

Covers Scala 3.x



SCALA

for the Impatient

Third Edition

Cay S. Horstmann

Foreword by Martin Odersky



Scala for the Impatient

Third Edition

Cay S. Horstmann

 Addison-Wesley

Table of Contents

Foreword to the First Edition

Preface

About the Author

Chapter 1 The Basics

Chapter 2 Control Structures and Functions

Chapter 3 Working with Arrays

Chapter 4 Maps, Options, and Tuples

Chapter 5 Classes

Chapter 6 Objects and Enumerations

Chapter 7 Packages, Imports, and Exports

Chapter 8 Inheritance

Chapter 9 Files and Regular Expressions

Chapter 10 Traits

Chapter 11 Operators

Chapter 12 Higher-Order Functions

Chapter 13 Collections

Chapter 14 Pattern Matching

Chapter 15 Annotations

Chapter 16 Futures

[Chapter 17 Type Parameters](#)

[Chapter 18 Advanced Types](#)

[Chapter 19 Contextual Abstractions](#)

[Chapter 20 Type Level Programming](#)

Contents

Foreword to the First Edition

Preface

About the Author

1 The Basics **A1**

- 1.1 The Scala Interpreter
 - 1.2 Declaring Values and Variables
 - 1.3 Commonly Used Types
 - 1.4 Arithmetic and Operator Overloading
 - 1.5 More about Calling Methods
 - 1.6 The `apply` Method
 - 1.7 Scaladoc
- Exercises

2 Control Structures and Functions **A1**

- 2.1 Conditional Expressions
 - 2.2 Statement Termination
 - 2.3 Block Expressions and Assignments
 - 2.4 Input and Output
 - 2.5 Loops
 - 2.6 More about the `for` Loop
 - 2.7 Functions
 - 2.8 Default and Named Arguments **L1**
-

2.9 Variable Arguments **L1**

2.10 The Main Function

2.11 Functions without Parameters

2.12 Lazy Values **L1**

2.13 Exceptions

Exercises

3 Working with Arrays **A1**

3.1 Fixed-Length Arrays

3.2 Variable-Length Arrays: Array Buffers

3.3 Traversing Arrays and Array Buffers

3.4 Transforming Arrays

3.5 Common Algorithms

3.6 Deciphering Scaladoc

3.7 Multidimensional Arrays

3.8 Interoperating with Java

Exercises

4 Maps, Options, and Tuples **A1**

4.1 Constructing a Map

4.2 Accessing Map Values

4.3 Updating Map Values

4.4 Iterating over Maps

4.5 Linked and Sorted Maps

4.6 Interoperating with Java

4.7 The `Option` Type

4.8 Tuples

4.9 Zipping

- Exercises
- 5 Classes **A1**

 - 5.1 Simple Classes and Parameterless Methods
 - 5.2 Properties with Getters and Setters
 - 5.3 Properties with Only Getters
 - 5.4 Private Fields
 - 5.5 Auxiliary Constructors
 - 5.6 The Primary Constructor
 - 5.7 Nested Classes **L1**
- Exercises
- 6 Objects and Enumerations **A1**

 - 6.1 Singletons
 - 6.2 Companion Objects
 - 6.3 Objects Extending a Class or Trait
 - 6.4 The `apply` Method
 - 6.5 Application Objects
 - 6.6 Enumerations
- Exercises
- 7 Packages, Imports, and Exports **A1**

 - 7.1 Packages
 - 7.2 Package Scope Nesting
 - 7.3 Chained Package Clauses
 - 7.4 Top-of-File Notation
 - 7.5 Package-Level Functions and Variables
 - 7.6 Package Visibility
 - 7.7 Imports

- [7.8 Imports Can Be Anywhere](#)
- [7.9 Renaming and Hiding Members](#)
- [7.10 Implicit Imports](#)
- [7.11 Exports](#)

Exercises

8 Inheritance **A1**

- [8.1 Extending a Class](#)
- [8.2 Overriding Methods](#)
- [8.3 Type Checks and Casts](#)
- [8.4 Superclass Construction](#)
- [8.5 Anonymous Subclasses](#)
- [8.6 Abstract Classes](#)
- [8.7 Abstract Fields](#)
- [8.8 Overriding Fields](#)
- [8.9 Open and Sealed Classes](#)
- [8.10 Protected Fields and Methods](#)
- [8.11 Construction Order](#)
- [8.12 The Scala Inheritance Hierarchy](#)
- [8.13 Object Equality **L1**](#)
- [8.14 Multiversal Equality **L2**](#)
- [8.15 Value Classes **L2**](#)

Exercises

9 Files and Regular Expressions **A1**

- [9.1 Reading Lines](#)
- [9.2 Reading Characters](#)
- [9.3 Reading Tokens and Numbers](#)

9.4 Reading from URLs and Other Sources

9.5 Writing Files

9.6 Visiting Directories

9.7 Serialization

9.8 Process Control **A2**

9.9 Regular Expressions

9.10 Regular Expression Groups

Exercises

10 Traits **L1**

10.1 Why No Multiple Inheritance?

10.2 Traits as Interfaces

10.3 Traits with Concrete Methods

10.4 Traits for Rich Interfaces

10.5 Objects with Traits

10.6 Layered Traits

10.7 Overriding Abstract Methods in Traits

10.8 Concrete Fields in Traits

10.9 Abstract Fields in Traits

10.10 Trait Construction Order

10.11 Trait Constructors with Parameters

10.12 Traits Extending Classes

10.13 What Happens under the Hood

10.14 Transparent Traits **L2**

10.15 Self Types **L2**

Exercises

11 Operators **L1**

- 11.1 Identifiers
 - 11.2 Infix Operators
 - 11.3 Unary Operators
 - 11.4 Assignment Operators
 - 11.5 Precedence
 - 11.6 Associativity
 - 11.7 The `apply` and `update` Methods
 - 11.8 The `unapply` Method **L2**
 - 11.9 The `unapplySeq` Method **L2**
 - 11.10 Alternative Forms of the `unapply` and `unapplySeq` Methods **L3**
 - 11.11 Dynamic Invocation **L2**
 - 11.12 Typesafe Selection and Application **L2**
- Exercises
- 12 Higher-Order Functions **L1**
 - 12.1 Functions as Values
 - 12.2 Anonymous Functions
 - 12.3 Parameters That Are Functions
 - 12.4 Parameter Inference
 - 12.5 Useful Higher-Order Functions
 - 12.6 Closures
 - 12.7 Interoperability with Lambda Expressions
 - 12.8 Currying
 - 12.9 Methods for Composing, Currying, and Tupling
 - 12.10 Control Abstractions
 - 12.11 The `return` Expression

Exercises

13 Collections **A2**

- 13.1 The Main Collections Traits
- 13.2 Mutable and Immutable Collections
- 13.3 Sequences
- 13.4 Lists
- 13.5 Sets
- 13.6 Operators for Adding or Removing Elements
- 13.7 Common Methods
- 13.8 Mapping a Function
- 13.9 Reducing, Folding, and Scanning **A3**
- 13.10 Zipping
- 13.11 Iterators
- 13.12 Lazy Lists **A3**
- 13.13 Interoperability with Java Collections

Exercises

14 Pattern Matching **A2**

- 14.1 A Better Switch
- 14.2 Guards
- 14.3 Variables in Patterns
- 14.4 Type Patterns
- 14.5 The `Matchable` Trait
- 14.6 Matching Arrays, Lists, and Tuples
- 14.7 Extractors
- 14.8 Patterns in Variable Declarations
- 14.9 Patterns in `for` Expressions

- 14.10 Case Classes
- 14.11 Matching Nested Structures
- 14.12 Sealed Classes
- 14.13 Parameterized Enumerations
- 14.14 Partial Functions **A3**
- 14.15 Infix Notation in `case` Clauses **L2**

Exercises

15 Annotations **A2**

- 15.1 What Are Annotations?
- 15.2 Annotation Placement
- 15.3 Annotation Arguments
- 15.4 Annotations for Java Features
 - 15.4.1 Bean Properties
 - 15.4.2 Serialization
 - 15.4.3 Checked Exceptions
 - 15.4.4 Variable Arguments
 - 15.4.5 Java Modifiers
- 15.5 Annotations for Optimizations
 - 15.5.1 Tail Recursion
 - 15.5.2 Lazy Values
- 15.6 Annotations for Errors and Warnings
- 15.7 Annotation Declarations

Exercises

16 Futures **A2**

- 16.1 Running Tasks in the Future
- 16.2 Waiting for Results

- 16.3 The `try` Class
- 16.4 Callbacks
- 16.5 Composing Future Tasks
- 16.6 Other `Future` Transformations
- 16.7 Methods in the `Future` Object
- 16.8 Promises
- 16.9 Execution Contexts

Exercises

17 Type Parameters **L2**

- 17.1 Generic Classes
- 17.2 Generic Functions
- 17.3 Bounds for Type Variables
- 17.4 Context Bounds
- 17.5 The `classTag` Context Bound
- 17.6 Multiple Bounds
- 17.7 Type Constraints **L3**
- 17.8 Variance
- 17.9 Co- and Contravariant Positions
- 17.10 Objects Can't Be Generic
- 17.11 Wildcards
- 17.12 Polymorphic Functions

Exercises

18 Advanced Types **L2**

- 18.1 Union Types
- 18.2 Intersection Types
- 18.3 Type Aliases

- 18.4 Structural Types
- 18.5 Literal Types
- 18.6 The Singleton Type Operator
- 18.7 Abstract Types
- 18.8 Dependent Types
- 18.9 Abstract Type Bounds

Exercises

19 Contextual Abstractions **L3**

- 19.1 Context Parameters
- 19.2 More about Context Parameters
- 19.3 Declaring Given Instances
- 19.4 Givens in `for` and `match` expressions
- 19.5 Importing Givens
- 19.6 Extension Methods
- 19.7 Where Extension Methods Are Found
- 19.8 Implicit Conversions
- 19.9 Rules for Implicit Conversions
- 19.10 Importing Implicit Conversions
- 19.11 Context Functions
- 19.12 Evidence
- 19.13 The `@implicitNotFound` Annotation

Exercises

20 Type Level Programming **L3**

- 20.1 Match Types
- 20.2 Heterogeneous Lists
- 20.3 Literal Type Arithmetic

- 20.4 Inline Code
 - 20.5 Type Classes
 - 20.6 Mirrors
 - 20.7 Type Class Derivation
 - 20.8 Higher-Kinded Types
 - 20.9 Type Lambdas
 - 20.10 A Brief Introduction into Macros
- Exercises

Foreword to the First Edition

When I met Cay Horstmann some years ago he told me that Scala needed a better introductory book. My own book had come out a little bit earlier, so of course I had to ask him what he thought was wrong with it. He responded that it was great but too long; his students would not have the patience to read through the eight hundred pages of *Programming in Scala*. I conceded that he had a point. And he set out to correct the situation by writing *Scala for the Impatient*.

I am very happy that his book has finally arrived because it really delivers on what the title says. It gives an eminently practical introduction to Scala, explains what's particular about it, how it differs from Java, how to overcome some common hurdles to learning it, and how to write good Scala code.

Scala is a highly expressive and flexible language. It lets library writers use highly sophisticated abstractions, so that library users can express themselves simply and intuitively. Therefore, depending on what kind of code you look at, it might seem very simple or very complex.

A year ago, I tried to provide some clarification by defining a set of levels for Scala and its standard library. There were three levels each for application programmers and for library designers. The junior levels could be learned quickly and would be sufficient to program productively. Intermediate levels would make programs more concise and more functional and would make libraries more flexible to use. The highest levels were for experts solving specialized tasks. At the time I wrote:

I hope this will help newcomers to the language decide in what order to pick subjects to learn, and that it will give some advice to teachers and book authors in what order to present the material.

Cay's book is the first to have systematically applied this idea. Every chapter is tagged with a level that tells you how easy or hard it is and whether it's oriented towards library writers or application programmers.

As you would expect, the first chapters give a fast-paced introduction to the basic Scala capabilities. But the book does not stop there. It also covers many of the more "senior" concepts and finally progresses to very advanced material which is not commonly covered in a language introduction, such as how to write parser combinators or make use of delimited continuations. The level tags serve as a guideline for what to pick up when. And Cay manages admirably to make even the most advanced concepts simple to understand.

I liked the concept of *Scala for the Impatient* so much that I asked Cay and his editor, Greg Doench, whether we could get the first part of the book as a free download on the Typesafe web site. They have graciously agreed to my request, and I would like to thank them for that. That way, everybody can quickly access what I believe is currently the best compact introduction to Scala.

Martin Odersky

January 2012

Preface

The evolution of traditional languages has slowed down considerably, and programmers who are eager to use more modern language features are looking elsewhere. Scala is an attractive choice; in fact, I think it is by far the most attractive choice for programmers who want to improve their productivity. Scala has a concise syntax that is refreshing after the Java boilerplate. It runs on the Java virtual machine (JVM), providing access to a huge set of libraries and tools. And Scala doesn't just target the JVM. The ScalaJS project emits JavaScript code, enabling you to write both the server-side and client-side parts of a web application in a language that isn't JavaScript. Scala embraces the functional programming style without abandoning object orientation, giving you an incremental learning path to a new paradigm. The Scala REPL lets you run quick experiments, which makes learning Scala very enjoyable. Last but not least, Scala is statically typed, enabling the compiler to find errors, so that you don't waste time finding them—or not—later in the running program. The compiler also helps you write error-free code, inferring types whenever possible so that you don't have to write (or read) them.

The first edition of this book was written when Java, C#, and C++ were mired in a malaise of ever-increasing complexity with little gain in expressive power. At the time, Scala was a welcome blast of fresh air. In the meantime, Java and other JVM languages such as Kotlin have embraced parts of the feature set of Scala. However, Scala has been blazing new trails in type-level programming, which make powerful libraries possible that you could simply not envision in a language such as Java or Kotlin.

I wrote this book for *impatient* readers who want to start programming in Scala right away. I assume you know Java, C#, JavaScript, Python, or C++, and I don't bore you with explaining variables, loops, or classes. I don't exhaustively list all the features of the language, I don't lecture you about

the superiority of one paradigm over another, and I don't make you suffer through long and contrived examples. Instead, you will get the information that you need in compact chunks that you can read and review as needed.

Scala has gained a reputation for being difficult to read, and this can certainly be true when library providers pay little attention to usability or assume that programmers are fluent in category theory. I assume that you are comfortable with object-oriented programming. I cover what you need for basic functional programming, similar in complexity to Java streams, but there are no monads to be seen. My goal is to teach you to write Scala code that is delightful rather than inscrutable.

