O'REILLY®

3rd Edition
Covers Scala 3.0

Programming Scala

Scalability = Functional Programming + Objects

Early Release
RAW & UNEDITED

Dean Wampler

Programming Scala

THIRD EDITION

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Dean Wampler

Programming Scala

by Dean Wampler

Copyright © 2020 Kevin Dean Wampler. All rights reserved.

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/safari). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Michele Cronin

Production Editor: Caitlin Ghegan

Copyeditor: TK

Proofreader: TK

Indexer: TK

Interior Designer: TK

Cover Designer: TK

Illustrator: TK

September, 2009: First Edition

November, 2014: Second Edition

Revision History for the Early Release

- 2020-09-30: First release

See http://oreilly.com/catalog/errata.csp?isbn=9781492077824 for release details.

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *Programming Scala*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-07782-4

[LSI]

Preface

Programming Scala introduces an exciting and powerful language that offers all the benefits of a modern object model, *functional programming* (FP), and an advanced type system, while leveraging the industry's investment in the Java Virtual Machine (JVM). 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 support concurrency and distribution. You'll also learn how Scala takes advantage of the advanced JVM as a platform for programming languages.

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

Welcome to Programming Scala, Third Edition

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 had slowed, in part because its steward, Sun Microsystems was sold to Oracle Corporation a few years

previously. Developers wanted improvements like more concise syntax for some constructs and features they saw in other languages, like support for functional programming.

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, where written in Scala and offered elegant Scala APIs, along with APIs in other languages.

A lot has changed in six years. Oracle deserves a lot of credit for reinvigorating Java after the Sun Microsystems acquisition. The pace of innovation has improved considerably. Java 8 was a ground-breaking release, as it introduced two of the most important improvements needed to address limitations compared to Scala. One was support for anonymous functions, called *lambdas*, which addressed the biggest missing feature needed for functional programming. The second feature was support for "default" implementations of the methods declared in interfaces, which made Java interfaces more useful as composable "mixins".

Also, the Kotlin language was created by the tool vendor Jet Brains, 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 data science called deep learning (DL - i.e., using neural networks), reinforcement learning (RL), and artificial intelligence (AI) are currently the hottest topics in the data world. All fit under the umbrella of machine learning (ML). A large percentage of the popular tools for data science, especially ML, 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. This edition introduces you to version 3 of the language, with significant changes to improve the expressiveness and correctness of Scala, and to remove deprecated and less useful features.

Also, Scala is now a viable language for targeting JavaScript applications through Scala.js. Early support for Scala as a native language is now available through Scala Native, although the version of Scala supported tends to lag the JVM version.

I currently split my time between the Python-based ML world and the Scala-based JVM world. When using Python, I miss the concision, power, and correctness of Scala, but when using Scala, I miss the wealth of data-centric libraries available in the Python world. So, I think we're entering a period of consolidation for Scala, where developers who want that power and elegance will keep the Scala community vibrant and growing, especially in larger enterprises that are JVM-centered. Scala will remain a preferred choice for hosted services, even while the data and mobile communities tend to use

other languages. Who knows what the next five or six years will bring, when it's time for the fourth edition of *Programming Scala*?

With each addition of this book, I have attempted to provide a comprehensive introduction to Scala features and core libraries, illustrated with plenty of pragmatic examples, tips, and tricks. However, each edition has shifted more towards pragmatism and away from comprehensive surveying of features.

I think this makes the book more useful to you, in an age when so much of our information is gathered in small, ad hoc snippets through Google searches. Libraries come and go. For example, when you need the best way to parse JSON, an Internet search for *Scala JSON libraries* is your best bet. Second, 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 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 getting bogged down in corner cases, obscure features, or advanced capabilities that you won't need to know until you become an advanced Scala developer.

Hence, I also won't discuss how to use Scala.js or Scala Native, as the best sources of information for targeting those platforms are their respective websites.

Finally, I wrote this book for professional programmers. I'll err on the side of tackling deeply technical topics, rather than keeping the material "light". Other books provide less thorough, but more gentle

introductions, if that's what you prefer. This is a book if you are serious about using Scala professionally.

Welcome to Programming Scala, Second Edition

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

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 Mac OS 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.

Both of us 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.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> 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 a CD-ROM of 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. Copyright 2020 Kevin Dean Wampler, 978-1-491-94985-6."

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*.

Getting the Code Examples

> **NOTE**
>
> TODO: At this time, the CLI tools are actually called `dotr` and `dotc`, not `scala` and `scalac`. The latter names are used in the expectation that when Scala 3 release candidates begin, the tools will be renamed. I'll refine this text depending on what final names are used for Scala 3 tools. Also, some features shown for `scala` are actually not yet implemented in `dotr`!

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*

> All Scala 3 source files built with SBT. The standard Scala file extension is *.scala*.

src/main/scala-2/.../.scala*

> All Scala 2 source files, some of which won't compile with Scala 3. They are not built with SBT.

src/test/.../.scala*

All Scala 3 test source files built and executed with SBT.

src/script/.../.scala*

"Script" files that won't compile with `scalac`, but can be interpreted with the `scala` interpreter.

O'Reilly Safari

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

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 *http://bit.ly/programmingScala_2E*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

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

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

Watch us on YouTube: *http://www.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 EPFL documentation for Dotty, *https://dotty.epfl.ch/docs/*, provided essential information.

Acknowledgments for the Second Edition

As I, Dean Wampler, 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 Huehnken, and Thomas Lockney, who reviewed drafts of the manuscript. Thanks to my long-time colleague and friend, Jonas Bonér, for writing an updated [Link to Come] 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 available at *http://programmingscala.com*. 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 [Link to Come] for the book. Jonas is a longtime friend and collaborator from the aspect-oriented

programming (AOP) 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* (Artima), 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

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at m.cronin@oreilly.com

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, interpreted scripts to large, sophisticated applications. So let's consider each of those ideas in more detail:

A 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* is an experimental Scala that compiles to native machine code, bypassing the JVM and JavaScript runtimes.

Statically typed

Scala embraces *static typing* as a tool for creating robust applications. It fixes many of the flaws of Java's type system and it uses type inference to eliminate much of the typing boilerplate.

Object-oriented programming

Scala fully supports *object-oriented programming* (OOP). Scala improves Java's object model with the addition of *traits*, providing a clean way to implement types using *mixin composition*. In Scala, everything *really* is an object, even numeric types, providing much more consistent handling, especially in collections.

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, functions without side effects, "higher-order" functions, and functional collections all contribute to concise, powerful, and correct code.

A sophisticated type system

Scala extends the type system of Java with more flexible generics and other enhancements to improve code correctness. 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 Java become concise idioms in Scala. Scala provides several facilities for building *domain-specific languages* (DSLs), APIs that feel "native" to users.[1]

Scalable—architectures

> You can write small, interpreted scripts 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 20th, 2004. Martin is a professor in the School of Computer and Communication Sciences at the Ecole 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 hired by Sun Microsystems to produce the reference implementation of `javac`, the descendant of which is the Java compiler that ships with the Java Developer Kit (JDK) today.

The Seductions of Scala

The growth of Scala users since it was introduced over fifteen 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 success.

I believe Scala is a language for *professional* developers. Not all users are professionals, of course, but Scala is the kind of language a professional in our field needs, rich in features, highly performant, 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.

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 it removes unimportant constructs to make the language smaller and more regular. The previous experimental approach to macros is replaced with a principled approach to meta-programming.

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 collections library as Scala 2.13. Hence, if you are using a Scala 2 version earlier than 2.13, I recommend upgrading to Scala 2.13 first, to update uses of the library, then upgrade to Scala 3.

However, to make this transition as painless as possible, there are several ways to compile Scala code that allows or disallows deprecated Scala 2 constructs. There are even compiler flags that will do some rewrites for you. See [Link to Come] and [Link to Come] in [Link to Come] for details.

Installing Scala

Let's learn how to install the command-line tools that you need to work with the book's code examples.[2] The examples used in this

book were written and compiled using Scala version 3.0, the latest release at the time of this writing.

All the examples will use Scala on the JVM. See the Scala.js and Scala Native websites for information on targeting those platforms.

At a minimum, you need to install a recent Java JDK and the de facto build tool for Scala, SBT. Then you can use the `sbt` command to bootstrap everything else for the examples. However, the following sections go into more details about tools you might want to install and how to install them.

The Scala website Getting Started page discusses even more options to get started with Scala.

Coursier

Coursier (get-coursier.io) 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.

Installing Coursier is not required, but it is recommended as it will install all the tools you need and more, in addition to providing many other convenient features. For example, the Coursier CLI is handy for managing dependency metadata and artifacts from Maven and Ivy repositories, although we'll do this indirectly through the SBT build for the examples. Coursier also has convenient commands for working with applications embedded in libraries and it can even manages installations of different Java JDK versions.

See the installation instructions at get-coursier.io/docs/cli-installation for details. After installing Coursier, see get-coursier.io/docs/cli-install for a description of the `coursier install` command for installing other tools, like SBT (`sbt-launcher`) and Scala. For example, this command shows the default set of available applications (at the time of this writing in early 2020):

```
$ unzip -l "$(cs fetch io.get-coursier:apps:0.0.8)" | grep
json
      188  02-09-2020 17:48   ammonite.json
      175  02-09-2020 17:48   coursier.json
      332  02-09-2020 17:48   cs.json
      241  02-09-2020 17:48   dotty-repl.json
      150  02-09-2020 17:48   echo-graalvm.json
      108  02-09-2020 17:48   echo-java.json
      172  02-09-2020 17:48   echo-native.json
      335  02-09-2020 17:48   giter8.json
      135  02-09-2020 17:48   mdoc.json
      323  02-09-2020 17:48   mill-interactive.json
      321  02-09-2020 17:48   mill.json
      524  02-09-2020 17:48   sbt-launcher.json
      222  02-09-2020 17:48   scala.json
      209  02-09-2020 17:48   scalac.json
      213  02-09-2020 17:48   scaladoc.json
      180  02-09-2020 17:48   scalafix.json
      184  02-09-2020 17:48   scalafmt.json
      204  02-09-2020 17:48   scalap.json
```

(Many of the tools shown here are discussed in [Link to Come].) For now, run the following command to install the `sbt` command. Note that the `.json` suffix is removed from the name. I recommend installing all the `scala*` commands, but they aren't strictly necessary, as discussed below.

```
coursier install --install-dir path sbt-launcher
```

Note that the `--install-dir path` arguments are optional. Depending on your platform, the default installation location will vary. Whatever location you use, make sure you add it to your `PATH`.

Java JDK

Use a recent Java JDK release. Version 11 or newer is recommended, although Java 8 should work. To install the JDK, go to the Oracle Java website and follow the instructions to install the full Java Development Kit (JDK).

SBT

The most popular build tool for Scala, *SBT*, version 1.3.8 or newer, is used for the code examples. Install using Coursier or follow the instructions at scala-sbt.org.

When you are finished, you will have an `sbt` command that you can run from a Linux or OS X terminal or Windows command window.

Scala

You actually don't need to install the Scala command-line tools separately, as SBT will install the basics that you need as dependencies. However, if you want to install these tools, use Coursier or see the instructions at scala-lang.org/download.

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".

Start SBT

Open a terminal and change to the root directory for the code examples. Type the command **`sbt test`**. It will download all the library dependencies you need, including the Scala compiler. This will take a while and you'll need an Internet connection. 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.

> **TIP**
>
> For most of the book, we'll use the Scala tools indirectly through SBT, which downloads the Scala compiler version we want, the Scala interpreter, Scala's standard library, and the required third-party dependencies automatically.

More Tips

In your browser, it's useful to bookmark the URL for the Scala library's *Scaladocs*, the analog of *Javadocs* for Scala. For your

convenience, most times when I mention a type in the Scala library, I'll include a link to the corresponding Scaladocs 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 the link on the line labeled "Source."

Any text editor or IDE (integrated development environment) will suffice for working with the examples. Scala plug-ins exist for all the popular editors and IDEs. For more details, see [Link to Come]. In general, the community for your favorite editor is your best source of up-to-date information on Scala support.

Using SBT

Let's cover the basics of using SBT, which you'll need to work with the code examples.

When you start the `sbt` command, if you don't specify a task to run, SBT starts an interactive REPL (*Read, Eval, Print, Loop*). 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`), where you start the `sbt` command, the > is the default SBT interactive prompt, and the `#` starts an `sbt` comment. You can type most of these commands in any order:

```
$ sbt
> help      # Describe commands.
> launchIDE # Open the project in Visual Studio Code.
> tasks     # Show the most commonly-used, available tasks.
> tasks     # 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 Scala REPL.
> run       # Run one of the "main" routines in the project.
> show      # Show the definition of variable "x".
> exit      # Quit the REPL (also control-d works).
```

TIP

The *SBT* project for the code examples is actually configured to show the following as the SBT prompt:

```
sbt:Programming Scala, Third Edition - Code examples>
```

We'll use the more concise prompt, >, the default for SBT, to save space.

The `launchIDE` task is convenient for those of you who prefer IDEs, although currently only Visual Studio Code is supported. The details can be found at dotty.epfl.ch/docs/usage/ide-support.html.

Both *IntelliJ IDEA* and *Visual Studio Code* can open SBT project, once you install a Scala plug-in.

A handy SBT technique is to add ~ 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 amd running my code and tests. SBT uses an incremental compiler, so you

don’t have to wait for a full rebuild every time. Break out of this loop by hitting the return key.

Scala has its own REPL. Invoke it using the `console` command in SBT. You will use this a lot to try examples in the book. The Scala REPL prompt is `scala>`.

Before starting the REPL, SBT will build your project and set up the `CLASSPATH` with your built artifacts and depedent libraries. This convenience means it’s rare to use the `scala` command-line tool outside of SBT.

You can exit both the SBT REPL and the Scala REPL with *Ctrl-D*.

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

Running the Scala Command-Line Tools

If you installed the Scala command-line tools separately, the Scala compiler is called `scalac`, analogous to the Java compiler `javac`. We will let SBT run it for us, but the command syntax is straightforward if you’ve ever run `javac`. Use `scalac -help` to see the options.

Similarly, the `scala` command, which is similar to `java`, is used to run programs, but it also supports the interactive REPL mode we just discussed *and* the ability to run Scala "scripts". Consider this example script from the code examples:

```
// 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()
println(up.convert(List("Hello", "World!")))
```

Let's run it with the `scala` command. Change your current working directory to the root of the code examples. For Windows, use backslashes in the next command:

```
$ scala 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!"

> **TIP**
>
> As you can see from the listing above for `Upper1.scala`, each of these files is listed starting with a comment that contains the file path in the code examples. That makes it easy to find the file.

If you invoke `scala` without a compiled `main` class to run or a script file, `scala` enters the REPL mode. Here is a REPL session illustrating some useful commands. If you didn't install Scala separately, just start `console` in `sbt`. The REPL prompt is `scala>` (some output elided):

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

:help                    print this summary
:load <path>             interpret lines in a file
:quit                    exit the interpreter
:type <expression>       evaluate the type of the given
expression
:doc <expression>        print the documentation for the
given expresssion
: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.
```

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

This `println` is effectively the same thing as Java's `System.out.println`. Also, Scala `Strings` are Java `Strings`.

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.

We didn't always explicit specify type information. When type information is shown, either when it is inferred or explicit type information is added to declarations, these *type annotations*, as they are called, follow a colon after the item name. The output of REPL shows several examples.

Why doesn't Scala follow Java conventions? When type annotations aren't explicitly in the code, then the type is *inferred*. Compared to Java's `type item` convention, the `item: type` convention is

easier for the compiler to analyze unambiguously when you omit the type annotation and just write `item`.

As a general rule, Scala follows Java conventions, departing from them for specific reasons, like supporting a new feature that would be difficult using Java syntax.

> **TIP**
>
> 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.

Finally, we used `:quit` to exit the REPL. Ctrl-D can also be used.

We'll see additional REPL commands as we go and we'll explore the REPL commands in depth in [Link to Come].

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.

> **TIP**
>
> When we mention a type in the Scala library, you might find it useful to read more in the Scaladocs about it. The Scaladocs for the current release of Scala can be found here. Note that Scala 3 uses the Scala 2.13 collections library, while other parts of the library have changed in Scala 3.

All examples shown in the book start with a comment line like this:

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

Scala follows the same comment conventions as Java, C#, C, etc. A `// comment` goes to the end of a line, while a `/* comment */` can cross line boundaries. *Scaladoc* comments follow Java conventions, `/** comment */`.

When the path starts with `src/script`, use `scala` to run the script, as follows:

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

However, this may not work if the script uses libraries. Instead, run the SBT `console`, then use the `:load` command:

```
scala> :load
src/script/scala/progscala3/introscala/Upper1.scala
```

Finally, you can also copy code and paste it at the `scala>` prompt.

Files named `src/test/scala/.../*Suite.scala` are tests written using MUnit (see [Link to Come]). 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`. SBT follows Maven conventions that directories for compiled source code go under `src/main/scala` and tests go under `src/test/scala`. So, in this example, the package definition for this test is `progscala3.objectsystem.equality`. The compiled class name is `EqualitySuite`.

> **NOTE**
>
> Java requires that package and file names must match the declared package and public class declared within the file. Scala doesn't require this practice. However, I follow these conventions most of the time for compiled code (less often for scripts) and I recommend you do this, too, for your production code.

Finally, many of the files under the `src/main/scala` define a `main` method that you can execute in one of several ways.

First, use SBT's `run` command. It will find all the classes with `main` methods and prompt you to pick which one. Note that SBT will only search `src/main/scala` and `src/main/java` directories, ignoring the other directories under `src`, including `src/script`.

Let's use another example we'll study later in the chapter, `src/main/scala/progscala3/introscala/UpperMain1.scala`. Invoke `run hello world`, then enter the number shown for `progscala3.introscala.UpperMain1`. Note we are passing arguments to `run`, `hello world`, which will be passed to the program to convert to upper case:

```
> run hello world
...

Multiple main classes detected, select one to run:

 ...
 [20] progscala3.introscala.UpperMain1
 ...

20

[info] running progscala3.introscala.UpperMain1 hello world
HELLO WORLD
[success] Total time: 2 s, completed Feb 29, 2020, 5:08:18
PM
```

The second way to run this program is to use `runMain` and specify the specific program class name. This skips the prompt:

```
> runMain progscala3.introscala.UpperMain1 hello world
[warn] Multiple main classes detected.  Run 'show
discoveredMainClasses' ...
[info] running progscala3.introscala.UpperMain1
HELLO WORLD
[success] Total time: 0 s, completed Feb 29, 2020, 5:18:05
PM
>
```

Finally, once your program is ready for production runs, you'll use the `scala` command, similar to how `java` is used. Now the correct `classpath` must be defined, including all dependencies. This example is relatively easy; we just point to the output directory for the compiled code:

```
$ scala -cp target/scala-3.0.0/classes/
progscala3.introscala.UpperMain1 words
```

Let's explore other differences between scripts, like the `Upper1` script we've used, and compiled code, like the `UpperMain1` example we just executed.

Here is the script again:

```
// 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()
println(up.convert(List("Hello", "World!")))
```

We declare a class, `Upper1`, using the `class` keyword. The entire class body is indented on the subsequent lines (or inside curly braces `{…}` if you use that syntax instead).

`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 and also avoids occasional surprises from the type inference process.

> **NOTE**
>
> I'll use *parameters* to refer to the things a method or function is defined as accepting when you call it. I'll use *arguments* to refer to values you actually pass to it when making the call.

Type *annotations* are specified using `name: type`. This is used here for both the parameter list and the return type of the method, the last `Seq[String]` before the equals sign.

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

One reason is to reduce ambiguity. Scala infers the return type if the colon and type are omitted. 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 equals sign also reminds us of the *functional programming* principle that variables and functions are treated uniformly. As we

saw in the invocation of `map`, functions can be passed as arguments to other functions, just like values. They can also be returned from functions, and assigned to variables. In fact, it's correct to say that *functions are values*.

This 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 type returned by this method will be the same concrete type that was passed into it as an argument, like `Vector` or `List` (both of which are immutable collections).

Collection types like `Seq` are *parameterized types*, very similar to *generic* types in Java. They are a "collection of something," in this example a sequence of strings. Scala uses square brackets (`[...]`) for parameterized types, whereas Java uses angle brackets (<...>).

NOTE

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 and allowed by Scala, whereas Java doesn't allow characters like that in identifiers. So, to avoid ambiguity, Scala uses square brackets instead for parameterized types and disallows them in identifiers.

Inside the body, we use one of the powerful methods available for most collections, `map`, which iterates through the collection, calls the

provided method on each element, and returns a new collection with the transformed elements.

