click()
  button2.click()
  button1.click()
  println(observer.clicks)
```

❶ A simple `Button` class.

❷ A concrete subtype of `SubjectObserver` for buttons, where `Subject` and `Observer` are both subtyped to the more specific types we want.

❸ `ObservableButton` overrides `Button.click` to notify the observers after calling `Button.click`.

❹ Implement `ButtonObserver` to track the number of clicks for each button in a UI.

If you run `progscala3.typesystem.selftype.TryButtonSubjectObserver`, the last line prints `HashMap(one -> 2, two -> 1)`.

So we can use self-type declarations to solve a typing problem when using abstract type members.

A related use is an old Scala design pattern called the Cake Pattern, which was a way of specifying components to wire together for an application. This pattern is seldom used now, due to typing challenges that I won't discuss here. See `SelfTypeCakePattern.scala` in the code examples for more details.

The second usage of self-type declarations is to alias `this` in narrow contexts so it can be referenced elsewhere:

```
// src/script/scala/progscala3/typesystem/selftype/ThisAlias.scala

class C1:
  c1this =>                                                       ❶

  def talk(message: String): String = "C1.talk: " + message
  class C2:
    class C3:
      def talk(message: String) = c1this.talk("C3.talk: " + message) ❷
    val c3 = C3()
  val c2 = C2()

val c1 = C1()
assert(c1.talk("Hello") == "C1.talk: Hello")                      ❸
assert(c1.c2.c3.talk("World") == "C1.talk: C3.talk: World")       ❹
```

❶ Define `c1this` to be an alias of `this` in the context of `C1`. There is nothing on the righthand side of the arrow.

❷ Use `c1this` to call `C1.talk`.

❸ Call `C1.talk` via the `c1` instance.

❹ Call `C3.talk` via the `c1.c2.c3` instance, which will itself call `C1.talk`.

We could also define self-type declarations inside `C2` and `C3`, if we needed them.

Without the self-type declaration, we can't invoke `C1.talk` directly from within `C3.talk` because the latter shadows the former, since they share the same name. `C3` is not a subtype of `C1` either, so `super.talk` can't be used. This use of self-type declarations is also a workaround that you can't use `super` beyond one supertype level.

You can think of the self-type declaration in this context as a generalized `this` reference.

Type Projections

Let's revisit our `Service` design problem in "Path-Dependent Types" on page 380. First, let's rewrite `Service` to extract some abstractions that would be more typical in real applications:

```
// src/main/scala/progscala3/typesystem/valuetypes/TypeProjection.scala
package progscala3.typesystem.valuetypes

trait Logger:                                                          ❶
  def log(message: String): Unit

class ConsoleLogger extends Logger:                                    ❷
  def log(message: String): Unit = println(s"log: $message")

trait Service:                                                         ❸
  type Log <: Logger
  val logger: Log

class ConsoleService extends Service:                                  ❹
  type Log = ConsoleLogger
  val logger: ConsoleLogger = ConsoleLogger()
```

❶ A `Logger` trait.

❷ A concrete `Logger` that logs to the console, for simplicity.

❸ A `Service` trait that defines an abstract type member for the `Logger` and declares a field for it.

❹ A concrete service that uses `ConsoleLogger`.

Suppose we want to reuse the `Log` type defined in `ConsoleService`. We can project the type we want with `#`:

```
// src/script/scala/progscala3/typesystem/valuetypes/TypeProjection.scala
scala> import progscala3.typesystem.valuetypes.*

scala> val l1: Service#Log = ConsoleLogger()
1 |val l1: Service#Log = ConsoleLogger()
  |                      ^^^^^^^^^^^^^^^
  |           Found:    progscala3.typesystem.valuetypes.ConsoleLogger
  |           Required: progscala3.typesystem.valuetypes.Service#Log

scala> val l2: ConsoleService#Log = ConsoleLogger()
val l2: ...ConsoleService#Log = ...ConsoleLogger@2287b06a
```

The first attempt doesn't type check. Although both `Service.Log` and `ConsoleLogger` are both subtypes of `Logger`, `Service.Log` is abstract so we don't yet know if it will actually be a supertype of `ConsoleLogger`. In other words, the final concrete definition could be another subtype of `Logger` that isn't compatible with `ConsoleLogger`. The only one that works is the second definition because the types check statically.

3 In Scala 3, a type projection `T#A` is not permitted if the type `T` is abstract. This was permitted in Scala 2, but it undermined type safety.

More on Singleton Types

Singleton objects define both an instance and a corresponding type. You can access the latter using `.type`. Even other instances have a singleton type:

```
// src/script/scala/progscala3/typesystem/valuetypes/SingletonTypes.scala

case object Foo:
  override def toString = "Foo says Hello!"

def fooString(foo: Foo.type) = s"Foo.type: $foo"                    ❶

case class C(s: String)
val c1 = C("c1")
println(c1)
val c1b: c1.type = c1                                               ❷
println(c1b)
val c1c: c1.type = C("c1")                                          ❸
```

❶ Use `Foo.type` to reference the type of the object `Foo`.

❷ The singleton type for the specific instance `c1` is `c1.type`.

❸ This doesn't compile because the type of the new instance `C("c1")` is not the same as `c1.type`. It will have its own unique singleton type.

Self-Recursive Types: F-Bounded Polymorphism

Self-recursive types, technically called *F-bounded polymorphic types*, are types that refer to themselves. A classic Java example, which has confused generations of programmers, is the `Enum` abstract class, the basis for all Java enumerations:

```
public abstract class Enum<E extends Enum<E>>
  extends Object implements Comparable<E>, Serializable
```

Where this recursion is useful is to constrain method arguments or return values in subtypes to have exactly the same type as the subtype, not a more generic type. Let's look at a Scala example, where a supertype will declare a custom factory method, `make`, that must return the same type as a subtype caller's actual type:

```
// src/script/scala/progscala3/typesystem/recursivetypes/FBound.scala

trait Super[T <: Super[T]]:                                    ❶
  def make: T                                                  ❷

case class Sub1(s: String) extends Super[Sub1]:                ❸
  def make: Sub1 = Sub1(s"Sub1: make: $s")

case class Sub2(s: String) extends Super[Sub2]:
  def make: Sub2 = Sub2(s"Sub2: make: $s")

// case class Foo(str:String)                                  ❹
// case class Odd(s: String) extends Super[Foo]:
//   def make: Foo = Foo(s"Foo: make: $s")
```

❶ `Super` has a recursive type. This syntax is the Scala equivalent for the preceding Java syntax for `Enum`.

❷ Whatever subtype of `Super`, `T`, is used, that's what implementations of `make` should return.

❸ Subtypes must follow the signature idiom `X extends Super[X]`.

❹ It's disallowed for `Odd` to pass `Foo` to the parent `Super`. It also can't return `Foo` from `make`.

Let's try it:

```
scala> val s1  = Sub1("s1")
     | val s2  = Sub2("s2")
val s1: Sub1 = Sub1(s1)
val s2: Sub2 = Sub2(s2)

scala> val s11 = s1.make
     | val s22 = s2.make
val s11: Sub1 = Sub1(Sub1: make: s1)
val s22: Sub2 = Sub2(Sub2: make: s2)
```

Notice that `s11` is of type `Sub1`, and `s22` is of type `Sub2`, not `Super`.

If we didn't declare the type parameter `T` to be a subtype of `Super[T]`, then a subclass could pass any arbitrary type `Super[T]`, as in the `Odd` comment. F-bounded polymorphism constrains the `T` type parameter to be one of the types in the `Super` hierarchy, the exact same subtype in this example, so that `make` always returns an instance of the type we want.

Higher-Kinded Types

Sometimes you'll see the term *type constructor* used for a parameterized type because such types are used to construct other types by providing specific types for the type parameters. This is analogous to how a class without type parameters is an *instance constructor*, where values are provided for the fields to create instances. For example, `Seq[T]` is used to construct the type `Seq[String]`, which can then be used to construct instances of `Seq[String]`, while `String` is used to construct instances like "hello world."

What if we want to abstract over all types that take one type parameter or two? The term *higher-kinded types* refers to all such parameterized types, such as `F[_]` and `G[_,_]` for one-parameter and two-parameter types. Scala provides tools for abstracting over higher-kinded types.

To get started, recall that the collections provide several `fold` methods, like `foldLeft`, which we first examined in "Folding and Reducing" on page 210. Here's one way you could sum a collection of numbers:

```
def add(seed: Int, seq: Seq[Int]): Int = seq.foldLeft(seed)(_ + _)

add(5, Vector(1,2,3,4,5))   // Result: 20
```

(`Seq.sum` also exists for all `Numeric` types.) Let's suppose that the collections didn't already provide `fold` methods. How might we implement them? We'll just worry about `foldLeft` and implement it as a separate module, not an extension method. Here's one possible approach that works for any subtype of `Seq[T]`:

```
// src/script/scala/progscala3/typesystem/higherkinded/FoldLeft.scala

object FoldLeft:
  def apply[IN, OUT](seq: Seq[IN])(seed: OUT)(f: (OUT, IN) => OUT): OUT =
    var accumulator = seed
    seq.foreach(t => accumulator = f(accumulator, t))
    accumulator

  def apply[IN, OUT](opt: Option[IN])(seed: OUT)(f: (OUT, IN) => OUT): OUT =
    opt match
      case Some(t) => f(seed, t)
      case None => seed
```

`FoldLeft` defines `apply` methods for two kinds of collections, `Seq` and `Option`. We could add an `apply` method for `Arrays` and other types too. Following the convention of the built-in `foldLeft` methods, the functions passed to the `apply` methods take two arguments, the accumulated output and each element. Hence the types for `f` are `(OUT, IN) => OUT`.

Let's verify that they work:

```
scala> FoldLeft(List(1, 2, 3))(0)(_+_)
val res0: Int = 6

scala> FoldLeft(List(1, 2, 3))("(0)")((s, i) => s"($s $i)")
val res1: String = ((((0) 1) 2) 3)

scala> FoldLeft(Array(1, 2, 3).toSeq)(0)(_+_)                    ❶
val res2: Int = 6

scala> FoldLeft(Vector(1 -> "one", 2 -> "two", 3 -> "three"))(0 -> "(0)"){
     |   case ((xs, ys), (x,y)) => (xs+x, s"($ys $y)")
     | }
val res3: (Int, String) = (6,((((0) one) two) three))

scala> FoldLeft(Some(1.1))(0.0)(_+_)
     | FoldLeft(Option.empty[Int])(0.0)(_+_)                     ❷
val res4: Double = 1.1
val none: Option[Int] = None
val res5: Double = 0.0
```

❶ Use `Array.toSeq` so the `Seq`-version of `apply` can be used.

❷ If we passed `None` here, the type can't be inferred. Using `Option.empty[Int]` returns `None`, but with the necessary type information. Try `None` instead and see what happens.

This implementation works, but it's unsatisfactory having to write multiple `apply` methods for the different cases, and we didn't cover all types we might care about, like

Arrays. We might not always have a suitable, common supertype where we can add such methods. Furthermore, we have to edit this code to broaden the support.

Now let's leverage the abstraction over higher-kinded types to write one apply method with a more composable and broadly applicable implementation:

```
// src/script/scala/progscala3/typesystem/higherkinded/HKFoldLeft.scala

object HKFoldLeft:       // "HK" for "higher-kinded"

  trait Folder[-M[_]]:                                          ❶
    def apply[IN, OUT](m: M[IN], seed: OUT, f: (OUT, IN) => OUT): OUT

  given Folder[Iterable] with                                   ❷
    def apply[IN, OUT](iter: Iterable[IN],
        seed: OUT, f: (OUT, IN) => OUT): OUT =
      var accumulator = seed
      iter.foreach(t => accumulator = f(accumulator, t))
      accumulator

  given Folder[Option] with                                     ❸
    def apply[IN, OUT](opt: Option[IN],
        seed: OUT, f: (OUT, IN) => OUT): OUT = opt match
      case Some(t) => f(seed, t)
      case None => seed

  def apply[IN, OUT, M[IN]](m: M[IN])(                          ❹
      seed: OUT)(f: (OUT, IN) => OUT)(using Folder[M]): OUT =
    summon[Folder[M]](m, seed, f)
```

❶ Define a helper trait, Folder, that abstracts over higher-kinded types with one type parameter. We'll implement given instances for different types of higher-kinded types. These instances will do most of the work. I'll explain why the type parameter is contravariant later on.

❷ Define a given instance that works for all subtypes of Iterable, such as all Seq subtypes. I could cheat and use iter.foldLeft, but I'll just assume that foreach is available, like before.

❸ Define a given instance for Option, which isn't an Iterable. New given instances can be defined elsewhere to support more higher-kinded types, as we'll see ahead.

❹ The apply method users will call. It has three parameter lists: the instance of a higher-kinded type M[IN]; the seed value of type OUT for the fold, the function that performs the fold for each element; and a using clause for the Folder[M] that does the real work.

The Folder type parameter -M[_] is contravariant because we implemented a given for the supertype Iterable, but users will pass types like List and Map. While Map[K,V] has two type parameters, it implements Iterable[(K,V)], so our implementation works for maps too.

The Scala collections use a more sophisticated and general approach for implementing methods like flatMap, while returning the correct concrete subtype.

Using HKFoldLeft instead of FoldLeft in the previous examples returns the same results. Here are some of the details:

```
scala> import HKFoldLeft.{given, *}    // Required everything

scala> summon[Folder[Iterable]]        // Verify the givens exist
     | summon[Folder[Option]]
val res0: HKFoldLeft.given_Folder_Iterable.type = ...
val res1: HKFoldLeft.given_Folder_Option.type = ...

scala> HKFoldLeft(List(1, 2, 3))(0)(_+_)
     | HKFoldLeft(List(1, 2, 3))("(0)")((s, i) => s"($s $i)")
val res2: Int = 6
val res3: String = ((((0) 1) 2) 3)

scala> HKFoldLeft(Some(1.1))(0.0)(_+_)
     | HKFoldLeft(Option.empty[Int])(0.0)(_+_)
val res4: Double = 1.1
val res5: Double = 0.0
```

What if we want to add support for a type with two or more type parameters, but we only need to fold over one of them? That's where type lambdas help.

Type Lambdas

3 A *type lambda* is the type analog of a function. Scala 3 introduces a syntax for them.[1] Suppose we want a type alias for Either where the first type is always String. We'll call it Trial for something that may fail or succeed:

```
type Trial[X] = Either[String,X]          // Syntax we've used before.
type Trial = [X] =>> Either[String,X]     // Type lambda syntax.
```

The second version shows the new type lambda syntax, which is deliberately like a corresponding function type: X => Either[String,X]. The term *lambda* is another name for function. One advantage of the lambda syntax is that you can use it most places where a type is expected, whereas the older type alias syntax requires you to define an alias like Trial[X] first and then use it.

1 In Scala 2, you had to use the Typelevel compiler plug-in kind-projector for similar capabilities.

Let's use this same type to add support for `Either[String,X]` to `HKFoldLeft`:

```
scala> given Folder[[X] =>> Either[String, X]]:
     |   def apply[IN, OUT](err: Either[String, IN],
     |       seed: OUT, f: (OUT, IN) => OUT): OUT = err match
     |     case Right(t) => f(seed, t)
     |     case _ => seed

scala> summon[Folder[[X] =>> Either[String, X]]]
val res10: given_Folder_Either.type = given_Folder_Either$@709c5def

scala> val bad: Either[String,Int] = Left("error")
     | val good: Either[String,Int] = Right(11)
     | HKFoldLeft(bad)(0.0)(_+_)
     | HKFoldLeft(good)(2.0)(_+_)
val bad: Either[String, Int] = Left(error)
val good: Either[String, Int] = Right(11)
val res11: Double = 0.0
val res12: Double = 13.0
```

We had to provide type declarations for `bad` and `good` to specify both types. Otherwise, the using clause when we call `HKFoldLeft.apply` wouldn't find the given `Folder` for `Either[String,X]`.

Type lambdas can have bounds, `<:` and `>:`, but can't use context bounds, like `T: Numeric`, nor can they be marked covariant `+` or contravariant `-`. Despite the latter limitation, the rules are enforced by the compiler when a type alias refers to a type that has covariant or contravariant behavior or both:

```
type Func1 = [A, B] =>> Function1[A, B]   // i.e., Function1[-A, +B]
```

Type lambdas can be curried:

```
type Func1Curried = [A] =>> [B] =>> Function1[A, B]
```

Finally, here's an example using a type lambda with a `Functor` type class so we can create a given instance that works over the values of a map:

```
// src/main/scala/progscala3/typesystem/typelambdas/Functor.scala
package progscala3.typesystem.typelambdas

trait Functor[M[_]]:
  extension [A] (m: M[A]) def map2[B](f: A => B): M[B]

object Functor:
  given Functor[Seq] with
    extension [A] (seq: Seq[A]) def map2[B](f: A => B): Seq[B] = seq map f

  type MapKV = [K] =>> [V] =>> Map[K,V]                          ❶

  given [K]: Functor[MapKV[K]] with                              ❷
    extension [V1] (map: MapKV[K][V1])
      def map2[V2](f: V1 => V2): MapKV[K][V2] = map.view.mapValues(f).toMap
```

❶ A curried type lambda to allow us to use the `Functor` over a map's values.

❷ The key type `K` is fixed, and we map from value type `V1` to `V2`.

Let's try it:

```
// src/script/scala/progscala3/typesystem/typelambdas/Functor.scala
scala> import progscala3.typesystem.typelambdas.Functor.given           ❶

scala> Seq(1,2,3).map2(_ * 2.2)
     | Nil.map2(_.toString)
val res0: Seq[Double] = List(2.2, 4.4, 6.6000000000000005)
val res1: Seq[String] = List()

scala> Map("one" -> 1, "two" -> 2, "three" -> 3).map2(_ * 2.2)
     | Map.empty[String,Int].map2(_.toString)
val res2: progscala3.typesystem.typelambdas.Functor.MapK[String][Double] =
  Map(one -> 2.2, two -> 4.4, three -> 6.6000000000000005)
val res3: progscala3.typesystem.typelambdas.Functor.MapK[String][String] = Map()
```

❶ Import the givens using the new given import syntax.

Polymorphic Functions

Methods have always supported polymorphism using a type parameter. Scala 3 extends this to functions. Let's start with a simple example, which maps over a sequence and returns tuples:

```
// src/script/scala/progscala3/typesystem/poly/PolymorphicFunctions.scala

val seq = Seq(1,2,3,4,5)

def m1[A <: AnyVal](seq: Seq[A]) = seq.map(e => (e,e))           ❶
val pf1 = [A <: AnyVal] => (seq: Seq[A]) => seq.map(e => (e,e)) ❷
val pf2: [A <: AnyVal] => Seq[A] => Seq[(A,A)] =                ❸
  [A] => (seq: Seq[A]) => seq.map(e => (e,e))
m1(seq)    // List((1,1), (2,2), (3,3), (4,4), (5,5))
pf1(seq)   // same
pf2(seq)   // same
```

❶ A polymorphic method. We've seen many of them already.

❷ The equivalent polymorphic function.

❸ With an explicit type signature. Note that `[A] =>` is also required on the right-hand side.

The syntax resembles type lambdas, but with the regular function arrow, `=>`. We have a type parameter, which can have upper and lower bounds, followed by the arguments, and ending with the literal body of the function. Note that we don't have a simple equals sign = before the body because the body is also used to infer the return type.

At the time of this writing, context bounds aren't supported:[2]

```
def m2[A : Numeric](seq: Seq[A]): A =                              // Okay
  seq.reduce((a,b) => summon.times(a,b))
val pf2 = [A] => (seq: Seq[A]) => (using n: Numeric[A]) =>         // ERROR
  seq.reduce((a,b) => n.times(a,b))
val pf2 = [A : Numeric] => (seq: Seq[A]) =>                        // ERROR
  seq.reduce((a,b) => n.times(a,b))
```

The method syntax is more familiar and easier to read, so why have polymorphic functions? They will be most useful when you need to pass a polymorphic function to a method and you want to implement function instances for a set of suitable types.

```
trait Base:
  def id: String
case object O1 extends Base:
  def id: String = "object O1"
case object O2 extends Base:
  def id: String = "object O2"

def work[B <: Base](b: B)(f: [B <: Base] => B => String) = s"<${f(b)}>"
val fas = [B <: Base] => (b: B) => s"found: $b"
work(O1)(fas)      // Returns: "<found: O1>"
work(O2)(fas)      // Returns: "<found: O2>"
```

See other examples in this source file that discuss two or more type parameters.

3 Type Wildcard Versus Placeholder

I mentioned in several places, starting with "Givens and Imports" on page 159, that Scala 3 has changed the wildcard for types from _ to ?. For example, when you don't need to know the type of a sequence's parameters:

```
def length(l: Seq[?]) = l.size
```

The rationale for this change is to reserve use of _ to be the placeholder for arguments passed to types and functions. (The use of _ for imports is also being replaced with *.) Specifically, `f(_)` is equivalent to `x => f(x)` and `C[_]` is equivalent to `X =>> C[X]`.

This change will be phased in gradually. In Scala 3.0, either character can be used for wild cards. In 3.1, using _ as a wildcard will trigger a deprecation warning. After 3.1, _

2 This is most likely a temporary limitation. Subsequent 3.X releases may support this ability.

will be the placeholder character and ? will be the wildcard character, exclusively. Note that Java also uses ? as the wildcard for type parameters.

Recap and What's Next

Shapeless is the Scala project that has pushed the limits of Scala's type system and led to several improvements in Scala 3. It is part of the Typelevel ecosystem of advanced, powerful libraries. While many of the techniques we surveyed in this chapter require a bit more work to understand and use, they can also greatly reduce subtle bugs and code boilerplate, while still providing wide reuse and composability.

While you don't have to master all the intricacies of Scala's rich type system to use Scala effectively, the more you learn the details, the easier it will be to understand and use advanced code.

Next we'll explore more advanced topics in FP. In particular, we'll see how higher-kinded types enable the powerful concepts of *category theory*.

CHAPTER 18

Advanced Functional Programming

Let's return to functional programming (FP) and discuss some more advanced concepts. You can skip this chapter if you are a beginner, but come back to it if you hear people using terms like *algebraic data types*, *category theory*, *functors*, *monads*, *semigroups*, and *monoids*.

The goal here is to give you a sense of what these concepts are and why they are useful without getting bogged down in too much theory and notation.

Algebraic Data Types

There are two common uses for the acronym ADT, abstract data types and algebraic data types. Abstract data types are familiar from object-oriented programming (OOP). An example is `Seq`, an abstraction for all the sequential collections in the library.

In contrast, algebraic data types, for which we'll use ADT from now on, are algebraic in the sense that they obey well-defined mathematical properties. This is important because if we can prove properties about our types, it raises our confidence that they are bug free.

Sum Types Versus Product Types

Scala types divide into *sum types* and *product types*. The names *sum* and *product* are associated with the number of instances possible for a particular type.

Most of the classes you know are product types. For example, when you define a `case class` or a tuple, how many unique instances can you have? Consider this simple example:

```
case class Person(name: Name, age: Age)   // or (Name, Age) tuples
```

You can have as many instances of Person as the allowed values for Name times the allowed values for age. Let's say that Name encapsulates nonempty strings and disallows nonalphabetic characters (for some alphabet). There will effectively still be infinitely many values, but let's suppose it is *N*. Similarly, Age limits integer values, let's say between 0 and 130.

Because we can combine any Name value with any Age value to create a Person, the number of possible Person instances is $N \times 131$. For this reason, such types are called *product types*. Here *product* refers to the number of possible instances of the type. Replace Person with the (Name, Age) tuple type, and the same argument applies.

It's also the source of the name for Scala's Product type, a supertype of tuple types and case classes (see "Products, Case Classes, Tuples, and Functions" on page 316).

We learned in "Reference Versus Value Types" on page 252 that the single instance of Unit has the mysterious name (). This odd-looking name actually makes sense if we think of it as a zero-element tuple. Whereas a two-element tuple of (Name, Age) values can have $N \times 131$ values, a no-element tuple can have just one instance because it can't carry any state.

Consider what happens if we start with a two-element tuple, (Name, Age), and construct a new type by adding Unit:

```
type unitTimesTuple2 = (Unit, Name, Age)
```

How many instances does this type have? It's exactly the same as the number of types that (Name, Age) has. In product terms, it's as if we multiplied the number of (Name,Age) values by 1. This is the origin of the name Unit, just as one is the unit of integer multiplication.

The zero for product types is scala.Nothing. Combining Nothing with any other type to construct a new type must have zero instances because we don't have an instance to inhabit the Nothing field, just as $0 \times N = 0$ in integer multiplication.

Now consider sum types. Enumerations are an example of sum types. Recall this LoggingLevel example from "Trait Parameters" on page 282:

```
// src/main/scala/progscala3/traits/Logging.scala
package progscala3.traits.logging

enum LoggingLevel:
  case Debug, Info, Warn, Error, Fatal
```

There are exactly five LoggingLevel values, which are mutually exclusive. We can't have combinations of them.

Another way to implement a sum type is to use a sealed hierarchy of objects. `Option` is a good example, which looks like the following, ignoring most implementation details:[1]

```
sealed trait Option[+T]
case class Some[T](value: T) extends Option[T]
case object None extends Option[Nothing]
```

Even though the number of instances of `T` itself might vary, the ADT `Option` only has two allowed values, `Some[T]` and `None`. We saw in "Sealed Class Hierarchies and Enumerations" on page 62 that `Option` could also be implemented with an `enum`. This correspondence between enumerations and sealed hierarchies is not accidental.

Properties of Algebraic Data Types

In mathematics, algebra is defined by three aspects:

A set of objects
: Not to be confused with our OOP notion of objects. They could be numbers, `Persons`, or anything.

A set of operations
: How elements are combined to create new elements that are still a member of the set.

A set of laws
: These are rules that define the relationships between operations and objects. For example, for numbers, $(x \times (y \times z)) == ((x \times y) \times z)$ (associativity law).

Let's consider product types first. The informal arguments we made about the numbers of instances are more formally described by operations and laws. Consider again the product of `Unit` and `(Name, Age)`. Because we are counting instances, this product obeys commutativity and associativity. Using a pseudocode for this arithmetic:

```
Unit * (Name * Age) == (Name * Age) * Unit
Unit * (Name * Age) == (Unit * Name) * Age
```

This generalizes to combinations with non-`Unit` types:

```
Trees * (Name * Age) == (Name * Age) * Trees
Trees * (Name * Age) == (Trees * Name) * Age
```

Similarly, multiplication with zero (`Nothing`) trivially works because the counts are all zero:

1 `Option` could also be implemented as an `enum`, but Scala 3.0 uses the Scala 2.13 library implementation with a sealed trait, for example.