Scala is a big language, but you can use it effectively without knowing all of its details intimately. Martin Odersky, the creator of Scala, has identified levels of expertise for application programmers and library designers—as shown in the following table.

Application Programmer	Library Designer	Overall Scala Level
Beginning A1		Beginning
Intermediate A2	Junior L1	Intermediate
Expert A3	Senior L2	Advanced
	Expert L3	Expert

For each chapter (and occasionally for individual sections), I indicate the experience level required. The chapters progress through levels A1, L1, A2, L2, A3, L3. Even if you don't want to design your own libraries, knowing about the tools that Scala provides for library designers can make you a more effective library user.

This is the third edition of this book, and I updated it thoroughly for Scala 3. Scala 3 brings major changes to the language. Classic features have been made more regular by removing awkward corner cases. Advanced features are now easier to learn. Even more powerful features have been added that were previously only available via macros. A “quiet syntax”, similar to that of Python, is easy on the eyes and is now the preferred way to write Scala 3 code.

I cover Scala 3 as it is and will be, and do not get into an elaborate evolutionary history of the past. If you need to work with Scala 2, get the second edition of this book.

I hope you enjoy learning Scala with this book. If you find errors or have suggestions for improvement, please visit <http://horstmann.com/scala> and leave a comment. On that page, you will also find up-to-date installation instructions, and an archive file containing all code examples from the book as executable programs or worksheets.

I am very grateful to Dmitry Kirsanov and Alina Kirsanova who turned my manuscript from XHTML into a beautiful book, allowing me to concentrate on the content instead of fussing with the format. Every author should have it so good!

Reviewers of this and prior editions include Adrian Cumiskey, Mike Davis, Rob Dickens, Steve Haines, Wei Hu, Susan Potter, Daniel Sobral, Craig Tataryn, David Walend, and William Wheeler. Thanks so much for your comments and suggestions!

Finally, as always, my gratitude goes to my editor, Greg Doench, for encouraging me to write this book, and for his insights during the development process.

Cay Horstmann

Berlin, 2022

About the Author

This content is currently in development.

Chapter 1

The Basics

Topics in This Chapter A1

- [1.1 The Scala Interpreter](#)
- [1.2 Declaring Values and Variables](#)
- [1.3 Commonly Used Types](#)
- [1.4 Arithmetic and Operator Overloading](#)
- [1.5 More about Calling Methods](#)
- [1.6 The `apply` Method](#)
- [1.7 Scaladoc](#)
- [Exercises](#)

In this chapter, you will learn how to use Scala as an industrial-strength pocket calculator, working interactively with numbers and arithmetic operations. We introduce a number of important Scala concepts and idioms along the way. You will also learn how to browse the Scaladoc documentation at a beginner's level.

Highlights of this introduction are:

- Using the Scala interpreter
- Defining variables with `var` and `val`
- Numeric types
- Using operators and functions
- Navigating Scaladoc

1.1 The Scala Interpreter

Depending on how you installed Scala, you can run the Scala interpreter from the command-line or from your integrated development environment. Since the installation instructions change ever so often, I put a set of instructions on the site <http://horstmann.com/scala>.

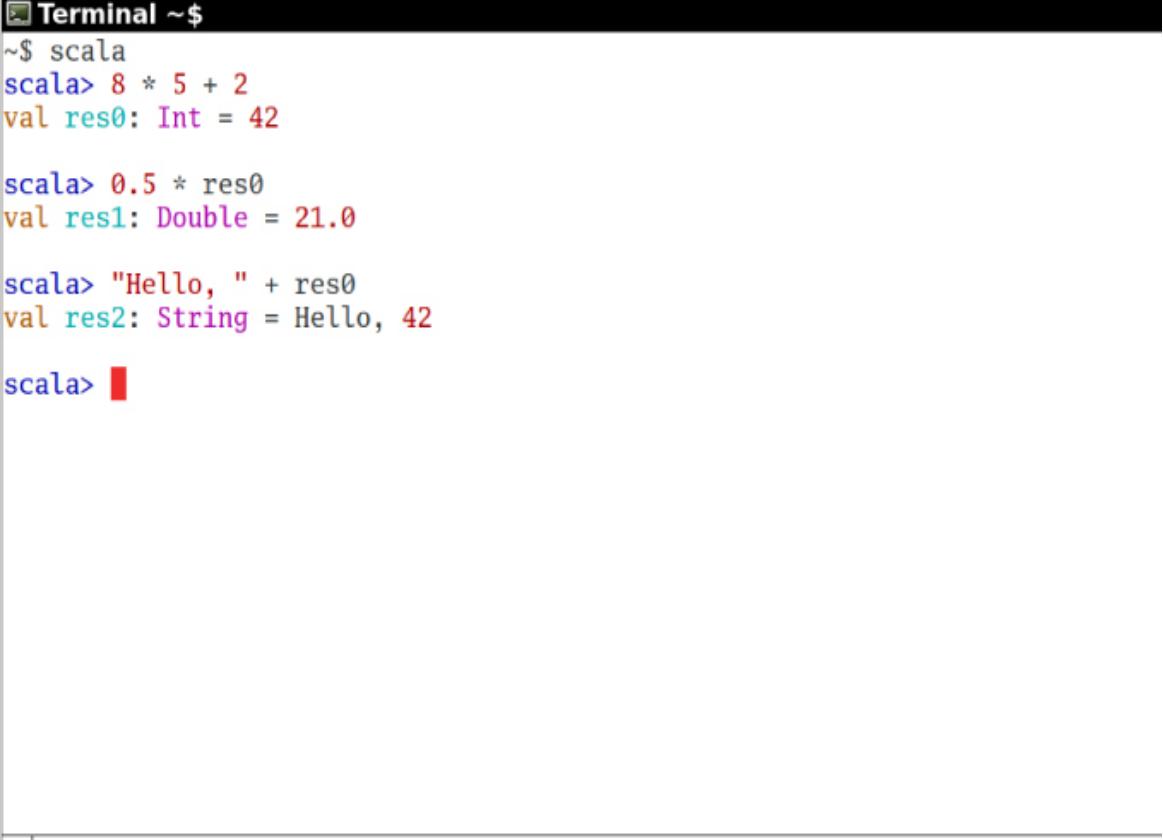
Start the interpreter and type commands followed by Enter. Each time, the interpreter displays the answer, as shown in Figure 1–1. For example, if you type **8 * 5 + 2** (as shown in boldface below), you get 42.

```
scala> 8 * 5 + 2
val res0: Int = 42
```

The answer is given the name `res0`. You can use that name in subsequent computations:

```
scala> 0.5 * res0
val res1: Double = 21.0 scala>
"Hello, " + res0
val res2: java.lang.String = Hello, 42
```

As you can see, the interpreter also displays the type of the result—in our examples, `Int`, `Double`, and `java.lang.String`.

A screenshot of a terminal window titled "Terminal ~\$". The window contains the following Scala interpreter session:

```
~$ scala
scala> 8 * 5 + 2
val res0: Int = 42

scala> 0.5 * res0
val res1: Double = 21.0

scala> "Hello, " + res0
val res2: String = Hello, 42

scala> █
```

The terminal window has a dark background with light-colored text. The prompt "scala>" appears in blue, and the results of the expressions are in green.

Figure 1–1 The Scala Interpreter



Don’t like the command shell? Several integrated development environments that support Scala have a “worksheet” feature for entering expressions and displaying their result whenever the sheet is saved.

[Figure 1–2](#) shows a worksheet in Visual Studio Code. An online version is at [!!!!!!!](#).

```
my.worksheet.sc - scala-3-project-template - VSCode
File Edit Selection View Go Run Terminal Help
EXPLORER ...
OPEN EDITORS my.worksheet.sc x Main.scala
src > main > scala > my.worksheet.sc > ...
Copy Worksheet Output
1 val answer = 6 * 7 // : Int = 42
2 answer * 0.5 // : Double = 21.0
3 ("Hello " + answer).toLowerCase() // : String = hello 42
SCALA-3-PROJECT-TEMPLATE
> .bloop
> .metals
> .vscode
> project
src
  main / scala
    > .metals
    Main.scala
    my.worksheet.sc
  test
  target
  build.sbt
  README.md
OUTLINE
0 0 △ 0
Ln 3, Col 32 Spaces: 2 UTF-8 LF Sc
```

Figure 1–2 A Scala Worksheet

When calling methods, try using *tab completion* for method names. Type `res2.to` and then hit the Tab key. If the interpreter offers choices such as

`toCharArray` `toLowerCase` `toString` `toUpperCase`

this means tab completion works in your environment. Type a U and hit the Tab key again. You now get a single completion:

`res2.toUpperCase`

Hit the Enter key, and the answer is displayed. (If you can't use tab completion in your environment, you'll have to type the complete method name yourself.)

Also try hitting the and arrow keys. In most implementations, you will see the previously issued commands, and you can edit them. Use the , , and Del keys to change the last command to

`res2.toLowerCase`

As you can see, the Scala interpreter reads an expression, evaluates it, prints it, and reads the next expression. This is called the *read-eval-print loop*, or REPL.

Technically speaking, the `scala` program is *not* an interpreter. Behind the scenes, your input is quickly compiled into bytecode, and the bytecode is executed by the Java virtual machine. For that reason, most Scala programmers prefer to call it “the REPL”.

Tip

The REPL is your friend. Instant feedback encourages experimenting, and you will feel good whenever something works.

It is a good idea to keep an editor window open at the same time, so you can copy and paste successful code snippets for later use. Also, as you try more complex examples, you may want to compose them in the editor and then paste them into the REPL.

Tip

In the REPL, type `:help` to see a list of commands. All commands start with a colon. For example, the `:type` command gives the type of an expression. You only have to enter the unique prefix of each command. For example, `:t` is the same as `:type`—at least for now, since there isn’t currently another command starting with `t`.

1.2 Declaring Values and Variables

Instead of using `res0`, `res1`, and so on, you can define your own names:

```
scala> val answer = 8 * 5 + 2
answer: Int = 42
```

You can use these names in subsequent expressions:

```
scala> 0.5 * answer
res3: Double = 21.0
```

A value declared with `val` is actually a constant—you can't change its contents:

```
scala> answer = 0
-- Error:
1 |answer = 0
|^^^^^^^^^^^
|Reassignment to val answer
```

To declare a variable whose contents can vary, use a `var`:

```
var counter = 0
counter = 1 // OK, can change a var
```

In Scala, you are encouraged to use a `val` unless you really need to change the contents. Perhaps surprisingly for Java, Python, or C++ programmers, most programs don't need many `var` variables.

Note that you need not specify the type of a value or variable. It is inferred from the type of the expression with which you initialize it. (It is an error to declare a value or variable without initializing it.)

However, you can specify the type if necessary. For example,

```
val message: String = null
val greeting: Any = "Hello"
```



In Scala, the type of a variable or function is written *after* the name of the variable or function. This makes it easier to read declarations with complex types.

I frequently move back and forth between Scala and Java. I find that my fingers write Java declarations such as `String greeting` on autopilot, so I have to rewrite them as `greeting: String`. This is a bit annoying, but when I work with complex Scala programs, I really appreciate that I don't have to decrypt C-style type declarations.



Note

You may have noticed that there were no semicolons after variable declarations or assignments. In Scala, semicolons are only required if you have multiple statements on the same line.

You can declare multiple values or variables together:

```
val xmax, ymax = 100 // Sets xmax and ymax to 100
var prefix, suffix: String = null
// prefix and suffix are both strings, initialized with null
```



In Scala, integer hexadecimal literals start with `0x`, such as `0xCAFEBABE`. There are no octal or binary literals. Long integer literals end in an `L`. Number literals can have underscores for the benefit of the human reader, such as `10_000_000_000L`.

1.3 Commonly Used Types

You have already seen some of the data types of Scala, such as `Int` and `Double`. Scala has seven numeric types: `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, and `Double`, and a `Boolean` type. In Scala, these types are *classes*. There is no distinction between primitive types and class types in Scala. You can invoke methods on numbers, for example:

```
1.toString() // Yields the string "1"
```

or, more excitingly,

```
1.to(10) // Yields Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

(We will discuss the `Range` class in [Chapter 13](#). For now, just view it as a collection of numbers.)

In Scala, there is no need for wrapper types. It is the job of the Scala compiler to convert between primitive types and wrappers. For example, if you make an array of `Int`, you get an `int[]` array in the virtual machine.

As you saw in [Section 1.1, “The Scala Interpreter,”](#) on page 1, Scala relies on the underlying `java.lang.String` class for strings. However, it augments that class with well over a hundred operations in the `StringOps` class. For example, the `intersect` method yields the characters that are common to two strings:

```
"Hello".intersect("World") // Yields "lo"
```

In this expression, the `java.lang.String` object `"Hello"` is implicitly converted to a `StringOps` object, and then the `intersect` method of the `StringOps` class is applied, as if you had written

```
scala.collection.StringOps("Hello").intersect("World")
```

So, remember to look into the `StringOps` class when you use the Scala documentation (see [Section 1.7, “Scaladoc,”](#) on page 10).

Similarly, there are classes `RichInt`, `RichDouble`, `RichChar`, and so on. Each of them has a small set of convenience methods for acting on their poor cousins—`Int`, `Double`, or `Char`. The `to` method that you saw above is actually a method of the `RichInt` class. The expression

```
1.to(10)
```

is equivalent to

```
scala.runtime.RichInt(1).to(10)
```

The `Int` value `1` is first converted to a `RichInt`, and the `to` method is applied to that value.

Finally, there are classes `BigInt` and `BigDecimal` for computations with an arbitrary (but finite) number of digits. These are backed by the `java.math.BigInteger` and `java.math.BigDecimal` classes, but, as you will see in the next section, they are much more convenient because you can use them with the usual mathematical operators.



In Scala, you use methods, not casts, to convert between numeric types. For example, `99.44.toInt` is `99`, and `99.toChar` is `'c'`. The `toString` method converts any object to a string.

To convert a string containing a number into the number, use `toInt` or `toDouble`. For example, `"99.44".toDouble` is `99.44`.

1.4 Arithmetic and Operator Overloading

Arithmetic operators in Scala work just as you would expect:

```
val answer = 8 * 5 + 2
```

The `+` `-` `*` `/` `%` operators do their usual job, as do the bit operators `&` `|` `^` `<<` `>>`. There is just one surprising aspect: These operators are actually methods. For example,

```
a + b
```

is a shorthand for

```
a.+(b)
```

Here, `+` is the name of the method. Scala has no silly prejudice against non-alphanumeric characters in method names. You can define methods with just about any symbols for names. For example, the `BigInt` class defines a method called `/%` that returns a pair containing the quotient and remainder of a division.

In general, you can write