The function passed to `map` is an unnamed *function literal* `(parameters) => body`, similar to Java's *lambda* syntax:

```
(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 UTF8 ⇒ characters was also allowed in Scala 2, but is now deprecated.) The body calls `toUpperCase()` on `s`. The result of this call is automatically returned by the function literal. In Scala, the last *expression* in a function or method 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. In fact, it is rarely used in methods.

METHODS VERSUS FUNCTIONS

Following the convention in most *object-oriented programming* languages, the term *method* is used to refer to a function defined within a class. Methods have an implied `this` reference to the object as an additional argument when they are called. Like most OO languages, the syntax used is `this.method_name(other_args)`. We'll use the term *method* this way. We'll use the term *function* to refer to non-methods, but also sometimes use it generically to include methods. The context will indicate the distinction.

The expression `(s: String) => s.toUpperCase()` in *Upper1.scala* is an example of a *function* that is not a method.

On the JVM, functions are implemented using JVM *lambdas*:

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

The last two lines create an instance of `Upper1`, named `up`, and use it to convert two strings to uppercase and finally print the resulting `Seq`. As in Java, the syntax `new Upper1()` creates a new instance. The `up` variable is declared as a read-only "value" using the `val` keyword. It behaves like a `final` variable in Java.

Now lets look at the compiled example, where I added `Main` to the name. 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
    params.map(s => s.toUpperCase()).foreach(s => printf("%s
",s))
    println("")
  end main
❸

  @main def hello(params: String*) = main(params.toArray)
❹
end UpperMain1
❺
```

❶ Declare the package location.

❷ Declare a `main` method, the program entry point.

❸ For long methods (unlike this one), you can use `end name`, but this is optional.

❹ An alternative way to define an entry point method.

❺ Optional end to the object definition.

Packages work much like they do in Java. Packages provide a "namespace" for scoping. Here we specify that this class exists in the `progscala3.introscala` package.

To declare a `main` method in Java, you would put it in a `class` and declare it `static`, meaning not tied to any one instance. You can then call it with the syntax `MyClass.main`. This pattern is so pervasive, that Scala builds it into the language. We instead declare an `object`, named `UpperMain1`, using the `object` keyword, then declare `main` as we would any other method. At the JVM level, it looks like a `static` method. When running this program like we did above, this `main` method is invoked.

Note the parameter list is an `Array[String]` and it returns `Unit`, which is analogous to `void` in programs like Java, where nothing useful is returned.

You use `end name` with all of the constructs like method and type definitions, `if`, `while` expressions, etc. that support the braceless, indentation-based structure. It is optional, intended for long definitions where it can be hard to see the beginning and end at the same time. Hence, for this short definition of `main`, you wouldn't use it normally. Here are other examples for control constructs:

```
if sequence.length > 0 then
  println(sequence)
end if      // "end if" is optional
```

```
while i < 10 do
  i += 1
end while   // "end while" is optional
```

The `end ...` lines shown are not required, but they are useful for readability if the indented blocks are many lines long.

The `@main def hello` method is another way to declare an entry point, especially useful when you don't need to process arguments, so you don't need to declare the parameter list. Note that in the list printed by the `sbt run` command, this entry point is shown as `progscala3.introscala.hello`, while the `main` method was shown by the type name, `progscala3.introscala.UpperMain1`. Try `run` and invoke `hello`.

Declaring `UpperMain1` as an `object` makes it a *singleton*, meaning there will always only be one instance of it, controlled by the Scala runtime library. 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 `class` declarations. Scala uses these `objects` to replace "class-level" members, like `statics` in Java. You use the declared `object` like an instance created from a regular class, reference the member variables and functions as required, e.g., `UpperMain1.hello`.

When you define the `main` method for your program, you *must* declare it inside an `object`, but in general, `UpperMain1` is a good candidate for a singleton, because we don't need more than one *instance* and it carries no state information.

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 performance. However, there are times when singletons make sense, as in this example where these concerns don't apply.

> **NOTE**
>
> Why doesn't Scala support `statics`? Compared to languages that allow static members (or equivalent constructs), Scala is more true to the vision that *everything* should be an object. The `object` construct keeps this policy more consistent than in languages that mix static and *instance* class members.

`UpperMain1.main` takes the user arguments in the `params` array, maps over them to convert to upper case, then uses another common method, `foreach`, to print them out.

The function we passed to `map` drops the type annotation for `s`:

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

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.

The `foreach` method is used when we want to process each element and do something with *complete side effects*, without returning a new value, unlike `map`. Here we print a string to *standard out*, without a newline after each one. The last `println` call prints the newline before the program exits.

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

Look again at the first line inside `main`, how concise it is, where we compose operations together. Sequencing transformations together 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 work the same was as they do in Java. Scala automatically imports many commonly used types and features, like the `Seq` and `List` types above and methods for I/O, like `println`, a method on the `scala.Console` object.

In Java, to use `println`, you have to write `System.out.println` or import `System.out` and then write `out.println`. In Scala, you can import objects and even individual methods from them. This is done automatically for you for `Console.println`, so we can just use `println` by itself. This

method is one of many methods and types imported automatically that are defined in a library object called `Predef`.

To run this code, you must first compile to a JVM-compatible `.class` file using `scalac`. (Sometimes multiple class files are generated.) SBT will do this for you, but let's so how to do it yourself. If you installed the Scala command-line tools separately, open a terminal window and change the working directory to the root of the project. Then run the following command (ignoring the `$` prompt):

```
$ scalac
src/main/scala/progscala3/introscala/UpperMain1.scala
```

You should now have a new directory named `progscala3/introscala` that contains several `.class` and `.tasty` files, including a file named `UpperMain1.class`. ("Tasty" files are an intermediate representation generated by the compiler.) Scala must generate valid JVM byte code and files. For example, the directory structure must match the package structure.

Now, you can execute this program to process a list of strings. Here is an example:

```
$ scala -cp . progscala3.introscala.UpperMain1 Hello World!
HELLO WORLD!
```

The `-cp .` option adds the current directory to the search classpath, although this is actually the default behavior.

Allowing SBT to compile it for us instead, we can run it at the SBT prompt using this command:

```
> runMain progscala3.introscala.UpperMain1 Hello World!
```

For completeness, if you compile with SBT, but run it using the `scala` command outside SBT, then set the classpath to point to the correct directory where SBT writes class files:

```
$ scala -cp target/scala-3.0.0/classes
progscala3.introscala.UpperMain1 hello
HELLO
```

INTERPRETING VERSUS COMPILING AND RUNNING SCALA CODE

To summarize, if you type `scala` on the command line without a file argument, the REPL is started. You type in commands, expressions, and statements that are evaluated on the fly. If you give the `scala` command a Scala source file argument, it compiles and runs the file as a script. Otherwise, if you provide a JAR file or the name of class with a `main` routine, `scala` executes it just like the `java` command.

Returning to the script version, we can actually simplify it even more. Consider this simplified version:

```
// src/script/scala/progscala3/introscala/Upper2.scala

object Upper2:
  def convert(strings: Seq[String]) =
strings.map(_.toUpperCase())

println(Upper2.convert(List("Hello", "World!")))
```

This code does exactly the same thing, but it’s more concise.

I omitted the return type for the method declaration, because it's "obvious" what's returned. However, we can't omit the type annotations for parameters. Technically, the type inference algorithm does *local type inference*, which means it doesn't work globally over your whole program, but locally within certain scopes. It can't infer caller's expectations for the parameter types, but it is able to infer the type of the method's returned value in most cases, because it sees the whole function body. Recursive functions are one exception where the return type must be declared.

However, explicit type annotations in the parameter lists and explicit return type annotations provide useful documentation for the reader. Just because Scala can infer the return type of a function, should you let it? For simple functions, where the return type is obvious to the reader, perhaps it's not that important to show it explicitly. However, sometimes the inferred type won't be what's expected, perhaps due to a bug or some subtle behavior triggered by certain input arguments or expressions in the function body. Explicit return types express what you *think* should be returned and the compiler confirms it. Hence, I recommend adding return types rather than inferring them, especially in public APIs.

We have also exploited a shorthand for the function literal. Previously we wrote it in the following two, equivalent ways:

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

We have now shortened it even further to the following expression:

```
_.toUpperCase()
```

The `map` method takes a single function parameter, where the function itself takes a single parameter. In this case, the function body only uses the parameter once, so we can use the anonymous variable _ instead of a named parameter. The string parameter will be assigned to it before `toUpperCase` is called.

Finally, using an `object` instead of a `class` simplifies the invocation, because we don't need to first create an instance with `new`. We just call `Upper2.convert` directly.

Let's do one last version of the compiled code (under `src/main/scala`) to show another way of working with collections for this situation:

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

object UpperMain2:
  def main(params: Array[String]): Unit =
    val output = params.map(_.toUpperCase()).mkString(" ")
    println(output)
```

Instead of using `foreach` to print each transformed word as before, we map the array to a new array of strings, then call a convenience method to concatenate the strings into a final string. There are two `mkString` methods. One takes a single parameter to specify the delimiter between the collection elements. The second version that takes three parameters, a leftmost prefix string, the delimiter, and a rightmost suffix string. Try changing the code to use `mkString("[", ", ", "]")`.

As a little exercise, return to the script `Upper2.scala` and try simplifying it further. Eliminate the `Upper2` object completely and just call `map` on the list of words directly. You should have just one line of code when you're done! (See the code examples for one implementation.)

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.

However, to keep it simple, this will be a single-threaded implementation. We won't actually do anything with concurrency right now.

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.

❷ Declare an abstract class for geometric shapes.

❸ Implement a `draw` method for "rendering" the shapes. The documentation comment uses the same conventions that Java uses.

❹ A circle with a center and radius.

❺ A rectangle with a lower-left point, height, and width. We assume for simplicity that the sides are 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* class body is the constructor, so you list the parameters for the *primary* constructor after the class name and before the class body. In this case, there is no

class body, so we can omit the colon (or curly braces if you use those instead).

Because we put the `case` keyword before the class declaration, each constructor parameter is automatically converted to a read-only (immutable) field of `Point` instances. That is, if 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 use `point.y = 3.0` triggers a compilation error.

You can also provide default values for method parameters, including constructors. 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.

You can also construct instances without using `new`.

Let's use our SBT project to explore these points:

```
> console
[info] ...

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
```

Running `console` will automatically compile the code first, so we don't need to run `compile` first.

The import statement uses _ as a wildcard to import everything in the `progscala3.introscala.shapes` package. It behaves the same way as using `*` in Java. Scala uses _ because you might want to use `*` as a method name for multiplication and since Scala lets you import methods into the local scope, using `*` as a wildcard would be ambiguous! This is the second use for _ that we've seen. The first use was in function literals where _ was an anonymous placeholder for parameters, used instead of naming them.

In the definition of `p00`, no arguments are specified, so Scala used `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 used the default value for the remaining argument, as shown for `p20` and `p20b`. We can even specify the arguments with the associated parameter name. The definition of `p02` uses the default value for `x`, but specifies the value for `y`, using `Point(y = 2.0)`.

> **TIP**
>
> Using named arguments explicitly, even when it isn't required, like `Point(x = 0.0, y = 2.0)`, can make your code easier to understand.

While there is no class body for `Point`, another feature of the `case` keyword is the compiler automatically generates several methods for us, including the familiar `toString`, `equals`, and `hashCode` methods in Java. 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. This is in contrast with Java, where == just compares *references*. In Java, you have to call `equals` explicitly to do a *logical* comparison.

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

> **NOTE**
>
> 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.

For *any* instance you use, either a declared `object` or an instance of a `class`, 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)
```

It's a compilation error if no `apply` method exists for the instance. Also, the argument list supplied must conform to the expected parameter list.

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 *without* the `new` keyword. The companion object generated is equivalent to this:

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

```
  ...
}
```

You can add methods to the companion object. A more sophisticated `apply` method might instantiate a different subclass 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.

An *instance* `apply` method defined on any `class` has whatever meaning is appropriate for instances. For example, `Seq.apply(index: Int)` retrieves the element at position `index` (counting from zero).

NOTE

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 argument list types. Syntacticly, 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`.

`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`. Shape is not declared `sealed`, because we intend for people to create their own subclasses.

The parameter `f` for `draw` is a function of type `String => Unit`. We saw `Unit` above. It is a real type, but it behaves roughly like `void` in other languages.[3]

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.

TIP

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 functional programming, 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.

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

`Shape.draw` is another example that functions are *first-class values*, just like instances of `Strings`, `Ints`, `Points`, etc. We saw this previously with `map` and `foreach`. Like other values, we can assign functions to variables, pass them to other functions as arguments, and return them from functions. 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` and it constructs the final string using an *interpolated string*.

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

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

Back to the code listing, `Circle`, `Rectangle`, and `Triangle` are concrete subclasses of `Shape`. They have no class bodies, because the `case` keyword defines 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 you could build a real graphics application that uses an `f` to 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` defines an interface for behavior and can be used as an abstract base class, which is how we use it here. All messages exchanged are subclasses of `Message`. The `sealed` keyword is explained below.

❷ A message to draw the enclosed `Shape`.

❸ `Response` is used to return an arbitrary string message to a caller in response to a message received from the caller.

❹ `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 subclasses of `Message` in the same file. This prevents bugs where users define their own `Message` subtypes that could break the code in the next file!

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 If we only need one instance, we can declare it an `object`, but it would be easy enough to make this a `class` and instantiate as many as we need, for scalability, etc.

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.

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

```
def apply(message: Message): Message =
  message match:
    case Exit =>
      expressions
    case Draw(shape) =>
      expressions
    case Response(unexpected) =>
      expressions
```

The `message match: ...` expression consists only of `case` clauses, which do *pattern matching* on the message passed into the function. This is an expression that returns a value, which you can assign to a variable or use as the return value, as we do here for `apply`.

Match expressions work a lot like "if/else" expressions. When one of the patterns matches, the expressions are evaluated after the arrow (=>) up to the next `case` keyword (or the end of the definition). 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 many cases where the match clauses don't handle all possible inputs. Note that our `sealed` hierarchy of messages is crucial here. If a user created 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`, 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 commonly taught tenets of object-oriented programming is that you should never use case statements that match on instance type, because inheritance hierarchies evolve, which breaks these case statements. Instead, polymorphic functions should be used. So, is the pattern-matching code just discussed an *antipattern*?

Recall that we defined `Shape.draw` to call the `toString` method on the `Shape`, which is automatically generated by the compiler for each concrete subclass because they are case classes. Hence, the code in the first `case` statement invokes a *polymorphic* `toString` operation and 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 class hierarchy by subclassing `Shape`, which we encourage.

The case clauses match on subtypes of `Message`, but we protected ourselves from unexpected change by making `Message` a `sealed` hierarchy. If we add a new message type here, we can modify the match expression accordingly.

Hence, we have combined polymorphic dispatch from object-oriented programming with pattern matching, a workhorse of functional programming. This is one way that Scala elegantly integrates these two programming paradigms.

PATTERN MATCHING VERSUS SUBTYPE POLYMORPHISM

Pattern matching plays a central role in functional programming just as *subtype polymorphism* (i.e., overriding methods in subtypes) plays a central role in object-oriented programming. Functional pattern matching is much more important and sophisticated than the corresponding `switch/case` statements found in most languages. 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.

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

```
//
src/main/scala/progscala3/introscala/shapes/ProcessShapesDri
ver.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.

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! At the `sbt` prompt, type `run`, which will compile the code if necessary and then present you with a list of all the code examples that have a `main` method:

```
> run
[info] Compiling ...

Multiple main classes detected, select one to run:

 ...
 [28] progscala3.introscala.shapes.ProcessShapesDriver
 ...

Enter number:
```

Enter `28` (or whatever number is shown for you) and the following output is written to the console (wrapped to fit):

```
Enter number: 28

[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),Point(2.0,0.0),...) 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 that are available on *http://scala-lang.org*. 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.

---

1 Lately, the acronym API has been used to refer to the interfaces exposed by services. I'll use API in its original sense, the types, methods, and values exposed by a code module.

2 For more details on these and other tools, see [Link to Come].

3 In fact, there is a single instance of `Unit` named `()`. If you are curious about that name, see [Link to Come].

Chapter 2. Type Less, Do More

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at m.cronin@oreilly.com

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.

New Scala 3 Syntax

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 Java-style curly braces (`{...}`) are replaced with indentation that becomes significant. The examples in the previous chapter and throughout the book use it.

This syntax has several benefits. It is more concise. For Python developers who decide to learn Scala, it will feel more familiar to them (and vice versa).

There is also a new syntax for control structures like `for` loops and `if` expressions. For example, `if condition then …` instead of the older `if (condition) …`. Also, `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 required. You also have to be careful to use tabs and spaces consistently for indentation.

The new constructs are the defaults supported by the compiler, but using the compiler flags `-old-syntax` and `-noindent` will enforce the old syntax constructs. Another flag, `-new-syntax` makes the new keywords `then` and `do` required. Finally, the compiler can now rewrite your code to use whichever style you prefer. Add the `-rewrite` compiler flag, for example `-rewrite -new-syntax`.

I opposed these changes initially, because they aren't strictly necessary. However, now that I have worked with them, I believe the advantages outweigh the disadvantages. I also suspect that the syntax changes are the future for Scala and eventually the old syntax could be deprecated. We'll see. Hence, I have chosen to use the new

conventions throughout this edition of the book. I will mention other pros and cons of these changes as we explore examples.

[Link to Come] provides examples of the old and new syntax.

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, even for the following example:

```
scala> val s = "hello"
     |   + "world"
     |   + "!"
val s: String = helloworld!
```

The Scala 2 REPL was more aggressive at interpreting the end of a line as the end of an expression, so the previous example would infer "hello" for the definition of `s` and then throw an error for the next two lines.

> **TIP**
>
> When using the Scala 2 REPL, use the `:paste` mode when multiple lines need to be parsed as a whole. Enter `:paste`, followed by the code you want to enter, then finish with Ctrl-D.

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 array: Array[String] = new Array(5)
```

Scala is like Java in that most variables are actually references to heap-allocated objects. Hence, the `array` reference cannot be changed to point to a different `Array`, *but the array elements themselves are mutable*, so the elements can be modified:

```
scala> val array: Array[String] = new Array(5)
val array: Array[String] = Array(null, null, null, null,
null)

scala> array = new Array(2)
1 |array = new Array(2)
  |^^^^^^^^^^^^^^^^^^^
  |Reassignment to val array

scala> array(0) = "Hello"

scala> array
val res1: Array[String] = Array(Hello, null, null, null,
null)
```

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 stockPrice: Double = 100.0
var stockPrice: Double = 100.0

scala> stockPrice = 200.0
stockPrice: Double = 200.0
```

To be clear, we changed the *value* of `stockPrice` itself. However, the "object" that `stockPrice` refers to can't be changed, because `Doubles` in Scala are immutable.

In Java, so-called *primitive* types, `char`, `byte`, `short`, `int`, `long`, `float`, `double`, and `boolean`, are fundamentally different than reference objects. Indeed, there is no object and no reference, just the "raw" value. Scala tries to be consistently object-oriented, so these types *are* actually objects with methods, like reference types (see [Link to Come]). However, Scala compiles to primitives where possible, giving you the performance benefit they provide (see [Link to Come] for details).

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

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

scala> val p = new 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
val res3: Int = 30

scala> p.age = 31; p.age  // Use semicolon to join two
expressions...
val res4: Int = 31
```

NOTE

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

Use immutable values whenever possible to eliminate a class of bugs caused by mutability. For example, a mutable object is dangerous as a key in hash-based maps. If the object 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 object 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 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` literal is just what we need. The following examples show how to create ranges for the types that support them, `Int`, `Long`, `Char`, `BigInt`, which represents integers of arbitrary size, and `BigDecimal`, which represents floating-point numbers of arbitrary size. `Float` and `Double` were supported in Scala 2, but floating point arithmetic makes range calculations error prone. Hence, Scala 3 drops ranges for `Float` and `Double`.

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
0 to 10

scala> 1 until 10             // Int range exclusive,
interval of 1, (1 to 9)
val res1: Range = Range 0 until 10

scala> 1 to 10 by 3           // Int range inclusive, every
```

```
third.
val res2: Range = inexact Range 0 to 10 by 3

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         // Char
val res5: ...immutable.NumericRange[Char] = NumericRange a
to g by

scala> BigInt(1) to BigInt(10) by 3
val res6: ...immutable.NumericRange[BigInt] = NumericRange 1
to 10 by 3

scala> BigDecimal(1.1) to BigDecimal(10.3) by 3.1
val res7: ...immutable.NumericRange.Inclusive[BigDecimal] =
  NumericRange 1.1 to 10.3 by 3.1
```

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 only of `case` clauses, which we saw in "A Sample Application", that do *pattern matching* on the input to the function. There is no function parameter shown explicitly, but when each input is processed, it is passed to the body of the partial function.

For comparison, is a regular function that does pattern matching and similar partial function, adapted from the example we explored in "A

Sample Application":

```
// 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)"

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

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)
```

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. It's 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* is simpler than it might appear. In essence, a partial function will only handle certain 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 a compiler warning for `func`, because it can determine that the match clauses don't handle all possible inputs. It won't complain about `pfunc`, because that situation 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:

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

```
val pfs: PartialFunction[Any,String] =
❶
  case s:String => "YES"
val pfd: PartialFunction[Any,String] = {
❷
  case d:Double => "YES"
}

val pfsd = pfs.orElse(pfd)
❸
```

❶ A partial function that only matches on strings, using the braceless syntax.