```
Nothing * (Name * Age) == (Name * Age) * Nothing
Nothing * (Name * Age) == (Nothing * Name) * Age
```

Turning to sum types, it's useful to recall that sets have unique members. Hence we could think of our allowed values as forming a set. That implies that adding `Nothing` to the set returns the same set. Adding `Unit` to the set creates a new set with all the original elements plus one, `Unit`. Similarly, adding a non-`Unit` type to the set yields a new set with all the original members and all the new members. The same algebraic laws apply that we expect for addition:

```
Nothing + (Some + None) == (Some + None) + Nothing
Unit    + (Some + None) == (Some + None) + Unit
FooBar  + (Some + None) == (Some + None) + FooBar
FooBar  + (Some + None) == (FooBar + Some) + None
```

Finally, there is even a distributive law of the form $x(a + b) = x \times a + x \times b$:

```
Name * (Some + None) = Name * Some + Name * None
```

I'll let you convince yourself that it actually works.

Final Thoughts on Algebraic Data Types

What does all this have to do with programming? This kind of precise reasoning encourages us to examine our types. Do they have precise meanings? Do they constrain the allowed values to just those that make sense? Do they compose to create new types with precise behaviors? Do they prevent ad hoc extensions that break the rules?

Category Theory

In one sense, the Scala community divides into two camps, those who embrace category theory, a branch of mathematics, as their foundation for programming, and those who rely less heavily on this approach. You could argue that this book falls into the latter grouping, but that's only because category theory is advanced and this book is aimed at a wide audience with different backgrounds.

This section introduces you to the basic ideas of category theory and to a few of the concepts most commonly used in FP. They are very powerful tools, but they require some effort to master.

In mathematics, category theory generalizes all aspects of mathematics in ways that enable reasoning about global properties. Hence, it offers deep and far-reaching abstractions. However, when it is applied to code, many developers struggle with the level of abstraction encountered. It's easier for most people to understand concrete examples, rather than abstractions. You may have felt this way while reading "Higher-Kinded Types" on page 388. Striking the right balance between concrete and abstract can be very hard in programming tasks. It's one of the most difficult trade-offs to get right. It is important to remember that even abstractions have costs, as well as many virtues.

Nevertheless, category theory now occupies a central place in advanced FP. Its use was pioneered in Haskell to solve various design problems and to push the envelope of functional thinking, especially around principled use of mutable versus immutable constructs.

If you are an advanced Scala developer, you should learn the rudiments of category theory as applied to programming, then decide whether or not it is right for your team and project. Unfortunately, I've seen situations where libraries written by talented proponents of category theory have failed in their organizations because the rest of the team found the libraries too difficult to understand and maintain. If you embrace category theory, make sure you consider the full life cycle of your code and the social aspects of software development.

Typelevel Cats is the most popular library in Scala for functional abstractions, including categories. It is a good vehicle for learning and experimentation. I'll use simplified category implementations in this chapter to minimize what's new to learn, overlooking details that Cats properly handles.

In a sense, this section continues where "Higher-Kinded Types" on page 388 left off. There, we discussed abstracting over parameterized types. For example, generalizing from specific type constructors like `Seq[A]` to all `M[A]`, where `M` is itself a parameter. We focused on folding there. Now we'll see how the functional combinators, like `map`, `flatMap`, `fold`, and so forth, are modeled by categories.

What Is a Category?

Let's start with the general definition of a category, which contains three entities that generalize what we said earlier about algebraic properties:

- A class consisting of a set of *objects*. These terms are used more generally than in OOP.
- A class of morphisms, also called *arrows*. A generalization of functions and written $f\colon A \to B$. Morphisms connect objects in the category. For each morphism, f, object A is the domain of f and object B is the codomain.
- A binary operation called *morphism composition* with the property that for $f\colon A \to B$ and $g\colon B \to C$, the composition $g \circ f\colon A \to C$ exists. Note that f is applied first, then g (i.e., right to left). It helps to say g follows f.

Two axioms are satisfied by morphism composition:

- Each object x has one and only one identity morphism, ID_x. That is, the domain and codomain are the same. Composition with identity has the following property: $f \circ ID_x = ID_x \circ f$.
- Associativity: for $f: A \rightarrow B$, $g: B \rightarrow C$, $h: C \rightarrow D$, $(h \circ g) \circ f = h \circ (g \circ f)$.

Now let's look at a few concepts from category theory that are used in software development (of the many categories known to mathematics): *functor* (and the special cases of *monad* and *arrow*), *semigroup*, *monoid*, and *applicative*.

Functor

`Functor` maps one category to another. In Scala terms, it abstracts the `map` operation. Let's define the abstraction and then implement it for two concrete types, `Seq` and `Option`, to understand these ideas:

```
// src/main/scala/progscala3/fp/categories/Functor.scala
package progscala3.fp.categories

trait Functor[F[_]]:                                                     ❶
  def map[A, B](fa: F[A])(f: A => B): F[B]                               ❷

object SeqF extends Functor[Seq]:                                        ❸
  def map[A, B](seq: Seq[A])(f: A => B): Seq[B] = seq map f

object OptionF extends Functor[Option]:
  def map[A, B](opt: Option[A])(f: A => B): Option[B] = opt map f
```

❶ The Scala library defines `map` as a method on most of the parameterized types. This implementation of `Functor` is a separate module. An instance of a parameterized type is passed as the first argument to `map`.

❷ The `map` parameters are the `F[A]` and a function `A => B`. An `F[B]` is returned.

❸ Define implementations for `Seq` and `Option`. For simplicity, just call the `map` methods on the collections!

Let's try these types:

```
// src/script/scala/progscala3/fp/categories/Functor.scala
scala> import progscala3.fp.categories.*

scala> val fid: Int => Double = i => 1.5 * i

scala> SeqF.map(Seq(1,2,3,4))(fid)
     | SeqF.map(Seq.empty[Int])(fid)
val res0: Seq[Double] = List(1.5, 3.0, 4.5, 6.0)
val res1: Seq[Double] = List()
```

```
scala> OptionF.map(Some(2))(fid)
     | OptionF.map(Option.empty[Int])(fid)
val res2: Option[Double] = Some(3.0)
val res3: Option[Double] = None
```

So why is the parameterized type with a `map` operation called a functor? Let's look at the `map` declaration again. We'll redefine `map` with `Seq` for simplicity (and rename it), then define a second version with the parameter lists switched:

```
def map1[A, B](seq: Seq[A])(f: A => B): Seq[B] = seq map f
def map2[A, B](f: A => B)(seq: Seq[A]): Seq[B] = seq map f
```

Now note the type of the new function returned when we use partial application on the second version:

```
scala> val fm = map2((i: Int) => i * 2.1)
val fm: Seq[Int] => Seq[Double] = Lambda...
```

So `map2` lifts a function `A => B` to a new function `Seq[A] => Seq[B]`! In general, `Functor.map` morphs `A => B`, for all types `A` and `B`, to `F[A] => F[B]` for any category `F`. Put another way, `Functor` allows us to apply a pure function (`f: A => B`) to a context (like a collection) holding one or more `A` values. We don't have to extract those values ourselves to apply `f`, then put the results into a new instance of the context.

In category theory terms, a `Functor` is a mapping between categories. It maps both the objects and the morphisms. For example, `List[Int]` and `List[String]` are two different categories, and so are all `Int`s and all `String`s.

`Functor` has two additional properties that fall out of the general properties and axioms for category theory:

- A functor `F` preserves identity; that is, the identity of the domain maps to the identity of the codomain.
- A functor `F` preserves composition: $F(f \circ g) = F(f) \circ F(g)$.

For an example of the first property, an empty list is the unit of lists; think of what happens when you concatenate it with another list. Mapping over an empty list always returns a new empty list, possibly with a different list element type.

Are the common and `Functor`-specific axioms satisfied? Let's try an example for associativity:

```
val f1: Int => Int = _ * 2
val f2: Int => Int = _ + 3
val f3: Int => Int = _ * 5

val l = List(1,2,3,4,5)

import progscala3.fp.categories.SeqF
```

```
val m12a = SeqF.map(SeqF.map(l)(f1))(f2)
val m23a = (seq: Seq[Int]) => SeqF.map(SeqF.map(seq)(f2))(f3)
assert(SeqF.map(m12a)(f3) == m23a(SeqF.map(l)(f1)))
```

Take the time to understand what each expression is doing. Verify that we are really checking associativity. Scala syntax is nice and concise, but sometimes the simple way of writing associativity in mathematics doesn't translate as concisely to code.

For a more extensive version of this example, see the code examples *src/test/scala/progscala3/fp/categories/FunctorPropertiesSuite.scala*, which use the property testing framework ScalaCheck to verify the properties for randomly generated collections.

It turns out that all the functions of `f: A => B` also form a category. Suppose I have a rich library of math functions for `Doubles` (`A =:= B` here). Can I use a functor to transform them into a set of functions for `BigDecimals`? Yes, but it can get tricky. See the code examples file `Functor2.scala` (in the same directory as `Functor.scala`) for several variations that support different scenarios.

Is it practical to have a separate abstraction for `map`? Abstractions with mathematically provable properties enable us to reason about program structure and behavior. For example, once we had a generalized abstraction for mapping, we could apply it to many different data structures, even functions. This reasoning power of category theory is why many people are so enthusiastic about it.

The Monad Endofunctor

If `Functor` is an abstraction for `map`, is there a corresponding abstraction for `flatMap`? Yes, `Monad`, which is named after the term *monas* used by the Pythagorean philosophers of ancient Greece, roughly translated as "the Divinity from which all other things are generated."

Technically, monads are specific kinds of functors, called *endofunctors*, that transform a category into itself.

Here is our definition of a `Monad` type, this time using a ScalaCheck property test:

```
// src/main/scala/progscala3/fp/categories/Monad.scala
package progscala3.fp.categories
import scala.annotation.targetName

trait Monad[M[_]]:                                              ❶
  def flatMap[A, B](fa: M[A])(f: A => M[B]): M[B]               ❷
  def unit[A](a: => A): M[A]                                    ❸

object SeqM extends Monad[Seq]:
  def flatMap[A, B](seq: Seq[A])(f: A => Seq[B]): Seq[B] =
    seq flatMap f
  def unit[A](a: => A): Seq[A] = Seq(a)
```

```
object OptionM extends Monad[Option]:
  def flatMap[A, B](opt: Option[A])(f: A => Option[B]):Option[B]=
    opt flatMap f
  def unit[A](a: => A): Option[A] = Option(a)
```

❶ Use `M[_]` for the type representing a data structure with monadic properties. As for `Functor`, it takes a single type parameter.

❷ Note that the function `f` passed to `flatMap` has the type `A => M[B]`, not `A => B`, as it was for `Functor`.

❸ `Monad` has a second function that takes a by-name value and returns it inside a `Monad` instance. In other words, it is a factory method for creating a `Monad`.

The name `unit` is conventional, but it works like our familiar `apply` methods. In fact, an abstraction with just `unit` is called `Applicative`, which is an abstraction over construction.

Let's try our `Monad` implementation. I'll explain the Monad Laws shortly.

```
// src/test/scala/progscala3/fp/categories/MonadPropertiesSuite.scala
package progscala3.fp.categories

import munit.ScalaCheckSuite
import org.scalacheck.*

class MonadPropertiesSuite extends ScalaCheckSuite:
  import Prop.forAll

  // Arbitrary function:
  val f1: Int => Seq[Int] = i => 0 until 10 by ((math.abs(i) % 10) + 1)

  import SeqM.*
  val unitInt: Int => Seq[Int] = (i:Int) => unit(i)
  val f2: Int => Seq[Int] = i => Seq(i+1)

  property("Monad law for unit works for Sequence Monads") {
    forAll { (i: Int) =>
      val seq: Seq[Int] = Seq(i)
      flatMap(unit(i))(f1)  == f1(i) &&
      flatMap(seq)(unitInt) == seq
    }
  }

  property("Monad law for function composition works for Sequence Monads") {
    forAll { (i: Int) =>
      val seq = Seq(i)
      flatMap(flatMap(seq)(f1))(f2) ==
              flatMap(seq)(x => flatMap(f1(x))(f2))
```

```
    }
  }
```

One way to describe `flatMap` is that it extracts an element of type `A` from the context on the left and binds it to a new kind of element in a new context instance. Like `map`, it removes the burden of knowing how to extract an element from `M[A]`. However, it looks like the function parameter now has the burden of knowing how to construct a new `M[B]`. Actually, this is not an issue because `unit` can be called to do this.

The Monad Laws are as follows.

`unit` behaves like an identity (so it's appropriately named):

```
flatMap(unit(x))(f) == f(x)      Where x is a value
flatMap(m)(unit) == m            Where m is a Monad instance
```

Like morphism composition for `Functor`, flat mapping with two functions in succession behaves the same as flat mapping over one function that is constructed from the two functions:

```
flatMap(flatMap(m)(f))(g) == flatMap(m)(x => flatMap(f(x))(g))
```

Monad's practical importance in software is the principled way it lets us wrap context information around a value, then propagate and evolve that context as the value evolves. Hence, it minimizes coupling between the values and contexts while the presence of the monad wrapper informs the reader of the context's existence.

This pattern is used frequently in Scala, inspired by the pioneer usage in Haskell. Examples include most of the collections, `Option`, and `Try`, which we discussed in "Options and Container Types" on page 232.

All are monadic because they support `flatMap` and unit construction with companion object `apply` methods. All can be used in `for` comprehensions. All allow us to sequence operations.

For example, the signature for `Try.flatMap` looks like this:

```
sealed abstract class Try[+A] {
  ...
  def flatMap[B](f: A => Try[B]): Try[B]
  ...
}
```

Now consider processing a sequence of steps where the previous outcome is fed into the next step, but we stop processing at the first failure:

```
// src/script/scala/progscala3/fp/categories/ForTriesSteps.scala
import scala.util.{ Try, Success, Failure }

type Step = Int => Try[Int]                                   ❶

val fail = RuntimeException("FAIL!")
```

```scala
val successfulSteps: Seq[Step] = List(                  // 2
  (i:Int) => Success(i + 5),
  (i:Int) => Success(i + 10),
  (i:Int) => Success(i + 25))
val partiallySuccessfulSteps: Seq[Step] = List(
  (i:Int) => Success(i + 5),
  (i:Int) => Failure(fail),
  (i:Int) => Success(i + 25))

def sumCounts(countSteps: Seq[Step]): Try[Int] =         // 3
  val zero: Try[Int] = Success(0)
  (countSteps foldLeft zero) {
    (sumTry, step) => sumTry.flatMap(i => step(i))
  }

assert(sumCounts(successfulSteps).equals(Success(40)))
assert(sumCounts(partiallySuccessfulSteps).equals(Failure(fail)))
```

1. Type alias for step functions.

2. Two sequences of steps, one successful, one with a failed step.

3. A method that works through a step sequence, passing the result of a previous step to the next step.

The logic of `sumCounts` handles sequencing, while `flatMap` handles the `Try` containers. Note that subtypes are actually returned, either `Success` or `Failure`.

The use of monads was pioneered in Haskell,[2] where functional purity is more strongly emphasized. For example, monads are used to compartmentalize input and output (I/O) from pure code. The `IO Monad` handles this separation of concerns. Also, because it appears in the type signature of functions that use it, the reader and compiler know that the function isn't pure. Similarly, `Reader` and `Writer` monads have been defined in many languages for the same purposes. The more general term now used is *effects* for these applications of monads for state management.

Cats Effect is a Scala effects library. FS2 and Zio use effects for robust, distributed computation.

A generalization of monad is *arrow*. Whereas monad lifts a value into a context (i.e., the function passed to `flatMap` is `A => M[B]`), an arrow lifts a function into a context, `(A => B) => C[A => B]`. Composition of arrows makes it possible to reason about sequences of processing steps (i.e., `A => B`, then `B => C`, etc.) in a referentially

2 Philip Wadler's home page has many of his pioneering papers on monad theory and applications.

transparent way, outside the context of actual use. In contrast, a function passed to `flatMap` is explicitly aware of its context, as expressed in the return type!

The Semigroup and Monoid Categories

We first encountered `Semigroup` and `Monoid` in "Scala 3 Type Classes" on page 143. `Semigroup` is the abstraction of addition, a category where there is a single morphism that is associative. `Monoid` is a `Semigroup` with an identity value, which we called `unit` previously. The obvious examples are numbers with addition, where 0 is the identity, and numbers with multiplication, where 1 is the identity.

It turns out that a lot of computation problems can be framed as monoids, especially in data analytics. This makes it possible to write infrastructure that knows how to add things, and specific problems are solved by defining a monoid that implements the computation.

For a great talk on this idea, see the Strange Loop 2013 talk by Avi Bryant called *Add All the Things!*. Avi discusses how Twitter and Stripe used monoids to solve many large-scale data problems in a generic and reusable way.

We saw an implementation of `Monoid` already in "Scala 3 Type Classes" on page 143, with examples. Let's see another example to refresh our memories.

A utility I often want is one that will merge two maps. Where the keys are unique in each one, it performs the union of the key-value pairs, but where the keys appear in both maps, it merges the values in some way. Merging of maps and the values for a given key are nicely modeled with monoids. The following `MapMergeMonoid` is defined to expect a `Monoid` instance for the values. The code is concise, yet general-purpose and flexible:

```
// src/main/scala/progscala3/fp/categories/MapMerge.scala
package progscala3.fp.categories
import progscala3.contexts.typeclass.Monoid

given MapMergeMonoid[K, V : Monoid]: Monoid[Map[K, V]] with       ❶
  def unit: Map[K, V] = Map.empty
  extension (map1: Map[K, V]) def combine(map2: Map[K, V]): Map[K, V] =
    val kmon = summon[Monoid[V]]
    (map1.keySet union map2.keySet).map { k =>
      val v1 = map1.getOrElse(k, kmon.unit)
      val v2 = map2.getOrElse(k, kmon.unit)
      k -> (v1 combine v2)
    }.toMap
```

❶ Require a `Monoid` for the values too.

We map over the union of the key sets, and for each key, extract the corresponding value or use the `Monoid[V].unit` as the default. Then all we have to do is combine the two values and return a new key-value pair. Finally, we convert back to a map.

Let's merge some maps. We'll use the given `StringMonoid` and `IntMonoid` we saw in Chapter 5:

```
// src/script/scala/progscala3/fp/categories/MapMerge.scala

scala> import progscala3.fp.categories.MapMergeMonoid                    ❶
     | import progscala3.contexts.typeclass.given

scala> val map1i = Map("one" -> 1, "two" -> 2)
     | val map2i = Map("two" -> 2, "three" -> 3)
     | val map1s = map1i.map{ (k,v) => (k, v.toString) }
     | val map2s = map2i.map{ (k,v) => (k, v.toString) }

scala> map1i.combine(map2i)
     | map1s.combine(map2s)
     | map1s <+> map2s         // Recall this operator is defined too.
val res0: Map[String, Int] = Map(one -> 1, two -> 4, three -> 3)
val res1: Map[String, String] = Map(one -> 1, two -> 22, three -> 3)
val res2: Map[String, String] = Map(one -> 1, two -> 22, three -> 3)
```

❶ Recall that when importing a named given, you just specify the name, `MapMergeMonoid` in this case, but you use `given` when importing all givens in a package.

Note the differences for key `two` when we merge the integer `2` versus a string.

Recap and What's Next

I hope this brief introduction to more advanced concepts, especially category theory, will help when you encounter these concepts in the Scala community. They are powerful, if also challenging to master.

Scala's standard library uses object-oriented conventions to add functions like `map` and `flatMap` as methods to many types, rather than implementing them as separate utilities. We learned in Chapter 8 that `flatMap`, along with `map` and `filter`, make `for` comprehensions so concise. Now we see that `flatMap` comes from monad, giving us monadic behaviors.

Unfortunately, categories have been steeped in mystery because of the mathematical formalism and their abstract names. However, they are abstractions of familiar concepts, with powerful implications for program correctness, reasoning, concision, and expressiveness.

I consider functor, monad, arrow, applicative, and monoid examples of Functional Design Patterns. The term *Design Patterns* has a bad connotation for some functional programmers, but really this confuses specific examples of patterns with the concept of patterns. Some of the classic OOP patterns are still valuable in FP or OOP. We've discussed `Observer` several times already, which is widely used in asynchronous toolkits. Other classic OOP patterns are less useful today. Now we have patterns from category theory too. They fit the definition of design patterns as reusable constructs within a context, where the context of use helps you decide when a pattern is useful and when it isn't.

For an exploration of category theory in Scala using Cats, see [Welsh2017]. The recommendations in "Recap and What's Next" on page 224 provide additional FP content. For a general introduction to category theory for programmers, see [Milewski2019].

The next chapter explores another practical subject, the important challenge of writing scalable, concurrent, and distributed software with Scala.

CHAPTER 19

Tools for Concurrency

Nearly twenty years ago, in the early 2000s, we hit the end of Moore's Law for the performance growth of single-core CPUs. We've continued to scale performance through increasing numbers of cores and servers, trading vertical scaling for horizontal scaling. The *multicore problem* emerged as developers struggled to write robust applications that leverage concurrency across CPU cores and across a cluster of machines.

Concurrency isn't easy because it usually requires coordinated access to shared, mutable state. Low-level libraries provide locks, mutexes, and semaphores for use on the same machine, while other tools enable distribution across a cluster. Failure to properly coordinate access to mutable state often leads to state corruption, race conditions, and lock contention. For cluster computing, you need to add networking libraries and coding idioms that are efficient and easy to use.

These problems drove interest in FP when we learned that embracing immutability and purity largely bypasses the problems of multithreaded programming. We also saw a renaissance of other mature approaches to concurrency, like the actor model, which lend themselves to cluster-wide distribution of work.

This chapter explores concurrency tools for Scala. You can certainly use any multithreading API, external message queues, etc. We'll just discuss Scala-specific tools, starting with an API for a very old approach, using separate operating system processes.

The scala.sys.process Package

Sometimes it's sufficient to coordinate state through database transactions, message queues, or simple pipes from one operating system process to another.

Using operating system processes, like the Linux/Unix shell tools, is straightforward with the `scala.sys.process` package. Here's a REPL session that demonstrates some of the features. Note that a `bash`-compatible shell is used for the commands:

```
scala> import scala.sys.process.*
     | import java.net.URL
     | import java.io.File

scala> "ls -l src".!
total 0
drwxr-xr-x  6 deanwampler  staff  192 Jan  2 10:53 main
drwxr-xr-x  4 deanwampler  staff  128 Jan 13 14:34 script
drwxr-xr-x  3 deanwampler  staff   96 Jan  2 10:53 test
val res0: Int = 0

scala> Seq("ls", "-l", "src").!!
val res1: String = "total 0
drwxr-xr-x  6 deanwampler  staff  192 Jan  2 10:53 main
drwxr-xr-x  4 deanwampler  staff  128 Jan 13 14:34 script
drwxr-xr-x  3 deanwampler  staff   96 Jan  2 10:53 test
"
```

The single `!` method prints the output and returns the command's exit status, 0 in this case. The double `!!` method returns the output as a string.

We can also connect processes. Consider the following methods:

```
// src/main/scala/progscala3/concurrency/process/Process.scala

scala> def findURL(url: String, filter: String) =
     |   URL(url) #> s"grep $filter" #>> File(s"$filter.txt")
     |
     | def countLines(fileName: String) =
     |   s"ls -1 $fileName" #&& s"wc -l $fileName"
def findURL(url: String, filter: String): scala.sys.process.ProcessBuilder
def countLines(fileName: String): scala.sys.process.ProcessBuilder
```

The `findURL` method builds a process sequence that opens a URL, redirects the output to the command `grep $filter`, where `filter` is a parameter to the method, and finally appends the output to a file. It doesn't overwrite the file because we used `#>>`. The `countLines` method runs `ls -l` on a file. If it exists, then it also counts the lines.

The `#>` method overwrites a file or pipes into `stdin` for a second process. The `#>>` method appends to a file. The `#&&` method only runs the process to its right if the process to its left succeeds, meaning that the lefthand process returns exit code zero.

Both methods return a `scala.sys.process.ProcessBuilder`. They don't actually run the commands. For that we need to invoke their `!` or `!!` method:

```
scala> findURL("https://www.scala-lang.org", "scala").!
val res2: Int = 0
```

```
scala> countLines("scala.txt").!
-rw-r--r--  1 deanwampler  staff  12236 Jan 13 15:50 scala.txt
     128 scala.txt
val res3: Int = 0
```

Run the two commands again and you'll see that the file size doubles because we append text to it each time `findURL` executes.

When it's an appropriate design solution, small, synchronous processes can be implemented in Scala or any other language, then glued together using the `process` package API.

For an alternative to `sys.process`, see Li Haoyi's Ammonite.

Futures

For many needs, process boundaries are too coarse-grained. We need easy-to-use concurrency APIs that use multithreading within a single process.

Suppose you have units of work that you want to run asynchronously, so you don't block while they are running. They might need to do I/O, for example. The simplest mechanism is `scala.concurrent.Future`.

When you construct a `Future`, control returns immediately to the caller, but the value is not guaranteed to be available yet. Instead, the `Future` instance is a handle to retrieve a result that will be available eventually. You can continue doing other work until the future completes, either successfully or unsuccessfully. There are different ways to handle this completion.[1]

We saw a simple example in "A Taste of Futures" on page 42. An instance of `scala.concurrent.ExecutionContext` is required to manage and run futures. We used the default value, `ExecutionContext.global`, which manages a thread pool for running tasks inside `Futures`, without locking up a thread per future. As users, we don't need to care about how our asynchronous processing is executed, except for special circumstances like performance tuning, when we might implement our own `ExecutionContext`.

To explore `Futures`, first consider the case where we need to do 10 things in parallel, then combine the results:

```
// src/main/scala/progscala3/concurrency/futures/FutureFold.scala
package progscala3.concurrency.futures

import scala.concurrent.{Await, Future}
```

1 `scala.concurrent.Promise` is also useful for working with `Futures`, but I won't discuss it further here.