```
a method b
```

as a shorthand for

```
a.method(b)
```

where *method* is a method with two arguments: the “receiver” `a` and the explicit argument `b`. For example, instead of

```
1.to(10)
```

you can write

```
1 to 10
```

Use whatever you think is easier to read. Beginning Scala programmers tend to stick to the dot notation, and that's fine. Of course, just about everyone seems to prefer `a + b` over `a.+(b)`.

Unlike Java, JavaScript, or C++. Scala does not have `++` or `--` operators. Instead, simply use `+=1` or `-=1`:

```
counter+=1 // Increments counter—Scala has no ++
```

Some people wonder if there is any deep reason for Scala's refusal to provide a `++` operator. (Note that you can't simply implement a method called `++`. Since the `Int` class is immutable, such a method cannot change an integer value.) The Scala designers decided it wasn't worth having yet another special rule just to save one keystroke.

You can use the usual mathematical operators with `BigInt` and `BigDecimal` objects:

```
val x: BigInt = 1234567890
x * x * x // Yields 1881676371789154860897069000
```

That's much better than Java, where you would have had to call `x.multiply(x).multiply(x)`.



Note

In Java, you cannot overload operators, and the Java designers claimed this is a good thing because it stops you from inventing crazy operators like `!@$&*` that would make your program impossible to read. Of course, that's silly; you can make your programs just as unreadable by using crazy method names like `qxywz`. Scala allows you to define operators, leaving it up to you to use this feature with restraint and good taste.

1.5 More about Calling Methods

You have already seen how to call methods on objects, such as

```
"Hello".intersect("World")
```

Methods without parameters are often invoked without parentheses. For example, the API of the `scala.collection.StringOps` class shows a method `sorted`, without `()`, which yields a new string with the letters in sorted order. Call it as

```
"Bonjour".sorted // Yields the string "Bjnooru"
```

The rule of thumb is that a parameterless method that doesn't modify the object has no parentheses. We discuss this further in [Chapter 5](#).

In Java, mathematical functions such as `sqrt` are defined as static methods of the `Math` class. In Scala, you define such methods in *singleton objects*, which we will discuss in detail in [Chapter 6](#). A package can have a *package object*. In that case, you can import the package and use the methods of the package object without any prefix:

```
import scala.math.*  
sqrt(2) // Yields 1.4142135623730951  
pow(2, 4) // Yields 16.0  
min(3, Pi) // Yields 3.0
```

If you don't import the `scala.math` package, add the package name:

```
scala.math.sqrt(2)
```



Note

If a package name starts with `scala.`, you can omit the `scala` prefix. For example, `import math.*` is equivalent to `import scala.math.*`, and `math.sqrt(2)` is the same as `scala.math.sqrt(2)`. However, in this book, I always use the `scala` prefix for clarity.

You can find more information about the `import` statement in [Chapter 7](#). For now, just use `import packageName.*` whenever you need to import a particular package.

Often, a class has a *companion object* providing methods that don't operate on instances. For example, the `BigInt` companion object to the `scala.math.BigInt` class declares `probablePrime`, which does not operate on a `BigInt`. Instead, it generates a random prime `BigInt` with a given number of bits:

```
BigInt.probablePrime(100, scala.util.Random)
```

Here, `Random` is a singleton random number generator object, defined in the `scala.util` package. Try this in the REPL; you'll get a number such as 1039447980491200275486540240713.

1.6 The `apply` Method

In Scala, it is common to use a syntax that looks like a function call. For example, if `s` is a string, then `s(i)` is the `i`th character of the string. (In C++, JavaScript, and Python, you would write `s[i]`; in Java, `s.charAt(i)`.) Try it out in the REPL:

```
val s = "Hello"  
s(4) // Yields 'o'
```

You can think of this as an overloaded form of the `()` operator. It is implemented as a method with the name `apply`. For example, in the documentation of the `StringOps` class, you will find a method

```
def apply(i: Int): Char
```

That is, `s(4)` is a shortcut for

```
s.apply(4)
```

Why not use the `[]` operator? You can think of a sequence `s` of element type `T` as a function from $\{0, 1, \dots, n - 1\}$ to `T` that maps i to $s(i)$, the i th element of the sequence.

This argument is even more convincing for maps. As you will see in [Chapter 4](#), you look up a map value for a given key as `map(key)`. Conceptually, a map is a function from keys to values, and it makes sense to use the function notation.



Caution

Occasionally, the `()` notation conflicts with another Scala feature: “contextual” parameters. For example, the expression

```
"Bonjour".sorted(3)
```

yields an error because the `sorted` method can optionally be called with an ordering, but `3` is not a valid ordering. You can use another variable:

```
val result = "Bonjour".sorted  
result(3)
```

or call `apply` explicitly:

```
"Bonjour".sorted.apply(3)
```

When you look at the documentation for the `BigInt` companion object, you will see `apply` methods that let you convert strings or numbers to `BigInt` objects. For example, the call

```
BigInt("1234567890")
```

is a shortcut for

```
BigInt.apply("1234567890")
```

It yields a new `BigInt` object, *without having to use* `new`. For example:

```
BigInt("1234567890") * BigInt("112358111321")
```

Using the `apply` method of a companion object is a common Scala idiom for constructing objects. For example, `Array(1, 4, 9, 16)` returns an array, thanks to the `apply` method of the `Array` companion object.



Note

All through this chapter, we have assumed that Scala code is executed on the Java virtual machine. That is in fact true for the standard Scala distribution. However, the Scala.js project (<https://www.scala-js.org>) provides tools to translate Scala to JavaScript. If you take advantage of that project, you can write both the client-side and the server-side code of web applications in Scala.

1.7 Scaladoc

You use Scaladoc to navigate the Scala API (see [Figure 1–3](#)).

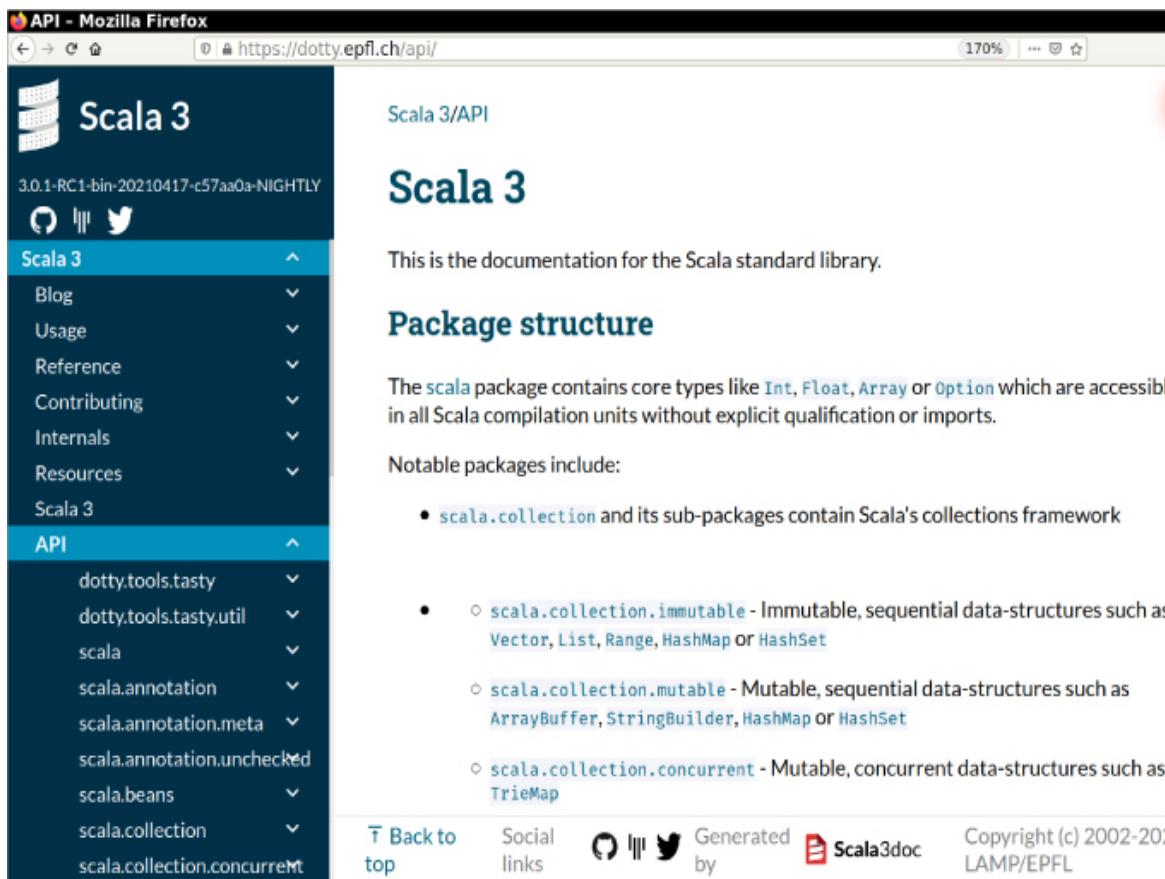


Figure 1–3 The entry page for Scaladoc

Using Scaladoc can be a bit overwhelming. Scala classes tend to have many convenience methods. Some methods use advanced features that are more meaningful to library implementors than to library users.

Here are some tips for navigating Scaladoc as a newcomer to the language.

You can browse Scaladoc online at <https://scala-lang.org/api/3.x/>, but it is a good idea to download a copy and install it locally.

Scaladoc is organized by packages. However, if you know a class or method name, don't bother navigating to the package. Simply use the search bar on the top of the entry page (see [Figure 1–4](#)).

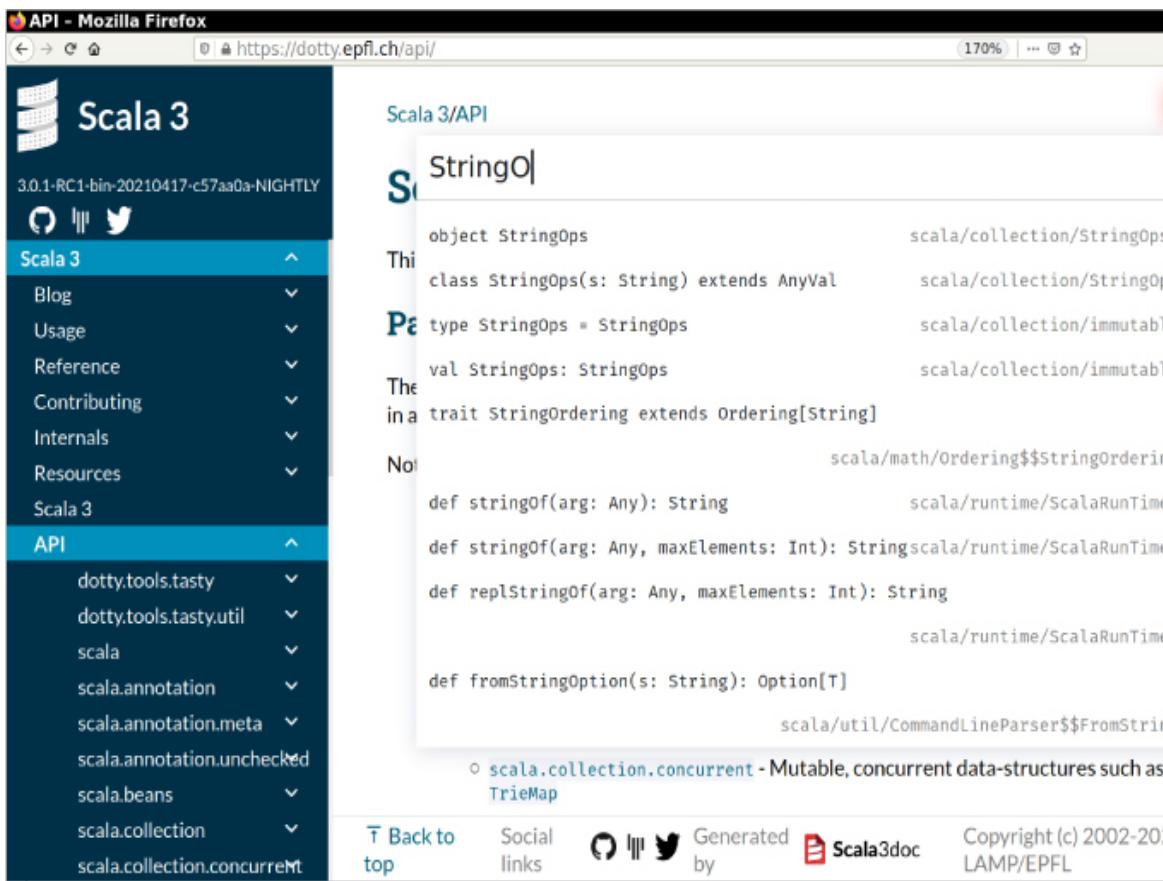


Figure 1–4 The search bar in Scaladoc

Then click on a matching class or method ([Figure 1–5](#)).

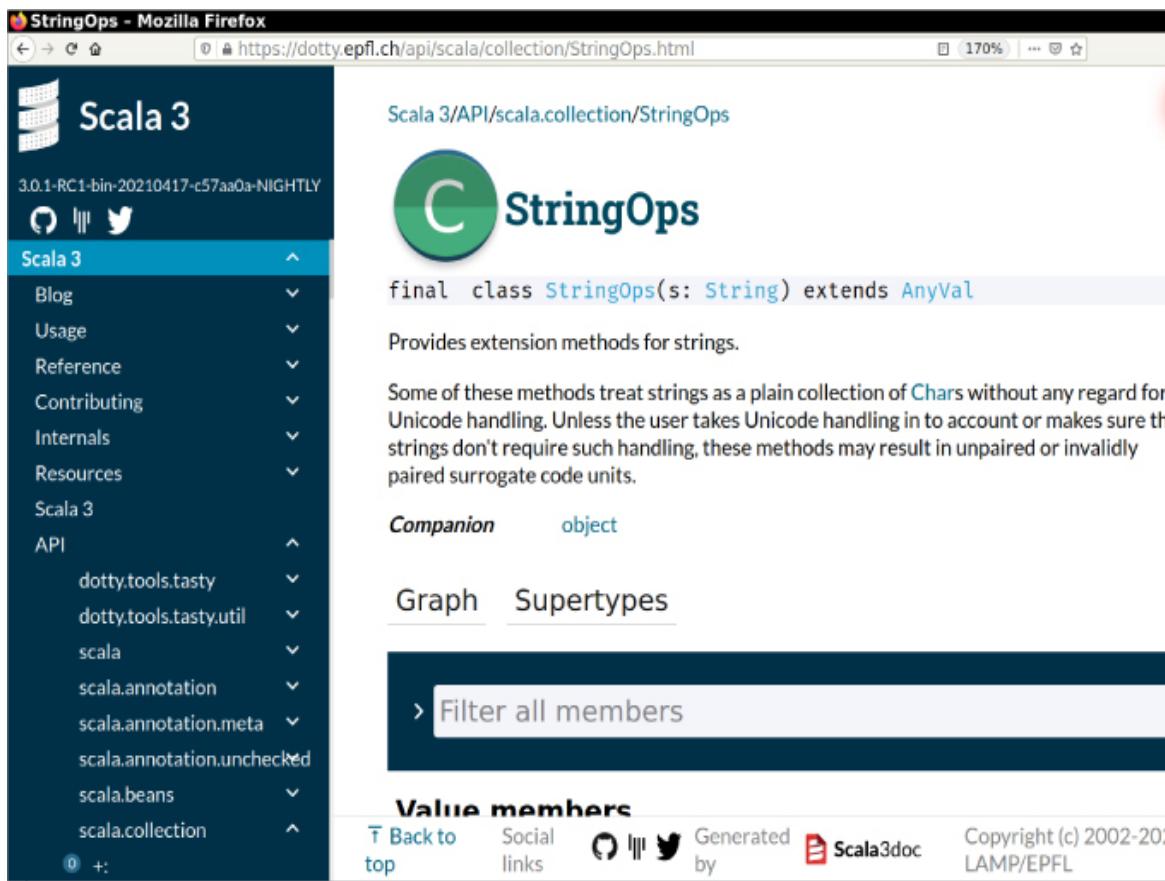


Figure 1–5 Class documentation in Scaladoc

Note the C and O symbols next to the class name. They let you navigate to the class (C) or the companion object (O). For traits (which are similar to Java interfaces and described in [Chapter 10](#)), you see t and O symbols instead.

Keep these tips in mind:

- Remember to look into `RichInt`, `RichDouble`, and so on, if you want to know how to work with numeric types. Similarly, to work with strings, look into `StringOps`.
- The mathematical functions are in the *package* `scala.math`, not in any class.
- Sometimes, you'll see methods with funny names. For example, `BigInt` has a method `unary_-`. As you will see in [Chapter 11](#), this is how you define the prefix negation operator `-x`.
- Methods can have functions as parameters. For example, the `count` method in `StringOps` requires a function that returns `true` or `false` for a `Char`, specifying which characters should be counted:

```
def count(p: (Char) => Boolean) : Int
```

You supply a function, often in a very compact notation, when you call the method. As an example, the call `s.count(_.isUpper)` counts the number of uppercase characters. We will discuss this style of programming in much more detail in [Chapter 12](#).

- You'll occasionally run into classes such as `Range` or `Seq[Char]`. They mean what your intuition tells you—a range of numbers, a sequence of characters. You will learn all about these classes as you delve more deeply into Scala.
- In Scala, you use square brackets for type parameters. A `Seq[Char]` is a sequence of elements of type `Char`, and `Seq[A]` is a sequence of elements of some type `A`.
- There are several slightly different types for sequential data structures such as `Iterable`, `IterableOnce`, `IndexedSeq`, `LinearSeq`, and so on. The differences between them are not very important for beginners. When you see such a construct, just think “sequence.” For example, the `StringOps` class defines a method

```
def concat(suffix: IterableOnce[Char]): String
```

The `suffix` can be just about any character sequence, since the ability to produce the elements once is very basic. For example, the characters could come from a file or socket. We won't see for a while how to do that, but here is another example, where the characters come from a range:

```
"bob".concat('c'.to('z')) // Yields "bobcdefghijklmnopqrstuvwxyz"
```

- Don't get discouraged that there are so many methods. It's the Scala way to provide lots of methods for every conceivable use case. When you need to solve a particular problem, just look for a method that is useful. More often than not, there is one that addresses your task, which means you don't have to write so much code yourself.
- Some methods have an “implicit” or “using” parameter. For example, the `sorted` method of `StringOps` is declared as

```
def sorted[B >: Char](implicit ord: scala.math.Ordering[B]): String
```

That means that an ordering is supplied “implicitly,” using a mechanism that we will discuss in detail in [Chapter 19](#). You can ignore `implicit` and `using`

parameters for now.

- Finally, don't worry if you run into the occasional indecipherable incantation, such as the `[B >: Char]` in the declaration of `sorted`. The expression `B >: Char` means “any supertype of `Char`,” but for now, ignore that generality.
- Whenever you are confused what a method does, just try it out in the REPL:

```
"Scala".sorted // Yields "Saacl"
```

Now you can clearly see that the method returns a new string that consists of the characters in sorted order.

- Scaladoc has a query language for finding methods by their argument and return types, separated by `=>`. For example, searching for `List[String] => List[String]` yields methods that transform a list of strings into another, such as `distinct`, `reversed`, `sorted`, and `tail`.

Exercises

1. In the Scala REPL, type `3.` followed by the Tab key. What methods can be applied?
2. In the Scala REPL, compute the square root of 3, and then square that value. By how much does the result differ from 3? (Hint: The `res` variables are your friend.)
3. What happens if you define a variable `res99` in the REPL?
4. Scala lets you multiply a string with a number—try out `"crazy" * 3` in the REPL. What does this operation do? Where can you find it in Scaladoc?
5. What does `10 max 2` mean? In which class is the `max` method defined?
6. Using `BigInt`, compute 2^{1024} .
7. What do you need to import so that you can get a random prime as `probablePrime(100, Random)`, without any qualifiers before `probablePrime` and `Random`?
8. One way to create random file or directory names is to produce a random `BigInt` and convert it to base 36, yielding a string such as `"qsnvbvtomcj38o06kul"`. Poke around Scaladoc to find a way of doing this in Scala.
9. How do you get the first character of a string in Scala? The last character?

10. What do the `take`, `drop`, `takeRight`, and `dropRight` string methods do?
What advantage or disadvantage do they have over using `substring`?

Chapter 2

Control Structures and Functions

Topics in This Chapter A1

- 2.1 Conditional Expressions
- 2.2 Statement Termination
- 2.3 Block Expressions and Assignments
- 2.4 Input and Output
- 2.5 Loops
- 2.6 More about the `for` Loop
- 2.7 Functions
- 2.8 Default and Named Arguments L1
- 2.9 Variable Arguments L1
- 2.10 The Main Function
- 2.11 Functions without Parameters
- 2.12 Lazy Values L1
- 2.13 Exceptions
- Exercises

In this chapter, you will learn how to implement conditions, loops, and functions in Scala. You will encounter a fundamental difference between

Scala and other programming languages. In Java or C++, we differentiate between *expressions* (such as `3 + 4`) and *statements* (for example, an `if` statement). An expression has a value; a statement carries out an action. In Scala, almost all constructs have values. This feature can make programs more concise and easier to read.

Here are the highlights of this chapter:

- An `if` expression has a value.
- A block has a value—the value of its last expression.
- The Scala `for` loop is like an “enhanced” Java `for` loop.
- Semicolons are (mostly) optional.
- In Scala 3, indentation is preferred over braces.
- The `void` type is `Unit`.
- Avoid using `return` in a function.
- Scala functions can have default, named, and variable arguments.
- The program entrypoint is the function annotated with `@main`.
- Exceptions work just like in Java or C++, but you use a “pattern matching” syntax for `catch`.
- Scala has no checked exceptions.

2.1 Conditional Expressions

Like most programming languages, Scala has an `if/else` construct. In programming languages such as Java and C++, `if/else` is a *statement*. It carries out one action or another. However, in Scala, an `if/else` is an *expression*. It has a value, namely the value of the expression that follows the `if` or `else`. For example,

```
if x > 0 then 1 else -1
```

has a value of `1` or `-1`, depending on the value of `x`. You can put that value in a variable:

```
val s = if x > 0 then 1 else -1
```

Contrast this with

```
var t = 0  
if x > 0 then t = 1 else t = -1
```

The first form is better because it can be used to initialize a `val`. In the second form, `t` needs to be a `var`.

As already mentioned, semicolons are mostly optional in Scala—see [Section 2.2, “Statement Termination,”](#) on page 20.



Note

Scala also supports the C-style syntax `if (condition) ...`, but this book uses the `if condition then ...` syntax that was introduced in Scala 3.

Java and C++ have a `?:` operator for conditionally selecting among two expressions:

```
x > 0 ? 1 : -1 // Java or C++
```

In Python, you write

```
1 if x > 0 else -1 // Python
```

Both are equivalent to the Scala expression `if x > 0 then 1 else -1`. In Scala, you don’t need separate forms for conditional expressions and statements.

In Scala, every expression has a type. For example, the expression `if x > 0 then 1 else -1` has the type `Int` because both branches have the type `Int`. The type of a mixed-type expression, such as

```
if x > 0 then "positive" else -1
```

is the common supertype of both branches. In this example, one branch is a `java.lang.String`, and the other an `Int`. As it happens, these two types have a common supertype `Matchable`. In the most extreme case, the expression has as its type the most general of all types, called `Any`.

If the `else` part is omitted, for example in

```
if x > 0 then "positive"
```

it is possible that the `if` expression yields no value. However, in Scala, every expression is supposed to have *some* value. This is finessed by introducing a class `Unit` that has one value, written as `()`. The `if` without an `else` is considered a statement and always has value `()`.

Think of `()` as a placeholder for “no useful value,” and of `Unit` as an analog of `void` in Java or C++.

(Technically speaking, `void` has no value whereas `Unit` has one value that signifies “no value.” If you are so inclined, you can ponder the difference between an empty wallet and a wallet with a bill labeled “no dollars.”)

The Scala REPL does not display the `()` value. To see the value in the REPL, print it:

```
println(if x > 0 then "positive")
```

You will get a warning when you use an `if` expression without an `else`. It is usually an error.

However, if the body of the `if` has type `Unit`, there is no problem:

```
if x < 0 then println("negative")
```

The `println` method is only called for its side effect—to display a string on the console. It has return type `Unit` and always returns `()`. Therefore, the value of the `if` expression is always `()`. This usage is correct, and no warning is displayed.



Scala has no `switch` statement, but it has a much more powerful pattern matching mechanism that we will discuss in [Chapter 14](#). For now, just use a sequence of `if` statements.



Caution

The REPL is more nearsighted than the compiler—it only sees one line of code at a time. For example, consider typing the following code into the REPL, one character at a time:

```
if x > 0 then 1  
else if x == 0 then 0 else -1
```

As soon as you hit the Enter key at the end of the first line, the REPL executes `if x > 0 then 1` and shows the answer. (The answer is `()`, which confusingly is not displayed, followed by a warning that an `if` without `else` is a statement.) Then an error is reported when you hit Enter after the second line since an `else` without an `if` is illegal.

To avoid this issue, put the `else` on the same line so that the REPL knows that more code is coming:

```
if x > 0 then 1 else  
if x == 0 then 0 else -1
```

This is only a concern in the REPL. In a compiled program, the parser will find the `else` on the next line.



Note

If you copy a block of code from a text editor or a web page and *paste* it into the REPL, then this problem doesn't occur. The REPL analyzes a pasted code snippet in its entirety.

2.2 Statement Termination

In Java and C++, every statement ends with a semicolon. In Scala—like in JavaScript and other scripting languages—a semicolon is never required if it falls just before the end of the line. A semicolon is also optional before an `}`, an `else`, and similar locations where it is clear from context that the end of a statement has been reached.

However, if you want to have more than one statement on a single line, you need to separate them with semicolons. For example,

```
if n > 0 then { r = r * n; n -= 1 }
```

A semicolon is needed to separate `r = r * n` and `n -= 1`. Because of the `}`, no semicolon is needed after the second statement.

If you want to continue a long statement over two lines, make sure that the first line ends in a symbol that *cannot be* the end of a statement. An operator is often a good choice:

```
s = s + v * t + // The + tells the parser that this is not the end
    0.5 * a * t * t
```

In practice, long expressions usually involve function or method calls, and then you don't need to worry much—after an opening `(`, the compiler won't infer the end of a statement until it has seen the matching `)`.

Many programmers coming from Java or C++ are initially uncomfortable about omitting semicolons. If you prefer to put them in, feel free to—they do no harm.

2.3 Block Expressions and Assignments

In Java, JavaScript, or C++, a block statement is a sequence of statements enclosed in `{ }` . You use a block statement whenever you need to put multiple actions in the body of a branch or loop statement.

In Scala, a `{ }` block contains a sequence of *expressions*, and the result is also an expression. The value of the block is the value of the last

expression.

This feature can be useful if the initialization of a `val` takes more than one step. For example,

```
var distance =
{ val dx = x - x0; val dy = y - y0; scala.math.sqrt(dx * dx +
dy * dy) }
```

The value of the `{ }` block is the last expression, shown here in bold. The variables `dx` and `dy`, which were only needed as intermediate values in the computation, are neatly hidden from the rest of the program.

If you write the block on multiple lines, you can use indentation instead of braces:

```
distance =
  val dx = x - x0
  val dy = y - y0
  scala.math.sqrt(dx * dx + dy * dy)
```

Block indentation is common with branches and loops:

```
if n % 2 == 0 then
  a = a * a
  n = n / 2
else
  r = r * a
  n -= 1
```

You can use braces, but in Scala 3, the “quiet” indentation style is preferred. Python programmers will rejoice.

In Scala, assignments have no value—or, strictly speaking, they have a value of type `Unit`. Recall that the `Unit` type is the equivalent of the `void` type in Java and C++, with a single value written as `()`.

A block that ends with an assignment, such as

```
{ r = r * a; n -= 1 }
```

has a `Unit` value. This is not a problem, just something to be aware of when defining functions—see [Section 2.7, “Functions,”](#) on page 27.

Since assignments have `Unit` value, don’t chain them together.