❷ A partial function that only matches on doubles, using braces.

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

The next block of code in the script tries different values with the three partial functions to confirm expected behavior. Note that integers are not handled by any combination. 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:

```
def tryPF(x: Any, f: PartialFunction[Any,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)
```

Finally, we can *lift* a partial function into a regular ("total") function that returns an option, `Some(value)`, when the partial function is defined for the input argument and `None` when it isn't. We can also *unlift* a single-parameter function. Here is a session that uses `pfs`:

```
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!
```

Infix Operator Notation

In the previous example, we combined two partial functions using `orElse`. This can be written equivalently in two ways:

```
val pfsd1 = pfs.orElse(pfd)
val pfsd2 = pfs orElse pfd
```

When a method takes a single parameter, you can drop the period after the object and drop the parentheses around the supplied argument. In this case, `pfs orElse pfd` has a cleaner appearance than `pfs.orElse(pfd)`, which is why this syntax is popular. This notation is called *infix notation*, because `orElse` is between the object and argument. This syntax is also called *operator notation*, because it is especially popular when writing libraries where algebraic notation is convenient. For example, you can write your own libraries for matrices and define a method named `*` for matrix multiplication using the `*` "operator". Then you can write expressions like `val matrix3 = matrix1 * matrix2`.

> **TIP**
>
> Scala method names can use most non-alphanumeric characters. When methods are called that take a single parameter, *infix operator notation* can be used where the period after the object and the parentheses around the supplied argument can be dropped.

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`. It uses the `copy` method that is also created automatically for case classes.

The `copy` method allows you to construct new instances of a case class while specifying just the fields that are changing. This is very useful for larger case classes:

```
scala> val p1 = new Point(x = 3.3, y = 4.4)     // Used named
arguments explicitly.
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, ${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 statements or 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)
}
```

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.

Unfortunately, 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 an experimental compiler flag `-Yindent-colons` that enables this capability, but it remains experimental (at the time of this writing), because it "is more contentious and less stable than the rest of the significant indentation scheme." (Quote from this Dotty documentation.)

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, like Java methods. If so, the client code would look like this:

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

That's not nearly as clear and elegant. It would also prevent us from 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, the are *not* identical, they just 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` or `implicit` 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 a suitable value that's 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

The `scala.concurrent.Future` API is another tool for concurrency. Akka uses `Futures`, but you can use them separately when you don't need the full capabilities of actors.

When you wrap some work in a `Future`, the work is executed asynchronously and the `Future` API provides various ways to process the results, such as providing callbacks that will be invoked

when the result is ready. Let's use callbacks here and defer discussion of the rest of the API until [Link to Come].

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 new 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 below.

❷ A `sleep` method to simulate staying busy for a period 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 1000, which it will also return. However, if `i` equals 3, we throw an exception.

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 subclasses of `scala.util.Try`, which encapsulates `try {…} catch {…}` clauses with less boilerplate. We can handle successful code *and* possible exceptions more uniformly. See [Link to Come] for further discussion.

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 might go like this, where the order of the results and the numbers on the right-hand side are arbitrary, as expected:

```
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` above. 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. The implementation of `Future.apply` evaluates this code.

Okay, let's finally get back to implicit parameters. Note the second import statement:

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

`Future` methods use an `ExecutionContext` to run code in separate threads, providing concurrency. These methods use a `given` value of an `ExecutionContext`. For example, here's the whole `Future.apply` declaration (using Scala 3 syntax):

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

In the Scala 2 library, the `implicit` keyword is used instead of `using`. The second parameter list is called a *using clause*.

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.global` value that is declared as `given` (or `implicit` in Scala 2). It 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 someExecutionContext)
```

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

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

The `global` value is declared in a similar way, but our `given` value will take precedence.

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

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

So, when `global` is imported into the current scope, the compiler will use it when methods are called that have a using clause with an `ExecutionContext` parameter, unless we specify a value explicitly. For this to work, only `given` instances that are type compatible with the parameter will be considered.

The details for the new idioms and the reasons for their existence are explained in Chapter 5.

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 type.

Here is an example for a factorial calculator:

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

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

  fact(i, 1L)

(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. We used `Long` because factorials grow in size quickly. So, we didn't want Scala to infer `Int` as the return type. Otherwise, we don't need the type annotation on `factorial`.

However, we *must* declare the return type for `fact`, because it is recursive and Scala's local-scope type inference can't infer the return

type of recursive functions.

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

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

Recursion is a hallmark of functional programming 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 so-called "trampoline" calls, 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): Long =
  @tailrec
  def fact(i: Int, accumulator: Long): Long =
    if i <= 1 then accumulator
    else fact(i - 1, i * accumulator)

  fact(i, 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): Long =
     |   if (i <= 1) 1L
     |   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, add them. So, this function is not tail recursive. (It's 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 very 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 functional programming languages, like Haskell, can infer almost all types, because they do global type inference. Scala can't do this, in part because Scala 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. However, `Map` is actually a trait with concrete subclasses, so you'll sometimes make declarations like this one for a `TreeMap`:

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

Here is a summary of the rules for when explicit type annotations are required in Scala.

WHEN EXPLICIT TYPE ANNOTATIONS ARE REQUIRED

In practical terms, you have to provide explicit type annotations for the following situations:

- 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 annotation.
    - When the inferred return type would be more general than you intended, e.g., `Any`.

The last case is somewhat rare, fortunately.

> **NOTE**
>
> 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 super type of all possible values.

We'll explore Scala's types in [Link to Come].

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

```
//
src/script/scala/progscala3/typelessdomore/MethodOverloadedR
eturn.scala

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

While the `Money` constructor expects a `Double`, we want the user to have the convenience of passing a `String` or a `BigDecimal` (ignoring possible errors). So, we *add* two more `apply` methods to the companion object. Both call the `apply(d: Double)` method the compiler automatically generates for the companion object, corresponding to the primary constructor `Money(value: Double)`.

The two methods have explicit return types. If you try removing them, you'll get a compiler error:

```
scala> case class Money(value: Double)
     | case object Money {
     |  def apply(s: String)     = apply(s.toDouble)  // no
return type
     |  def apply(d: BigDecimal) = apply(d.toDouble)  // no
return type
     | }
4 | def apply(d: BigDecimal) = apply(d.toDouble)
  |                            ^
  |                      Overloaded or recursive method apply
needs return type
3 | def apply(s: String)     = apply(s.toDouble)
  |                            ^
  |                      Overloaded or recursive method apply
needs return type
```

Variadic Argument Lists

Scala supports methods that take *variable argument lists* (sometimes just called *variadic methods*). Consider this contrived example that computes the mean of `Doubless`:

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

scala> object Mean1 {
     |   def calc1(ds: Double*): Double = calc2(ds :_*)
     |   def calc2(ds: Seq[Double]): Double = ds.sum/ds.size
     | }

scala> Mean1.calc1(1.0, 2.0)
val res0: Double = 1.5

scala> Mean1.calc2(Seq(1.0, 2.0))
val res1: Double = 1.5
```

The syntax `ds: Double*` means zero or more `Doubles`, a *variable argument list*. Since `calc1` calls `calc2`, which expects a `Seq[Double]` argument. the unusual syntax `(ds :_*)` is how

you take the variable argument list and convert to a sequence when needed.

Why have both functions? The examples show that both can be convenient for the user. In particular, using `calc1` doesn't require you to wrap values in a `Seq` first.

There are downsides. The API "footprint" is larger with two methods instead of one and the maintenance burden is larger.

Assuming you want both methods, why not use the same name, in particular, `apply`?

```
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.
```

In fact, `ds: Double*$$` is converted to a kind of sequence internally, so effectively the methods look identical at the byte code level.

There's a common idiom to break the ambiguity, add a first Double parameter in the first `apply`, then use a variable list for the rest of the supplied arguments:

```
scala> object Mean {
     |   def apply(d: Double, ds: Double*): Double = apply(d
+: ds)
     |   def apply(ds: Seq[Double]): Double = ds.sum/ds.size
     | }
// defined object Mean

scala> Mean(1.0,2.0)
val res10: Double = 1.5

scala> Mean(Seq(1.0,2.0))
val res11: Double = 1.5

scala> Mean()
1 |Mean()
  |^^^^
  |None of the overloaded alternatives of method apply in
object Mean with types
  | (ds: Seq[Double]): Double
  | (d: Double, ds: Double*): Double
  |match arguments ()

scala> Mean(Nil)
val res12: Double = NaN
```

When calling the second `apply`, the first one constructs a new sequence prepending `d` to `ds` using `d +: ds`. We'll explain this syntax in "Matching on Sequences".

Finally, `Nil` is an object representing an empty sequence with any type of elements.

Reserved Words

Scala reserves some words for defining constructs like conditionals, declaring variables, etc. Some of the reserved words are marked with *(soft)*, which means they can be used as regular identifiers for method and variable names, for example, but they are treated as keywords

when used in particular contexts. All of the soft words are new reserved words in Scala 3. The reason for treating them as soft is to avoid breaking older code that happens to use them as identifiers.

Table 2-1 lists the reserved keywords in Scala.Many are found in Java and they usually have the same meanings in both languages.

Table 2-1. Reserved words

Word	Description	See …
`abstract`	Makes a declaration abstract.	[Link to Come]
`as`	(soft) Used with `given`.	"Context Bounds"
`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"
`class`	Start a class declaration.	[Link to Come]
`def`	Start a method declaration.	"Method Declarations"
`do`	New syntax for `while` and `for` loops without braces. Old Scala 2 `do...while` loop.	"Scala while Loops"
`else`	Start an `else` clause for an `if` clause.	"Scala if Expressions"
`extends`	Indicates that the class or trait that follows is the parent type of the class or trait being declared.	[Link to Come]

Word	Description	See …
`extension`	(soft) Marks one or more extension methods for a type.	"Type Classes"
`false`	`Boolean` *false*.	"Boolean Literals"
`final`	Applied to a class or trait to prohibit deriving child types from it. Applied to a member to prohibit overriding it in a derived class or trait.	[Link to Come]
`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"
`for`	Start a `for` comprehension (loop).	"Scala for Comprehensions"
`forSome`	Used in Scala 2 for *existential type* declarations to constrain the allowed concrete types that can be used. Dropped in Scala 3.	[Link to Come]
`given`	Marks implicit definitions.	Chapter 5
`if`	Start an `if` clause.	"Scala if Expressions"

Word	Description	See …
`implicit`	Marks a method or value as eligible to be used as an *implicit* type converter or value. Marks a method parameter as optional, as long as a type-compatible substitute object is in the scope where the method is called.	Chapter 5
`import`	Import one or more types or members of types into the current scope.	"Importing Types and Their Members"
`lazy`	Defer evaluation of a `val`.	"lazy val"
`match`	Start a pattern-matching clause.	Chapter 4
`new`	Create a new instance of a class.	[Link to Come]
`null`	Value of a reference variable that has not been assigned a value.	[Link to Come]
`object`	Start a *singleton* declaration: a `class` with only one instance.	"A Taste of Scala"
`opaque`	(soft) Declares a special type alias with zero runtime overhead.	[Link to Come]
`open`	(soft) Declares a concrete class is open for subclassing.	[Link to Come]
`override`	Override a *concrete* member of a type, as long as the original is not marked `final`.	[Link to Come]

Word	Description	See …
`package`	Start a package scope declaration.	“Organizing Code in Files and Namespaces”
`private`	Restrict visibility of a declaration.	[Link to Come]
`protected`	Restrict visibility of a declaration.	[Link to Come]
`requires`	Deprecated. Was used for *self-typing*.	[Link to Come]
`return`	Return from a function.	“A Taste of Scala”
`sealed`	Applied to a parent type. Requires all derived types to be declared in the same source file.	“Sealed Class Hierarchies and Enumerations”
`super`	Analogous to `this`, but binds to the parent type.	[Link to Come]

Word	Description	See …
`then`	New syntax for `if` expressions	"Scala if Expressions"
`this`	How an object refers to itself. The method name for *auxiliary constructors*.	[Link to Come]
`throw`	Throw an exception.	"Using try, catch, and finally Clauses"
`trait`	A *mixin module* that adds additional state and behavior to an instance of a class. Can also be used to just declare methods, but not define them, like a Java interface.	[Link to Come]
`try`	Start a block that may throw an exception.	"Using try, catch, and finally Clauses"
`true`	`Boolean` *true*.	"Boolean Literals"
`type`	Start a *type* declaration.	"Abstract Types Versus Parameterized Types"
`using`	(soft) Scala 3 alternative to `implicit` for *implicit arguments*.	Chapter 5
`val`	Start a read-only "variable" declaration.	"Variable Declarations"

Word	Description	See …
`var`	Start a read-write variable declaration.	"Variable Declarations"
`while`	Start a `while` loop.	[Link to Come]
`with`	Include the trait that follows in the class being declared or the object being instantiated.	[Link to Come]
`yield`	Return an element in a `for` comprehension that becomes part of a sequence.	"Yielding New Values"
`_`	A placeholder, used in imports, function literals, etc.	Many
`:`	Separator between identifiers and type annotations.	"A Taste of Scala"
`=`	Assignment.	"A Taste of Scala"
`=>`	Used in *function literals* to separate the parameter list from the function body.	[Link to Come]
`<-`	Used in `for` comprehensions in *generator* expressions.	"Scala for Comprehensions"
`<:`	Used in *parameterized* and *abstract type* declarations to constrain the allowed types.	[Link to Come]
`<%`	Used in *parameterized* and *abstract type* "view bounds" declarations.	[Link to Come]
`>:`	Used in *parameterized* and *abstract type* declarations to constrain the allowed types.	[Link to Come]

Word	Description	See …
#	Used in *type projections*.	[Link to Come]
@	Marks use of an *annotation.*	[Link to Come]

Some Java methods use names that are reserved words by Scala, for example, `java.util.Scanner.match`. To avoid a compilation error, surround the name with single back quotes (“back ticks”), 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

Scala 3 expands 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_c5d_e6f_789  // 82090347159025545
val z: BigDecimal = 123_456_789.0123   // 123456789.0123
```

Scala 3 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.

The ability to use numeric literals for library and user-defined types like `BigInt` and `BigDecimal` is implemented with a new trait called `FromDigits`.

For `Long` literals, it is necessary to append the `L` or `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. 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 number is specified that 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
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, `3.` and `3.e5` are disallowed. Use `3.0` and `3.0e5` instead. Otherwise it would be ambiguous; do you mean 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.

Boolean Literals

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

```
scala> val b1 = true
b1: Boolean = true

scala> val b2 = false
b2: 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, i.e., 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'
```

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 (\)

Note that *nonprintable* Unicode characters like `\u0009` (tab) are not allowed. Use the equivalents like `\t`. Recall that three Unicode characters were mentioned in Table 2-1 as valid replacements for corresponding ASCII sequences, ⇒ for =>, → for ->, and ← for <-.

String Literals

A string literal is a sequence of characters enclosed in double quotes or *triples* of double quotes, i.e., """..."""".

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"
```

The string literals bounded by triples of double quotes are also called *multiline* string literals. These strings can cover several lines; 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 \ characters that don't form valid Unicode or escape sequences, like the valid sequences listed in Table 2-3. Regular expressions are a good example, which use lots of escaped characters with special meanings. Conversely, if escape sequences appear, they aren't interpreted.

Here are three example strings:

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

Fourth line"""
```

Note that we had to add a space before the trailing `"""` in the second example to prevent a parse error. Trying to escape the second `"` that ends the `"Scala is great!"` quote, i.e., `"Scala is great!\"`, doesn't work.

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, `|`. If you want some whitespace indentation, put the whitespace you want after the `|`. Consider this example:

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

def hello(name: String) = s"""Welcome!
  Hello, $name!
  * (Gratuitous Star!!)
  |We're glad you're here.
  |  Have some extra whitespace.""".stripMargin

val hi = hello("Programming Scala")
```

The last line prints the following:

```
val hi: String = Welcome!
  Hello, Programming Scala!
  * (Gratuitous Star!!)
```

```
We're glad you're here.
  Have some extra whitespace.
```

Note 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

Scala 2 supported *symbols*, which are *interned* strings, meaning that two symbols with the same "name" (i.e., the same character sequence) will actually refer to the same object in memory. They are deprecated, but still exist in Scala 3. However, the literal syntax has been removed:

```
scala> val sym1 = 'name              // Scala 2 only; single
"tick"
scala> val sym2 = Symbol("name")     // Scala 2 and 3
```

You might see symbols in older code, but don't use them yourself.

Function Literals

As we've seen already, `(i: Int, d: Double) => (i+d).toString` is a function literal of 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,String) => String        = (i, s) => s+i
val f2: Function2[Int,String,String] = (i, s) => s+i
```

Tuple Literals

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. A literal syntax for tuples uses a comma-separated list of values surrounded by parentheses.

Here is an example of a tuple declaration and how we can access elements and use *pattern matching* to extract them:

```
//
src/script/scala/progscala3/typelessdomore/TupleExample.scal
a

scala> val tup = ("Hello", 1, 2.3)                    ❶
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)

scala> tup._1                                         ❷
val res0: String = Hello
```

```
scala> tup._2
val res1: Int = 1

scala> tup._3
val res2: Double = 2.3

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

scala> println(s"s = $s, i = $i, d = $d")
s = Hello, i = 1, d = 2.3
```

❶ Use the literal syntax to construct a three-element tuple. Note the literal syntax is used for the type, too.

❷ Extract the first element of the tuple. Tuple indexing is one-based, by historical convention, not zero-based. The next two lines extract the second and third elements.

❸ Declare three values, `s`, `i`, and `d`, that are assigned the three corresponding fields from the tuple using pattern matching.

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

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

scala> 1 -> "one"
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 or type instance that's assigned to reference variables when there's nothing else for them to refer to. In Scala and Java, it's called `null`.

Using `null` is a giant source of nasty bugs. What `null` really 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 and its two concrete subclasses are `Some`, for when we have a value, and `None`, when we don't.

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 was 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`, if 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.

When You 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`.

Scala 3 introduces a new way to indicate a possible `null` through the type `Scala.Null`, which is a subtype of all other types. Suppose you have a Java `HashMap` to access:

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

import java.util.{HashMap => JHashMap}
❶
```

```
val jhm = new 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`.

Scala 3 introduces *union types*, which we use for `one2` and `two2`. They tell the reader that the value could be either a `String` or a `null`.

There is an optional feature to enable aggressive null checking. If you add the flag `-Yexplicit-nulls`, then 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. I have not enabled this option in the code examples build, because it forces lots of changes to otherwise safe code and it has other implications that are described in more detail in the documentation. However, if you use this same code in a REPL with this flag, 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, who invented the null reference in 1965 while working on a language called ALGOL W, called its invention his "billion dollar" mistake. Use `Option` instead. Consider enabling explicit nulls.

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

The `sealed` keyword tells the compiler that all subclasses 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`.

`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. We'll explain `Nothing` in more detail in [Link to Come], if you get what I mean…

You can also declare a type `final` if you want to prevent users from subtyping it. If you want to encourage subtyping of a concrete class, you can declare it `open`. (This is redundant for abstract types.)

This same constraint on subclassing can now be achieved more concise in Scala 3 with the new `enum` syntax:

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

Organizing Code in Files and Namespaces

Scala adopts the `package` concept that Java uses for namespaces, but Scala offers more flexibility. Filenames don't 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 conventional Java syntax:

```
//
src/main/scala/progscala3/typelessdomore/PackageExample1.sca
la
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/PackageExample2.sca
la
package com:
  package example:
    package pkg1:
      class Class11:
        def m = "m11"

      class Class12:
        def m = "m12"

    package pkg2:
      class Class21:
        def m = "m21"
        def makeClass11 = new pkg1.Class11

        def makeClass12 = new pkg1.Class12

    package pkg3.pkg31.pkg311:
      class Class311:
        def m = "m21"
```

Two packages, `pkg1` and `pkg2`, are defined under the `com.example` package. A total of three classes are defined between the two packages. 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.

The package `pkg3.pkg31.pkg311` shows that you can "chain" several packages together in one statement. It is not necessary to use a separate `package` statement for each package.

If you have package-level declarations, like types, in each of several parent packages that you want to bring into scope, use separate package statements as shown for each level of the package hierarchy with these declarations. Each subsequent package statement is interpreted as a subpackage of the previously specified package, as if we used the block-structure syntax shown previously. The first statement is interpreted as an absolete path.

Following the convention used by Java, 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.

WARNING

Scala does not allow package declarations in scripts, which are implicitly wrapped in an `object`, where package declarations are not permitted.

Importing Types and Their Members

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

```
import java.awt._                    ❶
import java.io.File                  ❷
import java.io.File._                ❸
import java.util.{Map, HashMap}      ❹
```

❶ Import everything in a package, using underscore (_) as a wildcard.

❷ Import an individual type.

❸ Import all static member fields and methods in `File`.

❹ Selectively import two types from `java.util`.

Java uses the asterisk character (`*`) as the wildcard for imports. In Scala, this character is allowed as a method name (e.g., for multiplication), so _ is used instead to avoid ambiguity.

The third line imports all the static methods and fields in `java.io.File`. The equivalent Java import statement would be `import static java.io.File.*;`. Scala doesn't have an

`import static` construct because it treats `object` types uniformly like other types.

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 stuffWithBigInteger() = {

  import java.math.BigInteger.{
    ONE => _,                          ❶
    TEN,                               ❷
    ZERO => JAVAZERO }                 ❸

  // println( "ONE: "+ONE )      // ONE is effectively
undefined
  println( "TEN: "+TEN )
  println( "ZERO: "+JAVAZERO )
}
```

❶ Alias to _ to make it invisible. Use this technique when you want to import everything except a few items.