```scala
import scala.concurrent.duration.*
import scala.concurrent.ExecutionContext.Implicits.global

@main def TryFutureFold =
  var accumulator = ""                                                    // 1
  def update(s:String) = accumulator.synchronized { accumulator += s}

  val futures = (0 to 9) map {                                            // 2
    i => Future {
      val s = i.toString                                                  // 3
      update(s)
      s
    }
  }

  val f = Future.reduceLeft(futures)((s1, s2) => s1 + s2)                 // 4
  println(f)

  val n = Await.result(f, 2.seconds)                                      // 5
  assert(n == "0123456789")

  println(s"accumulator: $accumulator")
```

1. To see asynchrony at work, append to a string, but do so using the low-level `synchronized` primitive for thread safety.

2. Create 10 asynchronous futures, each performing some work.

3. `Future.apply` takes two parameter lists. The first has a single, by-name `body` to execute asynchronously. The second list has the implicit `ExecutionContext`. We're allowing the `global` implicit value to be used. The body converts the integer to a string `s`, appends it to `accumulator`, and returns `s`. The type of `futures` is `IndexedSeq[Future[String]]`. In this contrived example, the `Futures` complete very quickly.

4. Reduce the sequence of `Future` instances into a single `Future[String]` by concatenating the strings. Note that we don't need to extract the strings from the futures because `reduceLeft` handles this for us. Afterward, it constructs a new `Future` for us.

5. Block until the `Future f` completes using `scala.concurrent.Await`. The `scala.concurrent.duration.Duration` parameter says to wait up to two seconds, timing out if the future hasn't completed by then. Using `Await` is the preferred way to block the current thread when you need to wait for a `Future` to complete. Adding a time-out prevents deadlock!

Because the futures are executed concurrently, the list of integers in the `accumulator` will not be in numerical order; for example, `0123574896`, `1025438967`, and `0145678932` are the outputs of three of my runs. However, because `fold` walks through the `Futures` in the same order in which they were constructed, the string it produces always has the digits in strict numerical order, `0123456789`.

`Future.fold` and similar methods execute asynchronously themselves; they return a new `Future`. Our example only blocks when we called `Await.result`.

> Well, not exactly. We did use `accumulator.synchronized` to update the string safely. Sometimes this low-level construct is all you need for safely updating a value, but be aware of how it introduces blocking into your application!

While the required time-out passed to `Await.result` avoids the potential of *deadlocks* in production, we still block the thread! Instead of using `Await`, let's register a callback that will be invoked when the `Future` completes, so our current thread isn't blocked. Here is a simple example:

```
// src/main/scala/progscala3/concurrency/futures/FutureCallbacks.scala
package progscala3.concurrency.futures

import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util.{Try, Success, Failure}

case class ThatsOdd(i: Int) extends RuntimeException(                  ❶
  s"odd $i received!")

val doComplete: Try[String] => Unit =                                  ❷
  case s: Success[String] => println(s)                                ❸
  case f: Failure[String] => println(f)

@main def TryFuturesCallbacks =
  val futures = (0 to 9).map {                                         ❹
    case i if i % 2 == 0 => Future.successful(i.toString)
    case i => Future.failed(ThatsOdd(i))
  }
  futures.map(_.onComplete(doComplete))                    ❺
```

❶ An exception we'll throw for odd integers.

❷ Define a callback handler for both successful and failed results. Its type must be a function, `Try[A] => B`, because the callback will be passed a `Try[A]`, where `A` is `String` here, encapsulating success or failure. The function's return type can be anything, but note that `onComplete` returns `Unit`; nothing can be returned from the handler, since it runs asynchronously. In real application, like a web server, a response could be sent to the caller at this point.

❸ If the `Future` succeeds, the `Success` clause will match. Otherwise the `Failure` will match. We just print either result.

❹ Create the `Futures` where odd integers are immediately completed as failures, while even integers are successes. We use two methods on the `Future` companion object for this purpose.

❺ Traverse over the `futures` to attach the callback, which will be called immediately since our `Futures` have already completed by this point.

Running the `TryFuturesCallbacks` produces output like the following, where the order will vary from run to run:

```
Failure(progscala3.concurrency.futures.ThatsOdd: odd 1 received!)
Success(2)
Success(0)
Success(4)
Failure(progscala3.concurrency.futures.ThatsOdd: odd 5 received!)
Failure(progscala3.concurrency.futures.ThatsOdd: odd 7 received!)
Failure(progscala3.concurrency.futures.ThatsOdd: odd 9 received!)
Success(8)
Failure(progscala3.concurrency.futures.ThatsOdd: odd 3 received!)
Success(6)
```

`Future` is monadic like `Option`, `Try`, `Either`, and the collections. We can use them in `for` comprehensions and manipulate the results with our combinator friends, `map`, `flatMap`, `filter`, and so forth.

3 When working with graphs of futures, use of callbacks can get complicated quickly. For Scala 2, the separate `scala-async` project provides a more concise DSL for working with futures. At the time of this writing, this library hasn't been ported to Scala 3, but it may be available by the time you read this section.

Robust, Scalable Concurrency with Actors

The *actor model* provides a reasonably intuitive and robust way to build, distributed applications with evolving state, where both the state and the computation can be distributed and parallelized. Fundamentally, an actor is an object that receives messages and takes action on those messages, one at a time and without preemption, thereby ensuring thread safety when local state is modified. The order in which messages arrive is unimportant in some actor systems, but not all. An actor might process a message itself, or it might forward the message or send a new message to another actor. An actor might create new actors as part of handling a message. A message might trigger the actor to change how it handles future messages, in effect implementing a state transition in a state machine.

Unlike traditional object systems that use method calls, actor message sending is usually asynchronous, so the global order of actions is nondeterministic. Like traditional objects, an actor may control some state that it evolves in response to messages. A well-designed actor system will prevent any other code from accessing and mutating this state directly.

These features allow actors to run in parallel, even across a cluster. They provide a *principled* approach to managing global state, mostly avoiding all the problems of low-level, multithreaded concurrency.

The two most important, production-ready implementations of the actor model are the Erlang implementation and Akka, which drew its inspiration from Erlang. Both implement an important innovation over the core actor model, a robust model of error handling and recovery, based on the idea of actor supervision.

Akka: Actors for Scala

In the *Actor Model of Concurrency*, independent software entities called *actors* share no mutable state access with each other. Instead, they communicate by exchanging messages, and each actor modifies its local state in a thread-safe way. By eliminating the need to synchronize access to shared, mutable state, it is far easier to write robust, concurrent applications without using tedious and error-prone synchronization primitives.

Akka is the popular concurrency library for Scala and Java that implements the actor model. Let's work through an example. You might find the Akka Scaladoc useful as we go.

Not only are actors created to do the routine work of the system, *supervisors* are created to watch the life cycle of one or more actors. Should an actor fail, perhaps because an exception is thrown, the supervisor follows a strategy for recovery that can include restarting, shutting down, ignoring the error, or delegating to its own supervisor for handling. See the Akka supervision documentation for details.

This architecture cleanly separates error-handling logic from normal processing. It enables an architecture-wide strategy for error handling. Most importantly, it promotes a principle of "let it crash."

In most software, it is common to mix error-handling logic with normal processing code, resulting in a complicated mess, which often fails to implement a complete, comprehensive strategy. Inevitably, some production scenarios will trigger a failed recovery that leaves the system in an inconsistent state. When the inevitable crash happens, service is compromised and diagnosing the root cause of the problem proves difficult. Akka actors cleanly separate error handling from normal processing.

The example we'll use simulates a client interface invoking a service, which delegates tasks to workers. This client interface (and location of the `main` entry point) is called `ServiceClient`. It passes user commands to a single `ServerActor`, which in turn delegates work to several `WorkerActors`, so that it never blocks. Each worker simulates a *sharded* data store. It maintains a map of keys (`Longs`) and values (`Strings`), and it supports CRUD (create, read, update, and delete) semantics. `ServiceClient` also provides a simple command-line interface to the user.

3 The following example uses the newer *typed actor* version of Akka. The classic API, where messages are effectively untyped (`Any`) will be difficult to use with the new `Matchable` feature in Scala 3 because pattern matching on `Any` is effectively deprecated. If your applications use the classic Akka API, plan to convert them to the new API!

Before walking through `ServiceClient`, let's look at `Messages`, which defines all the messages exchanged between the actors:

```
// src/main/scala/progscala3/concurrency/akka/Messages.scala
package progscala3.concurrency.akka
import scala.util.Try
import akka.actor.typed.ActorRef

object Messages:                                                      ❶
  sealed trait Request:                                               ❷
    val replyTo: ActorRef[Response]

  enum AdminRequest extends Request:                                  ❸
    case Start(numberOfWorkers: Int = 1, replyTo: ActorRef[Response])
    case Crash(whichOne: Int, replyTo: ActorRef[Response])
    case Dump(whichOne: Int, replyTo: ActorRef[Response])
    case DumpAll(replyTo: ActorRef[Response])

  enum CRUDRequest extends Request:                                   ❹
    val key: Long
    case Create(key: Long, value: String, replyTo: ActorRef[Response])
    case Read(key: Long, replyTo: ActorRef[Response])
    case Update(key: Long, value: String, replyTo: ActorRef[Response])
    case Delete(key: Long, replyTo: ActorRef[Response])

  case class Response(                                                ❺
    result: Try[String], replyTo: ActorRef[Response])
```

❶ Use a `Messages` object to hold all the message types.

❷ A common supertype for two groups of messages. All the messages will carry a reference to the actor to which replies should be sent. (This will always be `ServiceClient`.)

❸ A group of messages for administration purposes: start processing (where the number of workers is specified), simulate a crash of a worker, dump the current state of a worker, and dump the states of all workers.

❹ Enumeration for all CRUD requests: create, read, update (or create), and delete records. All of them have a record key.

❺ Wrap responses in a common message. A `Try` wraps the result of the corresponding request, indicating either success or failure.

`ServiceClient` constructs the `akka.actor.typed.ActorSystem`, which controls everything, and one instance of `ServerActor`. The file is quite long because of all the code for the command-line interface. I will elide most of it here, as it is less relevant to the discussion. See the full listing in the code examples. Here's the first part of `ServiceClient`:

```
// src/main/scala/progscala3/concurrency/akka/ServiceClient.scala
package progscala3.concurrency.akka

import akka.actor.typed.scaladsl.Behaviors
import akka.actor.typed.{ActorRef, ActorSystem, Behavior}
import java.lang.NumberFormatException as NFE
import scala.util.{Try, Success, Failure}

object ServiceClient:                                              ❶
  import Messages.*

  private var server: ActorRef[Request] = null                     ❷
  private var client: ActorRef[Response] = null

  def main(params: Array[String]): Unit =
    ActorSystem(ServiceClient(), "ServiceClient")                  ❸
    processUserInput()                                             ❹

  def apply(): Behavior[Response] =                                ❺
    Behaviors.setup { context =>
      client = context.self                                        ❻
      server = context.spawn(ServerActor(), "ServerActor")
      assert(client != null && server != null)
      val numberOfWorkers =
        context.system.settings.config.getInt("server.number-workers")
      server ! AdminRequest.Start(numberOfWorkers, client)         ❼
      Behaviors.receiveMessage { message =>                        ❽
        message match
```

```
          case Response(Success(s), _) =>
            printResult(s"$s\n")
            Behaviors.same
          case Response(Failure(th), _) =>
            printResult(s"ERROR! $th")
            Behaviors.same
      }
    }

  protected def printResult(message: String) =                          ❾
    println(s"<< $message")
    prompt()
  protected def prompt() = print(">> ")
```

❶ The client is an object with the `main` entry point.

❷ After initializing the server and client `ActorRef`s, these won't be `null`. An `akka.actor.typed.ActorRef` is a handle for the actual actor. The latter can be restarted, while the `ActorRef` remains durable. Hence, all interaction with an actor goes through the `ActorRef`.

❸ The single `ActorSystem` is passed to the top-level `Behavior`, which encapsulates the message handling logic of the client actor.[2] In idiomatic Akka, this `Behavior` is returned by calling the object's `apply`.

❹ After setting up the actors, process user command-line input. This method, which is very long, is not shown.

❺ Create the message handling `Behavior` for the client. Note that it is typed to accept only `Messages.Response` objects, meaning it doesn't handle `Messages.Request` messages, even though it sends them!

❻ Remember the client's `ActorRef` using `context.self`. Then use `context.spawn` and `ServerActor.apply` to construct the server actor.

❼ Send the `Start` message to the `ServerActor` to begin processing. Determine from the following configuration how many workers to use.

❽ Each match clause ends with `Behaviors.same`, which means the next message received will be handled exactly the same way. You can change the handling logic between messages to create a state machine.

2 If you know the classic API, `Behavior` plus an `ActorContext` replaces the `Actor` type.

❾ Here is where all results are printed to the console, followed by the prompt for the next command.

Akka uses the Lightbend Config library, which allows us to configure many aspects of the application in text files, rather than hardcoding information in the code:

```
// src/main/resources/application.conf
akka {                                                    ❶
  loggers  = [akka.event.slf4j.Slf4jLogger]               ❷
  loglevel = debug

  actor {                                                 ❸
    debug {
      unhandled = on
      lifecycle = on
    }
  }
}

server {                                                  ❹
  number-workers = 5
}
```

❶ Configure properties for the Akka system as a whole.

❷ Configure logging. The `sbt` build includes the `akka-slf4j` module required. There is a corresponding `logback.xml` in the same directory. By default, all debug and higher messages are logged.

❸ Configure properties for every actor. In this case, enable debug logging of occurrences when an actor receives a message it doesn't handle and any life cycle events.

❹ The `ServerActor` instance will be given the identifier `server`. Here is where properties for each kind of actor are specified, in this case the number of workers to create.

Next, let's look at `ServerActor`, sections at a time, omitting some details:

```
// src/main/scala/progscala3/concurrency/akka/ServerActor.scala
package progscala3.concurrency.akka
import akka.actor.typed.scaladsl.Behaviors
import akka.actor.typed.{ActorRef, Behavior, SupervisorStrategy}
import scala.util.Success

object ServerActor:
  import Messages.*

  var workers = Vector.empty[ActorRef[Request]]           ❶
```

```scala
  def apply(): Behavior[Request | Response] =                          // 2
    Behaviors.supervise(processRequests)
      .onFailure[RuntimeException](SupervisorStrategy.restart)
```

❶ Keep track of the workers. Note that they are expected to only receive and handle `Request`s.

❷ The `apply` method returns a behavior than can process both `Request` messages from `ServiceClient` and `Response` messages from the workers. This is a nice use of union types. A custom `akka.actor.SupervisorStrategy` is also defined, overriding the default strategy, "let it crash."

Continuing with the definition of `ServerActor`:

```scala
  protected def processRequests: Behavior[Request | Response] =
    Behaviors.receive { (context, message) =>                          // 1
      message match
        case AdminRequest.Start(numberOfWorkers, replyTo) =>
          workers = (1 to numberOfWorkers).toVector.map { i =>
            val name = s"worker-$i"
            context.spawn(WorkerActor(context.self, name), name)       // 2
          }
          replyTo ! Response(
            Success(s"Starting $numberOfWorkers workers"), replyTo)
          Behaviors.same
        case c @ AdminRequest.Crash(n, replyTo) =>                     // 3
          val n2 = n % workers.size
          workers(n2) ! c
          replyTo ! Response(
            Success(s"Crashed worker $n2 (from n=$n)"), replyTo)
          Behaviors.same
        case AdminRequest.DumpAll(replyTo) =>                          // 4
          (0 until workers.length).foreach { n =>
            workers(n) ! AdminRequest.DumpAll(replyTo)
          }
          Behaviors.same
        case AdminRequest.Dump(n, replyTo) =>
          val n2 = n % workers.size
          workers(n2) ! AdminRequest.Dump(n2, replyTo)
          Behaviors.same
        case request: CRUDRequest =>                                   // 5
          val key = request.key.toInt
          val index = key % workers.size  // in case key >= workers.size
          workers(index) ! request
          Behaviors.same
        case resp @ Response(_, replyTo) =>                            // 6
          replyTo ! resp
          Behaviors.same
    }
```

```
  end processRequests
end ServerActor
```

❶ The message handler can be defined by implementing one of two methods. Here we use `receive`, which takes a `context`, needed to spawn workers, and a `message` argument. Contrast with the simpler `receiveMessage` method implemented in `ServiceClient`.

❷ Spawn the workers and reply to the client (`replyTo`) with a success message, which is optimistic, since it doesn't confirm success first!

❸ Deliberately crash a worker by forwarding `c` to it. It will be restarted (see the following example).

❹ Most of the clauses look like this one; forward or send one or more messages to workers. Responses from them are handled by the `Response` clause at the end.

❺ All the CRUD requests are simply forwarded to the correct worker.

❻ Handle responses from workers and forward to the `replyTo` actor, which is always `ServiceClient`.

Finally, here is `WorkerActor`, where many details are similar to what we saw in the preceding example:

```
// src/main/scala/progscala3/concurrency/akka/WorkerActor.scala
package progscala3.concurrency.akka
import scala.util.{Try, Success, Failure}
import akka.actor.typed.scaladsl.Behaviors
import akka.actor.typed.{ActorRef, Behavior, SupervisorStrategy}
import Messages.*

object WorkerActor:

  def apply(
      server: ActorRef[Request | Response],
      name: String): Behavior[Request] =
    val datastore = collection.mutable.Map.empty[Long,String]            ❶
    def processRequests(                                                 ❷
        server: ActorRef[Request | Response],
        name: String): Behavior[Request] =
      Behaviors.receiveMessage {
        case CRUDRequest.Create(key, value, replyTo) =>                  ❸
          datastore += key -> value
          server ! Response(Success(s"$name: $key -> $value added"), replyTo)
          Behaviors.same
        case CRUDRequest.Read(key, replyTo) =>
          server ! Response(
            Try(s"$name: key = $key, ${datastore(key)} found"), replyTo)
```

```
          Behaviors.same
        case CRUDRequest.Update(key, value, replyTo) =>
          datastore += key -> value
          server ! Response(Success(s"$name: $key -> $value updated"), replyTo)
          Behaviors.same
        case CRUDRequest.Delete(key, replyTo) =>
          datastore -= key
          server ! Response(Success(s"$name: $key deleted"), replyTo)
          Behaviors.same
        case AdminRequest.Crash(n, replyTo) =>                          ❹
          val ex = CrashException(name)
          server ! Response(Failure(ex), replyTo)
          throw ex
          Behaviors.stopped
        case AdminRequest.Dump(n, replyTo) =>
          server ! Response(
            Success(s"$name: Dump($n): datastore = $datastore"), replyTo)
          Behaviors.same
        case AdminRequest.DumpAll(replyTo) =>
          server ! Response(
            Success(s"$name: DumpAll: datastore = $datastore"), replyTo)
          Behaviors.same
        case req: Request =>                                            ❺
          server ! Response(
            Failure(UnexpectedRequestException(req)),req.replyTo)
          Behaviors.same
      }
    Behaviors.supervise(processRequests(server, name))                 ❻
      .onFailure[RuntimeException](SupervisorStrategy.restart)
  end apply

  case class CrashException(name: String)
    extends RuntimeException(s"$name: forced to crash!")
  case class UnexpectedRequestException(request: Request)
    extends RuntimeException(s"Did not expect to receive $request!")
```

❶ Keep a mutable map of key-value pairs for this worker. Because the `Behavior` handler is thread-safe (enforced by Akka itself) and because this mutable state is private to the actor, it is safe to use a mutable object.

❷ This nested method will return the `Behavior[Request]` needed. It will be wrapped in a supervisor strategy ahead.

❸ The CRUD operation that adds a new key-value pair to the map and then sends a `Response` to the `server`. The other CRUD operations are very similar.

❹ Crash the actor by throwing a `CrashException`. It will be restarted automatically due to the supervisor strategy.

❺ This clause is undesirable, but workers don't handle all the `Request` messages. The message hierarchy could be fine-tuned to prevent the need for this clause.

❻ Restart the actor if it fails, such as the deliberate crashes.

After all this buildup, let us now run the application using `sbt`:

```
runMain progscala3.concurrency.akka.ServiceClient
```

Enter `h` or `help` to see the list of commands and try several. Try `dump` to see the actors and their contents, then `crash` one of them and use `dump` again. The actor should restart. Use `q` to exit. There is also a file of commands that can run through the program and copy and paste the contents of *misc/run-akka-input.txt* to the `>>` prompt.

You might be concerned that the `ServerActor`'s list of workers would become invalid when an actor crashes. This is why all access to an actor goes through the `ActorRef` handle, and direct access to the underlying actor is prevented. `ActorRef`s are very stable. When a supervisor restarts an actor, it resets the `ActorRef` to point to the new instance. If the actor is not restarted or resumed, all messages sent to the corresponding `ActorRef` are forwarded to the `ActorSystem.deadLetters`, which is the place where messages from dead actors go to die.

Actors: Final Thoughts

Our application demonstrates a common pattern for handling a high volume of concurrent input traffic, delegating results to asynchronous workers, then returning the results (or just printing them in this case).

We only scratched the surface of what Akka offers. Still, you now have a sense for how a typical, nontrivial Akka application works. Akka has excellent documentation at *https://akka.io*. [Roestenburg2014] is one of several books on Akka.

Akka actors are lightweight. You can easily create millions of them in a single, large JVM instance. Keeping track of that many autonomous actors would be a challenge, but if most of them are stateless workers, they can be managed. Akka also supports clustering across thousands of nodes for very high scalability and availability requirements.

The actor model is criticized for not being an FP model. Message sending is used instead of function calls with returned values. Effectively, everything is done through side effects! Furthermore, the model embraces mutable state when useful, as in our example, although it is encapsulated and handled in a thread-safe way.

> **Actors and OOP**
>
> The actor model is closely aligned with the vision of OOP espoused by Alan Kay, the coinventor of Smalltalk and the person who probably coined the term *object-oriented programming*. He argued that objects should be autonomous encapsulations of state, which only communicate through message passing. In fact, invoking a method in Smalltalk was called "sending a message."

Finally, the actor model is one approach to large-scale, highly available, event-driven applications. A few other concurrency and distribution models and corresponding libraries are worth investigating.

Stream Processing

If you think about how our collections methods work, they are like stream processors, working through the elements of a collection to map, filter, fold, etc. This idea generalizes to a more sophisticated model of distributed concurrent event or data stream processing. For many problems, the streaming model is more intuitive than the actor model, which is slightly biased toward state machines.

At very large scales, where data sharding over a cluster is essential to meet the demands for scalability, resiliency, and performance, the following tools have become the most popular for big-data stream processing:

- Apache Spark started as a batch-oriented tool for processing massive data sets. Now Spark also offers a streaming API. We'll revisit Spark in "Scala for Big Data: Apache Spark" on page 459.
- Apache Flink is an alternative to Spark that has always emphasized streaming more than batch-mode processing. It appears to be gaining popularity over Spark for stream processing.
- Apache Kafka is a distributed streaming platform. It provides durable stream storage, while applications can process the streams any way they want, including with Spark, Flink, and Kafka's own streaming API.

All three systems provide Scala APIs. Spark and Kafka are written in Scala. You can think of Kafka as the streaming analog of storage, while Spark, Flink, and Kafka streams are the processors that process the stored streams.

Not all applications are extreme scale, however. The streaming metaphor is still useful, and not just for data processing. For smaller-scale problems, where single-machine scalability is sufficient, the following libraries provide powerful streaming semantics:

- Functional Streams for Scala (FS2) is a more pure functional approach to stream processing.
- Zio is a more purely functional alternative to Akka actors, with a streaming component.
- Akka Streams is a layer on top of the actor model that provides streaming semantics without the need for writing actor boilerplate.

Recap and What's Next

We learned how to build scalable, robust, concurrent applications using Akka actors for large-scale systems. We also learned about Scala's support for process management and futures. We discussed streaming systems and their applications and even using shell tools from Scala.

The next chapter examines how to simulate a feature found in some dynamically typed languages, methods that don't actually exist but can be interpreted as such dynamically.

CHAPTER 20

Dynamic Invocation in Scala

Most of the time, Scala's static typing is a virtue. It adds safety constraints that are useful for ensuring correctness at runtime and easier comprehension when browsing code. Many errors are caught at compile time. These benefits are especially useful in large-scale systems.

Occasionally, you might miss the benefits of dynamic typing, however. For example, in "Structural Types" on page 362, we discussed a scenario where we would like to process SQL query result sets without necessarily defining a custom type, like a case class, for each and every query's returned record type. We'll explore this scenario in this chapter. We'll also avoid the completely untyped alternative of holding the column names and values as key-value pairs in a map or using a generic record type that requires casting column values to the correct type.

3 We'll leverage two mechanisms in Scala 3 that fall in the gap between specific types and generic records. The first is the new Scala 3 feature called *structural types* that I introduced in "Structural Types" on page 362. This chapter will start with a more sophisticated example for the query scenario. Then we'll discuss a less type-safe but more flexible mechanism that exists in both Scala 2 and 3, the `scala.Dynamic` trait. The `Dynamic` trait provides greater flexibility, but structural types provide better type safety.

3 Structural Types Revisited

Scala 3 structural types provide type-safe dynamic invocation. We saw a short example that uses the new `reflect.Selectable` trait in "Structural Types" on page 362. Here is a more extensive example adapted from the Dotty documentation. It treats SQL query records as a sequence of name-value pairs, while also providing a way to refer to the fields with "dot" notation and even handle updates:

```scala
// src/script/scala/progscala3/dynamic/SelectableSQL.scala

import reflect.ClassTag
import collection.mutable.HashMap as HMap

object SQL:
  open class Record(elems: (String, Any)*) extends Selectable:
    private val fields = HMap.from(elems.toMap)                        // 1

    def selectDynamic(name: String): Any = fields(name)                // 2

    def applyDynamic(                                                  // 3
        operation: String, paramTypes: ClassTag[?]*)(args: Any*): Any =
      val fieldName = operation.drop("update".length)  // remove prefix
      val fname = fieldName.head.toLower +: fieldName.tail             // 4
      fields += fname -> args.head

    override def toString: String = s"Record($fields)"

type Person = SQL.Record {
  val name: String                                                     // 5
  val age: Int
  def updateName(newName: String): Unit                                // 6
  def updateAge(newAge: Int): Unit
}
```

1. Use a mutable map to show how to update variables.

2. `Selectable.selectDynamic` is used to retrieve a value for a given name.

3. `Selectable.applyDynamic` is used to find methods to call. `scala.reflect.ClassTag` is used to retain type information that is otherwise erased. See Chapter 24.

4. The dynamic methods supported will be `updateFooBar`, etc. This line converts `FooBar` into `fooBar`, which should be a field name.

5. Specify the fields for a `Person`. These will be accessed using `Record.selectDynamic`.

6. Specify the methods for updating a `Person` field. These will be accessed using `Record.applyDynamic`.

Let's try it:

```scala
scala> val person = SQL.Record(
     |   "name" -> "Buck Trends", "age" -> 29,
     |   "famous" -> false).asInstanceOf[Person]
val person: Person = Record(HashMap(
```

```
  name -> Buck Trends, famous -> false, age -> 29))

scala> person.name
     | person.age
     | person.selectDynamic("name")                            ❷
val res0: String = Buck Trends
val res1: Int = 29
val res2: Any = Buck Trends

scala> person.famous                     // ERROR              ❸
1 |person.famous
  |^^^^^^^^^^^^^
  |value famous is not a member of Person

scala> person.selectDynamic("famous")
val res3: Any = false

scala> person.updateName("Dean Wampler")
     | person.updateAge(30)
     | person
val res4: Person = Record(HashMap(
  name -> Dean Wampler, famous -> false, age -> 30))

scala> person.updateFamous(true)         // ERROR              ❹
1 |person.updateFamous(true)         // ERROR
  |^^^^^^^^^^^^^^^^^^^
  |value updateFamous is not a member of Person

scala> person.applyDynamic("updateFamous", summon[ClassTag[Boolean]])(true)
val res5: Any = HashMap(name -> Buck Trends, famous -> true, age -> 29)

scala> person
val res6: Person = Record(HashMap(
  name -> Buck Trends, famous -> true, age -> 29))
```

❶ The idiom for constructing instances of `Person`.

❷ The alternative for finding a field in the internal `HashMap`, but note the return type, `Any`.

❸ `Person` doesn't define `famous`, but we can use `selectDynamic` to return the value (of type `Any`).

❹ We can't use `updateFamous` either, but `applyDynamic` works. It isn't intended to be invoked directly, so it's ugly to pass the summoned `ClassTag`.

Note that the types for the columns are what we want. When we type `person.name`, the compiler generates `person.selectDynamic("name").asInstanceOf[String]` and similarly for `age`. The `update*` methods are type-safe too, for the fields we defined.

Now let's explore a more general, but less type-safe, mechanism: `scala.Dynamic`, which will appear superficially similar to `Selectable`. `Dynamic` was also available in Scala 2.

A Motivating Example: ActiveRecord in Ruby on Rails

Our motivating example is the popular `ActiveRecord` API in the Ruby on Rails web framework. `ActiveRecord` is the original object-relational mapping (ORM) library integrated with Rails. Most of the details don't concern us here, but one of the useful features it offers is a DSL for composing queries that consist of chained method calls on a domain object.

However, the methods aren't actually defined explicitly. Instead, invocations are routed to Ruby's catch-all method for handling undefined methods, `method_missing`. Normally, this method throws an exception, but it can be overridden in classes to do something else. `ActiveRecord` does this to interpret the missing method as a directive for constructing a SQL query.

Suppose we have a simple database table of states in the USA (for some dialect of SQL):

```
CREATE TABLE states (
  name     TEXT,     -- Name of the state.
  capital  TEXT,     -- Name of the capital city.
  year     INTEGER   -- Year the state entered the union (USA).
);
```

With `ActiveRecord` you can construct queries as follows, where the Ruby domain object `State` is the analog of the table `states`:

```
# Find all states named "Alaska"
State.find_by_name("Alaska")
# Find all states named "Alaska" that entered the union in 1959
State.find_by_name_and_year("Alaska", 1959)
...
```

For a table with lots of columns, statically defining all permutations of the `find_by_*` methods would be unworkable. However, the protocol defined by the naming convention is easy to automate, so no explicit definitions are required. `ActiveRecord` automates all the boilerplate needed to parse the name, generate the corresponding SQL query, and construct in-memory objects for the results.

Hence, `ActiveRecord` implements an embedded or internal DSL, where the language is an idiomatic dialect of the host language Ruby rather than a separate language with its own grammar and parser.

Dynamic Invocation with the Dynamic Trait

Normally, a similar DSL in Scala would require all such methods to be defined explicitly. `Dynamic` works in an analogous way to Ruby's `method_missing`, allowing us to use methods but route the invocations through helper methods. It's a generalization of what we did earlier with `Selectable`.

The `Dynamic` trait is a marker trait; it has no method definitions. Instead, the compiler sees that this trait is used and follows a protocol described in the trait's Scaladoc page. For some `Foo` type that extends `Dynamic`, the protocol then works as follows:

What you write:	What the compiler generates:
`foo.method1("blah")`	`foo.applyDynamic("method1")("blah")`
`foo.method2(x="hi")`	`foo.applyDynamicNamed("method2")(("x","hi"))`
`foo.method3(x=1,2)`	`foo.applyDynamicNamed("method3")(("x",1),("",2))`
`foo.field1`	`foo.selectDynamic("field1")`
`foo.field2 = 10`	`foo.updateDynamic("field2")(10)`
`foo.array1(10) = 13`	`foo.selectDynamic("array1").update(10,13)`
`foo.array2(10)`	`foo.applyDynamic("array2")(10)`

`Foo` must implement any of these methods that might be called. The `applyDynamic` method is used for calls that don't use named parameters. If the user names any of the parameters, `applyDynamicNamed` is called. Note that the first parameter list has a single parameter for the method name invoked. The second parameter list has the actual parameters passed to the method.

You can declare these second parameter lists to allow a variable number of arguments to be supplied, or you can declare a specific set of typed parameters. It all depends on how you expect users to call the methods.

The methods `selectDynamic` and `updateDynamic` are for reading and writing fields that aren't arrays. The second to last example shows the special form used for array elements. For reading array elements, the invocation is indistinguishable from a method call with a single parameter. So, for this case, `applyDynamic` has to be used.

Let's create a simple query DSL in Scala using `Dynamic`. Actually, our example is closer to a query DSL in .NET languages called *language-integrated query*, or LINQ. LINQ enables SQL-like queries to be embedded into .NET programs and used with collections, database tables, etc. LINQ is one inspiration for Slick, a Scala functional-relational mapping library.

We'll implement just a few possible operators, so we'll call it CLINQ, for *cheap language-integrated query*. Also, we'll assume we only want to query in-memory data structures. The implementation is compiled with the code examples, so let's first try a script that both demonstrates the syntax we want and verifies that the implementation works (some output omitted):

```
// src/script/scala/progscala3/dynamic/CLINQ.scala
scala> import progscala3.dynamic.CLINQ

scala> def makeMap(
     |   name: String, capital: String, year: Int): Map[String,Any] =
     |   Map("name" -> name, "capital" -> capital, "year" -> year)

scala> val data = List(
     |   makeMap("Alaska",     "Juneau",      1959),
     |   makeMap("California", "Sacramento",  1850),
     |   makeMap("Illinois",   "Springfield", 1818))

scala> val states = CLINQ(data)
```

We'll study the imported `dynamic.CLINQ` case class in a moment. The `data` to query is a sequence of maps, representing records of fields and values.

Now write `SELECT`-like queries:

```
scala> states.name
     | states.capital
     | states.year
val res0: progscala3.dynamic.CLINQ[Any] =
  Map(name -> Alaska)
  Map(name -> California)
  Map(name -> Illinois)
val res1: progscala3.dynamic.CLINQ[Any] =
  Map(capital -> Juneau)
  Map(capital -> Sacramento)
  Map(capital -> Springfield)
val res2: progscala3.dynamic.CLINQ[Any] =
  Map(year -> 1959)
  Map(year -> 1850)
  Map(year -> 1818)

scala> states.name_and_capital
val res3: progscala3.dynamic.CLINQ[Any] =
  Map(name -> Alaska, capital -> Juneau)
  Map(name -> California, capital -> Sacramento)
  Map(name -> Illinois, capital -> Springfield)

scala> states.name_and_year
     | states.capital_and_year
...similar output...

scala> states.name_and_capital_and_year  // same as "states.all"
val res6: progscala3.dynamic.CLINQ[Any] =
  Map(name -> Alaska, capital -> Juneau, year -> 1959)
```

```
  Map(name -> California, capital -> Sacramento, year -> 1850)
  Map(name -> Illinois, capital -> Springfield, year -> 1818)
```

Finally, use where clauses for filtering:

```
scala> states.all.where("year").EQ(1818)
     | states.all.where("name").NE("Alaska")
val res7: progscala3.dynamic.CLINQ[Any] =
  Map(name -> Illinois, capital -> Springfield, year -> 1818)
val res8: progscala3.dynamic.CLINQ[Any] =
  Map(name -> California, capital -> Sacramento, year -> 1850)
  Map(name -> Illinois, capital -> Springfield, year -> 1818)

scala> states.name_and_capital.where("capital").EQ("Sacramento")
     | states.name_and_capital.where("name").NE("Alaska")
     | states.name_and_year.where("year").EQ(1818)
val res9: progscala3.dynamic.CLINQ[Any] =
  Map(name -> California, capital -> Sacramento)
val res10: progscala3.dynamic.CLINQ[Any] =
  Map(name -> California, capital -> Sacramento)
  Map(name -> Illinois, capital -> Springfield)
val res11: progscala3.dynamic.CLINQ[Any] =
  Map(name -> Illinois, year -> 1818)
```

CLINQ knows nothing about the keys in the maps, but the Dynamic trait allows us to support methods constructed from them. Here is the implementation of CLINQ:

```
// src/main/scala/progscala3/dynamic/CLINQ.scala
package progscala3.dynamic
import scala.language.dynamics                                          ❶

case class CLINQ[T](records: Seq[Map[String,T]]) extends Dynamic:

  def selectDynamic(name: String): CLINQ[T] =                           ❷
    if name == "all" || records.length == 0 then this                   ❸
    else
      val fields = name.split("_and_")
      val seed = Seq.empty[Map[String,T]]
      val newRecords = records.foldLeft(seed) {
        (results, record) =>
          val projection = record.filter {                              ❹
            case (key, _) => fields.contains(key)
          }
          // Drop records with no projection.
          if projection.size > 0 then results :+ projection
          else results
      }
      CLINQ(newRecords)

  def applyDynamic(name: String)(field: String): Where = name match
    case "where" => Where(field)                                        ❺
    case _ => throw CLINQ.BadOperation(field, """Expected "where".""")
```

```
  protected class Where(field: String) extends Dynamic:                    ❻
    def filter(op: T => Boolean): CLINQ[T] =
      val newRecords = records.filter {
        _ exists {
          case (k, v) => field == k && op(v)
        }
      }
      CLINQ(newRecords)

    def applyDynamic(op: String)(value: T): CLINQ[T] = op match
      case "EQ" => filter(x =>   value == x)                               ❼
      case "NE" => filter(x => !(value == x))
      case _ => throw CLINQ.BadOperation(field, """Expected "EQ" or "NE".""")

  override def toString: String = records.mkString("\n")

object CLINQ:
  case class BadOperation(name: String, msg: String) extends RuntimeException(
    s"Unrecognized operation $name. $msg")
```

❶ `Dynamic` is an optional language feature, so we use an import to enable it.

❷ Use `selectDynamic` for the projections of fields.

❸ Return all the fields for the keyword `all`. Also return immediately if there are no records. Otherwise, if two or more fields are joined by `_and_`, then split the name into an array of field names.

❹ Filter the maps to return just the named fields.

❺ Use `applyDynamic` for operators that follow projections. We will only implement `where` for the equivalent of SQL `where` clauses. A new `Where` instance is returned, which also extends `Dynamic`.

❻ The `Where` class is used to filter the `records` for particular values of the field named `field`. The helper method provides the ability to use different operators (`op` function).

❼ If `EQ` is the operator, call `filter` to return only records where the value for the given `field` is equal to the user-specified value. Also support `NE` (not equals). Note that supporting greater than, less than, etc., would require more careful handling of the types because not all possible value types support such expressions.

`CLINQ` is definitely cheap in several ways. It doesn't implement other useful operations from SQL, like the equivalent of `groupBy`. Nor does it implement other `where`-clause operators like greater than and less than. They are actually tricky to support because not all possible value types support them.

DSL Considerations

The `Selectable` and `Dynamic` traits are part of Scala's tools for implementing embedded DSLs (also called internal DSLs). We'll explore DSLs in depth in the next chapter. For now, note a few things.

First, the implementation is not easy to understand, which means it's hard to maintain, debug, and extend. It's very tempting to use a "cool" tool like this and live to regret the effort you've taken on. So use `Dynamic` judiciously, as well as any other DSL feature.

Second, a related challenge that plagues all DSLs is the need to provide meaningful, helpful error messages to users. Try experimenting with the examples we used in the previous section and you'll easily write something the compiler can't parse and the error messages won't be very helpful. (Hint: try using infix notation, where some periods and parentheses are removed.)

Third, a good DSL should prevent the user from writing something that's logically invalid. This simple example doesn't really have that problem, but it becomes a challenge for more advanced DSLs.

Recap and What's Next

We explored Scala's hooks for writing code with dynamically defined methods and values, which are familiar to users of dynamically typed languages like Ruby and Python. We used it to implement a query DSL that "magically" offered methods based on data values

However, we also summarized some of the challenges of writing DSLs with features like this. Fortunately, we have many tools at our disposal for writing DSLs, as we'll explore in the next chapter.

CHAPTER 21

Domain-Specific Languages in Scala

A *domain-specific language* (DSL) is a programming language that mimics the terms, idioms, and expressions used among experts in the targeted domain. Code written in a DSL reads like structured prose for the domain. Ideally, a domain expert with little experience in programming can read, understand, and validate this code, if not also write code in the DSL.

We will just scratch the surface of this large topic and Scala's support for it. For more in-depth coverage, see the DSL references in the Bibliography.

Well-crafted DSLs offer several benefits:

Encapsulation
: A DSL hides implementation details and exposes only those abstractions relevant to the domain.

Productivity
: Because implementation details are encapsulated, a DSL optimizes the effort required to write or modify code for application features.

Communication
: A DSL helps developers understand the domain and domain experts to verify that the implementation meets the requirements.

However, DSLs also have several drawbacks:

DSLs are difficult to create
: Although writing a DSL has been trendy, the effort shouldn't be underestimated. The implementation techniques can be nontrivial. It's difficult to account for all possible user errors and provide appropriate error handling and intuitive feedback. Good DSLs are harder to design than traditional APIs. The latter tend to follow language idioms for API design, where uniformity is important and easy to follow. In contrast, because each DSL is a unique language, the freedom to create code idioms that reflect the domain is hard to do well.

DSLs are hard to maintain
: DSLs can require more maintenance over the long term as the domain changes because of the nontrivial implementation techniques used. Implementation simplicity is often sacrificed for a better user experience.

It is hard to hide the implementation
: DSLs are often *leaky abstractions*. Especially when errors occur, it's difficult to hide the details from the user.

However, a well-designed DSL can be a powerful tool for accelerating user productivity.

From the implementation point of view, DSLs are classified as internal and external. An *internal* (or *embedded*) *DSL* is an idiomatic way of writing code in a general-purpose programming language, like Scala. No special-purpose parser is needed. In contrast, an *external DSL* is a custom language with its own custom grammar and parser.

Internal DSLs can be easier to create because they don't require a special-purpose parser. On the other hand, the constraints of the underlying language limit the options for expressing domain concepts, and it is harder to hide the underlying implementation. External DSLs remove this constraint. You can design the language any way you want, as long as you can write a reliable parser for it. Using a custom parser can be challenging too. Returning good error messages to the user has always been a challenge for parser writers.

Internal DSLs

Several features of Scala syntax support creation of internal (embedded) DSLs:

Flexible rules for names
: Because you can use almost any characters in a name, it's easy to create names that fit the domain, like algebraic symbols for types with corresponding properties. For example, if you have a `Matrix` type, you can implement matrix multiplication with a `*` method.

Infix notation
: Defining a `*` method wouldn't make much sense if you couldn't use infix notation; for example, `matrix1 * matrix2`.[1]

Using clause parameters, context functions, and default parameter values
: Three features that reduce boilerplate and hide complex details, such as a context that has to be passed to every method in the DSL, can be handled instead with a using clause or context function. Recall that many `Future` methods take an implicit `ExecutionContext`. Context functions, new to Scala 3, help eliminate boilerplate and provide flexible ways to build concise functionality. See "Context Functions" on page 172 for details.

Type classes and extension methods
: The ability to add methods to existing types. For example, the `scala.concurrent.duration` package has implicit conversions for numbers that allow you to write `1.25.minutes`, which returns a `FiniteDuration` instance equal to 75 seconds.

Dynamic method invocation
: As we discussed in Chapter 20, the `Selectable` (for structural typing) and `Dynamic` traits make it possible for an object to accept almost any apparent method or field invocation, even when the type has no such method or field defined with that name.

Higher-order functions and by-name parameters
: Both enable custom DSLs to look like native control constructs, like the examples we saw in "Call by Name, Call by Value" on page 94.

Self-type declarations
: Nested parts of a DSL implementation can refer to an instance in an enclosing scope if the latter has a self-type declaration visible to the nested parts. This could be used to update a state object in the enclosing scope, for example.

Macros
: Some advanced scenarios can be implemented using the new macros facility, which we'll discuss in Chapter 24.

Let's create an internal DSL for a payroll application that computes an employee's paycheck every pay period (two weeks). The DSL will compute the net salary by

1 There is also support for postfix expressions, like `50 dollars`, where `dollars` would be a method that takes no arguments. You must enable the `postfixOps` language feature to use it (e.g., `import scala.language.postfixOps`). Postfix expressions are often confusing and ambiguous, so they are strongly discouraged. Support for them may be removed in a future Scala 3 release.

subtracting the deductions from the gross salary, such as taxes, insurance premiums and retirement fund contributions.

Let's begin with some common types we'll use in both the internal and external DSLs. First, a collection of types for dollars and percentages:

```
// src/main/scala/progscala3/dsls/payroll/Money.scala
package progscala3.dsls.payroll
import progscala3.contexts.accounting.*                    ❶
import scala.util.FromDigits.Floating                       ❷

given Floating[Dollars] with                                ❸
  def fromDigits(digits: String): Dollars = Dollars(digits.toDouble)

given Floating[Percentage] with
  def fromDigits(digits: String): Percentage = Percentage(digits.toDouble)

implicit class dsc(sc: StringContext):                      ❹
  def $(tokens: Any*) =
    val str = StringContextUtil.foldTokens(tokens.toSeq, sc.parts)
    Dollars(str.toDouble)

extension (amount: Double)                                  ❺
  def dollars: Dollars = Dollars(amount)
  def percent: Percentage = Percentage(amount)

object StringContextUtil:                                   ❻
  def foldTokens(tokens: Seq[Any], parts: Seq[String]): String =
    val (str, toks) = parts.foldLeft("" -> tokens.toSeq){
      case ((s, toks), s2) =>
        if s2 == null || s2.length == 0 then s+toks.head -> toks.tail
        else s+s2 -> toks
    }
    assert(toks.size == 0)
    str
```

❶ Reuse the Dollars and Percentage types from "Scala 3 Implicit Conversions" on page 154.

❷ scala.util.FromDigits is a new Scala 3 feature for converting numeric literals to types. Discussed briefly in "Numeric Literals" on page 54.

❸ Two given instances for converting floating-point literals to Dollars and Percentages.

❹ Define a string interpolator that converts strings like $"123.40" to Dollars. We could try doing one for %"12.0" for percentages, but we run into parse issues with %.

❺ Extension methods on `Double` to convert to `Dollars` and `Percentages` .

❻ A utility for interpolated strings. It handles cases like `$"$dollars.$cents"`, where those values for `$dollars` and `$cents` are passed in as `tokens` and the fixed parts of the interpolated string are passed in as `parts` strings. To reconstruct values, you take a part when it's not empty or you take a token.

3 I won't actually use the floating-point literal initialization supported using the new `FromDigits` feature, but I added them here to show more tools you can use for your DSLs.

The rest of the shared code is for deductions:

```
// src/main/scala/progscala3/dsls/payroll/Deductions.scala
package progscala3.dsls.payroll
import progscala3.contexts.accounting.*

sealed trait Deduction:                                              ❶
  def name: String
  def amount(basis: Dollars): Dollars

case class PercentageDeduction(
    name: String, percentage: Percentage) extends Deduction:
  def amount(basis: Dollars): Dollars = basis * percentage
  override def toString = s"$name: $percentage"

case class DollarsDeduction(name: String, dollars: Dollars) extends Deduction:
  def amount(basis: Dollars): Dollars = dollars
  override def toString = s"$name: $dollars"

case class Deductions(                                               ❷
  name: String,
  annualPayPeriods: Int = 1,
  deductions: Vector[Deduction] = Vector.empty):

  def gross(annualSalary: Dollars): Dollars =                        ❸
    annualSalary / annualPayPeriods

  def net(annualSalary: Dollars): Dollars =                          ❹
    val g = gross(annualSalary)
    deductions.foldLeft(g) {
      (total, deduction) => total - deduction.amount(g)
    }

  override def toString =
    s"$name Deductions:" + deductions.mkString("\n  ", "\n  ", "")
```

❶ A sealed trait for a single deduction, with case classes for dollar-based and percentage-based deductions. The `basis` for calculating the `amount` is ignored for

Dollars because the amount is independent of gross salary and such. Really, basis is a hack for calculating a Dollars value from a Percentage.

❷ All the deductions for a given payroll period.

❸ The gross for the pay period is the total pay before any deductions, such as taxes.

❹ The net pay for the pay period is the total after deductions.

Here is the start of the internal DSL, including a main that demonstrates the DSL syntax:

```
// src/main/scala/progscala3/dsls/payroll/internal/DSL.scala
package progscala3.dsls.payroll.internal
import progscala3.dsls.payroll.*
import progscala3.contexts.accounting.*

@main def TryPayroll =
  import dsl.*                                                    ❶
  val biweeklyDeductions = biweekly { deduct =>                   ❷
    deduct federal_tax         25.0.percent
    deduct state_tax           5.0.percent
    deduct insurance_premiums 500.0.dollars
    deduct retirement_savings 10.0.percent
  }

  println(biweeklyDeductions)                                     ❸
  val annualGross = Dollars(100000.0)
  val gross = biweeklyDeductions.gross(annualGross)
  val net   = biweeklyDeductions.net(annualGross)
  print(f"Biweekly pay (annual: $annualGross): ")
  println(f"Gross: $gross, Net: $net")

object dsl:
  def biweekly(                                                   ❹
      db: DeductionsBuilder => DeductionsBuilder): Deductions =
    db(DeductionsBuilder("Biweekly", 26)).deductions

  case class DeductionsBuilder(                                   ❺
    name: String,
    annualPayPeriods: Int):

    private var all: Vector[Deduction] = Vector.empty

    def deductions: Deductions = Deductions(name, annualPayPeriods, all)

    infix def federal_tax(amount: Percentage): DeductionsBuilder =  ❻
      all = all :+ PercentageDeduction("federal taxes", amount)
      this

    infix def state_tax(amount: Percentage): DeductionsBuilder =
```

```
      all = all :+ PercentageDeduction("state taxes", amount)
      this

    infix def insurance_premiums(amount: Dollars): DeductionsBuilder =
      all = all :+ DollarsDeduction("insurance premiums", amount)
      this

    infix def retirement_savings(amount: Percentage): DeductionsBuilder =
      all = all :+ PercentageDeduction("retirement savings", amount)
      this
end dsl
```

❶ Import the DSL code in the following `dsl` object.

❷ The DSL in action. The idea is that a nonprogrammer could easily understand the rules expressed here and perhaps even write them without assistance. To be clear, this is Scala syntax.

❸ Print the deductions, then compute the net pay for the biweekly payroll.

❹ The method `biweekly` is the entry point for defining deductions. It constructs an empty `DeductionsBuilder` object that will be mutated in place (the easiest design choice) to add new `Deduction` instances.

❺ A builder for `Deductions`. The end user only sees the `Deductions` object, but the builder has extra methods for sequencing expressions.

❻ The first of the four kinds of deductions supported. It updates the `Deductions` instance in place. We declare these methods `infix` to support the DSL, but in general, you should limit use of `infix` for nonoperator methods.

The output of `progscala3.dsls.payroll.internal.TryPayroll` is the following:

```
Biweekly Deductions:
  federal taxes: 25.00%
  state taxes: 5.00%
  insurance premiums: $500.00
  retirement savings: 10.00%
Biweekly pay (annual: $100000.00): Gross: $3846.15, Net: $1807.69
```

The DSL works as written, but I would argue that it's far from perfect. Here are some issues:

It relies heavily on Scala syntax tricks
: It exploits infix notation, function literals, etc., to provide the DSL, but it would be easy for a user to break the code by adding periods, parentheses, and other changes that seem harmless.

The syntax uses arbitrary conventions
: Why are the curly braces and parentheses where they are? Why is the `deduct` parameter needed in the anonymous function for the example?

Poor error messages
: If the user enters invalid syntax, Scala error messages are presented, not domain-centric error messages.

The DSL doesn't prevent the user from doing the wrong thing
: Ideally, the DSL would not let the user invoke any construct in the wrong context. Here, too many constructs are visible in the `dsl` object. Nothing prevents the user from calling things out of order, constructing instances of internal implementation types (like `Percentage`), etc.

It uses mutable instances
: A DSL like this is not designed to be high performance nor would you run it in a multithreading context. The mutability simplifies the implementation without serious compromises.

Most of these issues could be fixed with more effort.

Examples of internal DSLs can be found in most of the Scala testing libraries, like ScalaTest, Specs2, and ScalaCheck. We are about to use another example of a nice internal DSL for parsing to write an external DSL!

External DSLs with Parser Combinators

When you write a parser for an external DSL, you can use a parser generator tool like ANTLR. However, several *parser combinator* libraries for Scala provide intuitive internal DSLs that make parser definitions look very similar to Extended Backus-Naur Form. Hence, they provide a very nice example of an internal DSL!

Some of the general-purpose parsing libraries include Fastparse, which aims for high performance and understandable error messages; Atto, a lightweight and fast library; a cats-parse a Typelevel project; and the parser combinators library that was originally part of the Scala library but is now packaged separately. I'll use the latter for the example. The `sbt` build dependencies for the code examples have been configured to use it.

About Parser Combinators

Just as the collection combinators we already know construct data transformations, parser combinators are building blocks for parsers. Parsers that handle specific bits of input, such as floating-point numbers, integers, whitespace, etc., are combined together to form parsers for larger expressions. A good parser library supports sequential and alternative cases, repetition, optional terms, etc.

A Payroll External DSL

We'll reuse the previous example, but with a simpler grammar because our external DSL does not have to be valid Scala syntax. Other changes will make parser construction easier, such as adding commas between each deduction declaration.

As before, let's start with the imports and `main` routine:

```
// src/main/scala/progscala3/dsls/payroll/parsercomb/DSL.scala
package progscala3.dsls.payroll.parsercomb
import scala.util.parsing.combinator.*
import progscala3.dsls.payroll.*
import progscala3.contexts.accounting.*

@main def TryPayroll =                                                   ❶
  import dsl.PayrollParser
  val input = """biweekly {
    federal tax          20.0  percent,
    state tax            3.0   percent,
    insurance premiums   250.0 dollars,
    retirement savings   15.0  percent
  }"""
  val parser = PayrollParser()
  val biweeklyDeductions = parser.parseAll(parser.biweekly, input).get

  println(biweeklyDeductions)
  val annualGross = Dollars(100000.0)
  val gross = biweeklyDeductions.gross(annualGross)
  val net   = biweeklyDeductions.net(annualGross)
  print(f"Biweekly pay (annual: $annualGross): ")
  println(f"Gross: $gross, Net: $net")
end TryPayroll

object dsl:
  class PayrollParser extends JavaTokenParsers:                          ❷

    /** @return Parser[(Deductions)] */
    def biweekly = "biweekly" ~> "{" ~> deductions <~ "}" ^^ { ds =>
      Deductions("Biweekly", 26, ds)                                      ❸
    }

    /** @return Parser[Vector[Deduction]] */
    def deductions = repsep(deduction, ",") ^^ { ds =>                   ❹
```

```
      ds.toVector
    }

    /** @return Parser[Deduction] .*/
    def deduction =                                                      ❺
      federal_tax | state_tax | insurance | retirement

    /** @return Parser[Deduction] */
    def federal_tax = parsePercentageDeduction("federal", "tax")         ❻
    def state_tax   = parsePercentageDeduction("state", "tax")
    def retirement  = parsePercentageDeduction("retirement", "savings")
    def insurance   = parseDollarsDeduction("insurance", "premiums")

    private def parsePercentageDeduction(word1: String, word2: String) =
      word1 ~> word2 ~> percentage ^^ {
        percentage => PercentageDeduction(s"${word1} ${word2}", percentage)
      }
    private def parseDollarsDeduction(word1: String, word2: String) =
      word1 ~> word2 ~> dollars ^^ {
        dollars => DollarsDeduction(s"${word1} ${word2}", dollars)
      }

    /** @return Parser[Dollars] */
    def dollars = doubleNumber <~ "dollars" ^^ { d => Dollars(d) }       ❼

    /** @return Parser[Percentage] */
    def percentage = doubleNumber <~ "percent" ^^ { d => Percentage(d) }

    def doubleNumber = floatingPointNumber ^^ (_.toDouble)
  end PayrollParser
end dsl
```

❶ A test program. Note how the input is defined as a multiline string, with slightly different values than the previous example. This choice means you don't get compile-time checking of the string, but it nicely supports loading definitions from a file at runtime.

❷ The class defining the grammar and parser. `JavaTokenParsers` provides some convenient utilities for parsing numbers and such.

❸ The top-level parser, created by building up smaller parsers. The entry method `biweekly` returns a `Parser[Deductions]`, which is a parser for a complete deductions specification. It returns a `Deductions` object. We'll discuss the syntax in a moment.

❹ Parse a comma-separated list of deductions. Adding the requirement to use a comma simplifies the parser implementation. (Notice the commas in the preceding `input` string.) The `repsep` method parses an arbitrary number of deduction expressions.

❺ Recognize four possible deductions.

❻ Call one of two helper functions to construct the four deduction parsers.

❼ Parse `Dollars` and such.

The output of `progscala3.dsls.payroll.parsercomb.TryPayroll` is the same as before, with slightly different numbers.

Let's look at `biweekly` more closely. Here it is rewritten a bit to aid the discussion:

```
"biweekly" ~> "{" ~> deductions <~ "}"
  ^^ { ds => Deductions("Biweekly", 26, ds) }
```

The first line finds three *terminal tokens*, `biweekly`, `{`, and `}`, with the results of evaluating the `deductions` production between the `{…}`. The arrow-like operators (actually methods, as always), `~>` and `<~`, mean drop the token on the side of the `~`. So the literals are dropped and only the result of `deductions` is retained.

In the second line, the `^^` separates the left side reduction from the right side *grammar rule* for the production. The grammar rule takes as parameters the tokens retained. If there is more than one, a partial function literal is used of the form `{ case t1 ~ t2 ~ t2 =>…}`, for example. In our case, `ds` is a `Vector` of `Deduction` instances, which is used to construct a `Deductions` instance.

Note that `DeductionsBuilder` in the internal DSL is not needed here.

Internal Versus External DSLs: Final Thoughts

Let's compare the internal and external DSL logic the end user writes. Here is the internal DSL example again:

```
val biweeklyDeductions = biweekly { deduct =>
  deduct federal_tax          (25.0  percent)
  deduct state_tax            (5.0   percent)
  deduct insurance_premiums   (500.0 dollars)
  deduct retirement_savings   (10.0  percent)
}
```

Here is the external DSL example again:

```
val input = """biweekly {
  federal tax          20.0  percent,
  state tax            3.0   percent,
  insurance premiums   250.0 dollars,
```

```
    retirement savings   15.0  percent
  }"""
```

You'll have to weigh which trade-offs make the most sense for your situation. The external DSL is simpler, but the user must embed the DSL in strings. Hence, compile-time checking, as well as niceties like IDE code completion, refactoring and color coding aren't available.

On the other hand, the external DSL is easier and actually more fun to implement. It should also be less fragile compared to relying on Scala parsing tricks.

Recall that we can implement our own string interpolators (see "Build Your Own String Interpolator" on page 142). This is a useful way to encapsulate a parser built with combinators behind a slightly easier syntax. For example, if you implement a SQL parser of some sort, let the user invoke it with `sql"SELECT * FROM table WHERE...;"`, rather than having to use the parser API calls explicitly like we did here.

Recap and What's Next

It's tempting to create DSLs with abandon. DSLs in Scala can be quite fun to work with, but don't underestimate the effort required to create robust DSLs that meet your clients' usability needs, while at the same time requiring reasonable effort for long-term maintenance and support.

In the next chapter, we'll explore the ecosystem of Scala tools and libraries.

CHAPTER 22

Scala Tools and Libraries

This chapter fills in some details about the Scala command-line tools, build tool options, IDE and text editor integration, and a look at some popular third-party libraries for Scala. Finally, this chapter explores mixing Java and Scala code.

Libraries and tools change quickly. I will avoid some details that are likely to change over time, focusing instead of suggestions for finding the best options for your needs, with some current examples. For the latest library options, search the Scala Library Index.

3 Scala 3 Versions

To better support migration of code bases from Scala 2 to 3, Scala 3 introduces a language version construct that allows the user to specify which version should be used, either with a command-line option or an import statement.

Here is the currently defined list of versions, adapted from the documentation on language versions:

Version 3.0
: The current default version. Some Scala 2 idioms are deprecated but still supported.

Version 3.0-migration
: Identical to 3.0 but with a Scala 2 compatibility mode enabled that helps migration of Scala 2.13 sources over to Scala 3. In particular:

- Flags some Scala 2 constructs that are disallowed in Scala 3 as migration warnings instead of hard errors.
- Changes some rules to be more lenient and backward compatible with Scala 2.13.
- Gives some additional warnings where the semantics have changed between Scala 2.13 and 3.0.
- Offers code rewrites from Scala 2.13 to 3.0, when used with the `-rewrite` flag.

Version future
: A preview of changes to be introduced in future releases of Scala 3, when deprecated Scala 2 idioms will be dropped and new Scala 3 features that break Scala 2 code will be enforced.

Version future-migration
: The same as future but with additional helpers to migrate from 3.0, including migration warnings and optional rewrites (using the `-rewrite` flag).

There are two ways to specify a language version:

- With a `-source` option for `scalac` (e.g., `-source:future` or `-source future`).
- With a `scala.language` import at the top of a compilation unit, as in the following example:

```
import scala.language.future
package p

class C { ... }
```

Language imports supersede command-line settings in the compilation units where they are specified. Only one language version import is allowed in a compilation unit, and it must come before all other definitions in that unit.

Command-Line Interface Tools

I rarely use the Scala CLI tools directly because it's easier to use them indirectly through build tools and IDEs. However, you'll need to configure compiler flags in your `build.sbt`.

"Installing the Scala Tools You Need" on page 3 described how to install a Java JDK and `sbt` using the instructions on their respective websites. The Scala website's Getting Started page discusses many options for installing and using Scala. Here, I'll discuss one of the newer options, using Coursier. Then I'll discuss the various Scala CLI tools themselves.

Coursier

Coursier is a new dependency resolver and tool manager. It replaces Maven and Ivy, the traditional dependency resolvers for Java and Scala projects. Written in Scala, it is fast and easy to embed in other applications. Coursier is embedded in `sbt`.

Installing the Coursier CLI is useful for managing other command-line tools, as well as libraries. Coursier can be used to install `sbt`, the various Scala tools, and it can even manage installations of different JDK versions.

Start with the Coursier installation instructions. See also Alex Archambault's very good blog post on using Coursier.

After installing Coursier, run the `cs` or `coursier` command to install `sbt` and several of the Scala CLI tools. Here's an example:

```
cs install sbt scala scalac scaladoc
```

I'll discuss these and other `scala*` tools. Use the `--help` option to show you how to configure where tools are installed and more.

Managing Java JDKs with Coursier

You can use Coursier to install and manage multiple JVMs. To see the list of available JVMs, run this command:

```
cs java --available
```

For example, to install the AdoptOpenJDK version 15.0.1:

```
cs java --jvm adopt:1.15.0-1
```

To pick a JVM to use, you can run the following command:

```
cs java --jvm 15.0.1 --setup
```

If you would rather print a definition for `JAVA_HOME` for the JVM specified, replace `--setup` with `--env`. Then put the definition in your shell initialization file (e.g., `~/.bashrc` or `~/.zshrc` on macOS or Linux). Also modify your `PATH` to begin with `$JAVA_HOME/bin` (or `%JAVA_HOME%\bin` for Windows). Putting this setting at the beginning of the `PATH` prevents other JVMs on your path from being used instead.

To switch between versions in the current shell environment, use these commands (macOS or Linux):

```
eval $(cs java --jvm 15.0.1 --env)    # Actually set JAVA_HOME
export PATH=$JAVA_HOME/bin:$PATH      # Put $JAVA_HOME/bin first on the PATH
```

The scalac Command-Line Tool

The `scalac` command compiles Scala source files and generates JVM class files. You invoke `scalac` like this:

```
scalac <options> <source files>
```

Recall from "A Taste of Scala" on page 9 that source filenames don't have to match the public class name in the file. You can define multiple public classes in a Scala source file too. Similarly, package declarations don't have to match the directory structure.

3 However, in order to conform to JVM requirements, a separate `.class` file is generated for each top-level type with a name that corresponds to the type's name. The class files are written to directories corresponding to the package declarations. Scala 3 also outputs *.tasty* files with an intermediate representation between source code and JVM byte code files. For teams with mixed Scala 2.13 and Scala 3 libraries, TASTy Reader was shipped in Scala 2.13.4, so the compiler can use Scala 3 libraries by reading their *.tasty* files. Scala 3 can already use Scala 2 libraries. For details, see the Scala 3 Migration Guide and Chapter 24.

Run `scalac -help` to see all the main options supported. Use `scalac -X` to see advanced options, and `scalac -Y` to see private (experimental) options, mostly of use to the language development team itself, and experimental options.

Here I'll just discuss some of the more interesting options, including those used for the code examples in `build.sbt`. Table 22-1 shows these options. The ones that aren't marked as used are actually commented out in `build.sbt`, for reasons I'll explain shortly.

Table 22-1. The `scalac` command options used in the code examples

Option	Used?	Description
`-d...`	X	Specify the output directory for build artifacts (set by `sbt` to `target/scala-3.X.Y/classes`).
`-encoding utf-8`	X	Specify character encoding used by source files.
`-deprecation`	X	Emit warnings and location for usages of deprecated APIs.
`-unchecked`	X	Enable additional warnings where generated code depends on assumptions.
`-feature`	X	Emit warnings and locations for usages of features that should be imported explicitly.
`-explain`	X	Explain errors in more detail.
`-explain-types`		Explain type errors in more detail.
`-indent`	X	Allow significant indentation.
`-noindent`		Require the classic `{...}` syntax, indentation is not significant.
`-new-syntax`	X	Require `then` in conditional expressions.
`-old-syntax`		Require `(...)` around conditional expressions.

Option	Used?	Description
`-language:Scala2`		Compile Scala 2 code, highlight what needs updating.
`-migration`		Emit warning and location for migration issues from Scala 2.
`-rewrite`		Attempt to fix code automatically.
`-source:future`	X	Enforce deprecation rules for future Scala 3 releases and such.
`-Xfatal-warnings`	X	Treat warnings as compilation errors.
`-Yexplicit-nulls`		Make reference types nonnullable. Nullable types can be expressed with unions (e.g., `String|Null`). (All `-Y` flags are experimental or internal! They are subject to change.)
`-classpath foo:bar`		Add to the classpath.

The options `-deprecation`, `-unchecked`, and `-feature` are recommended for maintaining good quality code. I like `-Xfatal-warnings` too. Scala 2 had the `-Xlint` option that was useful for flagging legal but questionable constructs.

Use `-noindent` and `-old-syntax` if you prefer to require Scala 2 syntax for conditionals and use of braces. (Omitting these flags, along with `-indent` and `-new-syntax`, allows old and new syntax.) For this book, I chose to use the new syntax conventions, more like Python-style syntax. Hence, I use the flags `-new-syntax` and `-indent`.

The three flags `-language:Scala2`, `-migration`, and `-rewrite` are very handy for migrating from Scala 2. I used them when I started migrating the code examples from the previous edition of this book.

The scala Command-Line Tool

The `scala` command runs a program, if specified. Otherwise, it starts the REPL. You invoke `scala` like this:

```
scala <options> [<file|class|jar> <arguments>]
```

The options are the same as for `scalac`.

The *file* argument is a source file to interpret. It must have a single `@main` or `main` method entry point:

```
$ scala src/main/scala/progscala3/introscala/UpperMain2.scala Hello World
HELLO WORLD
```

Note that none of the files in the code examples' *src/script* directory have entry points because these files are designed for interactive use in the REPL.

3 The Scala 2 `scala` command worked like a noninteractive REPL when given a file of Scala statements and expressions. It just executed them as if they were typed or pasted into the REPL. The Scala 3 `scala` command expects the file to contain an entry point it will run after compiling the file's contents.

The Scala 2 and 3 REPLs treat input files differently!

You can specify a compiled *class* or *jar* file. In the following example, note the use of the `-classpath` argument to specify the root location of the `sbt`-generated `.class` files:

```
$ scala -classpath target/scala-3.0.0/classes progscala3.introscala.Hello Dean
Hello: DEAN
```

If no file, class, or jar is specified, the interactive REPL is started. See "Running the Scala Command-Line Tools Using sbt" on page 7 for a discussion of the `:help` and other options inside the REPL.

The scaladoc Command-Line Tool

The `scaladoc` tool is used to generate documentation from code. It was reimplemented for Scala 3 with the ability to generate a range of static website content, not just documentation from Scala source files.

The easiest way to use `scaladoc` for your project is to run the `sbt doc` task. For more information, see the new Scaladoc documentation.

Other Scala Command-Line Tools

The Coursier `install` command can install other useful tools, including the following:

`scalafix`
: Refactoring and linting tool for Scala.

`scalafmt`
: Code formatter for Scala.

`scalap`
: Class file decompiler.

`scalap` may be ported to Scala 3 or replaced with a new tool focused on TASTy Inspection, where TASTy is the new intermediate format used by the compiler for Scala object code. The Java decompiler CFR is also very useful for this purpose.

Build Tools

sbt is the most common build tool for Scala projects. It also builds Java code. Table 22-2 lists popular alternatives:

Table 22-2. Build tools for Scala

Name	URL	Description
Maven (mvn)	*https://maven.apache.org*	JVM build tool with an available Scala plug-in.
Gradle	*https://www.gradle.org*	JVM build tool with an available Scala plug-in.
Bazel	*https://bazel.build*	A cross-language tool that is popular with large enterprises.
Mill	*https://github.com/lihaoyi/mill*	Li Haoyi's Java, Scala, and Scala.js build tool.

Maven and Gradle are widely used in enterprises for JVM-based projects. However, there are several reasons for choosing `sbt` as your build tool:

- Nobody gets fired for picking `sbt`. It's the ubiquitous, tried and true choice.
- The Scala plug-ins for most IDEs understand `sbt` projects, which they can import quickly and easily.
- There are lots of `sbt` plug-ins for different tasks, like publishing releases to Maven repositories.

Personally, I would not accept an alternative without the equivalent of `sbt console` and `sbt ~test`. Worst case, consider supporting two build systems, one for the corporate build and `sbt` for your development builds.

Integration with IDEs and Text Editors

Scala plug-ins exist for all the major IDEs, providing features like code completion, refactoring, navigation, and building. In most cases, they rely on your build tool to provide project-level information, such as dependencies and compiler flags.

Some IDE plug-ins and most text editor plug-ins are based on the Language Server Protocol (LSP), an open standard started by Microsoft. The Metals project implements LSP for Scala. The Metals website provides instructions for installing and using Metals in many IDEs and text editors.

Using Notebook Environments with Scala

The concept of an interactive notebook has become popular in the data science community. The most popular example is Jupyter, formerly known as iPython. Notebooks integrate nicely formatted documentation written in Markdown, executable code in many different languages, and the ability to graph data, all intermixed as needed.

Scala is one language option for most notebook environments. Notebooks are a better option than Scala worksheets for more permanent yet interactive work because they integrate documentation, graphing of data, and other tools. They are ideal for tutorials, for example.

One way to work with Scala in Jupyter is to use a Docker image that combines Jupyter with all the tools you need to run Spark, including Scala. The all-spark-notebook image is one example. It bundles Apache Toree to provide Spark and Scala support.[1]

Table 22-3 lists other notebook options you might consider.

Table 22-3. Notebook environments for Scala

Name	URL	Description
Polynote	*https://polynote.org*	A cross-language notebook environment with built-in Scala support, developed by Netflix.
BeakerX	*http://beakerx.com*	Extensions for Jupyter that add Spark and Scala support, graphing libraries, etc. It is developed by Two Sigma.
Zeppelin	*https://zeppelin-project.org*	A popular notebook environment that focuses on big-data environments.
Databricks	*https://databricks.com*	A feature-rich, commercial, cloud-based service for Spark with a notebook UI.

For an example that uses notebooks, see my spark-scala-tutorial on Apache Spark with Scala 2.

Testing Tools

In functional languages with rich type systems, like Scala, specifying the types is seen as a regression-testing capability, one that's exercised every time the compiler is invoked. The goal is to define types that eliminate the possibility of invalid states, when possible.

Still, tests are required. By now everyone should be using *test-driven development*, in some form. Table 22-4 lists some testing libraries to consider.

Table 22-4. Test libraries for Scala

Name	URL	Description
ScalaTest	*https://www.scalatest.org*	The most popular test library for Scala. It provides a rich set of DSL options, so you can use the style you want for writing tests.
Specs2	*https://github.com/etorreborre/specs2*	A testing library that emphasizes tests as specifications of correct behavior.

1 At the time of this writing, this environment has not yet been upgraded to Scala 3.

Name	URL	Description
MUnit	*https://scalameta.org/munit*	A new, lightweight library with basic syntax. (Used for this edition's code examples.)
ScalaCheck	*https://scalacheck.org*	A property-based testing library.
Hedgehog	*https://github.com/hedgehogqa/scala-hedgehog*	A property-based testing library.

I prefer a lightweight library with a minimal feature set. I chose MUnit for the code examples, which includes built-in support for ScalaCheck.

Types should have well-defined properties. *Property-based testing* is another angle on testing popularized by Haskell's QuickCheck and now ported to many languages. Conditions for a type are specified that should be true for all instances of the type. Recall our discussion in "Algebraic Data Types" on page 397. A property-based testing tool tries the conditions using a representative sample of instances that are automatically generated. It verifies that the conditions are satisfied for all the instances and reports when it finds instances that trigger failures. ScalaCheck and Hedgehog are Scala examples. One or both of them are integrated with the other general-purpose libraries.

Scala for Big Data: Apache Spark

I mentioned in Chapter 19 that the need to write concurrent programs has been a driver for adoption of FP. However, good concurrency models, like actors, make it easier for developers to continue using OOP techniques and avoid the effort of learning FP. So perhaps the multicore problem isn't driving change as fast as many of us originally thought.

Big data has been another driver of FP adoption. Around the time the second edition of this book was published, Scala adoption was growing rapidly, driven by exploding interest in big-data tools like Apache Spark and Apache Kafka, which are written in Scala.

In particular, the functional combinators in Scala's collection library, such as `map`, `flatMap`, `filter`, and `fold`, shine as tools for manipulating data sets with concise, composable expressions, many of which have logical mappings to `SQL` idioms.

I mentioned Spark in the context of stream processing in "Stream Processing" on page 426. Now we'll explore the original batch-mode RDD (resilient distributed dataset) API in a little more detail. Spark's RDD API was largely inspired by Scala's collection library, extending it to be an abstraction for processing massive, partitioning data sets in a cluster.

During this period, many Java developers I spoke with who had big-data experience and little prior interest in Scala would light up when they saw how concise their code could be if they made the switch to Scala. For this reason, Scala emerged as the de facto programming language for data engineering. Data scientists then and now mostly used their favorite languages, such as Python.

Another problem Spark has solved is how to optimize memory usage for very large data sets. The memory models for most languages that support garbage collection are ideal for graphs of heterogenous objects in memory with complex dependencies on each other. However, their overhead becomes suboptimal when you have billions of objects that are essentially homogeneous records in collections with few or no interdependencies. Spark's newer `Dataset` API stores the data *off heap* in a custom format that is highly optimized for space and access efficiency.

Let's see an example of Spark's RDD API used to implement a popular algorithm called Word Count. We load a corpus of documents, tokenize them (we'll just split on nonalphanumeric characters for simplicity), then count the occurrences of each unique word across the data set, which could be arbitrarily large:[2]