```
i = j = 1 // Does not set i to 1
```

The value of `j = 1` is `()`, and it’s highly unlikely that you wanted to assign a `Unit` to `x`. (In fact, it is not easy to do—the variable `i` would need to have type `Unit` or `Any`.) In contrast, in Java, C++, and Python, the value of an assignment is the value that is being assigned. In those languages, chained assignments are useful. In Scala, make two assignments:

```
j = 1  
i = j
```

2.4 Input and Output

To print a value, use the `print` or `println` function. The latter adds a newline character after the printout. For example,

```
print("Answer: ")  
println(42)
```

yields the same output as

```
println("Answer: " + 42)
```

Use *string interpolation* for formatted output:

```
println(f"Hello, ${name}! In six months, you'll be ${age +  
0.5}%7.2f years old.")
```

A formatted string is prefixed with the letter `f`. It contains expressions that are prefixed with `$` and optionally followed by C-style format strings. The expression `$name` is replaced with the value of the variable `name`. The expression `${age + 0.5}%7.2f` is replaced with the value of `age + 0.5`,

formatted as a floating-point number of width 7 and precision 2. You need `{...}` around expressions that are not variable names.

Using the `f` interpolator is better than using the `printf` method because it is typesafe. If you accidentally use `%f` with an expression that isn't a number, the compiler reports an error.



Note

Formatted strings are one of three predefined string interpolators in the Scala library. With a prefix of `s`, strings can contain delimited expressions with a `$` prefix, but not format directives. With a prefix of `raw`, escape sequences in a string are not evaluated. For example, `raw"\n` is a newline" starts with a backslash and the letter n, not a newline character.

To include `$` and `%` characters in a formatted string, double them. For example, `f"$$price: a 50% discount"` yields a dollar sign followed by the value of `price` and “a 50% discount”.

You can also define your own interpolators—see [Exercise 12](#) on page 37. However, interpolators that produce compile-time errors (such as the `f` interpolator) need to be implemented as “macros,” an advanced technique that is briefly introduced in [Chapter 20](#).

You can read a line of input from the console with the `readLine` method of the `scala.io.StdIn` class. To read a numeric, Boolean, or character value, use `readInt`, `readDouble`, `readByte`, `readShort`, `readLong`, `readFloat`, `readBoolean`, or `readChar`. The `readLine` method, but not the other ones, takes a prompt string:

```
import scala.io.*  
val name = StdIn.readLine("Your name: ")  
print("Your age: ")  
val age = StdIn.readInt()  
println(s"Hello, ${name}! Next year, you will be ${age + 1}.")
```



Caution

Some worksheet implementations do not handle console input. To make use of console input, compile and run a program, as shown in [Section 2.10, “The Main Function,”](#) on page 31.

2.5 Loops

Scala has the same `while` loop as Java, JavaScript, C++, and Python. For example,

```
while n > 0 do
    r = r * n
    n -= 1
```

This is the “quiet” braceless syntax that is favored in Scala 3. However, you can use braces if you prefer:

```
while (n > 0) {
    r = r * n
    n -= 1
}
```



Tip

If the body of a `while` loop gets long, and you use the braceless syntax, you can add `end while` at the end to more clearly show the end of the loop:

```
while n > 0 do
    r = r * n
    // Many more lines
    n -= 1
end while
```

Scala 3 does not have a `do/while` loop. In Java, JavaScript, or C++, one might use a loop such as the following for approximating a square root:

```
estimate = 1; // Initial estimate
do { // This is Java
    previous = estimate; // Save previous estimate
    estimate = (estimate + a / estimate) / 2; // Better estimate
} while (scala.math.abs(estimate - previous) > EPSILON)
// Keep going while consecutive estimates are too far apart
```

The `do/while` loop is used because one must enter the loop at least once.

In Scala, you can instead use a `while` loop whose condition is a block:

```
while
  val previous = estimate // Work done in the block
  estimate = (estimate + a / estimate) / 2 // More work done
  scala.math.abs(estimate - previous) > EPSILON
  // This is the value of the block and the loop condition
  do () // All work was done in the condition block
```

This may not be very pretty, but `do/while` loops are not common.

Scala has no direct analog of the `for (initialize; test; update)` loop either. If you need such a loop, you have two choices. You can use a `while` loop. Or, you can use a `for` statement like this:

```
for i <- 1 to n do
  r = r * i
```

You saw the `to` method of the `RichInt` class in [Chapter 1](#). The call `1 to n` returns a `Range` of the numbers from 1 to `n` (inclusive).

The construct

```
for i <- expr do
```

makes the variable `i` traverse all values of the expression to the right of the `<-`. Exactly how that traversal works depends on the type of the expression. For a Scala collection, such as a `Range`, the loop makes `i` assume each value in turn.

 **Note**

There is no `val` or `var` before the variable in the `for` loop. The type of the variable is the element type of the collection. The scope of the loop variable extends until the end of the loop.

When traversing a string, you can loop over the index values:

```
val s = "Hello"  
var sum = 0  
for i <- 0 to s.length - 1 do  
    sum += s(i)
```

In this example, there is actually no need to use indexes. You can directly loop over the characters:

```
sum = 0  
for ch <- "Hello" do sum += ch
```

In Scala, loops are not used as often as in other languages. As you will see in [Chapter 12](#), you can often process the values in a sequence by applying a function to all of them, which can be done with a single method call.

 **Note**

Scala has no `break` or `continue` statements to break out of a loop. What to do if you need a `break`?

You can always replace `break` statements with additional Boolean control variables. Alternatively, you can use the `break` method in

the `Breaks` object:

```
import scala.util.control.Breaks.*  
breakable {  
    for (c <- "Hello, World!") do  
        if (c == ',') then break // Exits the breakable block  
        else println(c)  
}
```

Here, the control transfer is done by throwing and catching an exception, so you should avoid this mechanism when time is of essence.



Note

In Java, you cannot have two local variables with the same name and overlapping scope. In Scala, there is no such prohibition, and the normal shadowing rule applies. For example, the following is perfectly legal:

```
val k = 42  
for (k <- 1 to 10) do  
    println(k) // Here k refers to the loop variable
```

2.6 More about the `for` Loop

In the preceding section, you saw the basic form of the `for` loop. However, this construct is much richer in Scala than in Java, JavaScript, or C++. This section covers the advanced features.

You can have multiple *generators* of the form *variable* \leftarrow *expression*. For example,

```
for
    i <- 1 to 3
    j <- 1 to 3
do
    print(f"${10 * i + j}%3d")
// Prints 11 12 13 21 22 23 31 32 33
```

A *guard* is a Boolean condition preceded by `if`:

```
for
    i <- 1 to 3
    j <- 1 to 3
    if i != j
do
    print(f"${10 * i + j}%3d")
// Prints 12 13 21 23 31 32
```

You can have any number of *definitions*, introducing variables that can be used inside the loop:

```
for
    i <- 1 to 3
    from = 4 - i
    j <- from to 3
do
    print(f"${10 * i + j}%3d")
// Prints 13 22 23 31 32 33
```



Note

If you prefer, you can use semicolons instead of newlines to separate generators and definitions of a `for` loop. Semicolons before an `if` guard are optional.

```
for i <- 1 to 3; from = 4 - i; j <- from to 3 if i != j
  do println(i * 10 + j)
```

The classic syntax uses parentheses instead of the `do` keyword:

```
for (i <- 1 to 3; from = 4 - i; j <- from to 3 if i != j)
  println(i * 10 + j)
```

Braces are ok too:

```
for { i <- 1 to 3; from = 4 - i; j <- from to 3 if i != j }
  println(i * 10 + j)
```

When the body of the `for` loop starts with `yield`, the loop constructs a collection of values, one for each iteration:

```
val result = for i <- 1 to 10 yield i % 3
// Yields Vector(1, 2, 0, 1, 2, 0, 1, 2, 0, 1)
```

This type of loop is called a `for` *comprehension*.

The generated collection is compatible with the generator.

```
for c <- "Hello" yield (c + 1).toChar
// Yields the string "Ifmmp"
```

2.7 Functions

Scala has functions in addition to methods. A method operates on an object, but a function doesn't. C++ has functions as well, but in Java, you have to imitate them with static methods.

To define a function, specify the function's name, parameters, and body. Then declare it outside a class or inside a block, like this:

```
def abs(x: Double) = if x >= 0 then x else -x
```

You must specify the types of all parameters. However, as long as the function is not recursive, you need not specify the return type. The Scala compiler determines the return type from the type of the expression to the right of the = symbol.



Caution

There is some disagreement about the terminology of methods and functions in Scala. I follow the classic terminology where, unlike a function, a method has a special “receiver” or `this` parameter.

For example, `pow` in the `scala.math` package is a function, but `substring` is a method of the `String` class.

When calling the `pow` function, you supply all arguments in parentheses: `pow(2, 4)`. When calling the `substring` method, you supply a `String` argument with the dot notation, and additional arguments in parentheses: `"Hello".substring(2, 4)`. The `"Hello"` argument is the “receiver” of the method invocation.

You declare methods with `def` inside a class, trait, or object. But you can also use `def` to declare “top-level” functions outside a class, and “nested” functions inside a block. They too will be compiled into methods in the Java virtual machine. Perhaps for that reason, some people call anything declared with `def` a method. But in this book, a method only refers to a member of a class, trait, or object.

Everyone agrees that the “lambda expressions” that you will see in [Chapter 12](#) are functions.

If the body of a function requires more than one expression, use a block. The last expression of the block becomes the value that the function returns. For example, the following function returns the value of `r` after the `for` loop.

```
def fac(n: Int) =  
    var r = 1  
    for i <- 1 to n do r = r * i  
    r
```

You can optionally add an `end` statement to denote the end of a function definition:

```
def fac(n: Int) =  
    var r = 1  
    for i <- 1 to n do r = r * i  
    r  
end fac
```

This makes sense when the function body spans many lines. You can also use braces:

```
def fac(n: Int) = {  
    var r = 1  
    for i <- 1 to n do r = r * i  
    r  
}
```

Note that there is no `return` keyword. You should not use the `return` statement in Scala. Instead, organize your code so that the last expression of the function body yields the value to be returned.



Caution

The Scala `return` statement is implemented by throwing and catching an exception. This is inefficient and brittle. Code that catches exceptions can interfere with the mechanism.

In an *anonymous function*, `return` doesn't return a value to its caller but to the caller of the enclosing named function. This behavior is dangerous and has been deprecated in Scala 3.

With a recursive function, you must specify the return type. For example,

```
def fac(n: Int): Int = if n <= 0 then 1 else n * fac(n - 1)
```

Without the return type, the Scala compiler couldn't verify that the type of `n * fac(n - 1)` is an `Int`.

Note

Some programming languages (such as ML and Haskell) *can* infer the type of a recursive function, using the Hindley-Milner algorithm. However, this doesn't work well in an object-oriented language. Extending the Hindley-Milner algorithm so it can handle subtypes is still a research problem.

A function need not return any value. Consider this example:

```
def log(sb: StringBuilder, message: String) =  
    sb.append(java.time.Instant.now())  
    sb.append(": ")  
    sb.append(message)  
    sb.append("\n")
```

The function appends a message with a time stamp.

Technically, the function returns a value, namely the value returned by the last call to `append`. We aren't interested in that value, whatever it may be. To clarify that the function is only called for its side effect, declare the return type as `Unit`:

```
def log(sb: StringBuilder, message: String) : Unit = ...
```

2.8 Default and Named Arguments L1

You can provide default arguments for functions that are used when you don't specify explicit values. For example,

```
def decorate(str: String, left: String = "[", right: String =
")] =  
    left + str + right
```

This function has two parameters, `left` and `right`, with default arguments "`[`" and "`]`".

If you call `decorate("Hello")`, you get "`[Hello]`". If you don't like the defaults, supply your own: `decorate("Hello", "<<<", ">>>")`.

If you supply fewer arguments than there are parameters, the defaults are applied from the end. For example, `decorate("Hello", ">>>[")` uses the default value of the `right` parameter, yielding "`>>>[Hello]`".

You can also specify the parameter names when you supply the arguments. For example,

```
decorate(left = "<<<", str = "Hello", right = ">>>")
```

The result is "`<<<Hello>>>`". Note that the named arguments need not be in the same order as the parameters.

Named arguments can make a function call more readable. They are also useful if a function has many default parameters.

You can mix unnamed and named arguments, provided the unnamed ones come first:

```
decorate("Hello", right = "]<<<") // Calls decorate("Hello", "[",
")]<<<")
```

2.9 Variable Arguments L1

Sometimes, it is convenient to implement a function that can take a variable number of arguments. The following example shows the syntax:

```
def sum(args: Int*) =  
    var result = 0
```

```
for arg <- args do result += arg
result
```

You can call this function with as many arguments as you like.

```
sum(1, 4, 9, 16, 25)
```

The function receives a single parameter of type `seq`, which we will discuss in [Chapter 13](#). For now, all you need to know is that you can use a `for` loop to visit each element.

If you already have a sequence of values, you cannot pass it directly to such a function. For example, the following is not correct:

```
sum(1 to 5) // Error
```

If the `sum` function is called with one argument, that must be a single integer, not a range of integers. The remedy is to tell the compiler that you want the parameter to be considered an argument sequence. Use a postfix `*`, like this:

```
sum((1 to 5)*) // Consider 1 to 5 as an argument sequence
```

This call syntax is needed in a recursive definition:

```
def recursiveSum(args: Int*) : Int =
  if args.length == 0 then 0
  else args.head + recursiveSum(args.tail*)
```

Here, the `head` of a sequence is its initial element, and `tail` is a sequence of all other elements. That's again a `seq` object, and we have to use a postfix `*` to convert it to an argument sequence.

2.10 The Main Function

Every program must start somewhere. When running a compiled executable, the entrypoint is the function defined with the `@main` annotation:

```
@main def hello() =  
  println("Hello, World!")
```

It does not matter what the main function is called.

To process command-line arguments, provide a parameter of type `String*`:

```
@main def hello2(args: String*) =  
  println(s"Hello, ${args(1)}!")
```

You can also specify types for the command-line arguments:

```
@main def hello3(repetition: Int, name: String) =  
  println("Hello " * repetition + name)
```

Then the first command-line argument must be an integer (or, more accurately, a string containing an integer).

Parsers for the types `Boolean`, `Byte`, `Short`, `Int`, `Long`, `Float`, and `Double` are supplied. You can parse other types—see [Exercise 13](#) on page 37.



Note

When compiling a program with a `@main` annotated function, the compiler produces a class file whose name is the name of that function, *not* the name of the source file. For example, if the `hello` function is in a file `Main.scala`, compiling that file yields `hello.class`.

2.11 Functions without Parameters

You can declare a function without any parameters:

```
def words =  
  scala.io.Source.fromFile("/usr/share/dict/words").mkString
```

You call the function as

```
words
```

without parentheses.

In contrast, if you define the function with an empty parameter list

```
def words() =  
  scala.io.Source.fromFile("/usr/share/dict/words").mkString
```

the function is invoked with parentheses:

```
words()
```

Why would you ever want to omit the parentheses? In Scala, the convention is to drop parentheses if the function is “idempotent”—that is, if it always returns the same value.

For example, if you assume that the contents of the file `/usr/share/dict/words` does not change, then the function without parentheses is the right choice.

Why wouldn't you just use a variable?