❷ Import `TEN` from `BigDecimal`. It can be referenced simply as `TEN`.

❸ Import `ZERO` but give it an alias. Use this technique to avoid shadowing other items with the same name. This is used a lot when mixing Java and Scala types, such as Java's `List` and Scala's `List`.

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

Finally, Scala 3 adds new ways to control how implicit definitions are imported. We'll discuss these details in "Givens and Imports", once we understand the new syntax and behaviors for *given instances*.

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/PackageObjects.scal
a
package progscala3.typelessdomore.api

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

class Class1:
  def m = "cm1"

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

In Scala 2, non-type definitions 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 DEFAULT_COUNT = 5
        def countTo(limit: Int = DEFAULT_COUNT) = (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.

Abstract Types Versus Parameterized Types

We mentioned in "A Taste of Scala" that Scala supports *parameterized types*, which are very similar to *generics* in Java. (The terms are somewhat interchangeable, but the Scala community uses parameterized types.) Java uses angle brackets (<...>), while Scala uses square brackets (`[...]`), because < and > are often used for method names.

Because we can plug in almost any type for a type parameter `A` in a collection like `List[A]`, this feature is called *parametric polymorphism*, because generic implementations of the `List` methods can be used with instances of any type `A`.

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 parent type 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 − 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 [Link to Come].

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

These types are declared as members of other types, just like methods and fields. Here is an example that uses an abstract type in a parent class, then makes the type member concrete in child classes:

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

abstract class BulkReader:
  type In
❶
  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, really just like any abstract field or method.

❷ Concrete subtype of `BulkReader` where `In` is defined to be `String`. Note that the type of the `source` parameter must match.

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

Strictly speaking, we don't need to declare the `source` field in the parent class, 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.

Using these readers:

```
scala> import progscala3.typelessdomore.{StringBulkReader,
FileBulkReader}

scala> new StringBulkReader("Hello Scala!").read
val res1: 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 `type` field is used like a type parameter in a parameterized type. In fact, as an exercise, try rewriting the example to use type parameters, e.g., `BulkReader[In]`.

So what's the advantage here of using type members instead of parameterized types? The latter are best for the case where the type parameter has no relationship with the parameterized type, like `List[A]` when `A` is `Int`, `String`, `Person`, etc. A type member works best when it "evolves" in parallel with the enclosing type, as in our `BulkReader` example, where the type member needed to match the "behaviors" expressed by the enclosing type, specifically the `read` method. Sometimes this characteristic is called *family polymorphism* or *covariant specialization*.

For completeness, another use for type members is to provide a convenient alias for a more complicated type. For 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 for a simple alternative:

```
scala> object Foo:
     |   type Record = (String, Double)
     |   def transform(record: Record): Record =
     |     (record._1.toUpperCase, 2*record._2)
// defined object Foo

scala> Foo.transform("hello", 10)
val res0: Foo.Record = (HELLO,20.0)
```

Notice the type shown for `res0`. In fact, concrete type members are always type aliases.

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, methods, and classes. 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

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at m.cronin@oreilly.com

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

Operator Overloading?

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. However, Scala will

compile to native platform primitives when possible for greater efficiency.

Also we've already seen that Scala identifiers can have most nonalphanumeric characters and single-parameter methods can be invoked with *infix operator notation*. So, `1 + 2` is the same as `1.+(2)`.

> **CAUTION**
>
> 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.

In fact, you aren't limited to "operator-like" method names when using infix notation. It is common to write code like the following, where the dot and parentheses around `println` are omitted:

```
scala> Seq("one", "two") foreach println
one
two
```

Use this convention cautiously, as these expressions can sometimes be confusing to read or parse.

To bring a little more discipline to this practice, Scala 3 now issue a deprecation warning if a one-parameter method is declared *without* the annotation `@infix`, but is used with infix notation. However, to support traditional practice, most of the familiar collection

"operators", like `map`, `foreach`, etc. are declared with `@infix`, so our `foreach` example still works in Scala 3 without a warning.

If you still want to use infix notation and you want to avoid the deprecation warning for a method that isn't annotated with `@infix`, put the name in "back ticks":

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

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

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

❶ Normal usage.

❷ Triggers a deprecation warning.

❸ Accepted usage, but odd looking.

You can also define your own operator methods with symbolic names. Suppose you want to allow users to create `java.io.File` objects 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.alpha
import java.io.File

case class Path(
    value: String, separator: String =
Path.defaultSeparator):  ❶
  val file = new File(value)
```

```
  override def toString: String = file.getPath
❷

  @alpha("concat") def / (node: String): Path =
❸
    copy(value + separator + node)
❹

object Path:
  val defaultSeparator = sys.props("file.separator")
```

❶ Use the operating systems default path separator string as the default separator when constructing a `File` object.

❷ How to override the default `toString` method. Here, I use the path string from `File`.

❸ I'll explain the `@alpha` in a moment.

❹ Use the case class copy method to create a new instance, changing only the `value`.

Now users can create new `File` objects 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
```

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 `@alpha` annotation is recommended for methods with symbolic names. It will be required in a future release of Scala 3. 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 Java code, which doesn't support invoking methods with symbolic names. However, the name `concat` can't be used in Scala code, as shown at the end of the example. It only affects the byte code produced by the compiler.

The `@infix` annotation is *not* required for methods that 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, in many contexts. The same rules using the `@infix` annotation apply:

```
// src/script/scala/progscala3/rounding/InfixType.scala
import scala.annotation.{alpha, infix}

@alpha("BangBang") 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 bangbang[A,B](a: A, b: B)            ❹
val ab1: Int bangbang String = 1 bangbang "one"
val ab2: Int bangbang String = bangbang(1, "one")
```

❶ Some type declaration with two type parameters.

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 non-infix notation on the right-hand side.

❹ These three lines behave the same, but note the use of `@infix` now.

To recap:

- Annotate symbolic operator definitions with `@alpha("some_name")`.
- Annotate alphanumeric types and methods with `@infix` if you want to allow their use with infix notation.

Related to infix notation is postfix notation, where a method with no parameters can be invoked without the period. However, postfix invocations can sometimes be even more ambiguous and confusing, so Scala requires that you explicitly enable this feature first, in one of several ways:

1. Add the import statement `import scala.language.postfixOps` to the source code.

2. Invoke the compiler with the option `-language:postfixOps` "feature flag".

There is no `@postfix` annotation.

> **TIP**
>
> The SBT build for the examples is configured to use the `-feature` flag that enables warnings when features are used, like postfix expressions, that should be enabled explicitly.

Dropping the punctuation by using infix or even postfix expresses can make your code cleaner and help create elegant programs that read more naturally, but avoid cases that are actually harder to understand.

Allowed Characters in Identifiers

Here is a summary of the rules for characters in identifiers for method and type names, variables, etc:

Characters

Scala allows all the printable ASCII characters, such as letters, digits, the underscore (`_`), and the dollar sign (`$`), with the exceptions of the "parenthetical" characters, `(`, `)`, `[`, `]`, `{`, and `}`, and the "delimiter" characters, `` ` ``, `'`, `'`, `"`, `.`, `;`, and `,`. Scala allows the characters between \u0020 and \u007F that are not in the sets just shown, such as mathematical symbols, the so-called *operator characters* like `/` and <, and some other symbols. This includes white space characters, discussed below.

Reserved words can't be used

We listed the reserved words in "Reserved Words". Recall that some of them are combinations of operator and punctuation characters. For example, a single underscore (_) is a reserved word!

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.

"Back-quote" literals

An identifier can also be an arbitrary string between two back quote characters, e.g., `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 white space in identifiers.) Also use back quotes to invoke a method or variable in a Java class when the name is identical to a Scala reserved word, e.g., `java.net.Proxy.`type`()`.

Pattern-matching identifiers

In pattern-matching expressions (for example, "A Sample Application"), 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.

Syntactic Sugar

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` where 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.

> **CAUTION**
>
> Avoid making up operator symbols when an established ASCII 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 the use of parentheses in methods with no parameters.

If a method takes no parameters, you can define it without parentheses. Callers must invoke the method without parentheses. Conversely, if you add empty parentheses to your definition, callers must add the parentheses.

For example, `List.size` has no parentheses, so you write `List(1, 2, 3).size`. If you try `List(1, 2, 3).size()`, you'll get an error.

However, exceptions are made for no-parameter methods in Java libraries. For example, the `length` method for `java.lang.String` does have parentheses in its definition, because Java requires them, but Scala lets you write both `"hello".length()` and `"hello".length`. This flexibility with Scala-defined methods was also allowed in earlier releases of Scala, but Scala 3 now requires that usage match the definition.

A convention in the Scala community is to omit parentheses for no-parameter methods that have no side effects, like returning the size of a collection, which could actually be a precomputed, immutable field in the object. So, many no-parameter methods could be interpreted as simply reading a field. However, when the method performs side effects or does extensive work, parentheses are added by convention, providing a hint to the reader of nontrivial activity, for example `myFileReader.readLines()`.

Why bother with optional parentheses in the first place? They make some method call chains read better as expressive, self-explanatory "sentences" of code:

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

def isEven(n: Int) = (n % 2) == 0

Seq(1, 2, 3, 4) filter isEven foreach println
```

The second line is "clean", but on the edge of non-obvious to read. Here is the same code repeated four times with progressively less explicit detail filled in. The last line is the original:

```
Seq(1, 2, 3, 4).filter((i: Int) => isEven(i)).foreach((i:
Int) => println(i))
Seq(1, 2, 3, 4).filter(i => isEven(i)).foreach(i =>
println(i))
Seq(1, 2, 3, 4).filter(isEven).foreach(println)
Seq(1, 2, 3, 4) filter isEven foreach println
```

The first three versions are more explicit and hence better for the beginning reader to understand. However, once you're familiar with the fact that `filter` is a method on collections that takes a single

argument, `foreach` loops over a collection, and so forth, the last, "Spartan" implementation is much faster to read and understand. The other two versions have more visual noise that just get in the way, once you're more experienced. Keep that in mind as you learn to read Scala code.

To be clear, this expression works because each method we used took a single parameter. If you tried to use a method in the chain that takes zero or more than one parameter, it would confuse the compiler. In those cases, put some or all of the punctuation back in.

The previous explanation glossed over an important detail. The four versions are *not exactly* the same code with different details inferred by the compiler, although all four behave the same way. Consider the function arguments passed to `filter` in the second and third examples. They are different as follows:

- `filter(i => isEven(i))` passes an *anonymous* function that *calls* `isEven`.
- `filter(isEven)` passes `isEven` itself as the function. Note that `isEven` is a *named* function.

The same distinction applies to the functions passed to `foreach` in the second and third examples.

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
```

Left vs. Right Associative Methods

Usually, method invocation using infix operator notation simply bind in left-to-right order, i.e., they are *left-associative*. Don't all methods work this way? No. In Scala, 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 introduced by 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*.

> **TIP**
>
> Any method whose name ends with a `:` binds to the *right*, not the *left*, in infix operator notation.

Enumerations and Algebraic Data Types

While it's common to declare a type hierarchy to represent all the possible "kinds" of some abstraction, sometimes we know the list of them is fixed and mostly what we need are unique "flags" to indicate each one.

Take for example the days of the week, where we have seven fixed values. An enumeration for the English days of the week can be declared as follows:

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

enum WeekDay(val fullName: String) derives Eql:
❶
  case Sun extends WeekDay("Sunday")
❷
  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)
❸

import WeekDay._
assert(Sun.fullName == "Sunday")
assert(Sun.ordinal == 0)
❹
assert(Sun.isWorkingDay == false)
assert(WeekDay.valueOf("Sun") == WeekDay.Sun)
❺

// Similar assertions for the other days not shown...

val sorted = WeekDay.values.sortBy(_.ordinal).toSeq
❻
assert(sorted == List(Sun, Mon, Tue, Wed, Thu, Fri, Sat))
```

❶

Declare an enumeration, similar to declaring a class. You can have optional fields as shown. Declare them with `val` if you want them to be accessible, e.g., `WeekDay.Sun.fullName`. The `derives Eql` clause lets use do comparisons with == and `!=`. We'll explain this construct in "Type Class Derivation" and [Link to Come].

❷ The values are declared using the `case` keyword.

❸ You can define methods.

❹ The ordinal value matches the declaration order.

❺ You can lookup a enumeration value by its name.

❻ The `WeekDay.values` order does not match the declaration order, so we sort by the `ordinal`.

Scala 3 introduced a new syntax for enumerations, which we saw briefly in "Sealed Class Hierarchies and Enumerations".[1] 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), such as transforming an element or combining two of them with an operation can only yield another element in the set.

Consider the following example:

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

object Scala2ADT:
  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 Scala3ADT:
  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))))

Scala2ADT.tree
❻
Scala3ADT.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 need to be imported.

❻ Is the output the same for these two lines?

We saw sealed type hierarchies before in "Sealed Class Hierarchies and Enumerations". The `enum` syntax provides the same benefits as sealed type hierarchies, but with much less code.

The types of the `tree` values are slightly different:

```
scala> Scala2ADT.tree
val res1: Scala2ADT.Branch[Int] = Branch(...)

scala> Scala3ADT.tree
val res2: Scala3ADT.Tree[Int] = Branch(...)
```

One last point; you may have noticed that `Branch` and `Leaf` don't extend `Tree`, while in `WeekDay` above, 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 `val fullName: String` field declared by `WeekDay`.

Interpolated Strings

We introduced *interpolated* strings in "A Sample Application". 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`, a `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")
```

When using interpolated strings, to write a literal dollar sign `$`, use two of them, `$$`.

There are two other kinds of interpolated strings. The first kind provides *printf* formatting and uses the prefix `f`. The second kind 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.

In this example, we use two dollar signs, $$, to print a literal US dollar sign, and two percent signs, `%%`, 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 the 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 control characters. 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 actually define our own string interpolators, but we'll need to learn more about *context abstractions* first. See "Build Your Own String Interpolator" for details.

Scala if 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.

Scala 2 `if` expressions used braces just like Java's `if` statements, but in Scala 3, `if` expressions can also use the new conventions discussed in the previous section. A simple example:

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

(0 until 6) foreach { n =>
  if n%2 == 0 then
    println(s"$n is even")
  else if n%3 == 0 then
    println(s"$n is divisible by 3")
  else
    println(n)
}
```

Recall from "New Scala 3 Syntax" that the `then` keyword is required only if you pass the `-new-syntax` flag to the compiler or REPL, which we use in the code examples SBT file. However, if you don't use that flag, you must wrap the predicate expressions, like `n%2 == 0`, in parentheses.

The bodies of each clause are so concise, we can write them on the same line as the `if` or `else` expressions:

```
// src/script/scala/progscala3/rounding/If2.scala

(0 until 6) foreach { n =>
  if n%2 == 0 then println(s"$n is even")
  else if n%3 == 0 then println(s"$n is divisible by 3")
  else println(n)
}
```

Here are the same examples using the curly brace syntax required by Scala 2 and optional for Scala 3:

```
// src/script/scala-2/progscala3/rounding/If.scala

(0 until 6) foreach { n =>
  if (n%2 == 0) {
    println(s"$n is even")
  } else if (n%3 == 0) {
    println(s"$n is divisible by 3")
  } else {
    println(n)
  }
}
```

```
// src/script/scala-2/progscala3/rounding/If2.scala

(0 until 6) foreach { n =>
  if (n%2 == 0) println(s"$n is even")
  else if (n%3 == 0) println(s"$n is divisible by 3")
  else println(n)
}
```

The older syntax can still be used, even with compiler flags for the new syntax, but you can also use flags to require the older syntax, as discussed in "New Scala 3 Syntax".

What is the type of the returned value if objects of different types are returned by different branches? The type of the returned value will be the so-called *least upper bound* (LUB) of all the branches, the closest parent type that matches all the potential values from each clause.

In the following example, the LUB 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 Operators

Scala uses the same conditional operators as Java. Table 3-1 provides the details.

Table 3-1. Conditional 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 equals	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 equals	The value on the left is less than or equal to the value on the right.
`==`	equals	The value on the left is the same as the value on the right.
`!=`	not equals	The value on the left is not the same as 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. What happens if you use `||` instead? Also, we don't us `if` here, because we just want to know the `Boolean` value of the conditional.

Most of the operators behave as they do in Java and other languages. An exception is the behavior of == and its negation, `!=`. In Java, == compares instance references only. It doesn't check logical equality, i.e., comparing field values. You must call the `equals` method for that purpose. Instead in Scala, == and `!=` call the `equals` method for the left-hand instance. You actually don't implement `equals` yourself very often in Scala, because you mostly only compare case class instances and the compiler generates `equals` automatically for case classes! You can override it when you need to, however.

If you need to determine if two instances are identical references, use the `eq` method.

See [Link to Come] for more details.

Scala for Comprehensions

Another familiar control structure that's particularly feature-rich in Scala is the `for` loop, called the `for` *comprehension*. They are expressions, not statements as in Java.

The term *comprehension* comes from functional programming. It expresses the idea that we are traversing one or more collections of

some kind, “comprehending” something new from it, such as another collection. Python *list comprehensions* are 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
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.”

Because this form doesn’t return anything, it only performs side effects. These kinds of `for` comprehensions are sometimes called `for` *loops*, analogous to Java `for` loops.

In the older Scala 2 syntax, this example would be 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)
```

> **TIP**
>
> From now on, in the examples that follow, 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 [Link to Come].

Generator Expressions

The expression `i <- 0 until 10` is called a *generator expression*, so named because it *generates* individual values in some way. The left arrow operator (`<-`) is used to iterate through any collection or iterator instance that supports sequential access, such as `Seq` and `Vector`.

Guards: Filtering Values

We can add `if` expressions, called *guards*, to filter for just elements we want to keep:

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

for
  n <- 0 to 6
  if n%2 == 0
do println(n)
```

The output is the number 0, 2, 4, and 6 (because we use `to` to make the `6` inclusive). 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
  n <- 0 to 10  // Note: 0 to 10, inclusive
  if n%2 == 0
yield n

assert(evens == Seq(0, 2, 4, 6, 8, 10))
```

Each iteration through the `for` expression "yields" a new value, named `n`. These are accumulated into a new collection that is assigned to the variable `evens`.

The type of the collection resulting from a comprehension expression is inferred from the type of the collection being iterated over. `Seq` is a *trait* and the actual concrete instance returned is of type `IndexedSeq`.

In the following example, a `Vector[Int]` is converted to a `Vector[String]`:

```
//
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:

```
// 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 ==
        Seq("Friday", "Monday", "Tuesday", "Wednesday",
"Thursday"))
```

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` is 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",
...))

val goodDays2 = for           // second, more concise pass
  case Some(day) <- dayOptions
  fn = day.fullName
yield fn
assert(goodDays1 == Seq("Monday", "Tuesday", "Wednesday",
...))
```

Imagine that we called some services to return days of the week. The services returned `Options`, because some of the services couldn't return anything, so they returned `None`. Now we want to remove and ignore the `None` values.

In the first expression of the first `for` comprehension, 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 `None`s. 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.

The second `for` comprehension makes this filtering even cleaner and more concise, using *pattern matching*. The expression `case Some(day) <- dayOptions` only succeeds when the instance is a `Some`, also skipping the `None` values, and it extracts the value into to `day`, all in one step.

To recap the difference between using the left arrow (`<-`) versus the equals sign (=), use the arrow when you are iterating through a collection and extracting values. Use the equals sign when you're assigning a value from another value that doesn't involve iteration. A limitation is that the first expression in a `for` comprehension has to be an extraction/iteration using the left arrow. If you really need to define a value first, put it before the `for` 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 seldom used. It executes a block of code as long as a condition is true:

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

var count = 0
while count < 10
  count += 1
  println(count)

assert(count == 10)
```

Scala do-while Loops

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.scal
a

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. But exceptions are still supported. In particular, they are needed when using Java 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 [Link to Come].

Scala uses pattern matching to specify the exceptions to be catched.

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 In Scala 3, we don't need an object to wrap the "main" method. By using the `@main` annotation, we can name the method whatever we want and we can specify the number and types of the arguments expected. Here, we just expect zero or more strings.