```
// src/script/scala-2/progscala3/bigdata/SparkWordCount.scala

val file = "README.md"                                          ❶
val input = sc.textFile(file).map(_.toLowerCase)                ❷
input
  .flatMap(line => line.split("""\W+"""))                      ❸
  .map(word => (word, 1))                                       ❹
  .reduceByKey((count1, count2) => count1 + count2)             ❺
  .saveAsTextFile(file+".wordcount")                            ❻
```

❶ Just use the code example *README* as the corpus.

❷ The `spark-shell` REPL wraps the Scala REPL and automatically defines an instance of a class called `SparkContext`, with the instance name `sc`. We use it to load the corpus of text, converting to lowercase. The type of `input` is `RDD` in Spark.

❸ Split on nonalphanumeric sequences of characters, flat-mapping from lines to words.

❹ Map each word to the tuple `(word, 1)` (i.e., a count of `1`).

❺ Use `reduceByKey`, which functions like a SQL `groupBy` followed by a reduction, in this case summing the values in the tuples, all the `1`s. The output is the total count for each unique word. In Spark, the first element of a tuple is the default key for operations like this, and the rest of the tuple is the value.

2 Adapted from my spark-scala-tutorial.

❻ Write the results to the path specified as the second input parameter. Spark follows Hadoop conventions and actually treats the path as a directory to which it writes one partition file per final task (with naming convention `part-n`, where `n` is a five-digit number, counting from `00000`).

See the code example *README* for details on how to run this example with the Spark REPL, `spark-shell`. This program is just seven lines of code! This concision is one reason Spark remains very popular.

Typelevel Libraries

Most of the state-of-the-art FP libraries for Scala under the Typelevel umbrella. Table 22-5 lists a few of these libraries. See the projects page for the full list.

Table 22-5. Typelevel libraries

Name	URL	Description
Cats	*https://github.com/typelevel/cats*	The most popular Scala library for pure-FP abstractions, including categories. It was discussed in "Category Theory" on page 400. See also subprojects like cats-effect.
Doobie	*https://github.com/tpolecat/doobie*	A pure, functional JDBC layer.
FS2	*https://fs2.io*	Functional streams for Scala. Mentioned in "Stream Processing" on page 426.
http4s	*https://http4s.org*	Functional, streaming HTTP.
Monix	*https://monix.io*	Composition of high-performance, asynchronous, event-based programs.
ScalaCheck	*https://scalacheck.org*	Property-based testing. Discussed previously.
Shapeless	*https://github.com/milessabin/shapeless*	Pushing the envelope of generic programming using type classes and dependent types.
Spire	*https://github.com/typelevel/spire*	Numerics library.
Squants	*https://github.com/typelevel/squants*	Quantities, units of measure, and dimensional analysis.

Li Haoyi Libraries

Li Haoyi is one of the most prolific Scala developers in our community. I mentioned a few of his tools previously. Table 22-6 lists several of his libraries.

Table 22-6. Li Haoyi's libraries for Scala

Name	URL	Description
Mill	*https://github.com/lihaoyi/mill*	Build tool. Discussed previously.
Ammonite	*https://ammonite.io/#Ammonite*	A set of libraries for scripting, including an excellent replacement for the default Scala REPL.

Name	URL	Description
Fastparse	*https://www.lihaoyi.com/post/Fastparse2EvenFasterScalaParserCombinators.html*	Parser combinators library (see "External DSLs with Parser Combinators" on page 446).

See his GitHub page for more projects. I also recommend his book, *Hands-on Scala Programming* (self-published). It's a fast introduction to Scala and to all his excellent libraries and tools.

Java and Scala Interoperability

It's common for organizations to mix Java and Scala code. This chapter finishes with a discussion of interoperability between code written in Java and Scala. The Scala.js and Scala Native websites discuss interoperability concerns for their target platforms.

Invoking Java APIs from Scala "just works" (with one exception). Going the other direction requires that you understand how some Scala features are encoded in byte code while still satisfying the JVM specification.

Using Java Identifiers in Scala Code

Java's rules for identifiers, the names of types, methods, fields, and variables, are more restrictive than Scala's rules. In almost all cases, you can just use the Java names in Scala code. You can create new instances of Java types, call methods, and access fields.

The exception is when a Java name is actually a Scala keyword. As we saw in "Language Keywords" on page 51, you can escape the name with single backticks. For example, if you want to invoke the `match` method on `java.util.Scanner`, then use ``myScanner.`match` ``.

Scala Identifiers in Java Code

On the JVM, identifiers are restricted to alphanumeric characters, underscores (_), and dollar signs ($). Scala encodes identifiers with operator characters, as shown in Table 22-7.

Table 22-7. Encoding of operator characters

Operator	Encoding	Operator	Encoding	Operator	Encoding	Operator	Encoding
`=`	`$eq`	`>`	`$greater`	`<`	`$less`		
`+`	`$plus`	`-`	`$minus`	`*`	`$times`	`/`	`$div`
`\`	`$bslash`	`|`	`$bar`	`!`	`$bang`	`?`	`$qmark`
`:`	`$colon`	`%`	`$percent`	`^`	`$up`	`&`	`$amp`

3 For your own operator definitions, use the `@targetName(...)` annotation to specify the desired name that can be called from Java code.

Java Generics and Scala Parameterized Types

All along, we've been using Java types in Scala code, like `String` and `Array`. You can use any Java generic class, including all of the Java collections.

You can also use Scala parameterized types in Java. Consider the following example using a two-element Scala tuple. You can't use Scala's literal syntax for tuples, but you can still create them:

```
// src/main/java/progscala3/javainterop/JavaWithScalaTuples.java
import scala.Tuple2;
...
Tuple2<String,Integer> si = new Tuple2<String,Integer>("one", 2);
```

Table 22-8 repeats Table 11-1 with an added column showing the equivalent Java syntax for covariant, contravariant, and invariant type specifications.

Table 22-8. Type variance annotations in Scala and Java

Scala	Java	Description
`+T`	`? extends T`	*Covariant* (e.g., Seq[T_{sub}] is a subtype of Seq[T]).
`-T`	`? super T`	*Contravariant* (e.g., X[T^{sup}] is a subtype of X[T]).
`T`	`T`	*Invariant* (e.g., can't substitute Y[T^{sup}] or Y[T_{sub}] for Y[T]).

An important difference between Java and Scala is that Java generics are not specified with variance behavior when they are defined. Instead, the variance behavior is specified when the type is used (i.e., at the call site), when instances are declared. Scala makes it the responsibility of the type designer to specify the correct behavior, rather than the user's responsibility to specify the correct variance.

Conversions Between Scala and Java Collections

A common occurrence when using Java libraries from Scala is the need to work with Java collections. Similarly, using a Scala API from Java may require working with Scala collections. Since most people will want to work with the native collections for each language, the Scala library provides conversion utilities.

Unfortunately, there are a number of deprecated converters in the library, so it can be confusing which group to use. The `scala.jdk.CollectionConverters` should be used when using Java collections (including Java `Streams`) in Scala code, so they feel native. The collections are actually wrapped, not converted, to avoid copying when possible.

When programming in Java and you want Java collection wrappers around instances of Scala collections, use the `scala.jdk.javaapi.CollectionConverters` API.

Some of these utilities leverage types in *scala/collection/convert* and may be useful for you to work with directly.

Java Lambdas Versus Scala Functions

When compiling for the JVM, Scala functions are implemented with Java lambdas in the generated byte code. This means that when calling a Scala library method from Java code where a function is required, you can pass a lambda. Similarly, when calling a Java library method from Scala code where a lambda is required, you can pass a Scala function.

Annotations for JavaBean Properties and Other Purposes

We saw in Chapter 9 that Scala does not follow the JavaBeans conventions for field reader and writer methods in order to support the more useful Uniform Access Principle. However, if you need these methods for use with dependency injection frameworks and other tools, there is an annotation that you can apply to fields, `@scala.beans.BeanProperty`, which tells the compiler to generate JavaBeans-style getter and setter methods.

Here is an example:

```scala
// src/main/scala/progscala3/javainterop/ComplexBean.scala
package progscala3.javainterop
import scala.annotation.targetName

/**
 * See also this Scala 2 version:
 *   src/main/scala-2/progscala3/javainterop/ComplexBean2.scala
 */
case class ComplexBean(
  @scala.beans.BeanProperty var real: Double,
  @scala.beans.BeanProperty var imaginary: Double):

  @targetName("plus") def +(that: ComplexBean) =
    ComplexBean(real + that.real, imaginary + that.imaginary)
  @targetName("minus") def -(that: ComplexBean) =
    ComplexBean(real - that.real, imaginary - that.imaginary)
```

Java requires checked exceptions to be declared in method signatures. In Scala code that will be used from Java, use the `@throws` annotation to indicate that a particular exception type may be thrown.

Recap and What's Next

This chapter filled in some details about the Scala command-line tools, build tools, and integration with IDEs and text editors. I also discussed a few of the third-party libraries and tools available, but I just scratched the surface. To search for the latest library options for your particular needs, see the Scala Library Index. Finally, I discussed mixing Scala and Java code.

Our next chapter covers application design considerations essential for truly succeeding with Scala.

CHAPTER 23
Application Design

Until now, we have mostly discussed language features. Most of the examples we've studied have been small, although I tried to make them realistic and useful. Actually, small is a very good thing. Drastic reduction in code size means all the problems of software development diminish in significance.

Not all applications can be small, however. This chapter considers the concerns of large, evolving APIs and applications. We'll discuss a few Scala language and API features that we haven't covered yet, consider a few design patterns and idioms, discuss architecture concepts, and balance object-oriented versus functional design techniques.

Recap of What We Already Know

Let's recap a few of the concepts we've covered already that make small design problems easier to solve and thereby provide a stable foundation for applications.

Functional composition
: Most of the book examples have been tiny in large part because we've used the concise, powerful combinators provided by collections and other containers. They allow us to compose logic with a minimum amount of code.

Types, especially parametric polymorphism
: Types enforce constraints. Ideally, they express as much information as possible about the behavior of our programs. For example, using `Option[T]` can eliminate the use of `nulls`. Parameterized types and abstract type members are tools for abstraction and code reuse.

Mixin traits
: Traits enable modularized and composable behaviors.

for comprehensions
: `for` comprehensions provide a convenient DSL for working with types using `flatMap`, `map`, and `filter`/`withFilter`.

Pattern matching
: Pattern matching makes quick work of data extraction.

Givens, extension methods, and implicit conversions
: Givens, extension methods, and implicit conversions solve many design problems, including boilerplate reduction, threading context through method calls, implicit conversions, ad hoc modifications of types, and even some type constraints.

Fine-grained visibility rules and exports
: Scala's fine-grained visibility rules and the Scala 3 export ability enable precise control over the visibility of implementation details in APIs, only exposing the public abstractions that clients should use. It takes discipline to do this, but doing so prevents avoidable coupling to the API internals, which makes evolution more difficult.

Open, sealed, enum, and final types
: By default, concrete classes are closed for extension unless they are declared open or the `adhocExtensions` language feature is enabled. Sealed type hierarchies and enums can't be extended outside their definition file. Types, methods, etc., that are marked `final` are closed for extension too. All contribute to careful design with fewer bugs, especially as a code base evolves.

Error handling strategies
: `Option`, `Either`, `Try`, and `cats.data.Validated` help us *reify* exceptions and other errors, making them part of the normal result returned from functions and preserving referential transparency. The type signature also tells the user what successful or error results to expect.

  `Future` exploits `Try` for the same purpose. The actor model implemented in Akka has a robust, strategic model for supervision of actors and handling failures (Chapter 19).

Let's consider other application-level concerns, starting with annotations.

Annotations

Annotations to tag elements with metadata are used in many languages. Some Scala annotations provide directives to the compiler or external tools.

Table 23-1 lists some the most common annotations, most of which we have already seen. Some of them are in the `scala.annotation` package, while others are in `scala`.

Table 23-1. Common Scala annotations

Name	Description
`@tailrec`	Assert to the compiler that the annotated method is tail recursive. If it isn't, a compilation error is thrown.
`@targetName`	Define an alphanumeric name for an operator identifier.
`@unchecked`	Don't issue a warning for potential pattern binding errors, usually related to typing.
`@unchecked Variance`	Don't check type variance.
`@deprecated`	Mark a declaration as deprecated. Issue a warning when used in code.

The `deprecated` annotation and related ones in the `scala` package are useful as your APIs evolve, allowing you to create a transition period for your users between the point when an alternative is implemented or planned and when the old construct is removed. These annotations take arguments for a message to the user about alternative choices, when the feature was deprecated, etc.

In "Annotations for JavaBean Properties and Other Purposes" on page 464, we discussed annotations that enable better interoperability with Java by changing how byte code is generated. In "Lazy Values" on page 97, we discussed `@threadUnsafe`.

Declaring an annotation in Scala doesn't require a special syntax. You declare a normal class as follows:

```
import scala.annotation.StaticAnnotation

final class marker(msg: String) extends StaticAnnotation

@marker("Hello!")
case class FooBar(name: String)
```

Using @main Entry Points

All applications need an entry point. A nice feature of `@main` is that Scala will parse the supplied argument list into whatever types we expect to see. Consider this example where some nonstring arguments are expected:

```
// src/main/scala/progscala3/appdesign/IntDoubleStringMain.scala
package progscala3.appdesign

@main def IntDoubleStringMain(i: Int, d: Double, s: String): Unit =
  println(s"i = $i")
  println(s"d = $d")
  println(s"s = $s")
```

Let's try it:

```
> runMain progscala3.appdesign.IntDoubleStringMain 1 2.2 three
...
i = 1
d = 2.2
s = three

> runMain progscala3.appdesign.IntDoubleStringMain three 2.2 1
Illegal command line: java.lang.NumberFormatException: For input string: "three"

> runMain progscala3.appdesign.IntDoubleStringMain
Illegal command line: more arguments expected
```

Scala parses the argument strings into the expected types. However, the error messages produced for invalid input are terse and may not be as user friendly as you want. Currently, there is no mechanism to plug in custom help messages, although hopefully this will change in a future release of Scala. So consider using a regular `main` method and parsing the strings yourself when you need more user-friendly error messages. Behind the scenes, Scala uses `scala.util.CommandLineParser`, which you can use too.

Design Patterns

Design patterns document reusable solutions to common design problems. Patterns become a useful part of the vocabulary that developers use to communicate.

Design patterns have taken a beating lately. Critics dismiss them as workarounds for missing language features. Newer languages like Scala provide built-in implementations or better alternatives for some of the popular *Gang of Four* ([GOF1995]) patterns, for example. Patterns are frequently misused or overused, becoming a panacea for every design problem, but that's not the fault of the patterns themselves.

I argued in "Category Theory" on page 400 that categories are FP design patterns adopted from mathematics.

Let's discuss ways in which the *Gang of Four* patterns occur in Scala as built-in features or common idioms. I'll follow the categories in the [GOF1995] book.

Creational Patterns

This section describes patterns for creating instances of types.

Abstract factory
: An abstraction for constructing instances from a type family without explicitly specifying the types. `Seq.apply` and `Map.apply` are examples where `apply` methods in `objects` are used for this purpose. They instantiate an instance of an appropriate subtype based on the parameters to the method.

Builder

Separate construction of a complex object from its representation so the same process can be used for different representations. We discussed in "Polymorphic Methods" on page 336 how a common method like `map` can be defined in a generic mixin trait, but specific instances of the correct collection type can be constructed using a pluggable builder object. Also, idioms like `seq.view.map(...)...filter(...).force()` build new sequences.

Factory method

Define an abstraction for instantiating objects and let subtypes implement the logic for what type to instantiate and how. An example of this pattern that we used in "Internal DSLs" on page 440 to convert from `Doubles` to `Dollars` and `Percentages` is `scala.util.FromDigits`. In this case, given instances are used, rather than subtyping.

Prototype

Start with a prototypical instance and copy it with optional modifications to construct new instances. Case class `copy` methods are the most common example I use, which permit cloning an instance while specifying just the arguments needed for changes.

Singleton

Ensure that a type has only one instance and all users of the type can access that instance. Scala implements this pattern as a first-class feature of the language with `objects`.

Structural Patterns

This section describes patterns for organizing types to minimize coupling while enabling collaboration.

Adapter

Create an interface a client expects around another abstraction, so the latter can be used by the client. Scala offers many mechanisms to implement this, including givens, extension methods, exports, and mixins.

Bridge

Decouple an abstraction from its implementation, so they can vary independently. Extension methods and type classes provide techniques that take this idea to a logical extreme. Not only is the abstraction removed from types that might need it, only to be added back in when needed, but the implementation of an extension method or type class can also be defined separately.

Composite
: Tree structures of instances that represent part-whole hierarchies with uniform treatment of individual instances or composites. Functional code tends to avoid ad hoc hierarchies of types, preferring to use generic structures like trees instead, providing uniform access and the full suite of combinators for manipulation of the tree. We also saw a simple `enum` declaration of tree structure in "Enumerations and Algebraic Data Types" on page 79.

Decorator
: Attach additional responsibilities to an object dynamically. Extension methods and type classes provide a principled way to do this.

Facade
: Provide a uniform interface to a set of interfaces in a subsystem, making the subsystem easier to use. The fine-grained visibility controls (see Chapter 15) and exports allow the developer to expose only the types and methods that should be visible without the need for separate facade code.

Flyweight
: Use sharing to support a large number of fine-grained objects efficiently. The emphasis on immutability in FP makes this straightforward to implement, as instances can be shared safely. An important set of examples are the persistent data structures, like `Vector` (see "What About Making Copies?" on page 222).

Proxy
: Provide a surrogate to another instance to control access to it. Immutability eliminates concerns about data corruption by clients.

Behavioral Patterns

This section describes patterns for collaboration between types to implement common interaction scenarios.

Chain of responsibility
: Avoid coupling a sender and receiver. Allow a sequence of potential receivers to try handling the request until the first one succeeds. Pattern matching and chaining partial functions support this pattern. Akka is great example of decoupling the sender and receiver.

Command
: Reify a request for service. This enables requests to be queued and supports undo, replay, etc. *Event-driven* and *message-driven* systems elevate this idea to an architectural principle. See, for example, the *Reactive Manifesto*. On a more "local" level, monadic collections are a good way to process commands sequentially using `for` comprehensions.

Interpreter
: Define a language and a way of interpreting expressions in the language. The term *DSL* emerged after the *Gang of Four* book. We discussed several approaches in Chapter 21.

Iterator
: Allow traversal through a collection without exposing implementation details. Almost all work with collections and other containers is done this way now, a triumph of functional thinking.

Mediator
: Avoid having instances interact directly by using a mediator to implement the interaction, allowing that interaction to evolve separately. Given instances is an interesting option here, where the value can be changed without forcing lots of other code changes. Similarly, message passing between Akka actors is mediated by the runtime system with minimal connections between the actors. While a specific `ActorRef` is needed to send a message, it can be determined through a query at runtime, without the need to hardcode dependencies programmatically, and it provides a level of indirection between actors.

Memento
: Capture an instance's state so it can be stored and used to restore the state later. Memoization is made easier by pure functions that are referentially transparent. A *decorator* could be used to add memoization, with the additional benefit that reinvocation of the function can be avoided when it is called with arguments previously used; the *memo* is returned instead.

Observer
: Set up a one-to-many dependency between a subject and observers of its state. When state changes occur, notify the observers. One of the more pervasive and successful patterns today, several variants are discussed throughout this book.

State
: Allow an instance to alter its behavior when its state changes. Functional programming provides deep, principled guidance about state management. Most of the time, values are immutable, so new instances are constructed to represent the new state. In principle, the new instance could exhibit different behaviors, although usually these changes are carefully constrained by a common supertype abstraction. The more general case is a state machine. We discussed in "Robust, Scalable Concurrency with Actors" on page 416 that Akka actors and the actor model in general can implement state machines in a principled, thread-safe way. Finally, monads are often used to encapsulate state.

Strategy
: Reify a family of related algorithms so that they can be used interchangeably. Higher-order functions make this easy. For example, when calling `fold` or `reduce`, the actual accumulator used to aggregate elements is specified by the caller using a function.

Template method
: Define the skeleton of an algorithm as a final method, with calls to other methods that can be overridden in subtypes to customize the behavior. This is one of my favorite patterns because it is far more principled and safe than overriding concrete methods, as discussed in "Overriding Methods? The Template Method Pattern" on page 251. Note that an alternative to defining abstract methods for overriding is to make the template method a higher-order function and then pass in functions to do the customization.

Visitor
: Insert a protocol into an instance so that other code can access the internals for operations that aren't supported by the type. This is the worst pattern in the catalog because it breaks a type's abstraction and complicates the implementation. Fortunately, we have far better options now. Defining an `unapply` or `unapplySeq` method lets the type designer define a low-overhead protocol for exposing only the internal state that is appropriate. Pattern matching uses this feature to extract these values and implement new functionality. Extension methods and type classes are another way of adding new behaviors to existing types in a principled way, although they don't provide access to internals that might be needed in special cases.

Better Design with Design by Contract

Our types make statements about allowed states for our programs. We use test-driven development (TDD) or other test approaches to verify behaviors that our types can't specify. Well before TDD and FP went mainstream, Bertrand Meyer described an approach called *Design by Contract* (DbC), which he implemented in the Eiffel language. TDD largely replaced interest in DbC, but the idea of contracts between clients and services is a very useful metaphor for thinking about design. We'll mostly use DbC terminology in what follows.

A contract of a module can specify three types of conditions:

Preconditions
: What constraints exist for inputs passed to a module in order for it to successfully perform its purpose? Preconditions constrain what *clients* of the module can do.

Postconditions
: What guarantees does the module make to the client about its results, assuming the preconditions were satisfied? Postconditions constrain the module.

Invariants
: What must be true before and after an invocation of the module?

These contractual constraints must be specified as code so they can be enforced automatically at runtime. If a condition fails, the system terminates immediately, forcing you to find and fix the underlying cause before continuing. If that sounds harsh, relaxing this requirement means contract failures are easy to ignore, undermining their value.[1]

It's been conventional to only test the conditions during testing, but not production, both to remove the extra overhead and to avoid crashing in production if a condition fails. Note that the "let it crash" philosophy of the actor model turns this on its head. If a condition fails at runtime, shouldn't it crash and let the runtime trigger recovery?

Scala provides several variants of `assert` that can be used to support Design by Contract in `Predef`: `assert`, `assume`, `require`, and `ensuring`. The following example shows how to use `require` and `ensuring` for contract enforcement:

```
// src/main/scala/progscala3/appdesign/dbc/BankAccount.scala
package progscala3.appdesign.dbc

import scala.annotation.targetName

case class Money(val amount: Double):                                    ❶
  require(amount >= 0.0, s"Negative amount $amount not allowed")

  @targetName("plus")  def +  (m: Money): Money = Money(amount + m.amount)
  @targetName("minus") def -  (m: Money): Money = Money(amount - m.amount)
  @targetName("ge")    def >= (m: Money): Boolean = amount >= m.amount

  override def toString = "$"+amount

case class BankAccount(balance: Money):
  def debit(amount: Money) =                                             ❷
    require(balance >= amount,
      s"Overdrafts are not permitted, balance = $balance, debit = $amount")
    (BankAccount(balance - amount)).ensuring(
      newBA => newBA.balance == this.balance - amount)

  def credit(amount: Money) = BankAccount(balance + amount)              ❸

import scala.util.Try
```

1 I speak from experience here.

```
@main def TryBankAccount: Unit =
  Seq(-10, 0, 10) foreach (i => println(f"$i%3d: ${Try(Money(i.toDouble))}"))

  val ba1 = BankAccount(Money(10.0))
  val ba2 = ba1.credit(Money(5.0))
  val ba3 = ba2.debit(Money(8.5))
  val ba4 = Try(ba3.debit(Money(10.0)))

  println(s"""
    |Initial state: $ba1
    |After credit of $$5.0: $ba2
    |After debit of $$8.5: $ba3
    |After debit of $$10.0: $ba4""".stripMargin)
```

❶ Encapsulate money, only allowing positive amounts using `require`, a precondition. `Money` and `BankAccount` could also be implemented as opaque type aliases or value classes if we were concerned about the overhead of these wrapper classes.

❷ Don't allow the balance to go negative. This is really an invariant condition of `BankAccount`, which is verified on entry with `require` and indirectly on output when a new `Money` instance is created for the changed balance. The deduction math is verified with `ensuring`, which takes the return value of the preceding block as an argument and returns it unchanged, unless the predicate fails.

❸ No contract violations are expected to occur, at least in this simple example without transactions, and so forth.

Running `runMain progscala3.appdesign.dbc.TryBankAccount`, we get the following output:

```
-10: Failure(java.lang.IllegalArgumentException: requirement failed:
     Negative amount -10.0 not allowed)
  0: Success($0.0)
 10: Success($10.0)

Initial state: BankAccount($10.0)
After credit of $5.0: BankAccount($15.0)
After debit of $8.5: BankAccount($6.5)
After debit of $10.0: Failure(...: requirement failed:
  Overdrafts are not permitted, balance = $6.5, debit = $10.0)
```

Each of the `assert`, `assume`, `require`, and `ensuring` methods have two overloaded versions, like this pair for `assert`:

```
final def assert(assertion: Boolean): Unit
final def assert(assertion: Boolean, message: => Any): Unit
```

In the second version, if the predicate argument is false, the message is converted to a `String` and used as part of the exception message.

The `assert` and `assume` methods behave identically. The names signal different intent. Both throw `AssertionError` on failure, and both can be completely removed from the byte code if you compile with the option `-Xelide-below assertion`.[2] `assertion` and other integer values are defined in the corresponding companion object. The `ensuring` methods call `assert`, so their conditional logic will be removed if assertions are elided. However, the body of code for which `ensuring` is invoked will not be elided.

The `require` methods are intended for testing method parameters (including constructors). They throw `IllegalArgumentException` on failure, and their code generation is not affected by the `-Xelide-below` option. Therefore, in our `Money` and `BankAccount` types, the `require` checks will never be turned off, even in production builds that turn off `assert` and `assume`. If that's not what you want, use one of those methods instead.

> Since calls to `assert` and `assume` can be completely removed by the compiler, do not put any logic in the conditional argument that must always be evaluated at runtime.

You can mark your own methods with the annotation `scala.annotation.elidable` and a constant value like `assertion` to suppress code generation. See the example *src/main/scala/progscala3/appdesign/dbc/Elidable.scala* in the code repo. See also a macro implementation of an `invariant` construct in "Macros" on page 496.

Type system enforcement is ideal, but the Scala type system can't enforce all constraints we might like. Hence, TDD (or variants) and assertion checks inspired by Design by Contract will remain useful tools for building correct software.

The Parthenon Architecture

Object-oriented programming emphasized the idea of mimicking domain language in code. A shared domain language is essential for discussions between all team members—business stakeholders as well as developers. However, faithfully implementing all domain concepts in code makes the applications bloated and harder to evolve. Some ideas should be expressed in code, while other concepts should be

---

2 At the time of this writing, Scala 3 does not support Scala 2's `-Xelide-below`, but this should be implemented in a subsequent release.

expressed in data, where the code remains agnostic. When I'm calculating payroll, do I really need to know I'm working with an `Employee` or is it sufficient to have `Money` instances for their salary and `Percentage`s for their tax deductions?

Functional programming provides useful guidance. While some domain types are useful for programmer comprehension, the real benefits come from contract enforcement, like the `Dollars`, `Money`, and `Percentage` types we've seen previously. Concepts with precise algebraic properties and other well-defined behaviors are good candidates for types.

The problem with implementing many real-world domain concepts is their inherent contextual nature, meaning, for example, that your idea of an `Employee` is different from mine because you have different use cases to implement than I do. Also, these domain concepts are fragile, subject to frequent change, especially as use cases are added and evolve.

If we boil our problems down to their essence, we have a bunch of numbers, dates, strings, and other fundamental types that we need to ingest from a data store, process according to some specific rules governed by tax law or other requirements, and then output the results. All programs are CRUD (create, read, update, and delete). I'm exaggerating, but only a little bit. Too many applications have far too many layers for the conceptually simple work they actually do.

The rules I follow for deciding whether or not to implement a domain concept in code are the following:

- The concept clarifies human understanding of the code.
- The concept improves encapsulation significantly.
- The concept has well-defined properties and behaviors.
- The concept improves overall correctness.

Otherwise, I'm much more likely to use generic container types like tuples, maps, and sequences.

`Money` is a good candidate type because it has well-defined properties. With a `Money` type, I can do algebra and enforce rules that the enclosed `Double` or `BigDecimal` is nonnegative, that arithmetic and rounding are done according to standard accounting rules, that `toString` shows the currency symbol, and so forth.

I might use opaque type aliases or value classes for types where efficiency is highly important.

I resist adding too many methods to my types. Instead, I use extension methods or type classes when extra behavior is needed in limited contexts. Or, I'll do pattern matching on instances with custom handling for the intended purpose.

But is there more we can do to gain the benefits of the domain language without the drawbacks? I've been thinking about an architectural style that tries to do just that.

> The following discussion is a sketch of an idea that is mostly theoretical and untested.

It combines four layers:

A DSL for the ubiquitous language
: It is used to specify use cases. The UI design is here, too, because it is also a tool for communication and hence a language.

A library for the DSL
: The implementation of the DSL, including the types implemented for some domain concepts, the UI, etc.

Use case logic
: Functional code that implements each use case. It remains as focused and concise as possible, relying primarily on standard library types, and a bare minimum of the domain-oriented types. Because this code is so concise, most of the code for each use case is a single vertical slice through the system. I don't worry too much about duplication here, but some extraction of reusable code occurs organically. If the code is concise and quick to write, I can easily throw it away when I want to rewrite it or I no longer need it.

Core libraries
: The Scala standard library, Typelevel libraries, Akka, and APIs for logging, database access, etc., plus a growing library of reusable code extracted from the use case implementations.

The picture that emerges reminds me of classical buildings because of the columns of code that implement each use case. So I'll be pretentious and call it *The Parthenon Architecture* (see Figure 23-1).

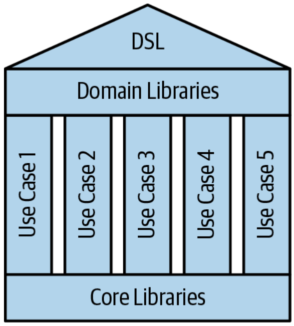


Figure 23-1. The Parthenon Architecture

Working from the top down, end users implement each use case using the DSL, the *pediment*. Next, the *entablature* represents the library of domain concepts, including the DSL implementation and UI. Below that, the *columns* represent the use case implementations created by users. Finally, the temple *foundation* represents the reusable core libraries.

The functional code for each use case should be very small, like many of the examples in this book, so that trivial duplication is not worth the cost of removal. Instead, the simple, in-place data flow logic is easy to understand, test, and evolve, or completely replace when that's easiest. It won't always work out that way, but places where duplication should be removed will suggest themselves, gradually building up the core libraries.

Finally, you could deploy each use case implementation in its own container, Kubernetes *pod*, etc. This is especially useful when you need to migrate from one version of the code to another, but you can't migrate all at once. If there is minimal coupling between use cases, such as a stable REST API, then it's easier to upgrade some use case deployments while others remain unchanged. You can even have concurrent versions of the same use case, when necessary.

Let's sketch an example that builds upon the payroll DSL from "External DSLs with Parser Combinators" on page 446. This time, we'll read comma-separated records for a list of employees, create strings from each record in the DSL format, then parse each DSL string to process the data. It feels a little convoluted going through the DSL string format, but the DSL is how stakeholders will provide the data. Finally, we'll implement two separate use cases: a report with each employee's pay and a report showing the totals for the pay period.

First, here is the code that implements the *domain library*, to use my terminology in the temple image:

```
// src/main/scala/progscala3/appdesign/parthenon/PayrollCalculator.scala
package progscala3.appdesign.parthenon
```

```
import progscala3.dsls.payroll.parsercomb.dsl.PayrollParser
import progscala3.dsls.payroll.*
import progscala3.contexts.accounting.*

object PayrollCalculator:                                          ❶
  val dsl = """biweekly {
      federal tax          %f  percent,
      state tax            %f  percent,
      insurance premiums   %f  dollars,
      retirement savings   %f  percent
    }"""

  case class Pay(                                                  ❷
    name: String, salary: Dollars, deductions: Deductions)

  def fromFile(inputFileName: String): Seq[Pay] =                  ❸
    val data = readData(inputFileName)
    for
      (name, salary, ruleString) <- data
    yield Pay(name, salary, toDeductions(ruleString))

  case class BadInput(message: String, input: String)
    extends RuntimeException(s"Bad input data, $message: $input")

  private type Record = (String, Dollars, String)                  ❹

  private def readData(inputFileName: String): Seq[Record] =
    for
      line <- scala.io.Source.fromFile(inputFileName).getLines.toVector
      if line.matches("\\s*#.*") == false    // skip comments
    yield toRule(line)

  private def toRule(line: String): Record =                       ❺
    line.split("""\s*,\s.*""") match
      case Array(name, salary, fedTax, stateTax, insurance, retirement) =>
        val ruleString = dsl.format(
          fedTax.toDouble, stateTax.toDouble,
          insurance.toDouble, retirement.toDouble)
        (name, Dollars(salary.toDouble), ruleString)
      case array => throw BadInput("expected six fields", line)

  private val parser = PayrollParser()                             ❻
  private def toDeductions(rule: String): Deductions =
    parser.parseAll(parser.biweekly, rule).get
```

❶ An object to hold the library code for payroll calculation. Note we use the string-based (external) DSL.

❷ A case class for each person's pay. The only new domain type we define here, where it's convenient to have good names for the three fields (versus using a tuple). We're reusing `Dollars`, `Percentage`, and `Deductions` from before.

❸ A utility function to read data from a comma-separated data file, skipping comment lines that start with `#`. It uses the private helper methods `readData` and `toRule`.

❹ An internal type definition for making the code more concise.

❺ The helper method that converts a comma-separated string of data into the expected DSL format. (It is not very forgiving about input errors!)

❻ The `PayrollParser` from the DSL chapter, used to convert a DSL-formatted rule string into a `Deductions` object.

Now we can use this library code to implement two use cases: calculate biweekly payroll for each employee and a report of biweekly totals (put in one file for convenience):

```
// src/main/scala/progscala3/appdesign/parthenon/PayrollUseCases.scala
package progscala3.appdesign.parthenon
import progscala3.dsls.payroll.parsercomb.dsl.PayrollParser
import progscala3.contexts.accounting.*

object PayrollUseCases:
  import PayrollCalculator.{fromFile, Pay}
  val fmt  = "%-10s %8.2f  %8.2f     %5.2f\n"
  val head = "%-10s %-8s  %-8s  %s\n"

  def biweeklyPayrollPerEmployee(data: Seq[Pay]): Unit =          ❶
    println("\nBiweekly Payroll:")
    printf(head, "Name", "   Gross", "     Net", "Deductions")
    println("-----------------------------------------")
    for
      Pay(name, salary, deductions) <- data
      gross = deductions.gross(salary)
      net   = deductions.net(salary)
    do printf(fmt, name, gross.amount, net.amount, (gross - net).amount)

  def biweeklyPayrollTotalsReport(data: Seq[Pay]): Unit =
    val (gross, net) = data.foldLeft(Dollars.zero -> Dollars.zero) {
      case ((gross, net), Pay(_, salary, deductions)) =>
      val g = deductions.gross(salary)
      val n = deductions.net(salary)
      (gross + g, net + n)
    }
    println("-----------------------------------------")
    printf(fmt, "Totals", gross.amount, net.amount, (gross - net).amount)

  @main def RunPayroll(inputFileNames: String*): Unit =
    val files =
      if inputFileNames.length == 0 then Seq("misc/parthenon-payroll.txt")
      else inputFileNames
```

```
    for (file <- files) do
      println(s"Processing input file: $file")
      val data = fromFile(file)
      biweeklyPayrollPerEmployee(data)
      biweeklyPayrollTotalsReport(data)
```

❶ The two use cases are implemented by `biweeklyPayrollPerEmployee` and `biweeklyPayrollTotalsReport`, respectively.

By default, it loads a data file in the `misc` directory, but you can pass another file as a parameter. If you run it in `sbt`, you get the following output:

```
> runMain progscala3.appdesign.parthenon.RunPayroll
...
Biweekly Payroll:
Name            Gross       Net  Deductions
-----------------------------------------
Joe CEO       7692.31   5184.62     2507.69
Jane CFO      6923.08   4457.69     2465.38
Phil Coder    4615.38   3080.77     1534.62
-----------------------------------------
Totals       19230.77  12723.08     6507.69
```

This rough sketch illustrates how the actual use case implementations can be small, independent columns of code. They use a few domain concepts from the entablature domain library and the foundation core Scala collections.

Recap and What's Next

We examined several pragmatic issues for application development, including Design Patterns and Design by Contract. We explored an architecture model I've been considering, which I pretentiously called The Parthenon Architecture.

Now it's time to look at Scala's facilities for reflection and metaprogramming.

CHAPTER 24

Metaprogramming: Macros and Reflection

Metaprogramming is programming that manipulates programs as data. In some languages, the difference between *programming* and *metaprogramming* isn't all that significant. Lisp dialects, for example, use the same *S-expression* representation for code and data, a property called *homoiconicity*. Dynamically typed languages like Python and Ruby make it easy to manipulate the program with other code, sometimes derisively called *monkey patching*. In statically typed languages like Java and Scala, metaprogramming is more constrained and less common. It's still useful for solving many advanced design problems, but more formality is required to separate compile-time versus runtime manipulation.

Metaprogramming comes in many forms. The word *reflection* refers to introspection of code at runtime, such as asking a value or type for metadata about itself. The metadata typically includes details about the type, methods and fields, etc.

Scala macros work like constrained compiler plug-ins because they manipulate the *abstract syntax tree* (AST) produced from the parsed source code. Macros are invoked to manipulate the AST before the final compilation phases leading to byte-code generation.

3 While Scala 2 had a metaprogramming system, called *Scalameta*, it was always considered experimental, even though it was widely used by library writers for advanced scenarios. Scala 3 introduces a new macro system, which is not considered experimental. Replacing Scala 2 macros with Scala 3 implementations is the biggest challenge some library maintainers face when migrating. If this affects you, see the guidance in the Scala 3 Migration Guide.

3 The Scala 3 metaprogramming documentation describes five fundamental features that support metaprogramming:

Inline
: The `inline` modifier directs the compiler to inline the definition at the point of use. Inlining reduces the overhead of method or function invocation and accessing values, but it can also greatly expand the overall size of the byte code, if the definition is used in many places. However, when used with conditionals and match clauses involving compile-time constants, inlining will remove the unused branches. Inlining happens early in the compilation process, in the *typer phase*, so that the logic can be used in subsequent phases for type-level programming (such as match types) and macro expansion.

Macros
: A combination of *quotation*, where sections of code are converted to a tree-like data structure, and *splicing*, which goes the other way, converting quotations back to code. Used with `inline` so the macros are applied at compile time.

Staging
: The runtime analog of macro construction of code. Staging also uses quotes and splices, but not `inline`. The term *staging* comes from the idea that manipulating the code at runtime breaks execution into multiple stages, intermixing stages of normal processing and metaprogramming.

TASTy reflection
: TASTy is the intermediate representation Scala 3 compilers generate. It enables richer introspection of code, better interoperation among modules compiled with different versions of Scala, and other benefits. TASTy reflection yields a typed abstract syntax tree (the origin of the name), which is a "white-box" representation of code versus the "black-box" view provided by quotations.

TASTy inspection
: When the syntax trees are serialized to binary files, they are given the extension *.tasty*. TASTy inspection provides a way to inspect the contents of those files.

These features are complemented by other constructs we've already explored, like match types ("Match Types" on page 369) and by-name context parameters ("Context Bounds" on page 167). It's a good idea to use metaprogramming to solve design problems as a last resort.

This chapter provides a brief introduction to the inline, macros, and staging features. For TASTy reflection and introspection, as well as additional documentation on Scala 3 metaprogramming, see the metaprogramming section of the Scala 3 Migration Guide.

However, let's first begin with some of the other compile-time and runtime reflection tools available.

Scala Compile Time Reflection

The `scala.compiletime` package, new in Scala 3, provides compile-time tools. We looked in depth at the `scala.compiletime.ops` operations in "Dependent Typing" on page 374. We met `uninitialized` in "Using try, catch, and finally Clauses" on page 90.

The `summonFrom` and `summonAll` methods extend the capabilities of `Predef.summon` that we've used before. First, `summonFrom`:

```
// src/script/scala/progscala3/meta/compiletime/SummonFrom.scala
import scala.compiletime.summonFrom

trait A; trait B

inline def trySummonFrom(label: String, expected: Int): Unit =   ❶
  val actual = summonFrom {
    case given A => 1
    case given B => 2
    case _ => 0
  }
  printf("%-9s trySummonFrom(): %d =?= %d\n", label, expected, actual)

def tryNone = trySummonFrom("tryNone:", 0)                        ❷

def tryA =                                                       ❸
  given A with {}
  trySummonFrom("tryA:", 1)

def tryB =
  given B with {}
  trySummonFrom("tryB:", 2)

def tryAB =
  given A with {}
  given B with {}
  trySummonFrom("tryAB:", 1)

tryNone; tryA; tryB; tryAB
```

❶ Example of `summonFrom`. You pattern match to determine which given instances are in scope, with a default cause to avoid a match error if none is in scope.

❷ Test the case where no given is in scope.

❸ Three methods to test when a given `A` or `B` is in scope, or both of them.

The last line prints the following:

```
scala> tryNone; tryA, tryB, tryAB
tryNone:  trySummonFrom(): 0 =?= 0
tryA:     trySummonFrom(): 1 =?= 1
tryB:     trySummonFrom(): 2 =?= 2
tryAB:    trySummonFrom(): 1 =?= 1   // Matched A first.
```

Similarly, `summonAll` returns all givens corresponding to types in a tuple:

```
// src/script/scala/progscala3/meta/compiletime/SummonAll.scala
scala> import scala.compiletime.summonAll

scala> trait C; trait D; trait E
     | given c: C with {}
     | given d: D with {}

scala> summonAll[C *: D *: EmptyTuple]
val res0: C *: D *: EmptyTuple = (...c...,...d...)  // "REPL noise" omitted.

scala> summonAll[C *: D *: E *: EmptyTuple]
1 |summonAll[C *: D *: E *: EmptyTuple]
  |^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |cannot reduce summonFrom with
  | patterns :  case t @ _:E
  | ...
```

The last line fails to compile because no given is found for `E`.

In the code examples, the *src/script/scala/progscala3/meta/compiletime* directory has a few other scripts demonstrating other `scala.compiletime` features that I won't discuss here. Some of those features are especially useful when working with `inline` code, discussed ahead.

Java Runtime Reflection

On the JVM, Java reflection is also available. First, here are types we can analyze:

```
// src/main/scala/progscala3/meta/reflection/JReflect.scala
package progscala3.meta.reflection

object JReflect:
  trait T[A]:
    val vT: A
    def mT = vT

  case class C[B](b: B) extends T[String]:
    val vT = "T"
    val vC = "C"
    def mC = vC
    class C2
```

Note that Java syntax is used for method names and such:

```
// src/script/scala/progscala3/meta/reflection/JReflect.scala

scala> import progscala3.meta.reflection.JReflect

scala> def as(array: Array[?]): String = array.mkString("[", ", ", "]") ❶

scala> val clazz = classOf[JReflect.C[Double]]                          ❷
val clazz: Class[progscala3.meta.reflection.JReflect.C[Double]] =
  class progscala3.meta.reflection.JReflect$C                           ❸

scala> clazz.getName
     | clazz.getModifiers
val res0: String = ...JReflect$C
val res1: Int = 9                                                       ❹

scala> val sup = clazz.getSuperclass
val sup: Class[? >: ...JReflect.C[Double]] = class java.lang.Object
scala> as(clazz.getTypeParameters)
     | as(clazz.getClasses)
     | as(clazz.getInterfaces)
val res2: String = [B]
val res3: String = [class ...JReflect$C$C2]
val res4: String = [interface ...JReflect$T,
  interface scala.Product, interface java.io.Serializable]

scala> as(clazz.getConstructors)
     | as(clazz.getMethods)
val res5: String = [public ...JReflect$C(java.lang.Object)]
val res6: String = [...]                                                ❺

scala> as(clazz.getFields)
     | as(clazz.getAnnotations)
val res7: String = []                                                   ❻
val res8: String = []
```

❶ A helper method to convert an array to a readable string.

❷ `Predef.classOf[T]` retrieves the Java `Class` object. Java syntax is `T.getClass()`.

❸ The package prefixes are shown here but elided in the subsequent output.

❹ Decode this value using `java.lang.reflect.Modifier`.

❺ A long list of elided methods, including `mT` and `mC`.

❻ It doesn't recognize the fields in `T` or `C`!

Scala Reflect API

The `scala.reflect` package expands on Java runtime reflection. We encountered a few members of this package already. In "Structural Types" on page 362, we used `scala.reflect.Selectable`. All Scala 3 `enums` are subtypes of `scala.reflect.Enum`.

Sometimes we need a context bound to provide a factory for a specific type. In "Constraining Allowed Instances" on page 175, we used given `Conversion` instances to convert from one type to another.

Another example is used by the standard library to allow us to construct arrays using parameterized methods—e.g., `makeArray[T](elems: T*): Array[T]`. This is trickier than it might look because Scala arrays are Java arrays, which don't permit abstracting over the type. In Scala, we can work around this limitation using `scala.reflect.ClassTag`. Here is an example adapted from the `ClassTag` documentation:

```
// src/script/scala/progscala3/meta/reflection/MakeArray.scala

scala> import scala.reflect.ClassTag

scala> def makeArray[T : ClassTag](elems: T*) = Array[T](elems*)   (1)

scala> makeArray(1, 2, 3)
     | makeArray(1.1, 2.2, 3.3)
     | makeArray("one", "two", "three")
     | makeArray("one" -> 1, "two" -> 2, "three" -> 3)
     | make(1, 2.2, 3L)
val res0: Array[Int] = Array(1, 2, 3)
val res1: Array[Double] = Array(1.1, 2.2, 3.3)
val res2: Array[String] = Array(one, two, three)
val res3: Array[(String, Int)] = Array((one,1), (two,2), (three,3))
val res4: Array[AnyVal] = Array(1, 2.2, 3)
```

(1) `T` must have a context-bound `ClassTag[T]` in scope.

The method calls one of the overloaded `Array.apply` methods, where there is one for each of the `AnyVal` types and one for all `AnyRef` types. These methods also require a `ClassTag` context bound on `T`, which they use to construct the array exactly as Java expects. For example, for `Int`s, an `int[]` is constructed with no boxing of the elements and hence no heap allocation for them. For an `AnyRef` type like `String`, a `String[]` is constructed where each element `String` is allocated on the heap.

However, this technique only works when constructing a new array (or a similar data structure) from elements. When a method is passed a previously constructed instance of a parameterized type, the crucial type information is already "erased." This is an issue if you're passing collections around and somewhere deep in the stack some method wants to use a ClassTag for introspection. Unless the collection and a corresponding ClassTag were passed around together, you're out of luck. However, we'll come back to this issue a little later.

Hence, ClassTags can't resurrect type information from byte code, but they can be used to capture and exploit type information before it is erased.

Type Class Derivation: Implementation Details

A specific form of reflection is the scala.deriving.Mirror trait that can be used for type class derivation, first discussed in "Type Class Derivation" on page 158, where we saw that the derives keyword causes the compiler to automatically instantiate certain type classes for us, such as CanEqual.

For the compiler to be able to generate a type class TC instance for some specific type A, the compiler needs the ability to introspect the structure of A and use that information to construct the TC instance for A. This information can be provided using the Mirror trait and its subtypes. Mirror is implemented as a type class itself, and the compiler can generate instances of it automatically for enums and enum cases, case classes and case objects, and sealed classes or traits that have only case classes and case objects as subtypes.

For information about how Mirror is used in derivation, along with a concrete example, see the derivation documentation.

Scala 3 Metaprogramming

Let's now return to three of the five features listed at the start of this chapter for Scala 3 metaprogramming: inline, macros, and staging. We'll evolve an example, a tool for enforcing invariants. In "Better Design with Design by Contract" on page 474, we mentioned one aspect of a contract is the invariants that should hold before and after evaluation of a block of code. Let's implement invariant enforcement.

Inline

The inline modifier directs the compiler to inline the definition at the point of use. Let's use it for our first version of an invariant enforcer:

```
// src/main/scala/progscala3/meta/Invariant1.scala
package progscala3.meta

object invariant1:
  inline val ignore = false                                    ❶

  /**
   * Throw an exception if the predicate is false before the block is
```

```
 * evaluated, then evaluate the block, then check the predicate again.
 * If all predicate checks pass, then return the block's value.
 */
inline def apply[T](                                                   ❷
    inline predicate: => Boolean)(
    inline block: => T): T =
  inline if !ignore then                                               ❸
    if !predicate then throw InvariantFailure("before")
    val result = block
    if !predicate then throw InvariantFailure("after")
    result
  else
    block                                                              ❹

case class InvariantFailure(beforeAfter: String) extends RuntimeException(
  s"FAILURE! predicate failed $beforeAfter evaluation!")