```
val words =  
  scala.io.Source.fromFile("/usr/share/dict/words").mkString
```

The value is set whether or not you use it. With a function, the computation is deferred until you invoke it.



Note

Scala 3 is stricter about the use of parentheses than prior versions. If you define a function with parentheses, you must invoke it with parentheses. Conversely, a function that was defined without parentheses must be called without them. For compatibility, this rule does not apply to legacy functions. The `scala.math.random` function is definitely not idempotent. You expect each invocation `scala.math.random()` to return a different value. Nevertheless, it can also be called without parentheses.



`@main def hello4 =
 println("Hello, World!")`

2.12 Lazy Values L1

When a `val` is declared as `lazy`, its initialization is deferred until it is accessed for the first time. For example,

```
lazy val words =  
  scala.io.Source.fromFile("/usr/share/dict/words").mkString
```

(We will discuss file operations in [Chapter 9](#). For now, just take it for granted that this call reads all characters from a file into a string.)

If the program never accesses `words`, the file is never opened. To verify this, try it out in the REPL, but misspell the file name. There will be no error when the initialization statement is executed. However, if you access `words`, you will get an error message that the file is not found.

Lazy values are useful to delay costly initialization statements. They can also deal with other initialization issues, such as circular dependencies. Moreover, they are essential for developing lazy data structures—see [Chapter 13](#).

You can think of lazy values as halfway between `val` and `def`. Compare

```
val words1 =  
  println("words1: Reading file")  
  scala.io.Source.fromFile("/usr/share/dict/words").mkString  
  // Evaluated as soon as words1 is defined  
  
lazy val words2 =  
  println("words2: Reading file")  
  scala.io.Source.fromFile("/usr/share/dict/words").mkString
```

```
// Evaluated the first time words2 is used
def words3 =
  println("words3: Reading file")
  scala.io.Source.fromFile("/usr/share/dict/words").mkString
// Evaluated every time words3 is used
```



Note

Laziness is not cost-free. Every time a lazy value is accessed, a method is called that checks, in a threadsafe manner, whether the value has already been initialized.

2.13 Exceptions

Scala exceptions work the same way as in Java, JavaScript, C++, or Python. When you throw an exception, for example

```
throw IllegalArgumentException("x should not be negative")
```

the current computation is aborted, and the runtime system looks for an exception handler that can accept an `IllegalArgumentException`. Control resumes with the innermost such handler. If no such handler exists, the program terminates.

As in Java, the objects that you throw need to belong to a subclass of `java.lang.Throwable`. However, unlike Java, Scala has no “checked” exceptions—you never have to declare that a function or method might throw an exception.



Note

In Java, “checked” exceptions are checked at compile time. If your method might throw an `IOException`, you must declare it. This forces programmers to think where those exceptions should be

handled, which is a laudable goal. Unfortunately, it can also give rise to monstrous method signatures such as `void doSomething()`
`throws IOException, InterruptedException,`
`ClassNotFoundException`. Many Java programmers detest this feature and end up defeating it by either catching exceptions too early or using excessively general exception classes. The Scala designers decided against checked exceptions, recognizing that thorough compile-time checking isn't *always* a good thing.

A `throw` expression has the special type `Nothing`. That is useful in `if/else` expressions. If one branch has type `Nothing`, the type of the `if/else` expression is the type of the other branch. For example, consider

```
if x >= 0 then scala.math.sqrt(x)
else throw IllegalArgumentException("x should not be negative")
```

The first branch has type `Double`, the second has type `Nothing`. Therefore, the `if/else` expression also has type `Double`.

The syntax for catching exceptions is modeled after the pattern matching syntax (see [Chapter 14](#)).

```
val url = URL("http://horstmann.com/fred.gif")
try
  process(url)
catch
  case _: MalformedURLException => println(s"Bad URL: $url")
  case ex: IOException => println(ex)
```

The more general exception types must come after the more specific ones.

Note that you can use `_` for the variable name if you don't need it.

The `try/finally` statement lets you dispose of a resource whether or not an exception has occurred. For example:

```
val in = URL("http://horstmann.com/cay-tiny.gif").openStream()
try
```

```
    process(in)
finally
    println("Closing input stream")
    in.close()
```

The `finally` clause is executed whether or not the `process` function throws an exception. The input stream is always closed.

This code is a bit subtle, and it raises several issues.

- What if the `URL` constructor or the `openStream` method throws an exception? Then the `try` block is never entered, and neither is the `finally` clause. That's just as well—`in` was never initialized, so it makes no sense to invoke `close` on it.
- Why isn't `val in = URL(...).openStream()` inside the `try` block? Then the scope of `in` would not extend to the `finally` clause.
- What if `in.close()` throws an exception? Then that exception is thrown out of the statement, superseding any earlier one. (This is just like in Java, and it isn't very nice. Ideally, the old exception would stay attached to the new one.)

Note that `try/catch` and `try/finally` have complementary goals. The `try/catch` statement handles exceptions, and the `try/finally` statement takes some action (usually cleanup) when an exception is not handled. You can combine them into a single `try/catch/finally` statement:

```
try
...
catch
...
finally
...
```

This is the same as

```
try
try
...
...
```

```
catch
...
finally
...
```

In practice, that combination is not often used because exceptions are usually caught far from where they are thrown, whereas cleanup is needed close to the point where exceptions can occur.

Note

The `Try` class is designed to work with computations that may fail with exceptions. We will look at it more closely in [Chapter 16](#). Here is a simple example:

```
import scala.io.*
import scala.util.*
val result =
  for
    a <- Try { StdIn.readLine("a: ").toInt }
    b <- Try { StdIn.readLine("b: ").toInt }
  yield a / b
```

If an exception occurs in either of the calls to `toInt`, or because of division by zero, then `result` is a `Failure` object, containing the exception that caused the computation to fail. Otherwise, `result` is a `Success` object holding the result of the computation.

Note

Scala does not have an analog to the Java `try-with-resources` statement. In Java, you can write

```
// Java
try (Reader in = openReader(inPath); Writer out =
```

```

openWriter(outPath)) {
    // Read from in, write to out
    process(in, out)
} // No matter what, in and out correctly closed

```

This statement calls the `close` method on all variables declared inside `try (...)`, handling all tricky cases. For example, if `in` was successfully initialized but the `newBufferedWriter` method throws an exception, `in` is closed but `out` is not.

In Scala, the `Using` helper handles this situation:

```

import scala.util._

Using.Manager { use =>
    val in = use(openReader(inPath))
    val out = use(openWriter(outPath))
    // Read from in, write to out
    process(in, out)
} // The reader and writer are closed

```

Exercises

1. What does `println(println("Hello"))` print, and why?
2. What is the value of an empty block expression `{}`? What is its type?
3. Come up with one situation where the assignment `x = y = 1` is valid in Scala. (Hint: Pick a suitable type for `x`.)
4. Write a Scala equivalent for this loop in Java/JavaScript/C++ syntax:

```
for (int i = 10; i >= 0; i--) System.out.println(i);
```

5. The *signum* of a number is 1 if the number is positive, -1 if it is negative, and 0 if it is zero. Write a function that computes this value.
6. Write a procedure `countdown(n: Int)` that prints the numbers from `n` to 0.

7. Write a `for` loop for computing the product of the Unicode codes of all letters in a string. For example, the product of the characters in `"Hello"` is `9415087488L`.

8. Solve the preceding exercise without writing a loop. (Hint: Look at the `StringOps` Scaladoc.)

9. Write a function `product(s: String)` that computes the product, as described in the preceding exercises.

10. Make the function of the preceding exercise a recursive function.

11. Write a function that computes x^n , where n is an integer. Use the following recursive definition:

- $x^n = y \cdot y$ if n is even and positive, where $y = x^{n/2}$.
- $x^n = x \cdot x^{n-1}$ if n is odd and positive.
- $x^0 = 1$.
- $x^n = 1/x^{-n}$ if n is negative.

Don't use a `return` statement.

12. Define a string interpolator `date` so that you can define a `java.time.LocalDate` as `date"$year-$month-$day"`. You need to define an “extension method”, like this:

```
extension (sc: StringContext)
  def date(args: Any*): LocalDate = ...
```

`args(i)` is the value of the i th expression. Convert each to a string and then to an integer, and pass them to the `LocalDate.of` method. If you already know some Scala, add error handling. Throw an exception if there aren't three arguments, or if they aren't integers, or if they aren't separated by dashes. (You get the strings in between the expressions as `sc.parts`.)

13. To parse a command-line argument into an arbitrary type, you need to provide a “given instance”. For example, to parse a `LocalDate`:

```
import java.time.*  
import scala.util.*  
given CommandLineParser.FromString[LocalDate] with  
  def fromString(s: String) = LocalDate.parse(s)
```

Write a Scala program that receives two dates on the command line and prints the number of days between them. Your main function should have two parameters of type `LocalDate`.

Chapter 3

Working with Arrays

Topics in This Chapter A1

- [3.1 Fixed-Length Arrays](#)
- [3.2 Variable-Length Arrays: Array Buffers](#)
- [3.3 Traversing Arrays and Array Buffers](#)
- [3.4 Transforming Arrays](#)
- [3.5 Common Algorithms](#)
- [3.6 Deciphering Scaladoc](#)
- [3.7 Multidimensional Arrays](#)
- [3.8 Interoperating with Java](#)
- [Exercises](#)

In this chapter, you will learn how to work with arrays in Scala. Java, JavaScript, Python, and C++ programmers usually choose an array or its close relation (such as lists or vectors) when they need to collect a bunch of elements. In Scala, there are other choices (see [Chapter 13](#)), but for now, I'll assume you are impatient and just want to get going with arrays.

Key points of this chapter:

- Use an `Array` if the length is fixed, and an `ArrayBuffer` if the length can vary.
- Use `()` to access elements.
- Use `for elem <- arr do ...` to traverse the elements.

- Use `for elem <- arr yield ...` to transform into a new array.
- Scala and Java arrays are interoperable; with `ArrayBuffer`, use `scala.jdk.CollectionConverters`.

3.1 Fixed-Length Arrays

If you need an array whose length doesn't change, use the `Array` type in Scala. For example,

```
val strings = Array("Hello", "World")
  // An Array[String] of length 2—the type is inferred
val moreStrings = Array.ofDim[String](5) //
  A string array with five elements, all initialized with null
val nums = Array.ofDim[Int](10)
  // An array of ten integers, all initialized with zero
```



Note

In Java, arrays are created with the `new` operator. However, in Scala 3, the use of the `new` operator is discouraged, as you will see in [Chapter 5](#). It is possible to call `new Array[Int](10)` to create an array of ten integers. But it is also a bit confusing since `Array[Int](10)` (without the `new`) is an array of length 1, containing the integer 10.

You use the `()` operator to access or modify the elements of an array:

```
strings(0) = "Goodbye"
// strings is now Array("Goodbye", "World")
```

Why not `[]` as in Java, JavaScript, C++, or Python? Scala takes the point of view that an array is like a function, mapping the index values to the elements.

Inside the JVM, a Scala `Array` is implemented as a Java array. The arrays in the preceding example have the type `java.lang.String[]` inside the JVM. An array of `Int`, `Double`, or another equivalent of the Java primitive types is a primitive type array. For example, `Array(2,3,5,7,11)` is an `int[]` in the JVM.

3.2 Variable-Length Arrays: Array Buffers

Java has `ArrayList`, Python has `list`, and C++ has `vector` for arrays that grow and shrink on demand. The equivalent in Scala is the `ArrayBuffer`.

```
import scala.collection.mutable.ArrayBuffer
val b = ArrayBuffer[Int]()
    // An empty array buffer, ready to hold integers
b += 1
    // ArrayBuffer(1)
    // Add an element at the end with  +=
b ++= Array(1, 2, 3, 5, 8)
    // ArrayBuffer(1, 1, 2, 3, 5, 8)
    // You can append any collection with the  +== operator
b.dropRightInPlace(3)
    // ArrayBuffer(1, 1, 2)
    // Removes the last three elements
```

Adding or removing elements at the end of an array buffer is an efficient (“amortized constant time”) operation.

You can also insert and remove elements at an arbitrary location, but those operations are not as efficient—all elements after that location must be shifted. For example:

```
b.insert(2, 6)
    // ArrayBuffer(1, 1, 6, 2)
    // Inserts before index 2
b.insertAll(2, Array(7, 8, 9))
    // ArrayBuffer(1, 1, 7, 8, 9, 6, 2)
    // Inserts the elements from another collection
b.remove(2)
    // ArrayBuffer(1, 1, 8, 9, 6, 2)
b.remove(2, 3)
    // ArrayBuffer(1, 1, 2)
    // The second parameter tells how many elements to remove
```

Sometimes, you want to build up an `Array`, but you don’t yet know how many elements you will need. In that case, first make an array buffer, then call

```
b.toArray  
// Array(1, 1, 2)
```

Conversely, call `a.toBuffer` to convert the array `a` to an array buffer.

3.3 Traversing Arrays and Array Buffers

In Java and C++, there are several syntactical differences between arrays and array lists/vectors. Scala is much more uniform. Most of the time, you can use the same code for both.

Here is how you traverse an array or array buffer with a `for` loop:

```
for i <- 0 until a.length do  
  println(s"$i: ${a(i)}")
```

The `until` method is similar to the `to` method, except that it excludes the last value. Therefore, the variable `i` goes from `0` to `a.length - 1`.

In general, the construct

```
for i <- range do
```

makes the variable `i` traverse all values of the range. In our case, the loop variable `i` assumes the values `0, 1, ...`, and so on until (but not including) `a.length`.

To visit every second element, let `i` traverse

```
0 until a.length by 2  
// Range(0, 2, 4, ...)
```

To visit the elements starting from the end of the array, traverse

```
a.length -1 to 0 by -1  
// Range(..., 2, 1, 0)
```



Instead of `0 until a.length` or `a.length -1 to 0 by -1`, you can use `a.indices` or `a.indices.reverse`.

If you don't need the array index in the loop body, visit the array elements directly:

```
for elem <- a do
    println(elem)
```

This is similar to the “enhanced” `for` loop in Java, the “`for in`” loop in JavaScript or Python, or the “range-based” `for` loop in C++. The variable `elem` is set to `a(0)`, then `a(1)`, and so on.

3.4 Transforming Arrays

In the preceding sections, you saw how to work with arrays just like you would in other programming languages. But in Scala, you can go further. It is easy to take an array (or array buffer) and transform it in some way. Such transformations don't modify the original array but yield a new one.

Use a `for` comprehension like this:

```
val a = Array(2, 3, 5, 7, 11)
val result = for elem <- a yield 2 * elem
// result is Array(4, 6, 10, 14, 22)
```

The `for/yield` loop creates a new collection of the same type as the original collection. If you started with an array, you get another array. If you started with an array buffer, that's what you get from `for/yield`.

The result contains the expressions after the `yield`, one for each iteration of the loop.

Oftentimes, when you traverse a collection, you only want to process the elements that match a particular condition. This is achieved with a *guard*: an `if` inside the `for`. Here we double every even element, dropping the odd ones:

```
for elem <- a if elem % 2 == 0 yield elem / 2
```

Keep in mind that the result is a new collection—the original collection is not affected.



Note

Some programmers with experience in functional programming prefer `map` and `filter` to `for/yield` and guards:

```
a.filter(_ % 2 == 0).map(_ / 2)
```

That's just a matter of style—the `for/yield` loop does exactly the same work. Use whichever you find easier.

Suppose we want to remove all negative elements from an array buffer of integers. A traditional sequential solution might traverse the array buffer and remove unwanted elements as they are encountered.