❸ Declare the `source` to be an `Option`, so we can tell in the `finally` clause if we have an actual instance or not. We use a mutable variable, but it's hidden inside the implementation and thread safety isn't a concern here.

❹ Start of `try` clause.

❺ `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`.

❻ Catch nonfatal errors. For example, out of memory would be fatal.

❼ 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 matches to pick the exceptions you want to catch. This is more compact and more flexible than Java's use of separate `catch` clauses for each exception. In this case, the clause `case NonFatal(ex) => …` uses `scala.util.control.NonFatal` to match any exception that isn't considered fatal.

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 "main" method, the handles would be cleaned up anyway on exit, but you'll want to close resources in other contexts.

> **TIP**
>
> 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 116 lines
Closing README.md...
Non fatal exception! java.io.FileNotFoundException: foo/bar
(...)
```

You throw an exception by writing `throw new MyBadException(...)`. If your custom exception is a `case` class, you can omit the `new`.

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.

```
// 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 previous `TryCatch` example, which continues processing the arguments specified.

See the `scala.util.Using` documentation for a few other ways this utility can be used. For more sophisticated approaches to error handling, see [Link to Come].

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 subclass 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 another *optional feature*, like postfix expressions, which we saw earlier. So we add the import statements to tell the compiler we know what we're doing.

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".

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" 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())
```

Okay, 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 a call by name, this time repeatedly evaluating *two* by-name parameters; an implementation of 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 with traditional brace syntax.

It's important to note that the by-name parameters are evaluated every time they are referenced. So, by-name parameters are in a sense *lazy*, because evaluation is deferred, but possibly repeated over and over again. Scala also provides lazy values. By the way, this implementation shows how "loop" constructs can be replaced with recursion.

Unfortunately, this ability to define our own control structures only works with the old Scala 2 syntax using parentheses and braces. If `continue` really behaved like `while` or similar built-in constructs, we would be able to write the last two examples. The syntax uniformity with user-defined constructs is a nice feature we must give up if we use the new syntax… or do we??

```
count = 0
continue (count < 5)
  println(s"at $count")
  count += 1
assert(count == 5)
```

This actually parses, but it executes in an unexpected way. Here it is again, annotated to explain what actually happens:

```
continue (count < 5)      // Returns a "partially-applied
function"
  println(s"at $count")   // A separate statement that prints
"at 0" once
  count += 1              // Increments count once
```

A *partially-applied function* is one where we provide some, but not all arguments, returning a new function that will accept the remaining arguments (see [Link to Come]). Here's what the REPL will print for that line:

```
scala> continue (count < 5)
val res1: (=> Unit) => Unit =
Lambda$11103/0x00000008048c0040@4d0f9ec0
```

This is an anonymous function, implemented with a JDK *lambda* on the JDK, that takes a by-name parameter returning `Unit` (recall the second parameter for `continue`) and then returns `Unit`.

The second and third lines are not treated as part of the `body` that should be passed to `continue`. They are treated as single statements that are evaluated once.

So, the optional braces don't work.

lazy val

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 object, 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 object needs to be initialized lazily so that other initializations can happen first.

We'll explore the last scenario when we discuss [Link to Come].

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 when we referenced `connection` the first time.

One-time evaluation makes little sense for a mutable field. Therefore, the `lazy` keyword is not allowed on `vars`.

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 [Link to Come]).

There is a `@threadUnsafe` annotation you can add to a lazy val. It causes the initialization to use a faster mechanism which is not thread-safe, so use with caution.

Traits: Interfaces and "Mixins" in Scala

Until now, I have emphasized the power of functional programming in Scala. I waited until now to discuss Scala's features for object-oriented programming, 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 [Link to Come], 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

class Service(name: String):
  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)}"))
```

We ask the service to do some (random) work and get this output:

```
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 are two traits, one that defines the abstraction with no concrete members and the other that implements the abstraction for "logging" to standard output:

```
trait Logging:
  def info    (message: String): Unit
  def warning(message: String): Unit
  def error   (message: String): Unit

trait StdoutLogging extends Logging:
  def info    (message: String) = println(s"INFO:
$message")
```

```
  def warning(message: String) = println(s"WARNING:
$message")
  def error   (message: String) = println(s"ERROR:
$message")
```

Note that `Logging` is pure abstract. It works *exactly* like a Java interface. It is even implemented the same way in JVM byte code.

Finally, let's declare a service that "mixes in" logging and use it:

```
val service2 = new Service("two") 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

(1 to 3) foreach (i => println(s"Result:
${service2.work(i)}"))
```

We override the `work` method to log when we enter and before we leave the method. Scala requires the `override` keyword when you override a concrete method in a parent class. This prevents mistakes when you didn't know you were overriding a method, for example from a library parent class and it catches misspelled method names that aren't actually overrides! Note how we access the parent class `work` method, using `super.work`.

Here is the output:

```
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)
```

> **WARNING**
>
> Be very careful about overriding concrete methods! In this case, we don't change the behavior of the the parent-class method. We just log activity, then call the parent method, then log again. We are careful to return the result unchanged that was returned by the parent 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're actually *modifying* the behavior of `work`, in order to inject logging, but we are not changing its "contract" with clients, that is, its external behavior.[3]

If we needed multiple instances of `Service with StdoutLogging`, we should declare a class:

```
class LoggedService(name: String)
  extends Service(name) with StdoutLogging:
  ...
```

Note how we pass the `name` argument to the parent class `Service`. To create instances, `new LoggedService("three")` works as you would expect it to work.

There is a lot more to discuss about traits and *mixin composition*, as we'll see.

Recap and What's Next

We've covered a lot of ground in these first 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.

---

1 You can find a Scala 2 version of `WeekDay` in the code examples, `src/script/scala-2/progscala3/rounding/WeekDay.scala`.

2 Not to be confused with the keyword `using` that we discussed in "A Taste of Futures".

3 That's not strictly true, in the sense that the extra I/O has changed the code's interaction with the "world."

Chapter 4. Pattern Matching

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at m.cronin@oreilly.com

Scala's pattern matching provides deep inspection and decomposition of objects in a variety of ways. 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" and "Partial Functions". We'll start with a quick tour of common and straightforward usage examples, then explore more advanced scenarios.

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.sc
ala

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[Any]`.

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.

❸ When two cases are handled the same, how to match on either case.

❹ Here's another example with strings.

❺ Match on a two-element tuple where the elements are of any type, extract the elements into the variables `x` and `y`.

❻

> Match all other inputs, where `unexpected` is the variable to which the value of `x` is assigned. The variable name is arbitrary. Because no type annotation is given, `Any` is inferred. This functions as the "default" clause.

I pass a partial function to `Seq.map()`. It's actually *total* because the last clause matches anything.

I didn't include a clause with a floating-point literal, because matching on floating-point literals is a bad idea, as rounding errors mean two values that appear to be the same often differ in the least significant digits.

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.

Pattern matching is an expression, so it returns a value. In this case, all clauses return strings, so the return type of the whole thing is `List[String]`. The compiler infers the closest supertype (also called the *least upper bound*) for types of values returned by all the `case` clauses.

We passed a *partial function literal* to `map`, rather than a typical anonymous function. Recall that the literal syntax requires `case` statements.

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.s
cala

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. We don't actually need to, because we already have `x`.

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`. Otherwise, just using `x` wouldn't be sufficient, as it has type `Any`.

We used the braceless syntax inside the function, because now we have a `match` expression, whereas before we had a partial function literal where the braces were necessary to mark the whole anonymous function. In general, using a partial function is more concise, because we eliminate the need for `x => x match`.

> **TIP**
>
> When you use any of the collection methods like `map` and `foreach`, and you will pattern match, use a partial function.

There are a few rules and gotchas to keep in mind when writing `case` clauses. The compiler assumes that a term that starts with a capital letter is a type name, while a term that begins with a lowercase letter is assumed to be the name of a variable that will hold an extracted or matched value.

This rule can cause surprises, as shown in the following example, where we want to match on a particular value that's passed into a method:

```
//
src/script/scala/progscala3/patternmatching/MatchSurprise.sc
ala

def checkY(y: Int): Seq[String] =
  for
    x <- Seq(99, 100, 101)
  yield
    x match
      case y => "found y!"       ❶
      case i: Int => "int: "+i  // <2> ERROR: Unreachable
code

checkY(100)
```

We want the ability to pass in a specific value for the first `case` clause, rather than hard-code it. So, we might expect the first `case` clause to match when `x` equals `y`, which holds the value of `100`, but this is what we actually get:

```
def checkY(y: Int): Seq[String]
10 |      case i: Int => "int: "+i  // <2> ERROR:
Unreachable code
   |           ^^^^^^
   |           Unreachable case
```

Calling `checkY(100)` returns `found y!` for all three numbers.

The `case y` actually means, "match anything (because there is no type annotation) and assign it to this *new* variable named `y`." The `y` here is not interpreted as a reference to the method parameter `y`. So, we actually wrote a default, match-all clause first, triggering the "Unreachable case" warning, so we will never reach the second `case` expression.

The solution is to use "back ticks" to indicate we really want to match against the value held by `y`:

```
//
src/script/scala/progscala3/patternmatching/MatchSurpriseFix
.scala

def checkY(y: Int): Seq[String] =
  for
    x <- Seq(99, 100, 101)
  yield
    x match
      case `y` => "found y!"    // Note the backticks
      case i: Int => "int: "+i
```

> **WARNING**
>
> 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 back ticks. Conversely, a term that begins with an uppercase letter is assumed to be a type name.

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)  => "int 1"
     | }
scala.MatchError: None (of class scala.None$)
  ...

2 |  case Some(i)  => "int 1"
  |  ^
  |  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]`, but the partial function isn't exhaustive, so we get a warning that `None` isn't covered and and a `MatchError` when a `None` is encountered. The fix is straightforward:

```
//
src/script/scala/progscala3/patternmatching/MatchExhaustiveF
ix.scala
```

```
scala> val result3 = seq3 map {
     |   case Some(i)  => "int 1"
     |   case None     => ""
     | }
val result3: Seq[String] = List(int 1, "", int 1, "")
```

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

val nonEmptySeq    = Seq(1, 2, 3)
❶
val emptySeq       = Seq.empty[Int]
val nonEmptyVector = Vector(1, 2, 3)
❷
val emptyVector    = Vector.empty[Int]
val nonEmptyMap    = Map("one" -> 1, "two" -> 2, "three" ->
3)       ❸
val emptyMap       = Map.empty[String,Int]

def seqToString[T](seq: Seq[T]): String = seq match
❹
  case head +: tail => s"($head +: ${seqToString(tail)})"
❺
  case Nil => "Nil"
❻

val results = Seq(
❼
    nonEmptySeq, emptySeq, nonEmptyVector, emptyVector,
    nonEmptyMap.toSeq, emptyMap.toSeq) map {
  seq => seqToString(seq)
}

assert(results == Seq(
  "(1 +: (2 +: (3 +: Nil)))",
  "Nil",
  "(1 +: (2 +: (3 +: Nil)))",
```

```
  "Nil",
  "((one,1) +: ((two,2) +: ((three,3) +: Nil)))",
  "Nil"))
```

❶ Construct a nonempty scala-lang.org/api/current/scala/collection/immutable/Seq.html[`Seq[Int]`]. A `List` is actually returned, which uses a linked-list implementation. Accessing the head of a list is *O(1)*, while accessing an arbitrary element is *O(N)*, where *N* is the length of the list. The next line shows the idiomatic way of constructing an empty `Seq[Int]`.

❷ Construct a nonempty `Vectors[Int]`, a subtype of `Seq` with *O(1)* access patterns, followed by an empty `Vector[Int]`.

❸ Construct a nonempty `Map[String,Int]`, which *isn't* a subtype of `Seq`; we'll use `toSeq` to return a sequence of key-value tuples.

❹ 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. Note this method is not tail recursive.

❺ There are two match clauses and they are exhaustive. The first matches on any nonempty `Seq`, extracting the head, the first element, and the tail, which is the rest of the `Seq`. (`Seq` has `head` and `tail` methods, but here these terms are interpreted as variable names as usual for `case` clauses.) The body of the clause constructs a `String` with the head followed by `+:` followed by the result of calling `seqToString` on the tail.

❻ 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

`Seq` can always be interpreted as terminating with an empty instance of the same type, although only some types, like `List`, are actually implemented that way.

7. Put the `Seq`s in another `Seq`, calling `Map.toSeq` as required, then iterate through it and call `seqToString` on each one.

`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 `Map`s and not the general case.

There are two new kinds of `case` clauses. The first, `head +: tail`, matches the head element and the tail `Seq` (the remainder) of a sequence. 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.

I'm calling them "operators" and "methods," but actually that's not quite right in this context; we'll come back to this expression shortly and examine what's going on.

The Scala library has an object called `Nil` for empty lists, but we can use it for matching any empty sequence. The types don't have to match exactly.

Because `Seq` behaves conceptually like a linked list, where each head node holds an element and it points to the tail (the rest of the

sequence), creating a hierarchical structure that is indicated by parentheses inserted, like this example:

```
(1 +: (2 +: (3 +: Nil)))
```

So, we process sequences with just two `case` clauses and recursion. Note that this implies something fundamental about all sequences; they are either empty or not. That sounds trite, but once you recognize simple patterns like this, it gives you a surprisingly general tool for "divide and conquer." The idiom used by `processSeq` is widely reusable.

We can copy and paste the output of the previous examples to reconstruct the original objects, which is not accidental:

```
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 variable argument list of two-element tuples. So, in order to use the sequence `kvs`, we had to use the `:_*` idiom for the compiler to convert it to a variable-argument list.

So, there's an elegant symmetry between construction and pattern matching ("deconstruction") when using `+:`.

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
}
assert(results == Seq(
  "Scala",
  "Clojure, creator Rich Hickey",
  "Lisp, creator John McCarthy"))
```

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.

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"))
```

Parameter Untupling

Consider this example of tuples with no nesting:

```
//
src/script/scala/progscala3/patternmatching/ParameterUntupli
ng.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 tuple syntax here is the implication that it's not exhaustive, when we know it is for three-element tuples. It is also a bit inconvenient. 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 happens to one level:

```
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
```

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 `i` is even.

❷ Match the only other possibility, that `i` 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. This was true in Scala 2's conditional 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, which also works for enums. 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 `Person`,

while also extracting fields inside the instance, in this case `age`.

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 => ...`.

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":

```
// src/script/scala/progscala3/patternmatching/MatchTreeADT.scala

enum Tree[T]:
  case Branch(left: Tree[T], right: Tree[T])
  case Leaf(elem: T)

import Tree._
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 using a sealed class hierarchy in the original example.

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.

> **WARNING**
>
> 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.

Scala wraps Java's regular expressions.[1] Here is an example:

```
//
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 2020",
  "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 2020""",
  "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.

Normally you'll want to use triple-quoted strings for the regexes. Otherwise, you must escape the regex "backslash" constructs, like `\\s` instead of `\s`. You can also define regular expressions by creating new instances of the `Regex` class, as in `new Regex("""\W+""")`, but this isn't as common.

WARNING

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.

More on Type Matching

Consider the following example, where we attempt to discriminate between `List[Double]` and `List[String]` inputs:

```
//
src/script/scala/progscala3/patternmatching/MatchTypes.scala

scala> val results = Seq(Seq(5.5,5.6,5.7), Seq("a", "b"))
map {
     |   case seqd: Seq[Double] => ("seq double", seqd)
     |   case seqs: Seq[String] => ("seq string", seqs)
     |   case other             => ("unknown!", other)
     | }
val results: Seq[(String, Seq[Double | String])] =
  List((seq double,List(5.5, 5.6, 5.7)), (seq double,List(a,
b)))
2 |  case seqd: Seq[Double] => ("seq double", seqd)
  |       ^^^^^^^^^^^^^^^^^
  |       the type test for Seq[Double] cannot be checked at
runtime
3 |  case seqs: Seq[String] => ("seq string", seqs)
  |       ^^^^^^^^^^^^^^^^^
  |       the type test for Seq[String] cannot be checked at
runtime
```

When Scala code runs on the JVM these warnings result from the JVM's *type erasure*, a historical legacy of Java's introduction of *generics* in Java 5, long ago. In order to avoid breaking older code, the JVM byte code doesn't retain information about the actual type parameters that were used for instances of generic (parameterized) types, like `Seq[T]`.

So, the compiler is warning us that, while it can check that a given object is a `Seq`, it can't check at *runtime* that it's a `Seq[Double]` or a `Seq[String]`.

In fact, the second `case` clause for `Seq[String]` is effectively unreachable. Note the output for `results`. It shows that "seq double" was written for both inputs, event the `Seq[String]`!

One ugly, but effective 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.sc
ala

def doSeqMatch[T](seq: Seq[T]): String = seq match
  case Nil => "Nothing"
  case head +: _ => head match
    case _ : Double => "Double"
    case _ : String => "String"
    case _ => "Unmatched seq element"

val results = Seq(Seq(5.5,5.6,5.7), Seq("a", "b"), Nil) map
{
  case seq: Seq[_] => (s"seq ${doSeqMatch(seq)}", seq)
}

assert(results == Seq(
  ("seq Double",  Seq(5.5, 5.6, 5.7)),
  ("seq String",  Seq("a", "b")),
  ("seq Nothing", Seq())))
```

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. As an example, suppose we define the following code to represent the allowed "methods" for HTTP:

```
//
src/script/scala/progscala3/patternmatching/HTTPMethods.scal
a

enum HttpMethod:
❶
  def body: String
❷

  case Connect(body: String)
❸
  case Delete  (body: String)
  case Get     (body: String)
  case Head    (body: String)
  case Options (body: String)
  case Post    (body: String)
  case Put     (body: String)
  case Trace   (body: String)

import HttpMethod._
def handle (method: HttpMethod) = method match
❹
  case Connect (body) => s"Connect: length = ${body.length},
body = $body"
  case Delete  (body) => s"Delete:  length = ${body.length},
body = $body"
  case Get     (body) => s"Get:     length = ${body.length},
body = $body"
  case Head    (body) => s"Head:    length = ${body.length},
body = $body"
  case Options (body) => s"Options: length = ${body.length},
body = $body"
  case Post    (body) => s"Post:    length = ${body.length},
body = $body"
  case Put     (body) => s"Put:     length = ${body.length},
body = $body"
  case Trace   (body) => s"Trace:   length = ${body.length},
body = $body"
```

```
assert(handle(Connect("CONNECT")) == "Connect: length = 7,
body = CONNECT")
assert(handle(Delete ("DELETE"))  == "Delete:  length = 6,
body = DELETE")
assert(handle(Get    ("GET"))     == "Get:     length = 3,
body = GET")
assert(handle(Head   ("HEAD"))    == "Head:    length = 4,
body = HEAD")
assert(handle(Options("OPTIONS")) == "Options: length = 7,
body = OPTIONS")
assert(handle(Post   ("POST"))    == "Post:    length = 4,
body = POST")
assert(handle(Put    ("PUT"))     == "Put:     length = 3,
body = PUT")
assert(handle(Trace  ("TRACE"))   == "Trace:   length = 5,
body = TRACE")
```

❶ Define an `enum HttpMethod`, the equivalent of a sealed class hierarchy. Only the values defined in this file are allowed `HttpMethod` members. Similarly, for a sealed base type, all the derived types must be defined in the same file.

❷ Define a method for the body of the HTTP message.

❸ Define eight methods. Note that each declares a constructor parameter `body: String`, which is a `val` because each of these types is effectively a case class. This `val` *implements* the abstract `def` method in `HttpMethod`.

❹ An *exhaustive* pattern-match expression, even though we don't have a default clause, because the `method` argument can only be an instance of one of the eight cases we defined.

> **TIP**
>
> When pattern matching on an instance of an `enum` or a sealed base class, the match is exhaustive if the `case` clauses cover all the derived types defined in the same source file. Because no user-defined derived types are allowed, the match can never become non-exhaustive as the project evolves, since users are prevented from defining new types.

A corollary is to avoid using `sealed` if the type hierarchy is at all likely to change (see [Link to Come]). Instead, rely on your traditional object-oriented inheritance principles, including polymorphic methods. What if you added a new derived type, either in this file or in another file, and you removed the `sealed` keyword on `HttpMethod`? You would have to find and fix all pattern-match clauses in your code base *and* all your users would have to fix their code.

As a side note, we are exploiting a useful feature for implementing certain methods. An abstract, no-parameter method declaration in a parent type can be implemented by a `val` in a subtype. This is because a `val` has a single, fixed value (of course), whereas a no-parameter method returning the same type can return any value of the type. Hence, the `val` implementation is more restrictive in the return type, which means using it where the method is "called" is always just as safe as calling a method. Why? Because clients of the base-type method have to account for any value being returned, but any one concrete implementation only returns one value. In fact, this is an application of *referential transparency*, where we are substituting a value for an expression that *should* always return the same value!

> **TIP**
>
> When declaring an abstract field in a parent type, use a no-parameter method declaration instead. This allows concrete implementations to choose whether to use a `val` or a method to implement it.

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.sc
ala

scala> for opt <- Seq(Some(1), None)
     | yield opt match {
     |  case None => ""
     |  case Some(i) => i.toString
     | } match {
     |  case "" => false
     |  case _ => true
     | }
val res10: Seq[Boolean] = List(true, false)
```