@main def TryInvariant1 =
  var i = 0
  invariant1(i >= 0)(i += 1)
  println(s"success: $i")
  println(s"Will now fail:")
  invariant1(i >= 0)(i -= 2)                                           ❺
```

❶ When not testing and you want to eliminate the overhead of the two predicate checks, compile with this global flag to `true`.

❷ Pass a `predicate` that is checked twice, before and after the `block` is evaluated. Note that both are by-name parameters.

❸ Only evaluate the predicate around the block if not ignored. The reason for the `inline` here will be discussed.

❹ Otherwise, just evaluate the block.

❺ Raises an `InvariantFailure` exception with the message "…after evaluation!"

The `inline` modifier on the value `ignore` means `true` or `false` and is inlined wherever it is used. The byte code won't contain the `ignore` field.

Furthermore, because we inline the conditional `if !ignore then…`, the whole conditional expression itself will be replaced with either the `then` or `else` branch, depending on whether the expression is true or false, because the compiler knows at compile time which branch will be taken.

Finally, the `apply` method body is also inlined where invoked, since `ignore` and the conditional are both inlined. Therefore, `if !ignore then…else…` reduces to either the `then` clause or the `else` clause. Specifically, if `ignore` is `false`, the entire `invariant1(…)(…)` is inlined to the following:

```
if !predicate then throw InvariantFailure("before")
val result = block
if !predicate then throw InvariantFailure("after")
result
```

This is further transformed because both the `predicate` and `block` are themselves declared `inline`. For example:

```
var i=0
invariant(i >= 0)(i += 1)
```

This finally results in this code:

```
if !(i >= 0) then throw InvariantFailure("before")
val result = i += 1
if !(i >= 0) then throw InvariantFailure("after")
result    // The type is Unit in this case
```

If `ignore` is `true`, then the whole body reduces to the content of `block`.

If we declared `ignore` as a `var` (which can't be inlined), it would enable convenient change at runtime. However, we would lose most of the inlining. We're giving up the convenience of runtime change for the smaller code footprint and better performance made possible with extensive inlining.

Actually, I'm not certain that inlining `predicate` and `block` are always beneficial. If they are large blocks of code, probably not. In your code base, you might experiment with inlining and not inlining these expressions to see what differences you observe, if any, in compile times, output byte code sizes, and runtime performance.

This is a nice tool already, but if an invariant test fails at runtime, we get a not very informative error message:

```
[error] ...$InvariantFailure: FAILURE! predicate failed after evaluation!
[error] ...
```

Sure, we'll have the stack trace from the thrown exception, but we'll really want to see a lot more information about what actually failed and why. That's were macros come in, as we'll see shortly, but first let's finish our discussion of `inline`.

The `inline` keyword is a soft modifier, so the word can be used as an identifier in other contexts.

Inline methods can be recursive, but care is required when invoking them:

```
// src/script/scala/progscala3/meta/inline/Recursive.scala

scala> inline def repeat(s: String, count: Int): String =
     |   if count == 0 then ""
```

```
     |   else s + repeat(s, count-1)

scala> repeat("hello", 3)
val res0: String = hellohellohello

scala> var n=3
var n: Int = 3

scala> repeat("hello", n)
1 |repeat("hello", n)
  |^^^^^^^^^^^^^^^^^^
  |Maximal number of successive inlines (32) exceeded,
  |Maybe this is caused by a recursive inline method?
  |You can use -Xmax-inlines to change the limit.
  | This location contains code that was inlined from rs$line$14:1
  | This location contains code that was inlined from rs$line$11:3
  ...
```

All invocations of `repeat` must pass a compile-time constant for `count`!

Inline methods can override or implement noninline methods:

```
// src/script/scala/progscala3/meta/inline/Overrides.scala

trait T:
  def m1: String
  def m2: String = m1

object O extends T:
  inline def m1 = "O.m1"
  override inline def m2 = m1 + " called from O.m2"

val t: T = O
assert(O.m1 == t.m1)
assert(O.m2 == t.m2)
```

Method dispatch works normally as it does for noninlined methods. Even though `t` is of type `T`, the inlined implementations of `O.m1` and `O.m2` are invoked.

Abstract methods can be declared inline, but the implementations must also be inline. However, the abstract method can't be invoked directly, unlike the previous example:

```
trait T2:
  inline def m: String

object O2 extends T2:
  inline def m: String = "O2.m"

val t2: T2 = O2
O2.m
t2.m        // ERROR
```

The last line produces the error "Deferred inline method m in trait T2 cannot be invoked."

If an inline declaration is also declared `transparent`, the compiler can return a more specific type than the code is declared to return:

```
// src/script/scala/progscala3/meta/inline/Transparent.scala

scala> open class C1
     | class C2 extends C1:
     |   def hello = "hello from C2"

scala> transparent inline def make(b: Boolean): C1 = if b then C1() else C2()

scala> val c1: C1 = make(true)              ❶
     | // c1.hello                           // C1.hello doesn't exist!
val c1: C1 = C1@28548fae

scala> val c2: C2 = make(false)             ❷
     | c2.hello                              // Allowed!
val c2: C2 = C2@7bdea065
val res0: String = hello from C2
```

❶ The declared type and actual type are both `C1`, as would be the case for nontransparent and noninline code.

❷ Even though `make` is declared to return a `C1`, the compiler allows us to declare the returned value to be `C2` because this is in fact true and determined at compile time. This lets us call `c2.hello`. A compilation type error would result for the declaration of `c2` if `make` weren't declared transparent.

We saw in Chapter 17 several other ways to declare methods that return specific types based on dependent typing. Using `transparent` is another way to achieve this goal for the specific case of subtypes in a hierarchy.

Recall from the preceding `repeat` example that we encountered an error exceeding the maximum number of inlines allowed. Note what happens if we define a new version that adds `inline` before the conditional expression:

```
// src/script/scala/progscala3/meta/inline/ConditionalMatch.scala

scala> inline def repeat2(s: String, count: Int): String =
     |   inline if count == 0 then ""          // <-- inline added here.
     |   else s + repeat2(s, count-1)

scala> repeat2("hello", 3)    // Okay
val res0: String = hellohellohello

scala> var n=3
     | repeat2("hello", n)    // ERROR!
```

```
9 |repeat2("hello", n)    // ERROR!
  |^^^^^^^^^^^^^^^^^^^
  |Cannot reduce `inline if` because its condition is not a constant value:
  ...
```

This is a little better than the previous error.

Finally, `match` expressions can be marked inline, if there is enough static information to decide which branch to take. Only the code for that branch will be inlined in the generated byte code:

```
scala> inline def repeat3(s: String, count: Int): String =
     |   inline count match
     |     case 0 => ""
     |     case _ => s + repeat3(s, count-1)
     |

scala> repeat3("hello", 3)    // Okay
val res13: String = hellohellohello

scala> var n=3
     | repeat3("hello", n)    // ERROR!
1 |repeat3("hello", n)    // ERROR!
  |^^^^^^^^^^^^^^^^^^^
  |Maximal number of successive inlines (32) exceeded,
  |Maybe this is caused by a recursive inline method?
  |You can use -Xmax-inlines to change the limit.
  | This location contains code that was inlined from rs$line$29:1
  ...
```

If you're unsure when an expression is constant or not, you can use one of the `scala.compiletime.{constValue, constValueOpt, constValueTuple}` methods. (See examples in the code repo in *src/script/scala/progscala3/meta/compiletime*.)

So you can see that `inline` is a powerful tool, but it requires some care to use it effectively. It works at compile time, which constrains inlined methods to accept only constant arguments, inlined `if` statements to evaluate only constant predicates, and inlined `match` clauses to match only on constant values. Also, remember that inlining code can create byte code bloat. For many more details on `inline`, including other uses for the `scala.compiletime` features, see the Dotty documentation for `inline`. The book's examples contain additional inline (and macro) examples in *src/main/scala/progscala3/meta*.

Macros

Scala's previous, experimental macro system was used to implement clever solutions to difficult design problems in many advanced libraries. However, to use it required advanced knowledge. The new Scala 3 macro system offers nearly the same level of power but is more approachable.

Macros are built using two complementary operations: quotation and splicing. Quotation converts a code expression into a typed abstract syntax tree representing the expression, an instance of type `scala.quoted.Expr[T]`, where `T` is the type of the expression. For a type T itself, quotation returns a type structure for it of type `scala.quoted.Type[T]`. These trees and type structures can be manipulated to build new expressions and types.

Splicing goes the opposite direction, converting a syntax tree `Expr[T]` to an expression of type `T` and a type structure `Type[T]` to a type `T`.

The syntax for expression quotation is `'{…}`. For types, it is `'[…]`. The splicing syntax is `${…}`, analogous to the way we do string interpolation. Identifiers can be quoted (`'expr`) or spliced (`$expr`) within larger quote or splice expressions.

Let's return to our invariant example and use quotes and splices to create a better error message:

```
// src/main/scala/progscala3/meta/Invariant.scala
package progscala3.meta
import scala.quoted.*                                                    ❶

object invariant:
  inline val ignore = false

  inline def apply[T](
      inline predicate: => Boolean, message: => String = "")(            ❷
      inline block: => T): T =
    inline if !ignore then
      if !predicate then fail(predicate, message, block, "before")       ❸
      val result = block
      if !predicate then fail(predicate, message, block, "after")
      result
    else
      block

  inline private def fail[T](
      inline predicate: => Boolean,
      inline message: => String,
      inline block: => T,
      inline beforeAfter: String): Unit =
    ${ failImpl('predicate, 'message, 'block, 'beforeAfter) }            ❹

  case class InvariantFailure(msg: String) extends RuntimeException(msg)

  private def failImpl[T](
      predicate: Expr[Boolean], message: Expr[String],
      block: Expr[T], beforeAfter: Expr[String])(
      using Quotes): Expr[String] =
    '{ throw InvariantFailure(                                           ❺
      s"""FAILURE! predicate "${${showExpr(predicate)}}" """
```

```
      + s"""failed ${$beforeAfter} evaluation of block:"""
      + s""" "${${showExpr(block)}}". Message = "${$message}". """)
    }

  private def showExpr[T](expr: Expr[T])(using Quotes): Expr[String] =
    val code: String = expr.show                                      ❻
    Expr(code)
```

❶ Import the reflection and macro features required.

❷ Add an optional message, analogous to the optional messages the `assert` methods support.

❸ Call another inline method `fail` to process the error.

❹ Splice the `Expr[String]` returned by `failImpl`. This will insert the string as code.

❺ Quote the expression we want to insert, including string representations of the `Exprs` for `predicate` and `block`, which have to be converted to `Expr[String]`. The nested expressions like `${${showExpr(…)}}` are required to first return the `Expr[String]` and then splice it into the large `String`.

❻ Note that `expr.show` returns a `String`, which is then lifted to an `Expr[String]`.

Quoting and splicing are combined with `inline` to cause this macro implementation to do compile-time metaprogramming.

If you think of each quote or splice as a *stage change*, they have to sum to zero in a program, meaning for a given expression or type, the number of quotes has to equal the number of splices. This should make intuitive sense because the purpose of the macro system is to transform code from one valid form to a final form, such as inserting logic automatically that would otherwise have to be written explicitly.

For the line marked with "4," we splice a quoted expression returned by `failImpl`. The body of `failImpl` is a little harder to understand. Consider the example of `${$beforeAfter}`, where `beforeAfter` is an `Expr[String]`. Calling `$beforeAfter` returns a string, then normal string interpolation is used, `${…}`, to insert this string into the larger string. Similarly, `showExpr(predicate)` returns an `Expr[String]`, is spliced with the innermost `${…}`, and then is interpolated into the string with the outermost `${…}`. Finally, `finalImpl` returns a quote of the code `throw InvariantFailure("…")`, which `fail` will splice back into the source code stream.

Recall that `Invariant1.scala` had a `TryInvariant1` entry point in the same file that demonstrated `invariant1` in action. It is not possible to use a compile-time macro definition in the same compilation unit where it is defined. Therefore, `TryInvariant` is defined in a separate file:

```
// src/main/scala/progscala3/meta/TryInvariant.scala
package progscala3.meta

@main def TryInvariant =
  var i = 0
  invariant(i >= 0, s"i = $i")(i += 1)
  println(s"success: $i")
  println(s"Will now fail:")
  invariant(i >= 0, s"i = $i")(i -= 2)
```

The last line now results in a much more informative error message (wrapped to fit):

```
> runMain progscala3.meta.TryInvariant
[info] running progscala3.meta.TryInvariant
...
[error] progscala3.meta.invariant$InvariantFailure:
  FAILURE! predicate "i.>=(0)" failed after evaluation of block: "i = i.-(2)".
  Message = "i = -1".
[error]   at ...InvariantFailure$.apply(Invariant.scala:26)
[error]   at ...TryInvariant(TryInvariant.scala:9)
[error]   ...
```

Note how the user-supplied `message` is used to show the actual value that `i` had at failure. This argument to `invariant.apply` is a by-name parameter, so it is evaluated after the failure occurs, not when `apply` is called. If it were a regular string parameter, the message would be "i = 1," which would be confusing with the rest of the error message. (The built-in `assert` and related `Predef` methods also do this.) Another advantage of using a by-name parameter is that you won't waste cycles building an interpolated string that rarely gets used.

I really love the fact that this output stack trace doesn't show a lot of uninteresting levels for the implementation of `invariant` because almost all of it was inlined. There is just the one line for constructing the exception. The second line is where the failure actually happened, the line you really care about.

Finally, note that `invariant` will always evaluate the `block`, even when `ignore` is true, but the predicate will not be evaluated. Recall the discussion in "Better Design with Design by Contract" on page 474 about how `assert` and related `Predef` methods behave.

The Dotty macro documentation discusses far more details, including pattern-matching support, as well as additional examples.

Staging

In the previous section, I said that the number of quote versus splice stages needs to be equal, but that's not quite correct. If you want to apply quoting and splicing at runtime, your code constructs an `Expr[T]` (or `Type[T]`) at compile time and evaluates at runtime. Hence, the compiled code has one more quote than splice stage. This is useful when some information, like a data structure, is dynamic rather than known at compile time. Note that `inline` is not used.

In principle, you can also have more splices than quotes, which will be purely compile-time evaluation. The term *multistage programming* covers the general case of *N* additional quote versus splice stages. We'll discuss only *N = 1*.

Consider the following program that folds over a list of integers, multiplying or adding them, with a starting seed value. It is loosely based on an example in the documentation, which you should visit for additional details and examples:

```
// src/main/scala/progscala3/meta/Staging.scala
package progscala3.meta
import scala.quoted.*
import scala.quoted.staging.*                                        ❶

object Fold:
  given Compiler = Compiler.make(getClass.getClassLoader)            ❷

  /**
   * Fold operation:
   * @param operation for folding, + or *
   * @param the seed value
   * @param the array to fold.
   */
  val f: (String, Int, Array[Int]) => Int = run {                    ❸
    val stagedFold: Expr[(String, Int, Array[Int]) => Int] = '{
      (op: String, seed: Int, arr: Array[Int]) =>
        val combine = if op == "*" then (x:Int, y:Int) => x*y        ❹
          else (x:Int, y:Int) => x+y
        ${ fold[Int]('seed, 'arr)('combine) }
    }
    println(s"\nStaged fold code after expansion:\n\n${stagedFold.show}")
    stagedFold                                                       ❺
  }

  def fold[T](seed: Expr[T], arr: Expr[Array[T]])(                   ❻
      combine: Expr[(T,T) => T])(
      using Type[T], Quotes): Expr[T] = '{
    var accum: T = ($seed)
    var i = 0
    while i < ($arr).length do {
      val element: T = ($arr)(i)
      i += 1
```

```
      accum = ${combine}(accum, element)
    }
    accum
  }

@main def TryStaging(operator: String, seed: Int, args: Int*) = ❼

  val result = Fold.f(operator, seed, args.toArray)
  println(s"fold of ($args) with operator $operator and seed $seed: $result")
```

❶ Import staging support.

❷ The necessary toolbox for runtime code generation.

❸ The user provides these arguments when running the program: the operator, either `*` or `+`, a seed value, and an array of integers. The `run` method from `scala.quoted.staging` has this signature: `def run[T](expr: Quotes ?=> Expr[T])(using Compiler): T`. We pass it a context function (see "Context Functions" on page 172) that returns an `Expr[T]`.

❹ Construct a function, `combine`, that either multiplies or adds `Int`s.

❺ After printing the expanded source code, return it.

❻ A generic implementation of folding using a `while` loop. Note how the `Expr` arguments are spliced into this code block. A `using Type[T]` is needed since we use a generic type `T`.

❼ The entry point expects either `*` or `+`, a seed value, and one or more integers to fold over.

This program uses a library already in the `sbt` build, `org.scala-lang.scala3-staging`.

Try running it with a command like this:

```
> runMain progscala3.meta.TryStaging + 10 1 2 3 4 5
```

Try using `*` instead of `+` and a different seed value (the first argument).

> If you are not sure what code gets inlined at compile time, use the compiler option `-Xprint:typer` to print the code after compile-time macro expansion.

Wrapping Up and Looking Ahead

Scala 3 metaprogramming is powerful but takes more effort to master. Fortunately, Scala 3 macros are no longer experimental, so any investment you make in learning the macro system and writing macros should provide benefits for a long time. However, make sure that other Scala idioms aren't sufficient for your requirements before using these techniques.

Congratulations, you have now completed all the main chapters in *Programming Scala*, third edition! The appendix compares old versus new syntax in a concise way. This is followed by a bibliography of references I hope you'll investigate to learn more. At this point, you have learned about all the major features of the language and how to use them. I hope you'll find the code examples useful as templates for your own projects.

I'm grateful to you for reading *Programming Scala*, third edition. Best wishes in your Scala journey!

APPENDIX A

Significant Indentation Versus Braces Syntax

In "New Scala 3 Syntax—Optional Braces" on page 31, I introduced the new, optional braces syntax in Scala 3, which is also called *significant indentation*. This appendix provides concise examples of both forms. The examples are available as scripts in the code examples folder *src/script/scala/progscala3/*: *BracesSyntax.scala* and *IndentationSyntax.scala*. They demonstrate more details than are shown here, such as how to use the optional `end` markers for significant indentation syntax.

Table A-1 shows examples of each form. Under the "Indentation" column, the `for` and `while` loops, and the `if` expression examples also demonstrate the optional, alternative control syntax. If you don't pass the flag `-new-syntax` to the compiler or REPL, you can omit the keywords `then` and `do` and use parentheses around the conditions, as shown in the corresponding braces examples.

Also shown are the changes to the import syntax.

For the braces examples, all braces around single expressions can be omitted, but they are shown here to emphasize where they are used in the general case.

Table A-1. Significant indentation versus braces and other syntax changes

Construct	Indentation	Braces

Package definition	`package mypkg:`
	`  // ...`
		`package mypkg {`
		`  // ...`
		`}`

Import statements	`import foo.bar.{given, *}`
	`import foo.X as FooX`
	`import baz.{A as _, *}`
		`import foo.bar._`
		`import foo.{X => FooX}`
		`import baz.{A => _, _}`

<table>
<tr><th>Construct</th><th>Indentation</th><th>Braces</th></tr>
<tr><td><code>for</code> comprehension</td><td><pre>val evens = for
  i <- 0 until 10
  if i%2 == 0
yield i</pre></td><td><pre>val evens = for {
  i <- 0 until 10
  if i%2 == 0
} yield { i }</pre></td></tr>
<tr><td><code>for</code> loop</td><td><pre>for
  i <- 0 until 10
  if i%2 == 0
do println(i)</pre></td><td><pre>for {
  i <- 0 until 10
  if i%2 == 0
} { println(i) }</pre></td></tr>
<tr><td><code>if</code> expression</td><td><pre>if 8 < 10 then
  println(true)
else
  println(false)</pre></td><td><pre>if (8 < 10) {
  println(true)
} else {
  println(false)
}</pre></td></tr>
<tr><td><code>while</code> loop</td><td><pre>var i = 0
while i < 10 do i+=1</pre></td><td><pre>var i = 0
while (i < 10) { i+=1 }</pre></td></tr>
<tr><td><code>match</code> expression</td><td><pre>0 match
  case 0 => "zero"
  case _ => "other value"</pre></td><td><pre>0 match {
  case 0 => "zero"
  case _ => "other value"
}</pre></td></tr>
<tr><td>Partially defined function</td><td><pre>val o: Option[Int] => Int =
  case Some(i) => i
  case None => 0</pre></td><td><pre>val o: Option[Int] => Int = {
  case Some(i) => i
  case None => 0
}</pre></td></tr>
<tr><td><code>try</code>, <code>catch</code>, <code>finally</code> expressions</td><td><pre>import scala.io.Source
import scala.util.control.NonFatal
var source: Option[Source] = None
try
  source =
    Some(Source.fromFile("..."))
  // ...
catch
  case NonFatal(ex) => println(ex)
finally
  if source != None then
    source.get.close</pre></td><td><pre>import scala.io.Source
import scala.util.control.NonFatal
var source: Option[Source] = None
try {
  source =
    Some(Source.fromFile("..."))
  // ...
} catch {
  case NonFatal(ex) => println(ex)
} finally {
  if (source != None) {
    source.get.close
  }
}</pre></td></tr>
<tr><td>Multiline method definition</td><td><pre>def m(s: String): String =
  println(s"input: $s")
  val result = s.toUpperCase
  println(s"output: $result")
  result</pre></td><td><pre>def m(s: String): String = {
  println(s"input: $s")
  val result = s.toUpperCase
  println(s"output: $result")
  result
}</pre></td></tr>
<tr><td>Trait, class, object definitions</td><td><pre>trait Monoid[A]:
  def add(a1: A, a2: A): A
  def zero: A</pre></td><td><pre>trait Monoid[A] {
  def add(a1: A, a2: A): A
  def zero: A
}</pre></td></tr>
</table>

Construct	Indentation	Braces

Instantiate an anonymous instance	`val mon = new Monoid[Int]:`
	`  def add(i1: Int, i2: Int): Int =`
	`    i1+i2`
	`  def zero: Int = 0`
		`val mon = new Monoid[Int] {`
		`  def add(i1: Int, i2: Int): Int =`
		`    i1+i2`
		`  def zero: Int = 0`
		`}`

New type class definition	`given intMonoid: Monoid[Int] with`
	`  def add(i1: Int, i2: Int): Int =`
	`    i1+i2`
	`  def zero: Int = 0`
		`given intMonoid: Monoid[Int] with {`
		`  def add(i1: Int, i2: Int): Int =`
		`    i1+i2`
		`  def zero: Int = 0`
		`}`

Alias given	`given Monoid[Int] = new Monoid[Int]:`
	`  def add(i1: Int, i2: Int): Int =`
	`    i1+i2`
	`  def zero: Int = 0`
		`given Monoid[Int] = new Monoid[Int] {`
		`  def add(i1: Int, i2: Int): Int =`
		`    i1+i2`
		`  def zero: Int = 0`
		`}`

Extension method definition	`extension (s: String)`
	`  def bold: String =`
	`    s.toUpperCase + "!"`
	`  def meek: String =`
	`    s"(${s.toLowerCase}, maybe?)"`
		`extension (s: String) {`
		`  def bold: String =`
		`    s.toUpperCase + "!"`
		`  def meek: String =`
		`    s"(${s.toLowerCase}, maybe?)"`
		`}`

If you know Python, you'll notice that semicolons are not used in `if` and `for` expressions and method definitions like they are used in Python, but they are used in a similar way for `trait`, `class`, and `object` declarations.

Most of the remaining uses for curly braces are for passing anonymous functions to collections methods, like `seq.map{ item => …}`. A future release of Scala 3 will probably offer support for passing anonymous functions while using the braceless syntax.

Bibliography

[Abelson1996] Harold Abelson, Gerald Jay Sussman, and Julie Sussman, *Structure and Interpretation of Computer Programs*. The MIT Press, 1996.

[Agha1986] Gul Agha, *Actors*. The MIT Press, 1986.

[Alexander2013] Alvin Alexander, *Scala Cookbook: Recipes for Object-Oriented and Functional Programming* 2nd edition. O'Reilly Media, 2021.

[Alexander2017] Alvin Alexander, *Functional Programming Simplified*. CreateSpace Independent Publishing Platform, 2017.

[Bird2010] Richard Bird, *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010.

[Bloch2008] Joshua Bloch, *Effective Java* 2nd edition. Addison-Wesley, 2008.

[Bryant2013] Avi Bryant, "Add All the Things!" Strange Loop, 2013. *https://oreil.ly/3PTcM*.

[Chiusano2013] Paul Chiusano and Rúnar Bjarnason, *Functional Programming in Scala*. Manning Publications, 2013.

[DesignByContract] "Building Bug-Free O-O Software: An Introduction to Design by Contract™." Eiffel Software. *https://oreil.ly/kwxqW*.

[Dzilums2014] Lauris Dzilums, "Awesome Scala." GitHub. *https://github.com/lauris/awesome-scala*.

[Ghosh2010] Debasish Ghosh, *DSLs in Action*. Manning Press, 2010.

[GOF1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides ("Gang of Four"), *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Hewitt1973] Carl Hewitt, Peter Bishop, and Richard Steiger, "A Universal Modular Actor Formalism for Artificial Intelligence." *IJCAI '73*, August 20–23, 1973, Stanford, California.

[Hewitt2014] Carl Hewitt, "Actor Model of Computation: Scalable Robust Information Systems." 2014. *https://arxiv.org/pdf/1008.1459.pdf*.

[Lawvere2009] F. William Lawvere and Stephen H. Schanuel, *Conceptual Mathematics: A First Introduction to Categories*. Cambridge University Press, 2009.

[LiHaoyi2020] Li Haoyi, *Hands-on Scala Programming*. Self-published, 2020. See also "Hands-on Scala.js".

[Meyer1997] Meyer, Bertrand, *Object-Oriented Software Construction*, 2nd edition. Prentice Hall, 1997.

[Milewski2019] Bartosz Milewski, *Category Theory for Programmers*. Blurb, (GitHub). A version with Scala examples is also available.

[Naftalin2006] Maurice Naftalin and Philip Wadler, *Java Generics and Collections*. O'Reilly Media, 2006.

[Nedelcu2020] Alexandru Nedelcu, "Retry Failing Tasks with Cats and Scala." August 2020. *https://oreil.ly/Cuo44*.

[Odersky2009] Martin Odersky, Lex Spoon, and Bill Venners, "How to Write an Equality Method in Java." June 2009. *https://oreil.ly/XDqBz*.

[Odersky2019] Martin Odersky, Lex Spoon, and Bill Venners, *Programming in Scala*, 4th edition. Artima Press, 2019.

[Okasaki1998] Chris Okasaki, *Purely Functional Data Structures*. Cambridge University Press, 1998.

[Patryshev2020] Vlad Patryshev, *A Brief Course in Modern Math for Programmers*. Gumroad, 2020. *https://gumroad.com/l/lcbk02*.

[Pierce2002] Benjamin C. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[Rabhi1999] Fethi Rabhi and Guy Lapalme, *Algorithms: A Functional Programming Approach*. Addison-Wesley, 1999.

[Roestenburg2014] Raymond Roestenburg, Rob Bakker, and Rob Williams, *Akka in Action*. Manning, 2014.

[Scala3Migration] Scala 3 Migration Guide. *https://oreil.ly/dap2o*.

[Vector2020] "Rewrite Vector (now 'radix-balanced finger tree vectors'), for performance." GitHub. *https://oreil.ly/WOB7e*.

[Volpe2020] Gabriel Volpe, *Practical FP in Scala, A Hands-on Approach*. LeanPub, 2020.

[Welsh2017] Noel Welsh and Dave Gurnell, *Scala with Cats 2*. Underscore, April 2020. *https://www.scalawithcats.com*.

[Whaling2020] Richard Whaling, *Modern Systems Programming with Scala Native*. Pragmatic Programmer, 2020.

Index

Symbols

A

B

C

D

E

F

G

H

I

N

O

P

T

U

About the Author

Dean Wampler is an expert in data engineering for scalable streaming data systems and applications of machine learning and artificial intelligence. He is a principal software engineer at Domino Data Lab. Previously he worked at Anyscale and Lightbend, where he worked on scalable machine learning with Ray and distributed streaming data systems with Apache Spark, Apache Kafka, Kubernetes, and other tools. Besides *Programming Scala*, Dean is also the author of *What Is Ray?*, *Distributed Computing Made Simple*, *Fast Data Architectures for Streaming Applications*, and *Functional Programming for Java Developers*, as well as the coauthor of *Programming Hive*, all from O'Reilly. He is a contributor to several open source projects and a frequent conference speaker, and he co-organizes several conferences around the world and several user groups in Chicago. Dean has a PhD in physics from the University of Washington. Find Dean on Twitter: @deanwampler.

Colophon

The animal on the cover of *Programming Scala* is a Malayan tapir (*Tapirus indicus*), also called an Asian tapir. It is a black-and-white hoofed mammal with a round, stocky body similar to that of a pig. At 6–8 feet long and 550–700 pounds, the Malayan is the largest of the four tapir species. It lives in tropical rain forests in Southeast Asia.

The Malayan tapir's appearance is striking: its front half and hind legs are solid black, and its midsection is marked with a white saddle. This pattern provides perfect camouflage for the tapir in a moonlit jungle. Other physical characteristics include a thick hide, a stumpy tail, and a short, flexible snout. Despite its body shape, the Malayan tapir is an agile climber and a fast runner.

The tapir is a solitary and mainly nocturnal animal. It tends to have very poor vision, so it relies on smell and hearing as it roams large territories in search of food, tracking other tapirs' scents and communicating via high-pitched whistles. The Malayan tapir's predators are tigers, leopards, and humans, and it is considered endangered due to habitat destruction and overhunting.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from the Dover Pictorial Archive. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.