```
var n = b.length
var i = 0
while i < n do
  if b(i) >= 0 then i += 1
  else
    b.remove(i)
    n -= 1
```

That's a bit fussy—you have to remember *not* to increment `i` when you remove the element, and to decrement `n` instead. It is also not efficient to remove elements from the middle of the array buffer. This loop unnecessarily moves elements that will later be removed.

In Scala, the obvious solution is to use a `for/yield` loop and keep all non-negative elements:

```
val nonNegative = for elem <- b if elem >= 0 yield elem
```

The result is a new array buffer. Suppose that we want to modify the original array buffer instead, removing the unwanted elements. Then we can collect their positions:

```
val positionsToRemove = for i <- b.indices if b(i) < 0 yield i
```

Now remove the elements at those positions, starting from the back:

```
for i <- positionsToRemove.reverse do b.remove(i)
```

Alternatively, remember the positions to keep, copy them over, and then shorten the buffer:

```
val positionsToKeep = for i <- b.indices if b(i) >= 0 yield i
for j <- positionsToKeep.indices do b(j) = b(positionsToKeep(j))
b.dropRightInPlace(b.length - positionsToKeep.length)
```

The key observation is that it is better to have *all index values together* instead of seeing them one by one.

3.5 Common Algorithms

It is often said that a large percentage of business computations are nothing but computing sums and sorting. Fortunately, Scala has built-in functions for these tasks.

```
Array(1, 7, 2, 9).sum
// 19
// Works for ArrayBuffer too
```

In order to use the `sum` method, the element type must be a numeric type: either an integral or floating-point type or `BigInteger/BigDecimal`.

Similarly, the `min` and `max` methods yield the smallest and largest element in an array or array buffer.

```
ArrayBuffer("Mary", "had", "a", "little", "lamb").max
// "little"
```

The `sorted` method sorts an array or array buffer and *returns* the sorted array or array buffer, without modifying the original:

```
val b = ArrayBuffer(1, 7, 2, 9)
val bSorted = b.sorted
// b is unchanged; bSorted is ArrayBuffer(1, 2, 7, 9)
```

You can also supply a comparison function, but then you should use the `sortWith` method:

```
val bDescending = b.sortWith(_ > _) // ArrayBuffer(9, 7, 2, 1)
```

See [Chapter 12](#) for the function syntax.

You can sort an array or array buffer in place:

```
val a = Array(1, 7, 2, 9)
a.sortInPlace()
// a is now Array(1, 2, 7, 9)
```

For the `min`, `max`, and `sortInPlace` methods, the element type must have a comparison operation. This is the case for numbers, strings, and other types with a “given” `Ordering` object.

Finally, if you want to display the contents of an array or array buffer, the `mkString` method lets you specify the separator between elements. A second variant has parameters for the prefix and suffix. For example,

```
a.mkString(" and ")
// "1 and 2 and 7 and 9"
a.mkString("<", ", ", ">")
// "<1,2,7,9>"
```

Contrast with `toString`:

```
a.toString
// "[I@b73e5"
// This is the useless toString method for arrays, straight from Java
b.toString
// "ArrayBuffer(1, 7, 2, 9)"
```

3.6 Deciphering Scaladoc

There are lots of useful methods on arrays and array buffers, and it is a good idea to browse the Scala documentation to get an idea of what’s there.

Because the `Array` class compiles directly to a Java array, most of the useful array methods are found in the `ArrayOps` and `ArraySeq` classes.

Scala has a rich type system, so you may encounter some strange-looking syntax as you browse the Scala documentation. Fortunately, you don’t have to understand all nuances of the type system to do useful work. Use [Table 3–1](#) as a “decoder ring.”

For now, you'll have to skip methods that use types such as `Option` or `PartialFunction` that we haven't discussed yet.

Table 3–1 Scaladoc Decoder Ring

Scaladoc	Explanation
<pre>def count(p: (A) => Boolean): Int</pre>	This method takes a <i>predicate</i> , a function from A to Boolean. It counts for how many elements in the collection the predicate function is true. For example, <code>a.count(_ > 0)</code> counts how many elements of <code>a</code> are positive.
<pre>def insert(@deprecatedName("n", "2.13.0") index: Int, elem: A): Unit</pre>	Ignore the <code>@deprecatedName</code> annotation. At one point the index parameter was called <code>n</code> . You don't care about it.
<pre>def appendAll(xs: IterableOnce[A]): Unit</pre>	The <code>xs</code> parameter can be any collection with the <code>IterableOnce</code> trait, a trait in the Scala collection hierarchy. Other common traits that you may encounter in Scaladoc are <code>Iterable</code> and <code>Seq</code> . All Scala collections implement these traits, and the difference between them is academic for library users. Simply think "any collection" when you see one of these.
<pre>def combinations(n: Int): Iterator[ArrayBuffer[A]]</pre>	This method computes a potentially large result and therefore returns it as an iterator instead of a collection, so that you can visit the elements one by one instead of placing them in a collection. For now, just call <code>toArray</code> or <code>toBuffer</code> on the result.
<pre>def copyToArray[B >: A] (xs: ArrayBuffer[A]): Unit</pre>	Note that the function copies an <code>ArrayBuffer[A]</code> to an <code>Array[B]</code> . Here, <code>B</code> is allowed to be a supertype of <code>A</code> . For example, you can copy from an <code>ArrayBuffer[Int]</code> to an <code>Array[Any]</code> . At first reading, just ignore the <code>[B >: A]</code> and mentally replace <code>B</code> with <code>A</code> .
<pre>def sorted[B >: A] (implicit ord: Ordering[B]): ArrayBuffer[A]</pre>	The element type <code>A</code> must have a supertype <code>B</code> which an "implicit" or "given" object of type <code>Ordering[B]</code> exists. Such an ordering exists for numbers and strings, as well as for classes that implement the Java <code>Comparable</code> interface. We will discuss the mechanism in detail in Chapter 11. With other implicit types such as <code>Numeric</code> in <code>def sorted[B >: A] (implicit num: Numeric[B]): ArrayBuffer[A]</code> , follow your intuition. To form a sum, there must be some way of adding the values.
<pre>def ++[B >: A](xs: Array[_ <: B])(implicit evidence\$23: ClassTag[B]): ArrayBuffer[B]</pre>	This method concatenates two arrays. The second array can be a subtype of the first. The "implicit" of type <code>ClassTag</code> is a formality that is required to construct the resulting array in the JVM. You can safely ignore any such <code>ClassTag</code> parameters.
<pre>def stepper[S <: Stepper[_]](implicit shape: StepperShape[A, S]): S with EfficientSplit</pre>	Here, you just have to admit defeat. This method is required for a technical reason that is only of interest to the Scala library implementors.

3.7 Multidimensional Arrays

Like in Java, JavaScript, or Python, multidimensional arrays are implemented as arrays of arrays. For example, a two-dimensional array of `Double` values has the type `Array[Array[Double]]`. To construct such an array, use the `ofDim` method:

```
val matrix = Array.ofDim[Double](3, 4) // Three rows, four columns
```

To access an element, use two pairs of parentheses:

```
matrix(row)(column) = 42
```

You can make ragged arrays, with varying row lengths:

```
val triangle = Array.ofDim[Array[Int]](10)
for i <- triangle.indices do
  triangle(i) = Array.ofDim[Int](i + 1)
```

3.8 Interoperating with Java

Since Scala arrays are implemented as Java arrays, you can pass them back and forth between Java and Scala.

This works in almost all cases, except if the array element type isn't an exact match. In Java, an array of a given type is automatically converted to an array of a supertype. For example, a Java `String[]` array can be passed to a method that expects a Java `Object[]` array. Scala does not permit this automatic conversion because it is unsafe. (See [Chapter 17](#) for a detailed explanation.)

Suppose you want to invoke a Java method with an `Object[]` parameter, such as `java.util.Arrays.binarySearch(Object[] a, Object key)`:

```
val a = Array("Mary", "a", "had", "lamb", "little")
java.util.Arrays.binarySearch(a, "beef") // Does not work
```

This does not work because Scala will not convert an `Array[String]` into an `Array[Object]`. You can force the conversion like this:

```
java.util.Arrays.binarySearch(a.asInstanceOf[Array[Object]], "beef")
```



Note

This is just an example to show how to overcome element type differences. If you want to carry out binary search in Scala, do it like this:

```
import scala.collection.Searching.*  
val result = a.search("beef")
```

The result is `Found(n)` if the element was found at position `n` or `InsertionPoint(n)` if the element was not found but should be inserted before position `n`.

If you call a Java method that receives or returns a `java.util.List`, you could, of course, use a Java `ArrayList` in your Scala code—but that is unattractive. Instead, import the conversion methods in `scala.jdk.CollectionConverters`. Then you can call the `asJava` method to convert any sequence (such as a Scala buffer) into a Java list.

For example, the `java.lang.ProcessBuilder` class has a constructor with a `List<String>` parameter. Here is how you can call it from Scala:

```
import scala.jdk.CollectionConverters.*  
import scala.collection.mutable.ArrayBuffer  
val command = ArrayBuffer("ls", "-al", "/usr/bin")  
val pb = ProcessBuilder(command.asJava) // Scala to Java
```

The Scala buffer is wrapped into an object of a Java class that implements the `java.util.List` interface.

Conversely, when a Java method returns a `java.util.List`, you can convert it into a `Buffer`:

```
import scala.jdk.CollectionConverters.*  
import scala.collection.mutable.Buffer  
val cmd: Buffer[String] = pb.command().asScala // Java to Scala  
// You can't use ArrayBuffer—the wrapped object is only guaranteed to  
be a Buffer
```

If the Java method returns a wrapped Scala buffer, then the implicit conversion unwraps the original object. In our example, `cmd == command`.

Exercises

1. Write a code snippet that sets `a` to an array of `n` random integers between `0` (inclusive) and `n` (exclusive).
2. Write a loop that swaps adjacent elements of an array of integers. For example, `Array(1, 2, 3, 4, 5)` becomes `Array(2, 1, 4, 3, 5)`.
3. Repeat the preceding assignment, but produce a new array with the swapped values. Use `for/yield`.
4. Given an array of integers, produce a new array that contains all positive values of the original array, in their original order, followed by all values that are zero or negative, in their original order.
5. How do you compute the average of an `Array[Double]`?
6. How do you rearrange the elements of an `Array[Int]` so that they appear in reverse sorted order? How do you do the same with an `ArrayBuffer[Int]`?
7. Write a code snippet that produces all values from an array with duplicates removed. (Hint: Look at Scaladoc.)
8. Suppose you are given an array buffer of integers and want to remove all but the first negative number. Here is a sequential solution that sets a flag when the first negative number is called, then removes all elements beyond.

```
var first = true
var n = a.length
var i = 0
while i < n do
  if a(i) >= 0 then i += 1
  else
    if first then
      first = false
      i += 1
    else
      a.remove(i)
      n -= 1
```

This is a complex and inefficient solution. Rewrite it in Scala by collecting positions of the negative elements, dropping the first element, reversing the sequence, and calling `a.remove(i)` for each index.

9. Improve the solution of the preceding exercise by collecting the positions that should be moved and their target positions. Make those moves and truncate the buffer. Don't copy any elements before the first unwanted element.

10. Make a collection of all time zones returned by `java.util.TimeZone.getAvailableIDs` that are in America. Strip off the "America/" prefix and sort the result.

11. Import `java.awt.datatransfer.*` and make an object of type `SystemFlavorMap` with the call

```
val flavors =  
  SystemFlavorMap.getDefaultFlavorMap().asInstanceOf[SystemFlavorMap]
```

Then call the `getNativesForFlavor` method with parameter `DataFlavor.imageFlavor` and get the return value as a Scala buffer. (Why this obscure class? It's hard to find uses of `java.util.List` in the standard Java library.)

Chapter 4

Maps, Options, and Tuples

Topics in This Chapter A1

- [4.1 Constructing a Map](#)
- [4.2 Accessing Map Values](#)
- [4.3 Updating Map Values](#)
- [4.4 Iterating over Maps](#)
- [4.5 Linked and Sorted Maps](#)
- [4.6 Interoperating with Java](#)
- [4.7 The Option Type](#)
- [4.8 Tuples](#)
- [4.9 Zipping](#)
- [Exercises](#)

A classic programmer’s saying is, “If you can only have one data structure, make it a hash table.” Hash tables—or, more generally, maps—are among the most versatile data structures. As you will see in this chapter, Scala makes it particularly easy to use them.

When looking up a key in a map, there may not be a matching value. An option is perfect for describing that situation. An option is used in Scala whenever a value may be present or absent.

Maps are collections of key/value pairs. Scala has a general notion of tuples—aggregates of n objects, not necessarily of the same type. A pair is simply a tuple with $n = 2$. Tuples are useful whenever you need to aggregate two or more values together.

Highlights of the chapter are:

- Scala has a pleasant syntax for creating, querying, and traversing maps.
- You need to choose between mutable and immutable maps.
- By default, you get a hash map, but you can also get a tree map.
- You can easily convert between Scala and Java maps.
- Use the `Option` type for values that may or may not be present—it is safer than using `null`.
- Tuples are useful for aggregating values.

4.1 Constructing a Map

You can construct a map as

```
val scores = Map("Alice" -> 10, "Bob" -> 3, "Cindy" -> 8)
```

This constructs an immutable `Map[String, Int]` whose contents can't be changed. See the next section for mutable maps.

If you want to start out with a blank map, you have to supply type parameters:

```
val scores = scala.collection.mutable.Map[String, Int]()
```

In Scala, a map is a collection of *pairs*. A pair is simply a grouping of two values, not necessarily of the same type, such as `("Alice", 10)`.

The `->` operator makes a pair. The value of

```
"Alice" -> 10
```

is

```
("Alice", 10)
```

You could have equally well defined the map as

```
Map(("Alice", 10), ("Bob", 3), ("Cindy", 8))
```

The `->` operator is just a little easier on the eyes than the parentheses. It also supports the intuition that a map data structure is a kind of function that maps keys to values. The difference is that a function computes values, and a map just looks them up.

4.2 Accessing Map Values

In Scala, the analogy between functions and maps is particularly close because you use the `()` notation to look up key values.