Pattern Matching in Other Contexts

Fortunately, this powerful feature is not limited to `match` expressions. You can use pattern matching in assignment statements, called *pattern bindings*:

```
//
src/script/scala/progscala3/patternmatching/OtherUses1.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 pers = Person("Dean", 29, addr)
val addr: Address = Address(1 Scala Way,CA,USA)
val pers: Person = Person(Dean,29,Address(1 Scala
Way,CA,USA))

scala> val Person(name, age, Address(_, state, _)) = pers
val name: String = Dean
val age: Int = 29
val state: String = CA
```

This works with sequences, too:

```
scala> val seq = 0 to 4
val seq: scala.collection.immutable.Range.Inclusive = Range
0 to 4
scala> val head1 +: head2 +: tail = seq
val head1: Int = 0
val head2: Int = 1
val tail: IndexedSeq[Int] = Range 2 to 4
```

They work in `for` comprehensions:

```
scala> val people = seq.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 na = for
     |   Person(name, age, address) <- people
     | yield (name, age)
val na: IndexedSeq[(String, Int)] =
  Vector((Name0,10), (Name1,11), (Name2,12), (Name3,13),
(Name4,14))
```

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/OtherUsesTuples.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))
```

Finally, we can use pattern matching on a regular expression to decompose a string. Here's an example extracted from tests I once wrote for parsing (simple!) SQL strings:

```
// src/script/scala/progscala3/patternmatching/RegexAssignments.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, col2 FROM atable WHERE col1 = 'foo';"
val distinct: String = DISTINCT
val cols: String = "col1, col2 "
val table: String = atable
val otherClauses: String = WHERE col1 = 'foo'
```

Note that I had to add extra backslashes, e.g., `\\s` instead of `\s`, in the regular expression string, because I used string interpolation. See

the source file for other examples.

Obviously, using regular expressions to parse complex text, like XML or programming languages, has its limits. Beyond simple cases, consider a parser library, like the ones we'll discuss in [Link to Come].

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. For example:

```
// src/script/scala/progscala3/patternmatching/FragileAssignments.scala

scala> val h4a +: h4b +: t4 = Seq(1,2,3,4)
val h4a: Int = 1
val h4b: Int = 2
val t4: Seq[Int] = List(3, 4)

scala> val h2a +: h2b +: t2 = Seq(1,2)
val h2a: Int = 1
val h2b: Int = 2
val t2: Seq[Int] = List()

scala> val h1a +: h1b +: t1 = Seq(1)
scala.MatchError: List(1) (of class
scala.collection.immutable.$colon$colon)
  ...
```

In fact, if you use the `-strict` flag, expressions like the last one will fail to compile, even though Scala 2 allowed them. In Scala 3.1, this expression or any expression of the following form will fail to compile:

```
val head +: tail = someSequence  // Error in Scala 3.1, but
not 3.0
```

Hence, while Scala 3.0 still allows some expressions like this, subsequent Scala 3 versions will tighten the type-checking rules for pattern bindings, closing some loopholes in Scala 2 that would parse successfully, but fail with type errors at runtime, like the last several examples.

If you *know* that `someSequence` has at least one element, you can use the `@unchecked` annotation to allow the binding:

```
val head +: tail : @unchecked = someSequence  // Compiles
without error
```

Note how the annotation is used in the type location, because it controls how Scala types the binding.

Consider this final example of a `for` comprehension:

```
scala> val elems: Seq[Any] = Seq((1, 2), "hello", (3, 4))
scala> val what = for ((x, y) <- elems) yield (y, x)
val what: Seq[(Any, Any)] = List((2,1), (4,3))
```

In Scala 2 and 3.0, the `elems` sequence is filtered to retain only the elements of tuple type that match the pattern `(x, y)`. In Scala 3.1, this code will trigger a compile-time error. If filtering is the behavior you want in Scala 3.1 and later, use a case expression:

```
scala> val what = for case (x, y) <- elems yield (y, x)
val what: Seq[(Any, Any)] = List((2,1), (4,3))
```

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 also for many types in the Scala library. You can implement these extractors yourself to customize the behavior for your types. I'll start with how this process typical works in Scala 2, then discuss how it has been generalized in recent releases of Scala, including Scala 3.

Since you rarely need to implement your own extractors, you can safely skip to "Concluding Remarks on Pattern Matching" near the end of the chapter and return to this discussion later, as 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 extraction. Indeed there is such an *extractor* method and it is invoked in pattern-match expressions for most 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. All Scala 2-compatible `unapply` methods return an `Option[(...)]`, where the tuple type corresponds to the number of values and their types that can be extracted from the object, three for `Person` (of types `String`, `Int`, and `Address`) and two for `Address` (both of types `String`). So, the `Person` companion object that the Scala 2 compiler generates looks like this:

```
object Person {
  def apply(name: String, age: Int, address: Address) =
    new Person(name, age, address)
  def unapply(p: Person): Option[(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'll explore the details in "unapplySeq Method", but for now, just note that the extracted fields are returned in a `Some` wrapping a three-element tuple. The compiler then extracts those tuple elements for comparison with literal values, when used, assignment to variables, or they are dropped for _ placeholders.

The `unapply` methods are invoked recursively when necessary. Here we process the nested `Address` object first, then `Person`.

Recall the `head +: tail` expression we used above. 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 *destruction/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 schematically as follows:[2]

```
def unapply[T, Coll](collection: Coll): Option[(T, Coll)]
```

The head is of type `T`, which is inferred, and some collection type `Coll`, which represents the type of the input collection and the output tail collection. So, an `Option` of a two-element tuple with the head and tail is returned.

How can the compiler see the expression `case head +: tail => ...` and use a method `+:.unapply(collection)`? We might expect that the `case` clause would have to be written `case +: (head, tail) => ...` to work consistently with the behavior we just examined for pattern matching with `Person`, `Address`, and tuples.

As a matter of fact, we can write it that way:

```
scala> def seqToString2[T](seq: Seq[T]): String = seq match
     |   case +:(head, tail) => s"($head +:
${seqToString2(tail)})"
     |   case Nil => "Nil"
def seqToString2[T](seq: Seq[T]): String

scala> seqToString2(Seq(1,2,3,4))
val res0: String = (1 +: (2 +: (3 +: (4 +: Nil))))
```

But we can also use *infix* notation, `head +: tail`, because the compiler exploits another bit of syntactic sugar. Types with two type parameters can be written with infix notation and so can `case` clauses:

```
// src/script/scala/progscala3/patternmatching/Infix.scala

case class With[A,B](a: A, b: B)

val with1: With[String,Int] = With("Foo", 1)
val with2: String With Int  = With("Bar", 2)
// val with3: String With Int  = "Baz" With 3  // ERROR

val results = Seq(with1, with2).map {
  case s With i => s"$s with $i"
}
assert(results == Seq("Foo with 1", "Bar with 2"))
```

For completness, there is an analog of `+:` that can be used to process the sequence elements in reverse, `:+`:

```
//
src/script/scala/progscala3/patternmatching/MatchReverseSeq.
scala
// Compare to match-seq.sc

val nonEmptySeq = Seq(1, 2, 3, 4, 5)

def reverseSeqToString[T](l: Seq[T]): String = l match
  case prefix :+ end => s"(${reverseSeqToString(prefix)} :+
$end)"
  case Nil => "Nil"

assert(reverseSeqToString(nonEmptySeq) ==
       "(((((Nil :+ 1) :+ 2) :+ 3) :+ 4) :+ 5)")
```

Note that `Nil` comes first this time.

As before, you could use this output to reconstruct collections (skipping the duplicate second line of the previous output):

```
scala> val revList = (((((Nil :+ 1) :+ 2) :+ 3) :+ 4) :+ 5)
val revList: List[Int] = List(1, 2, 3, 4, 5)
```

The parentheses are unnecessary. Try it!

We mentioned above that you can pattern match pairs with `->`. This feature is implemented with a `val` defined in `Predef`, `->`. This is an alias for `Tuple2`, which subclasses `Product2`, which defines an `unapply` method that is used for these pattern matching expressions.

Alternatives to Option Return Values

Back to implementation requirements for `unapply`, recent Scala releases relaxed the requirement to return an `Option`. As of Scala 2.11, any type with this signature is allowed (which `Option` also implements):

```
def isEmpty: Boolean
def get: T
```

Scala 3 also allows a `Boolean` to be returned or a `Product` type, which is a parent class of Tuples, but more broad than that. Here's an example using `Boolean`, where we want to discriminate between two kinds of strings:

```
//
src/script/scala/progscala3/patternmatching/UnapplyBoolean.s
cala

object ScalaSearch:
❶
        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
❷
        case (_ @ ScalaSearch(), index) => s"$index: found
Scala"
        case (_, index) => s"$index: no Scala"

assert(result == Seq("0: found Scala", "1: no Scala", "2:
found Scala"))
```

❶ Define an object with an `unapply` method that takes a string, converts to lower case, 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”.

Other single values can be returned. Here is an example that converts a Scala `Map` to a Java `HashMap`:

```
//
src/script/scala/progscala3/patternmatching/UnapplySingleVal
ue.scala

import java.util.{HashMap => 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)
     | map match
     |   case JHashMapWrapper(jmap) => jmap
val map: Map[String, Int] = Map(one -> 1, two -> 2)
val res2: 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, not one possible return values):

```
//
src/script/scala/progscala3/patternmatching/UnapplySingleVal
ue2.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    // Not shown;
see the file
... errors ...
```

Anyway, the Scala collections already have tools for converting between Scala and Java collections (see [Link to Come]).

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 by an index or iterating over them. Tuples implement `Product`. We can use it as a way to provide several return values extracted by `unapply`:

```
//
src/script/scala/progscala3/patternmatching/UnapplyProduct.s
cala

class Words(words: Seq[String], index: Int) extends Product:
❶
  def _1 = words
❷
  def _2 = index

  def canEqual(that: Any): Boolean = ???
❸
  def productArity: Int = ???
  def productElement(n: Int): Any = ???

object Words:
  def unapply(si: (String, Int)): Words =
❹
    val words = si._1.split("""\W+""").toSeq
❺
    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"))
```

❶ Now we need a class to instantiate with the results, when a match succeeds. It implements `Product`

❷ Define two methods for retrieving the first and second items. Note the method names are the same as for two-element tuples.

❸ 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 `Predef.???`, which always throws `NotImplementedError`.

❹ Matches on a tuple of `String` and `Int`.

❺ 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`:

```
def apply[A](elems: A*): Seq[A]
final def unapplySeq[A](x: Seq[A]): UnapplySeqWrapper[A]
```

`UnapplySeqWrapper` is a helper class. Recall that `A*` means that `elems` is a variable argument list.

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:

```
//
src/script/scala/progscala3/patternmatching/MatchUnapplySeq.
scala

val nonEmptyList   = List(1, 2, 3, 4, 5)
val emptyList      = Nil
val nonEmptyMap    = Map("one" -> 1, "two" -> 2, "three" ->
3)

// Process pairs
def windows[T](seq: Seq[T]): String = seq match
  case Seq(head1, head2, _*) =>
❶
    s"($head1, $head2), " + windows(seq.tail)
❷
  case Seq(head, t: _*) =>
❸
    s"($head, _), " + windows(t)
  case Nil => "Nil"
❹

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"))
```

❶ 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 and ignore the rest of the variable argument list

with `_*`. Think of the `*` as matching zero to many, like in regular expressions.

❷ Format a string with the first two elements, then move the "window" by one (not two) calling `seq.tail`.

❸ 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 capture the tail as `t` and use it in the recursive call, although we actually know that this call to `windows(t)` will simply return `Nil`

❹ The `Nil` case with temninates the recursion.

You could rewrite the second case statement to skip the final invocation of `windows(t)`, but I left it in to show capturing of the tail using `t: +*`.

We could still use the `+:` matching we saw before, which is more elegant and what I would do:

```
//
src/script/scala/progscala3/patternmatching/MatchWithoutUnap
plySeq.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> 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 sliding windows contain our last element, `5`.

Implementing unapplySeq

Let's implement an `unapplySeq` method adapted from our `Words` example above. We'll tokenize the words as before, but also remove

all words shorter than a specified value.

```
// src/script/scala/progscala3/patternmatching/UnapplySeq.scala

object Tokenize:
  // 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 of length 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 variable argument list (`d`).

Try simplifying this example to not do length filtering. Uncomment the line for comment 1 and work from there.

Concluding Remarks on Pattern Matching

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 ("Extractors"). These methods let you extract that information while hiding the implementation details. In fact, the information returned by `unapply` might be a transformation of the actual fields in the instance.

When designing pattern-matching statements, 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 the design should be refined so you know more precisely all the possible matches that might occur. Often the right design uses sealed class hierarchies or enums, especially those that implement algebraic data types.

Recap and What's Next

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.

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.

---

1 See *The Java Tutorials. Lesson: Regular Expressions*.

2 I have simplified the actual declaration, because we haven't yet covered details about the type system that we need to understand the actual signature.

Chapter 5. Abstracting over Context, Part I

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at m.cronin@oreilly.com

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, both to make learning and using these idioms easier and to address some shortcomings of the prior implementations. 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.

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". 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.

The most sweeping changes introduced in Scala 3 are to the type system and to how we define and use context abstractions. The changes to the latter 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. Not only do the changes make intention more clear, they also eliminate some boilerplate previously required when using implicits and 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:[1]

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 deemphasizes the previous distinction where you had to know when to declare an instance vs. a class. Most of the time

you will specify that a particular class should be used to satisfy the need for an implicit value and the compiler does the rest.

Using Clauses

Instead of using the keyword `implicit` to declare an implicit parameter list for a method, you use the keyword `using`. The different keyword eliminates several ambiguities and it allows a method definition to have more than one implicit parameter list, which are now called *using clauses*. (We'll explore them in the next chapter.)

Given Imports

When you use wild cards in import statements, they no longer import `given` instances along with everything else. Instead, you use the `given` keyword to explicitly ask for givens to 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. The next chapter discusses *using clauses*, previous called *implicit parameter lists* in Scala 2.

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 address 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. Keep in mind that 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", then only those users are affected.

Hence, the goal is to enable ad hoc additions to types in a principled way. By *principled*, type implementations can remain focused on their core abstractions, while additional behaviors can be added only where needed, as opposed to making global modifications that affect all users. These tools also preserve type safety.

However, a drawback of this *separation of concerns* is that the separate `toJSON` functionality needs to track changes in the code for 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) {
  @infix def -> [B](y: B): (A, B) = (self, y)
}
```

Here is how implicit conversion works in Scala 2. 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 can emit code to create an instance of `ArrowAssoc[String]`, with the string `"one"` passed as the `self` argument, followed by code to call `->(1)` 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 re-implement this using a Scala 3 extension method. To avoid ambiguity with `->`, let's use `~>` instead, but it works identically:

```
//
src/script/scala/progscala3/contexts/ArrowAssocExtension.sca
la

scala> import scala.annotation.{alpha, infix}
❶

scala> extension [A,B] (a: A):
❷
     |   @infix @alpha("arrow2") def ~>(b: B): (A, B) = (a,
b)
def extension_~>[A, B](a: A)(b: B): (A, B)

scala> "one" ~> 1
val res0: (String, Int) = (one,1)
```

❶ Use the `@infix` annotation to allow *infix operator notation*, i.e., `"one" ~> 1`. Use `@alpha` to define an alphanumeric name in byte code for this method.

❷ The syntax for defining an extension method. Any type parameters used by the methods that follow must go after the keyword `extension`. (The whitespace is arbitrary.) Next we define the method `~>`, like we would if this were a "regular" type.

Note the signature the compiler reports for the generated method. It is named `extension_~>` and it takes two parameter lists. The first one is for the target instance of type `A` being extended. The second list is the same one specified when we declared the `~>` method. We'll see this naming convention again for anonymous given instances below.

Now, when the compiler sees `"one" ~> 1`, it will look for a corresponding `extension_~>` that is in scope and type compatible in the left-hand and right-hand types. Our definition satisfies this

requirement for all types. Then the compiler will emit code to call the extension method. No wrapping in a new instance is required. Hence, implicit conversion is eliminated.

Let's complete an example we started in "Operator Overloading?", where we showed that parameterized types with two parameters can be written with infix notation, but 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.scal
a

scala> import scala.annotation.{alpha, infix}

scala> @alpha("BangBang") case class !![A,B](a: A, b: B)
❶

scala> extension [A,B] (a: A) def !!(b: B): A !! B = !!(a,
b)     ❷
def extension_!![A, B](a: A)(b: B): A !! B

scala> val ab1: Int !! String = 1 !! "one"
❸
     | val ab2: Int !! String = !!(1, "one")
val ab1: Int !! String = !!(1,one)
val ab2: Int !! String = !!(1,one)
```

❶ The same case class defined in "Operator Overloading?".

❷ The extension method definition. When only one method is defined, you can omit the colon (or curly braces) and even define

it on the same line as shown.

❸ This line failed to compile before, but now the extension method is applied to `Int` and invoked with the `String` argument `"one"`.

> **TIP**
>
> When defining just one extension method for a type, the colon at the end of the opening line or curly braces can be omitted. For consistency and easier reading, consider always using the colon or braces.

Both of our examples did not need to add additional fields to the target type nor was there an interface that made sense to implement. When those are required, we'll use *type classes*, discussed next.

There was a "context" for both extensions. Users only need `->` or `!!` in certain, limited circumstances. These would not be good methods to add to the source code for *all* types! With extension methods, we get the best of both worlds, calling "methods" like `->` when we need them, while keeping types as focused 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`:

```
scala> object Foo:
     |   def one: Int = 1

scala> extension (foo: Foo.type) def two: String = "two"
def extension_two(foo: Foo.type): String
```

```
scala> Foo.one
     | Foo.two
val res0: Int = 1
val res1: String = two
```

Type Classes

The next step beyond extension methods is to add not only methods to types, but also state (fields) and to implement an abstraction, so all type extensions are done uniformly. A term that is popular for these kinds of extensions is *type classes*, which comes from the Haskell language, where this idea was pioneered. The word *class* in this context is not the same as Scala's OOP concept of classes, which can be confusing.

As an example, suppose we have a collection of `Shapes` from "A Sample Application" and we want the ability to call a `toJSON` method on them that returns a JSON representation appropriate for each type? If we write `someShape.toJSON`, We want the Scala compiler to invoke some mechanism that implements this functionality.

Scala 3 Type Classes

A type class defines an abstraction with optional state (fields) and behavior (methods). They provide a another way to implement *mixin composition* ("Traits: Interfaces and "Mixins" in Scala"). The abstraction is valuable for ensuring that all "instances" of the type class follow the same protocol uniformly.

First, we need a `trait` for the state and behavior we want to add. To keep things simple, we'll return JSON-formatted strings, not objects from some JSON library (of which there are many…):

```
// src/main/scala/progscala3/contexts/json/ToJSON.scala
package progscala3.contexts.json

trait ToJSON[T]:
  extension (t: T) def toJSON(name: String, level: Int):
String

  protected val INDENTATION = "  "
  protected def indentation(level: Int): (String,String) =
    (INDENTATION * level, INDENTATION * (level+1))
```

This is the Scala 3 type class pattern. We define a trait with a type parameter and we define extension methods. Although we don't actually use the type parameter in the body of this particular type class, the parameter will be essential for disambiguating one type class instance, such as the one for `Circle`, from another, such as the one for `Rectangle`.

The public method users care about is `toJSON`. The `protected` method, `indentation`, and immutable state, `INDENTATION`, are implementation details.

The type class pattern solves the limitation discussed above for extension methods alone. We can define and implement an abstraction and we can add arbitrary state as fields.

Now we create instances for our `Shapes`:

```
//
src/main/scala/progscala3/contexts/typeclass/new1/ToJSONType
```

```
Classes.scala
package progscala3.contexts.typeclass.new1

import progscala3.introscala.shapes.{Point, Shape, Circle,
Rectangle, Triangle}
import progscala3.contexts.json.ToJSON

given ToJSON[Point]:
❶
  extension (point: Point) def toJSON(name: String, level:
Int): String =
    val (outdent, indent) = indentation(level)
    s"""$name": {
      |${indent}"x": "${point.x}",
      |${indent}"y": "${point.y}"
      |$outdent}""".stripMargin

given ToJSON[Circle]:
❷
  extension (circle: Circle) def toJSON(name: String, level:
Int): String =
    val (outdent, indent) = indentation(level)
    s"""$name": {
      |${indent}${circle.center.toJSON("center", level +
1)},
      |${indent}"radius": ${circle.radius}
      |$outdent}""".stripMargin

// And similarly for Rectangle and Triangle

@main def TryJSONTypeClasses() =
  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 `given` keyword declares a conversion for `ToJSON[Point]`. The extension method for `ToJSON` is implemented as required for point

❷ A given for `ToJSON[Circle]`.

Running `TryJSONTypeClasses` prints the following:

```
> runMain
progscala3.contexts.typeclass.new1.TryJSONTypeClasses
...
"circle": {
  "center": {
    "x": "1.0",
    "y": "2.0"
  },
  "radius": 1.0
}
...
```

> **NOTE**
>
> If you use braces, the colon on the `given` line is replaced with an opening curly brace and a corresponding closing brace is required. The same applies for the definition of the anonymous `ToJSON` subtype, of course.

There's a flaw with our implementation, though. If we put those shapes in a sequence, `shapes`, and try `shapes.foreach(s ⇒ println(s.toJSON("shape", 0)))`, we get an error that `Shape` doesn't have a `toJSON` method. Polymorphic dispatch doesn't work here.

What if we add a given for `Shape` that delegates to the others?

```
given ToJSON[Shape]:
  extension (shape: Shape) def toJSON(name: String, level:
Int): String =
    shape.toJSON(name, level)
```

Seems legit, but the compiler says we have an “infinite recursion”. Again, we aren’t actually calling a polymorphic method `toJSON` defined in the old-fashioned way for the hierarchy. So, the call `shape.toJSON(level)` attempts to call the extension method recursively.

What about pattern matching on the type of `Shape`?

```
given ToJSON[Shape]:
  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)
```

We still get an infinite recursion at runtime, but the compiler can’t detect it! So, instead, let’s call the compiler generated `toJSON` implementations directly. A synthesized `object` is output for each specific `given`:

```
//
src/main/scala/progscala3/contexts/typeclass/new2/ToJSONType
Classes.scala
...
given ToJSON[Shape]:
  extension (shape: Shape) def toJSON(name: String, level:
Int): String =
    shape match
      case c: Circle    =>
        given_ToJSON_Circle.extension_toJSON(c)(name, level)
      case r: Rectangle =>
        given_ToJSON_Rectangle.extension_toJSON(r)(name,
level)
      case t: Triangle  =>
        given_ToJSON_Triangle.extension_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 `...typeclass.new2.TryJSONTypeClasses` (not shown) indicates that calling `shape.toJSON` and `circle.toJSON` (for example) now both work as desired, but we are relying on an obscure implementation detail that could change in a future release of the compiler. Note the naming conventions, `given_...` and `extension_...`, which we saw previously.

There is an easy fix. We can give names to the givens and then call them:

```
//
src/main/scala/progscala3/contexts/typeclass/new3/ToJSONType
Classes.scala
...
given ToJSON[Shape]:
  extension (shape: Shape) def toJSON(name: String, level:
Int): String =
    shape match
      case c: Circle    => circleToJSON.extension_toJSON(c)
(name, level)
      case r: Rectangle =>
rectangleToJSON.extension_toJSON(r)(name, level)
      case t: Triangle  =>
triangleToJSON.extension_toJSON(t)(name, level)

given circleToJSON as ToJSON[Circle]:
  ...
```

Note the `as` keyword after the name. Instead of the synthesized name `given_ToJSON_Circle`, it will be `circleToJSON`, and similarly for the other shapes. We must still use the synthesized method name for the extension method, but at least we have more control over the object name.

> **NOTE**
>
> When used as shown, `as` is a "soft" reserved word. You can still use it as an identifier elsewhere. However, `given` is always reserved.

At this point, if we still want to simulate polymorphic behavior, consider refactoring the code to move the implementations of the `toJSON` methods for each concrete Shape to a regular helper object. Then call its methods instead of using the compiler generated objects. See `...typeclass.new4.TryJSONTypeClasses` in the code examples for one approach.

> **TIP**
>
> Keep the given instances as concise as possible. Consider moving some of the code to "regular" types that operate as helpers.

Finally, note that `Point` and the concrete subtypes of `Shape` are not related in the type hierarchy (well, except for `AnyRef` way at the top). Hence, this extension mechanism is *ad hoc polymorphism*, because the polymorphic behavior of `toJSON` is not tied to the type

system, as it would be in *subtype polymorphism*. Subtype polymorphism is nice for allowing parent types to declare behaviors that can can be defined in subtypes. We had to hack around this missing feature for `toJSON`! For completeness, recall that we discussed a third kind of polymorphism, *parametric polymorphism*, in "Abstract Types Versus Parameterized Types", where containers like `Container[A]` behave uniformly for any type `A`.

Scala 2 Type Classes

To implement the same type class and instances in Scala 2, you write an implicit conversion that wraps the `Point` and `Shape` instances in a new instance 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 in `ToJSON` won't work with Scala 2:

```
//
src/main/scala/progscala3/contexts/typeclass/old/ToJSONOldTy
peClasses.scala
package progscala3.contexts.typeclass.old

import scala.language.implicitConversions
❶

trait ToJSONOld[T]:
  def toJSON(level: Int): String
❷

  protected val INDENTATION = "  "
  protected def indentation(level: Int): (String,String) =
    (INDENTATION * level, INDENTATION * (level+1))
```

❶ We must enable implicit conversions.

❷ Now this is a regular method. in the original `ToJSON`, it was an extension method.

Because indiscriminate use of implicit conversions can be confusing for code comprehension and sometimes lead to unexpected behavior, implicit conversions are treated as an optional feature by Scala. This means you must enable the feature explicitly with the import statement used for `implicitConversions` or use the global `-language:implicitConversions` compiler flag.

Now here is an implementation of a `toJSON` type class instances for Scala 2, which also works in Scala 3. We'll only show the implementations for `Point` and `Circle`:

```
implicit final class PointToJSON(
    point: Point) extends ToJSONOld[Point]:
❶
  def toJSON(name: String, level: Int): String =
    val (outdent, indent) = indentation(level)
    s"""$name": {
      |${indent}"x": "${point.x}",
      |${indent}"y": "${point.y}"
      |$outdent}""".stripMargin

implicit final class CircleToJSON(
    circle: Circle) extends ToJSONOld[Circle]:
❷
  def toJSON(name: String, level: Int): String =
    val (outdent, indent) = indentation(level)
    s"""$name": {
      |${indent}${circle.center.toJSON("center", level +
1)},
      |${indent}"radius": ${circle.radius}
      |$outdent}""".stripMargin
...

@main def TryJSONOldTypeClasses() =
  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))
```

❶ The type class "instance" that implements `ToJSONOld.toJSON` for `Point`. It is called a type class *instance* following Haskell conventions, but we actually declare a *class* for it, which can be confusing.

❷ The type class *instance* that implements `toJSON` for `Circle`.

Because these classes are declared as implicit, when the compiler sees `circle.toJSON()`, for example, it will look for an 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, auto-generated 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 the original 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 implicit conversions.

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 could 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 givens 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 will want to convert between types for other reasons. Consider the following example, where types are defined 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`:

```
//
src/main/scala/progscala3/contexts/NewImplicitConversions.sc
ala
package progscala3.contexts
import scala.language.implicitConversions

case class Dollars(amount: Double):
  override def toString = f"$$$amount%.2f"
```

```
case class Percentage(amount: Double):
  override def toString = f"${(amount*100.0)}%.2f%%"

case class Salary(gross: Dollars, taxes: Percentage):
  def net: Dollars = Dollars(gross.amount * (1.0 -
taxes.amount))
```

Note that we import `scala.language.implicitConversions`. 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.

Let's try it:

```
@main def TryImplicitConversions() =
  given Conversion[Double,Dollars] = d => Dollars(d)
❶
  given Conversion[Double,Percentage] = d => Percentage(d)

  val salary = Salary(100_000.0, 0.20)
  println(s"salary: $salary. Net pay: ${salary.net}")
```

❶ The syntax for declaring a given conversion from `Double` to `Dollars` and a second conversion from `Double` to `Percentage`.

Running this example prints the following:

```
salary: Salary($100000.00,20.00%). Net pay: $80000.00
```

The declaration of the `Conversion[Double,Dollars]` given is shorthand for the following longer form:

```
given Conversion[Double,Dollars]:
  def apply(d: Double): Dollars = Dollars(d)
```

By the way, if you define the given for converting `Doubles` to `Dollars` in the REPL, observe what happens:

```
...
scala> given Conversion[Double,Dollars] = d => Dollars(d)
def given_Conversion_Double_Dollars: Conversion[Double,
Dollars]

scala> given dd as Conversion[Double,Dollars] = d =>
Dollars(d)
def dd: Conversion[Double, Dollars]
```

In our type classes above, `objects` were generated. Here, methods are generated. As we saw previously, for an anonymous given, the generated name follows the convention `given_...`.

Rules for Implicit Conversion Resolution

Here is a summary of the lookup rules used by the compiler to find and apply conversions conversions. I'll use `given` and given instances to refer to both new and old style conversions:

1. No conversion will be attempted if the object 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 classes as `given` instances from other types as long as they obey a set of common properties. A good example is `Eql`, which we've seen used in a few examples, but not yet explained. Another good candidate is `Ordering`, but support for it is not yet implemented at the time of this writing.

TODO: * Verify which mixins support this at the time Scala 3 is released. * `derives Eql` shouldn't be necessary for `enum` definitions, but it appears to be necessary for `WeekDay`. Is this a Scala 3 bug?

In "Enumerations and Algebraic Data Types", we defined an `enum` called `WeekDay` for the days of the week. It had the clause `derives Eql`. Here is part of the declaration:

```
// src/script/scala/progscala3/rounding/WeekDay.scala
package progscala3.rounding

enum WeekDay(val fullName: String) derives Eql:
  case Sun extends WeekDay("Sunday")
  ...
```

When `derives Eql` is used, the compiler automatically derives a type class instance for the `Eql` type class. This type class allows us to

use == and != to compare instances. The code examples compiler flags include `-language:strictEquality`, which requires that we use `derives Eql` when comparing instances with == and !=, but what does that actually mean?

This clause causes the compiler to generate the following type class instantiations in the compiler-generated companion object named `WeekDay`:

```
object WeekDay:
  given Eql[WeekDay] = Eql.derived
  ...
```

The terminology used is `WeekDay` is the *deriving type* and the `Eql` instance is a *derived instance*.

For the following kinds of types, the compiler automatically derives from `Eql`:

- Enums and enum cases
- Case classes and case objects
- Sealed classes or traits that have only case classes and case objects as children

In general, any type `T` defined with a companion object that has the `derived` method (like `Eql.derived` above) can be used with `derives T` clauses. We'll discuss the details in [Link to Come], after we have learned the metaprogramming details required.

Givens and Imports

In "A Taste of Futures" 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 wild-card imports frequently while restricting 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"
  val m(s: String) = s"$s, hello from $name"
  class C1
  given c1 as C1
  class C2
  given c2 as C2
```

Now consider these import statements:

```
import O1._               // Imports everything EXCEPT the
givens, c1 and c2
import O1.{given _}       // Imports ONLY the givens, c1 and
c2
import O1.{given c1}      // Imports just c1 explicitly
import O1.{given _, _}    // Imports everything in O1
```

A `given` import selector also brings old style implicits into scope.

What if the `given` instances are anonymous and you don't want to use the wild card?

```
object O2:
  class C1
  given C1
  class C2
  given C2
  given intOrd as Ordering[Int]
  given listOrd[T: Ordering] as Ordering[List[T]]
```

You can import *by type*. Note the `?` wild card for the type parameter, which means both `Ordering` givens will be imported:

```
import O2.{given C1, given Ordering[?]}
```

Because this is a breaking change in how _ wild cards work for imports, it is being implemented gradually:

- In Scala 3.0 an old-style implicit definition can be brought into scope either by a _ or a given _ wildcard selector.
- 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.

Finally, the older Scala 2 implicit conversions are still allowed, where an `implicit def` is used, for example:

```
implicit def doubleToDollars(d: Double): Dollars =
Dollars(d)
```

Unlike the Scala 2 alternative to extension methods, we don't need an implicit class here, like `ArrowAssoc` above, because `Dollars` has all the methods we need. This method would be invoked to do the conversion exactly the same way as the `given Conversion[Double,Dollars]` above.

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`, `new2`, etc. 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 Scala 3's 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.

I'll use the term "given" in the following discussion to include given instances, extension methods, and Scala 2 implicit methods, values, and classes, depending on the scenario. Resolving to a particular given happens in the following order:

RULES FOR GIVEN RESOLUTION

1. Any type-compatible given that doesn't require a prefix path. In other words, it is defined in the same scope, such as within the same block of code, within the same type, within its companion object (if any), or within a parent type.
2. A given that was imported into the current scope. It also doesn't require a prefix path to use it.
3. Imported givens take precedence over the already-in-scope givens. That is, when an import brings in new givens.
4. In some cases, several possible matches are type compatible. The most specific match wins. For example, if a `Foo` is needed in a context, then a given in scope of type `Foo` will be chosen over a given of type `AnyRef`, if both are in scope.
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 rational is that the other givens are likely to be user defined or imported from special libraries, and hence more "important" to the user.

Build Your Own String Interpolator

Let's look at a final example of extension, one that lets us define our own string interpolation capability. Recall from "Interpolated Strings" that Scala has several built-in ways to format strings through interpolation. For example:

```
val (first, last) = ("Buck", "Trends")
println(s"Hello, ${first} ${last}")
```

There are also `StringContext` methods named `f` and `raw`, where `f` supports `printf` format directives and `raw` doesn't interpret escape characters:

```
scala> val pi=3.14159

scala> f"$pi%5.3f or ${pi}%7.5f"
val res0: String = 3.142 or 3.14159

scala> raw"\t $pi \n $pi again"
val res1: String = \t 3.14159 \n 3.14159 again
```

We'll look at a simplistic implementation of a SQL query compiler named `sql`. When the compiler sees an expression like `sql"SELECT $column FROM $table;"`, it will be translated to the following:

```
StringContext("SELECT ", "FROM ", ";").sql(column, table)
```

Note how the embedded expressions become arguments to `sql`, while the other string tokens are arguments to `StringContext.apply`. However, `scala.StringContext` doesn't have a `sql` method, so an implicit conversion to another type or an extension method is required.

Let's define `sql` for `StringContext`. For simplicity, it will only handle SQL queries of the form `sql"SELECT columns FROM table;"` with the columns and table strings specified as part of the string or using embedded expressions. The extracted column and table names are returned in an instance of a simple case class `SQLQuery`. It would be possible to use the same approach with a real SQL parser and return a query object for a library like JDBC. I

won't show the whole implementation (which is somewhat of a "hack" for this simple case), but just the declarations:

```
//
src/main/scala/progscala3/contexts/SQLStringInterpolator.sca
la
package progscala3.contexts

object SimpleSQL:
  case class SQLQuery(columns: Seq[String] = Nil, table:
String = "")

  extension (sc: StringContext):
    def sql(values: String*): SQLQuery =
      // Extract the column names and table name.
      SQLQuery(columns, table)
```

See the `SQLStringInterpolator` source code for the full details. Here is how to use it:

```
scala> import progscala3.contexts.SimpleSQL._

scala> val query1 = sql"SELECT one, two, three FROM t1;"
val query1: ...SimpleSQL.SQLQuery = SQLQuery(Vector(one,
two, three),t1)

scala> val cols = Seq("four", "five").mkString(", ")
     | val table = "t2"
     | val query2 = sql"SELECT $cols FROM $table;"
val cols: Seq[String] = List(four, five)
val table: String = t2
val query2: ...SQLQuery = SQLQuery(Vector(four, five),t2)
```

As shown, custom string interpolators can return any type you want, not just a new `String`, like `s`, `f`, and `raw` return. Hence, they can function as instance factories that are driven by data encapsulated in strings.

The Expression Problem

Let's step back for a moment and ponder what we just accomplished in the previous example. We added new functionality to an existing library type without editing the source code for it!

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 subtyping, more precisely called *subtype polymorphism*. We program to abstractions and use derived classes when we need changed behavior. Bertrand Meyer coined the term *Open/Closed Principle* to describe a principled OOP approach, where base types declare the behaviors as abstract that should be open for extension or variation in subtypes, while keeping invariant behaviors closed to modification. The base types are never modified for extension.

Scala certainly supports this technique, but it has drawbacks. What if it's questionable that we should have that behavior defined in the type hierarchy in the first place? What if the behavior is only needed in a few contexts, while for most contexts, it's just a burden that the code carries around?

It can be a burden for several reasons. First, the extra, mostly-unused code is a maintenance burden. Developers have to understand it, even when working on other aspects of the code, so they don't break it inadvertently. Second, it's also inevitable that most defined behaviors will be refined over time. Every change to a feature that some clients aren't using forces unwanted updates on client code.

This problem led to the *Single Responsibility Principle*, a classic design principle that encourages us to define abstractions and implement types with just a single behavior.

Still, in realistic scenarios, it's sometimes necessary for an object to combine several behaviors. For example, a service often needs to "mix in" the ability to log messages. Scala makes these *mixin* features relatively easy to implement, as we saw in "Traits: Interfaces and "Mixins" in Scala". We can even declare instances that mix in traits without first defining a class.

In general, mixins, extension methods, and type classes provide robust and principled solutions to the Open/Closed Principle while allowing the core implementations of types to obey the Single Responsibility Principle.

Wise Use of Type Extensions

Why not take an extreme approach and define types with very little state and behavior (sometimes called *anemic* types), then add most behaviors using mixin traits, type classes, extension methods, even implicit conversions?

First, when using a type class or implicit conversion, the resolution algorithm requires more work by the compiler than just finding the logic inside the type's original definition. Also, there can be more boilerplate writing extensions compared to the alternative of defining constructs inside the type. Therefore, a project that over-uses these

tools is a project that is slow to build, as well as potentially hard to comprehend when reading the code.

Another problem for the extension mechanisms explored in this chapter is that you effectively lose several benefits of object orientation!

First, code evolution can be challenging if the extensions depend on details of the types they extend, like our `ToJSON` example in the last chapter. The details have to be coordinated in more than one place, the locations of the extensions, as well as the type definitions themselves. Fortunately, coupling between type definitions and extensions are limited to the public interfaces, as an extension has no access to private or protected members of a type.

A second issue is the loss of object-oriented method dispatch. We had to do some hacking to support `Shape.toJSON` in a polymorphic way.

If instead, `toJSON` were declared abstract in `Shape` and implemented in `Circle`, `Rectangle`, etc., then this code would work with the usual object-oriented dispatch rules.

Most of the time, the core domain logic belongs in the type definition. Ancillary behaviors, like serializing to JSON and logging, belong in mixins or type classes. However, if *your* applications use `toJSON` frequently for your domain classes, it might be a good idea to move this behavior into the type definitions, on balance.

When should use use type classes and extension methods vs. *mix-in* composition? For example, recall the `Logging` trait example we saw in "Traits: Interfaces and "Mixins" in Scala". If the trait has *orthogonal* state and behavior, like logging, that can be mixed into many different objects, then a mixin trait is often best. If the behavior has to be defined carefully for each type, like `toJSON`, then type classes are best.

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.

Part 2 explores *using clauses*, which work with given instances to address particular design scenarios and to simplify user code.

---

1 Adapted from this Dotty documentation.

Chapter 6. Abstracting over Context, Part II

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at m.cronin@oreilly.com

In the previous chapter, 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* (explored in the previous chapter) are in scope that the compiler can use instead.

In Scala 2 terminology, those parameters were called *implicit parameters* and the whole list of parameters is an *implicit parameter list* or *implicit parameter clause*. In Scala 3, they are *context parameters* and the whole parameter list is a *using clause*.[1] Here is an example:

```
class BankAccount(...):
  def debit(amount: Money)(using transaction: Transaction)
  ...
```

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 that we studied last chapter.

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 function 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 definition at compile time and they 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 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 an implicit, and then have all sequences of the same element type use it automatically.

This first implementation uses syntax valid for both Scala 2 an 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]=
    new SortableSeq(seq.sortBy(transform)(o))

  def sortBy2[B : Ordering](transform: A => B):
SortableSeq[A] =
    new 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() = {
  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 Uses the default ordering provided by `math.Ordering` for `Int`s.

❹ Define a custom `oddEven` ordering, which will be the "closest" implicit value in 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 `Seq.sortBy` method in the collections library. 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 "bound" 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 bound `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]`.)

In the generated byte code for Scala 2, this is just short hand 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] =
    new SortableSeq(seq.sortBy(transform)(o))

  def sortBy1b[B](transform: A => B)(using Ordering[B]):
SortableSeq[A] =
    new SortableSeq(seq.sortBy(transform)
(summon[Ordering[B]]))

  def sortBy2[B : Ordering](transform: A => B):
SortableSeq[A] =
    new 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`, we see that the name can be omitted and a new `Predef` method, `summon` is used to bind the value, instead. (It is 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 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 as Ordering[Int] = new Ordering[Int]:
    def compare(i: Int, j: Int): Int = i%2 compare j%2 match
      case 0 => i compare j
      case c => -c

  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)
❷
  assert(seq.sortBy1b(i => -i)(using evenOdd) == expected)
  assert(seq.sortBy2(i => -i)(using evenOdd)  == expected)

evenOddGivenOrdering()
```

❶ Use the `given evenOdd` instance implicitly.

❷ Use the `given evenOdd` instance explicitly with `using`.

The syntax `given foo as Type[T]` is used instead of `implicit val foo: Type[T]`, essentially the same way we

used givens when discussing type classes. Recall the use of `as`, too.

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 the `implicit` keyword here. The reason `using` is now required is two fold. First, it's better documentation for the reader. Second, it removes an ambiguity that is illustrated in the following contrived Scala 2 example:

```
case class FastSeq[T](implicit storage: Storage[T]):
❶
  def apply(i: Int): Option[T] = storage.get(i)
❷

implicit val customStorage: Storage = ???
val opt = FastSeq[String](5)
❸
```

❶ A "fast" sequence implementation with user-pluggable storage.

❷ Optionally return the item at index `i`.

❸ Does this line return a `None`, because the sequence is empty?

In Scala 2, the last line would cause a compiler error for the argument `5`, saying that a `Storage` instance was expected as the argument. The actual user intention was for the instance to be constructed with the implicit value `customStorage`, then the apply method was to be called with `5`. Instead, you would have to use the unintuitive expression `FastSeq[String].apply(5)`.

Now this ambiguity is removed by requiring `using` when the implicit is provided explicitly. As written, the compiler knows that you want to use the implicit for the storage *and then* call `apply(5)`.

> **TIP**
>
> The intent of the new `given name as ...` syntax and the `using ...` syntax is to make their purpose more explicit, but it functions almost identically to Scala 2 `implicit` definitions and parameters.

By-Name Context Parameters

Context parameters can be by-name parameters. Here is an example adapted from this Dotty documentation.

```
//
src/script/scala/progscala3/contexts/ByNameContextParameters
.scala

trait Codec[T]:
  def write(x: T): Unit

given intCodec as Codec[Int]:
  def write(i: Int): Unit = print(i)

given optionCodec[T](using ev: => Codec[T]) as
Codec[Option[T]]:
  def write(xo: Option[T]) = xo match
    case Some(x) => ev.write(x)
    case None =>

val s = summon[Codec[Option[Int]]]

s.write(Some(33))
s.write(None)
```

Note that `ev` for `optionCodec[T]` is a by-name parameter, which means its evaluation is delayed until used. Using a by-name parameter here can avoid certain cases where a *divergent expansion* can happen, as the compiler chases its tail trying to resolve all using clause parameters.

Other Context Parameters

In "A Taste of Futures", 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
implicit value
Future(...)(using customExecutionContext)      // Explicit
argument with "using"
```

`Future` supports a lot of the operations like `filter`, `map`, etc. All take two argument lists, like `Future.apply`. Have a using clause for the `ExecutionContext` makes the code much cleaner:

```
given customExecutionContext: ExecutionContext = ???
val f1 = Future(...)(using customExecutionContext)
  .map(...)(using customExecutionContext)
  .filter(...)(using customExecutionContext)
// versus:
val f2 = Future(...).map(...).filter(...)
```

Other example contexts where this idiom is useful include transaction identifiers, database connections, and web sessions.

Passing Context Functions

Context Functions are functions with context parameters only. Scala 3 introduces a new *context function type* for them, indicated by `? =>` `T`, with special handling depending on how they are used.

In essence, context functions abstract over context parameters.

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)
❺
val duration1 = Await.result(future1, 2.second)
❻
val duration2 = Await.result(future2, 2.second)
```

❶ Type alias for a context function with an `ExecutionContext`.

❷ Compare this definition of `FutureCF.apply()` to `Future.apply()` above, which we are calling here. The implicit `ExecutionContext` is passed to `Future.apply()`.

❸ Define some work that will be passed to futures; sleep for some `Duration` and return the actual elapsed time as a `Duration`.

❹ Create a future with the implicit value, like calling `Future.apply()`.

❺ Create another future specifying the implicit argument explicitly.

❻ 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 look at a more extensive example inspired by the context functions documentation page. We'll create a *domain-specific language* (DSL) for constructing JSON. For simplicity, we won't construct instances for some JSON library, just JSON-formatted strings. To motivate the example, let's begin with an entry point that shows the DSL in action:

```
// src/main/scala/progscala3/contexts/json/JSONBuilder.scala
package progscala3.contexts.json

@main def TryJSONBuilder(): Unit =
  val js = obj {
    "config" -> obj {
      "master" -> obj {
        "host" -> "192.168.1.1"
        "port" -> 8000
        "security" -> "null"
        // "foo" -> (1, 2.2, "three")  // doesn't compile!
      }
      "nodes" -> array {
        aobj {                          // "array object"
          "name" -> "node1"
          "host" -> "192.168.1.10"
        }
        aobj {
          "name" -> "node2"
          "host" -> "192.168.1.20"
        }
        "otherThing" -> 2
      }
    }
  }
  println(js)
```

Let's try it in SBT. I reformatted the output for better legibility:

```
> runMain progscala3.contexts.json.TryJSONBuilder
...
{
  "config": {
    "master": {
      "host": "192.168.1.1", "port": 8000, "security": null
```

```
    },
    "nodes": [
      {"name": "node1", "host": "192.168.1.10"},
      {"name": "node2", "host": "192.168.1.20"},
      "otherThing" -> 2
    ]
  }
}
```

Now let's work through the implementation (same source file):

```
object JSONElement:
  def valueString[T](t: T): String = t match
❶
    case "null" => "null"
    case s: String => "\""+s+"\""
    case _ => t.toString

sealed trait JSONElement
❷
case class JSONKeyedElement[T](key: String, element: T)
extends JSONElement:
  override def toString = "\""+key+"\":
"+JSONElement.valueString(element)
case class JSONArrayElement[T](element: T) extends
JSONElement:
  override def toString = JSONElement.valueString(element)
```

❶ Return the correct string representation for a value. JSON allows `null`s, for which we'll expect the user to use the string `"null"` (as shown in the example). Hence, `valueString` returns `null` without quotes, all other strings in double quotes, and for everything else, the output of `toString`.

❷ We can model everything as either a "keyed" element of the form `"key": value` or just a value, but the latter only appear as elements in arrays.

Continuing, we have types for JSON objects and arrays:

```
import scala.collection.mutable.ArrayBuffer

trait JSONContainer(open: String, close: String) extends
JSONElement:  ❶
  val elements = new ArrayBuffer[JSONElement]
  def add(e: JSONElement): Unit = elements += e
  override def toString = elements.mkString(open, ", ",
close)

class JSONObject extends JSONContainer("{", "}")
class JSONArray extends JSONContainer("[", "]")
```

❶ For both JSON objects and arrays, we add elements to a mutable array buffer. There are two places the `add` method is called, discussed below.

Note that traits can define constructor parameters, like classes. For our purposes, only the opening and closing delimiter differ between objects and arrays. The concrete classes for them define the correct delimiters.

```
sealed trait ValidJSONValue[T]
❶
given ValidJSONValue[Int]
given ValidJSONValue[Double]
given ValidJSONValue[String]
given ValidJSONValue[Boolean]
given ValidJSONValue[JSONObject]
given ValidJSONValue[JSONArray]

extension [T : ValidJSONValue] (name: String)
❷
  def ->(element: T)(using jc: JSONContainer) =
    jc.add(JSONNamedElement(name, element))
```

❶ These given instances of `ValidJSONValue[T]` are *witnesses*, constraining the allowed types of JSON values (see "Constraining Allowed Instances").

❷ This `String` extension method that is constrained by `ValidJSONValue[T]`. It constructs `JSONKeyedElements` using `"key" -> value`, just like tuple pairs, but constrained by the *context bound* `T : ValidJSONValue`. We use a `JSONContainer` because these key-value pairs only occur inside containers (objects or arrays) in the DSL. It is here that we add the key-value pairs to the container `jc`.

If you try a tuple value, it will fail to compile, as shown in a comment in `TryJSONBuilder`!

In Scala 2, you would need to declare derived classes of `ValidJSONValue[T]`, like this:

```
implicit object VJSONInt extends ValidJSONValue[Int]
...
```

Finally, we see the actual context functions in action:

```
def obj(init: JSONObject ?=> Unit) =
❶
  given jo as JSONObject
  init
  jo

def aobj(init: JSONObject ?=> Unit)(using jc: JSONContainer)
=          ❷
  given jo as JSONObject
  init
  jc.add(jo)

def array(init: JSONArray ?=> Unit) =
❸
  given ja as JSONArray
  init
  ja
```

❶ A whole JSON object, as well as nested objects, starts with `obj`. Refer to the example in `TryJSONBuilder`. Where does the `init` context function of type `JSONObject ?=> Unit` come from? It is constructed by the compiler from the expressions inside the braces passed as the argument to `obj`. Or, as it appears in the DSL, the braces after the `obj` "keyword". Next, the `given` clause creates an instance of `JSONObject` named `jo`. Then, `init` is evaluated, where `jo` will be used to satisfy using clauses inside those nested expressions. Finally, we return `jo`.

❷ Use `aobj` to define objects as array elements. Note that this function has a using clause, unlike `obj`, which will always a `JSONArray`. Unfortunately, the name `obj` can't be overloaded here, because the compiler would consider the two definitions ambiguous. The body of `aobj` is the second place where the `add` method is called. Recall that the other location is inside the `String` extension method `->`.

❸ Define an array. This body is very similar to `obj`.

So, if you find you need a small DSL for expressing structure, context functions is one tool at your disposal. We'll explore more tools for DSLs in [Link to Come].

Constraining Allowed Instances

The "given" instances of `ValidJSONValue[T]` in the previous example were used as context bounds that constrained the allowed types that could be used for type parameter `T` in the `String` extension method `->(element: T)`.

What was new is that we did no actual work with these instances. Only their existence mattered. They "witnessed" the allowed types for JSON elements. So, because we didn't provide an instance for three-element tuples, for example, attempting to use a tuple value in the DSL, such as `"stuff" -> (1, "two", 3.3)`, causes a compilation error.

Sometimes a context bound is used in both ways, as a witness and 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 instances.

```
// 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")

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](nameValue: (String, T))(using Conv[T]): Record
=         ❸
    Record(contents + nameValue)
  def get[T](colName: String)(using toT: Conv[T]): Try[T] =
❹
    Try(toT(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] as 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 -> $one, $two,
$three, $four, $five")
  println(s"bad1, bad2, bad3 -> $bad1, $bad2, $bad3")
```

❶ The companion object defines `make` to start "safe" construction of a `Record`. It also defines a type alias for `Conversion`, where we always use `Any` as the first type parameter. This alias is necessary when we define `given ab` below.

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 anonymous context parameter is used only to constrain the allowed values for `T`. It's `apply` method won't be used. Since `Records` are immutable, a new instance is returned.

❹ A method to retrieve a field value with the desired type `T`. Here the context parameter both constrains the allowed `T` types and it 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 below.

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 `given ab` declares a given for pairs, but the `A` and `B` types are constrained to be other allowed types, including other pairs!

❻ Attempting to retrieve columns with the wrong types. Attempting to retrieve an unsupported `Byte` column 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 runtime failure we have we can't prevent at compile time is attempting to get a column with the wrong type.

The type alias `Conv[T]` not only made the code more concise than using `Conversion[Any,T]`, it is necessary for the context bounds on `A` and `B` in `ab`. This is 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 the type alias `Conv[T]` and use it for the bounds in `ab`. Using a type alias like this is a useful trick when the number of type parameters in a type don't match what your need!

As a reminder, use of `given` provides a more concise syntax than the Scala 2 way of declaring an implicit value (which is still supported):

```
given Conv[Int] = _.asInstanceOf[Int]                    //
Scala 3
// vs.
implicit val toInt: Conv[Int] = new Conv[Int]:           //
Scala 2
  def apply(any: Any): Int = any.asInstanceOf[Int]
```

To recap, we limited the allowed types that can be used for a parameterized method by passing an implicit parameter and only defining given values that match the types we want to allow.

This example was inspired by an API I once wrote to work with Cassandra.

Implicit Evidence

In the previous example, the `Record.add` method showed one way to constrain the allowed types without doing anything else with the context bounds. Now we'll discuss another technique called *implicit evidence*.

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 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 `toMap` if the sequence is *not* a sequence of pairs.

The `toMap` method 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 `<:`, e.g., `A <: (K,V)`.

Recall we said that types with two type parameters can be written in "infix" notation. So, the following two expressions are equivalent:

```
<:<[A, (T,U)]
A <:< (T,U)
```

Now, when we have a traversable collection that we want to convert to a `Map`, the implicit evidence `ev` value we need will be synthesized by the compiler, but only if `A <: (T,U)`; that is, if `A` is actually a pair of types. If true, then `toMap` can be called and it simply passes the elements of the traversable to the `Map` constructor. However, if `A` is not a pair type, the code fails to compile.

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 `=:=`.

Working Around Type Erasure with Context Bounds

Context bounds can also be used to work around limitations due to *type erasure* on the JVM.

For historical reasons, the JVM "forgets" the supplied type arguments for parameterized types. For example, consider the following

definitions for an *overloaded* method with unique type signatures, at least to human readers:

```
scala> object O:
     |   def m(seq: Seq[Int]): Unit = println(s"Seq[Int]:
$seq")
     |   def m(seq: Seq[String]): Unit =
println(s"Seq[String]: $seq")
     |
3 |  def m(seq: Seq[String]): Unit = println(s"Seq[String]:
$seq")
  |      ^
  |      Double definition:
  |      def m(seq: Seq[Int]): Unit in object O at line 2
and
  |      def m(seq: Seq[String]): Unit in object O at line 3
  |      have the same type after erasure.
```

So, the compiler disallows the definitions because they are effectively the same in byte code.

> **WARNING**
>
> Try inputting these two method definitions in the REPL without the enclosing `object O`. You'll see no complaints, because the REPL lets you *redefine* anything, for your convenience. You will have one method definition, for `Seq[String]`, instead of two.

However, we can add an implicit parameter to disambiguate the methods:

```
//
src/script/scala/progscala3/contexts/UsingTypeErasureWorkaro
und.scala
object M:
  implicit object IntMarker
❶
  implicit object StringMarker
```

```
  def m(seq: Seq[Int])(using IntMarker.type): String =
❷
    s"Seq[Int]: $seq"

  def m(seq: Seq[String])(using StringMarker.type): String =
    s"Seq[String]: $seq"

import M._
```

The last three lines produce the following results:

```
scala> m(Seq(1,2,3))
     | m(Seq("one", "two", "three"))
val res0: String = Seq[Int]: List(1, 2, 3)
val res1: String = Seq[String]: List(one, two, three)

scala> m(Seq("one" -> 1, "two" -> 2, "three" -> 3))    //
ERROR
1 |m(Seq("one" -> 1, "two" -> 2, "three" -> 3))
  |^
  |None of the overloaded alternatives of method m in object
M with types
  |...
```

❶ Define two special-purpose implicit objects that will be used to disambiguate the methods affected by type erasure.

❷ Redefinition of the method that takes `Seq[Int]`. It now has a second parameter list expecting an implicit `IntMarker.type` (because `IntMarker` is an `object`). Then define a similar method for `Seq[String]`.

Now the compiler considers the two `m` methods to be distinct after type erasure.

You might wonder why I didn't use implicit `Int` and `String` values, rather than invent new types. Using implicit values for very

common types is not recommended. It would be too easy for one or more implicit `String` values, for example, to show up in a particular scope. If you don't expect one to be there, you might 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 all of them are valid choices and the compiler can't decide which to use.

At least the second scenario triggers an immediate error rather than allowing unintentional behavior to occur.

The safer bet is to limit your use of implicit parameters and values to very specific, purpose-built types.

> **WARNING**
>
> Avoid using context parameters for very common types like `Int` and `String`, as they are more likely to cause confusing behavior or compilation errors.

We'll discuss type erasure in more detail in [Link to Come].

Rules for Using Clauses

The sidebar lists the general rules for using clauses.

> **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, including what happens when a regular parameter list follows an using clause, which is allowed in Scala 3, but not in Scala 2:

```
//
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 as U1[Int](0)
given u2s as U2[String]("one")
```

❶ One using clause with two values.

❷ Two using clauses, each with one value.

❸ One using clause sandwiched between two regular parameter lists.

Now use them:

```
scala> f1("f1a")
     | f1("f1b")(using u1i, u2s)
     | f2("f2a")
     | f2("f2b")(using u1i)(using u2s)
     | f3("f3a")
     | f3("f3b")(using u1i)
     | f3("f3c")(using u1i)(u2s)
```

```
val res0: String = f1: f1a: U1(0), U2(one)
val res1: String = f1: f1b: U1(0), U2(one)
val res2: String = f2: f2a: U1(0), U2(one)
val res3: String = f2: f2b: U1(0), U2(one)
val res4: U2[Any] => String =
Lambda$7814/0x000000080360d040@4aa25f5d
val res5: U2[Any] => String =
Lambda$7815/0x000000080360c840@521dc499
val res6: String = f3: f3c: U1(0), U2(one)
```

The results for `f1` and `f2` should make sense; they are functionally equivalent. Recall that when passing values explicitly, the `using` keyword is required.

Now consider `res4` through `res6`. First, `res6` should be unsurprising, as we explicitly provided arguments for all three lists.

Partial application is the explanation for `res4` and `res5`. For methods with more than one parameter list, if you invoke them with a subset of the *leading* parameter lists, a new function is returned expecting the *rest* of the parameter lists. For `res4`, the given is used for the second parameter list, the using clause, while a value is explicitly provided for the using clause in the `res5` definition. Hence, both return the same thing.

For both `res4` and `res5`, the third parameter list is not a using clause and it was not provided explicitly. Therefore, the expressions returned a *function* that expects the remaining parameter list, which takes an instance of `U2`, and returns a `String`:

```
scala> val u2a = U2[Any](1.1)     // Declare a U2[Any] we can
use.
     | res4(u2a)                  // Pass it to the res4 and
res5 functions.
```

```
     | res5(u2a)
val u2a: U2[Any] = U2(1.1)
val res7: String = f3: f3a: U1(0), U2(1.1)
val res8: String = f3: f3b: U1(0), U2(1.1)
```

Because we didn't provide the third parameter list when we constructed `res4` and `res5`, the type parameter `T2` for `f3` was inferred to be the widest possible type, `Any`. Try calling `res4(m1)` or `res4(m2)` and you'll get a type error, as `m1` and `m2` are not type compatible with `U2[Any]`. It doesn't matter that `m1` and `m2` were declared as givens; we can still use them as regular parameters, as long as the types are compatible.

Improving Error Messages

Let's finish the discussion of using clauses by discussing how to improve the errors reported when an value isn't found. The compiler's default messages are usually sufficiently descriptive, but you can customize them with the `implicitNotFound` annotation,[2] as follows:

```
//
src/script/scala/progscala3/contexts/ImplicitNotFound.scala[
]

import scala.annotation.implicitNotFound

@implicitNotFound("Stringer: No implicit found ${T} :
Tagify[${T}]")
trait Tagify[T]:
  def toTag(t: T): String

case class Stringer[T : Tagify](t: T):
  override def toString: String =
    s"Stringer: ${implicitly[Tagify[T]].toTag(t)}"
```

```
object O:
  def makeXML[T](t: T)(
      implicit @implicitNotFound("No Tagify[${T}] implicit
found")
        tagger: Tagify[T]): String =
    s"<xml>${tagger.toTag(t)}</xml>"

given Tagify[Int]:
  def toTag(i: Int): String = s"<int>$i</int>"
given Tagify[String]:
  def toTag(s: String): String = s"<string>$s</string>"
```

Let's try it:

```
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)
  |                 ^
  |                Stringer: No implicit found Double :
Tagify[Double]
2 |O.makeXML(3.14569)
  |                  ^
  |                  Stringer: No implicit found Double :
Tagify[Double]
```

TODO: Still true in Scala 3 final? What about a new annotation??

Only the annotation on `Tagify` is used. The annotation on the parameter to `O.makeXML` is supposed to take precedence for the last output. This appears to be a current limitation in Scala 3.

You can only annotate types intended for use as givens. This is another reason for creating custom types for this purpose, rather than using more common types, like `Int`, `String`, or our `Person` type. 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 backwards 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 functional programming. 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.

---

1 A "regular" parameter list is also known as a *normal parameter clause*, but I have just used the more familiar *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.

2 At the time of this writing, there is no `givenNotFound` or similar replacement annotation in Scala 3.

About the Author

Dean Wampler is an expert in data engineering for scalable streaming data systems and applications of machine learning and artificial intelligence (ML/AI). He is a Principal Software Engineer at *Domino Data Lab*. Previously he worked at *Anyscale* and *Lightbend*, where he worked on scalable ML 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*, *Functional Programming for Java Developers*, and the coauthor of *Programming Hive*, all from O'Reilly. He is a contributor to several open source projects, a frequent conference speaker, and he also co-organizes several conferences around the world and several user groups in Chicago. Dean has a Ph.D. in Physics from the University of Washington. Find Dean on Twitter: @deanwampler.