```
scores("Bob")
// Like scores.get("Bob") in Java or scores["Bob"] in
JavaScript, Python, or C++
```

If the map doesn't contain a value for the requested key, an exception is thrown.

To check whether there is a key with the given value, call the `contains` method:

```
if (scores.contains("Donald")) scores("Donald") else 0
```

Since this call combination is so common, there is a shortcut:

```
scores.getOrElse("Donald", 0)
// If the map contains the key "Bob", return the value; otherwise,
return 0.
```

Finally, the call `map.get(key)` returns an `Option` object that is either `Some(value for key)` or `None`. We discuss the `Option` class in [Section 4.7, “The Option Type,”](#) on page 57.

 **Note**

Given an immutable map, you can turn it into a map with a fixed default value for keys that are not present, or a function to compute such values.

```
val scores2 = scores.withDefaultValue(0)
scores2("Zelda")
    // Yields 0 since "Zelda" is not present
val scores3 = scores.withDefault(_.length)
scores3("Zelda")
    // Yields 5, applying the length function to the key that is not
present
```

4.3 Updating Map Values

In this section, we discuss mutable maps. Here is how to construct one:

```
val scores = scala.collection.mutable.Map("Alice" -> 10, "Bob" ->
3, "Cindy" -> 8)
```

In a mutable map, you can update a map value, or add a new one, with a () to the left of an = sign:

```
scores("Bob") = 10
    // Updates the existing value for the key "Bob" (assuming scores
is mutable)
scores("Fred") = 7
    // Adds a new key/value pair to scores (assuming it is mutable)
```

Alternatively, you can use the `++=` operator to add multiple associations:

```
scores += Map("Bob" -> 10, "Fred" -> 7)
```

To remove a key and its associated value, use the `--` operator:

```
scores -= "Alice"
```

You can't update an immutable map, but you can do something that's just as useful—obtain a new map that has the desired update:

```
val someScores = Map("Alice" -> 10, "Bob" -> 3)
val moreScores = someScores + ("Cindy" -> 7) // Yields a new
immutable map
```

The `moreScores` map contains the same associations as `someScores`, as well as a new association for the key "Cindy".

Instead of saving the result as a new value, you can update a `var`:

```
var currentScores = moreScores
currentScores = currentScores + ("Fred" -> 0)
```

You can even use the `+=` operator:

```
currentScores += "Donald" -> 5
```

Similarly, to remove a key from an immutable map, use the `-` operator to obtain a new map without the key:

```
currentScores = currentScores - "Alice"
```

or

```
currentScores -= "Alice"
```

You might think that it is inefficient to keep constructing new maps, but that is not the case. The old and new maps share most of their structure. (This is possible because they are immutable.)

4.4 Iterating over Maps

The following amazingly simple loop iterates over all key/value pairs of a map:

```
for ((k, v) <- map) process(k, v)
```

The magic here is that you can use pattern matching in a Scala `for` loop. ([Chapter 14](#) has all the details.) That way, you get the key and value of each pair in the map without tedious method calls.

If for some reason you want to visit only the keys or values, use the `keySet` and `values` methods. The `values` method returns an `Iterable` that you can use in a `for` loop.

```
val scores = Map("Alice" -> 10, "Bob" -> 7, "Fred" -> 8, "Cindy"  
-> 7)  
scores.keySet // Yields a set with elements "Alice", "Bob",  
"Fred", and "Cindy"  
for (v <- scores.values) println(v) // Prints 10 7 8 7
```



Note

For a mutable map, the collections returned by `keySet` and `values` are “live”—they are updated as the contents of the map changes.

To invert a map—that is, switch keys and values—use

```
for ((k, v) <- map) yield (v, k)
```

4.5 Linked and Sorted Maps

There are two common implementation strategies for maps: hash tables and binary trees. Hash tables use the hash codes of the keys to scramble entries. Tree maps use the sort order of the keys to build a balanced tree. By default, Scala gives you a map based on a hash table because it is usually more efficient.

Immutable hash maps are traversed in insertion order. This is achieved by additional links in the hash table. For example, when iterating over

```
Map("Fred" -> 1, "Alice" -> 2, "Bob" -> 3)
```

the keys are visited in the order "Fred", "Alice", "Bob", no matter what the hash codes of these strings are.

However, in a mutable map, insertion order is not maintained. Iterating over the elements yields them in unpredictable order, depending on the hash codes of the keys.

```
scala.collection.mutable.Map("Fred" -> 1, "Alice" -> 2, "Bob" -> 3)  
// Printed as HashMap(Fred -> 1, Bob -> 3, Alice -> 2)
```

If you want to visit the keys in insertion order, use a `LinkedHashMap`:

```
scala.collection.mutable.LinkedHashMap("Fred" -> 1, "Alice" -> 2,  
"Bob" -> 3)  
// Printed as LinkedHashMap(Fred -> 1, Alice -> 2, Bob -> 3)
```

To visit the keys in sorted order, use a `SortedMap` instead.

```
scala.collection.SortedMap("Fred" -> 1, "Alice" -> 2, "Bob" -> 3)  
scala.collection.mutable.SortedMap("Fred" -> 1, "Alice" -> 2,  
"Bob" -> 3)  
// Printed as TreeMap(Alice -> 2, Bob -> 3, Fred -> 1)
```

4.6 Interoperating with Java

If you get a Java map from calling a Java method, you may want to convert it to a Scala map so that you can use the pleasant Scala map API.

Simply add an `import` statement:

```
import scala.jdk.CollectionConverters.*
```

Then use the `asScala` method to turn the Java map into a Scala map:

```
val ids = java.time.ZoneId.SHORT_IDS.asScala
// Yields a scala.collection.mutable.Map[String, String]
```

In addition, you can get a conversion from `java.util.Properties` to a `Map[String, String]`:

```
val props = System.getProperties.asScala
// Yields a Map[String, String], not a Map[Object, Object]
```

Conversely, to pass a Scala map to a method that expects a Java map, provide the opposite conversion. For example:

```
import java.awt.font.TextAttribute.* // Import keys for map below
val attrs = Map(FAMILY -> "Serif", SIZE -> 12) // A Scala map
val font = java.awt.Font(attrs.asJava) // Expects a Java map
```

4.7 The `option` Type

The `Option` class in the standard library expresses values that might or might not be present. The subclass `Some` wraps a value. The object `None` indicates that there is no value.

```
var friend: Option[String] = Some("Fred")
friend = None // No friend
```

This is less ambiguous than using an empty string and safer than using `null` for a missing value.

`Option` is a generic type. For example, `Some("Fred")` is an `Option[String]`.

The `get` method of the `Map` class returns an `Option`. If there is no value for a given key, `get` returns `None`. Otherwise, it wraps the value inside `Some`.

```
val scores = Map("Alice" -> 10, "Bob" -> 7, "Cindy" -> 8)
val alicesScore = scores.get("Alice") // Some(10)
```

```
val dansScore = scores.get("Dan") // None
```

To find out what is inside an `Option` instance, you can use the `isEmpty` and `get` methods:

```
if alicesScore.isEmpty  
  then println("No score")  
  else println(alicesScore.get)
```

That's tedious, though. It is better to use the `getOrElse` method:

```
println(alicesScore.getOrElse("No score"))
```

If `alicesScore` is `None`, then `getOrElse` returns "No score".



Note

The argument of the `getOrElse` method is executed *lazily*. For example, in the call

```
alicesScore.getOrElse(System.getProperty("DEFAULT_SCORE"))
```

the call to `System.getProperty` only happens when `alicesScore` is empty.

This delayed evaluation is achieved with a “by name” parameter. See [Chapter 12](#) for details.

A more powerful way of working with options is to consider them as collections that have zero or one element. You can visit the element with a `for` loop:

```
for score <- alicesScore do println(score)
```

If `alicesScore` is `None`, nothing happens. If it is a `Some`, then the loop executes once, with `score` bound to the contents of the option.

You can also use methods such as `map`, `filter`, or `foreach`. For example,

```
val biggerScore = alicesScore.map(_ + 1) // Some(score + 1) or  
None  
val acceptableScore = alicesScore.filter(_ > 5) // Some(score) if  
score > 5 or None  
alicesScore.foreach(println) // Prints the score if it exists
```

Tip

When you create an `Option` from a value that may be `null`, you can simply use `Option(value)`. The result is `None` if `value` is `null` and `Some(value)` otherwise.

4.8 Tuples

Maps are collections of key/value pairs. Pairs are the simplest case of *tuples*—aggregates of values of different types.

A tuple value is formed by enclosing individual values in parentheses. For example,

```
(1, 3.14, "Fred")
```

is a tuple of type

```
(Int, Double, String)
```

If you have a tuple, say,

```
val t = (1, 3.14, "Fred")
```

then you can access its components as `t(0)`, `t(1)`, and `t(2)`.

As long as the index value is an integer constant, the types of these expressions are the component types:

```
val second = t(1) // second has type Double
```

 **Note**

Alternatively, you can access components with the methods `_1`, `_2`, `_3`, for example:

```
val third = t._3 // Sets third to "Fred"
```

Note that these component accessors start with `_1`. There is no `_0`.

If the index is variable, the type of `t(n)` is the common supertype of the elements:

```
var n = 1
val component = t(n) // component has type Any
```

 **Caution**

If the tuple index is a `val`, you get a complex “match type” that is no more useful than `Any`:

```
val m = 0
val first = t(m) /* first has type
  m.type match {
    case 0 => Int
    case scala.compiletime.ops.int.S[n1] =>
      scala.Tuple.Elem[(Double, String), n1]
  } */
```

Often, it is simplest to use the “destructuring” syntax to get at the components of a tuple:

```
val (first, second, third) = t // Sets first to 1, second to
  3.14, third to "Fred"
```

You can use a `_` if you don't need all components:

```
val (first, second, _) = t
```

You can concatenate tuples with the `++` operator:

```
("x", 3) ++ ("y", 4) // Yields ("x", 3, "y", 4)
```

Tuples are useful for functions that return more than one value. For example, the `partition` method of the `StringOps` class returns a pair of strings, containing the characters that fulfill a condition and those that don't:

```
"New York".partition(_.isUpper) // Yields the pair ("NY", "ew  
ork")
```



Caution

Sometimes, the Scala compiler wants to “help out” to convert between tuples and function argument lists. This can lead to surprises. Here is a typical example:

```
val destination = "world"  
val b = StringBuilder()  
b.append("Hello") // b holds "Hello"  
b.append(" ", destination) // Oops, b holds "Hello( ,world)"
```

There is no `append` method with multiple arguments. Therefore, Scala turns the arguments into a tuple and passes that as the single argument of type `Any`.

This “auto tupling” behavior can be surprising, and it may be restricted or removed in future versions of Scala.

4.9 Zipping

One reason for using tuples is to bundle together values so that they can be processed together. This is commonly done with the `zip` method. For example, the code

```
val symbols = Array("<", "-", ">")  
val counts = Array(2, 10, 2)  
val pairs = symbols.zip(counts)
```

yields an array of pairs

```
Array(("<", 2), ("-", 10), (">", 2))
```

The pairs can then be processed together:

```
for ((s, n) <- pairs) print(s * n) // Prints <<----->>
```



The `toMap` method turns a collection of pairs into a map.

If you have a collection of keys and a parallel collection of values, zip them up and turn them into a map like this:

```
keys.zip(values).toMap
```

Exercises

1. Set up a map of prices for a number of gizmos that you covet. Then produce a second map with the same keys and the prices at a 10 percent discount.
2. Write a program that reads words from a file. Use a mutable map to count how often each word appears. To read the words, simply use a `java.util.Scanner`:

```
val in = java.util.Scanner(new java.io.File("myfile.txt"))
while (in.hasNext()) process in.next()
```

Or look at [Chapter 9](#) for a Scalaesque way.

At the end, print out all words and their counts.

3. Repeat the preceding exercise with an immutable map.
4. Repeat the preceding exercise with a sorted map, so that the words are printed in sorted order.
5. Repeat the preceding exercise with a `java.util.TreeMap` that you adapt to the Scala API.
6. Define a linked hash map that maps "Monday" to `java.util.Calendar.MONDAY`, and similarly for the other weekdays. Demonstrate that the elements are visited in insertion order.
7. Print a table of all Java properties reported by the `getProperties` method of the `java.lang.System` class, like this:

<code>java.runtime.name</code>	Java (TM) SE Runtime Environment
<code>sun.boot.library.path</code>	
/home/apps/jdk1.6.0_21/jre/lib/i386	
<code>java.vm.version</code>	17.0-b16
<code>java.vm.vendor</code>	Sun Microsystems Inc.
<code>java.vendor.url</code>	http://java.sun.com/
<code>path.separator</code>	:
<code>java.vm.name</code>	Java HotSpot (TM) Server VM

You need to find the length of the longest key before you can print the table.

8. Write a function `minmax(values: Array[Int])` that returns a pair containing the smallest and the largest values in the (nonempty) array.
9. Reimplement the function from the preceding exercise to return an `Option` that is `None` if the array happens to be empty.
10. Write a program that prompts the user for a first and last letter and then prints a matching word from

`scala.io.Source.fromFile("/usr/share/dict/words").mkString.split("\n")`. Use `find`. What alternatives do you have for dealing with the returned `Option`?

11. Write a program that demonstrates that the argument of the `getOrElse` method in the `Option` class is evaluated lazily.
12. Write a function `lteqgt(values: Array[Int], v: Int)` that returns a triple containing the counts of values less than `v`, equal to `v`, and greater than `v`.
13. What happens when you zip together two strings, such as `"Hello".zip("World")`? Come up with a plausible use case.

Chapter 5

Classes

Topics in This Chapter A1

- [5.1 Simple Classes and Parameterless Methods](#)
- [5.2 Properties with Getters and Setters](#)
- [5.3 Properties with Only Getters](#)
- [5.4 Private Fields](#)
- [5.5 Auxiliary Constructors](#)
- [5.6 The Primary Constructor](#)
- [5.7 Nested Classes L1](#)
- [Exercises](#)

In this chapter, you will learn how to implement classes in Scala. If you know classes in Java, Python, or C++, you won't find this difficult, and you will enjoy the much more concise notation of Scala.

The key points of this chapter are:

- An accessor method with no parameters does not need parentheses.
- Fields in classes automatically come with getters and setters.
- You can replace a field with a custom getter/setter without changing the client of a class—that is the “uniform access principle.”
- Every class has a primary constructor that is “interwoven” with the class definition. Its parameters turn into the fields of the class. The primary constructor executes all statements in the body of the class.

- Auxiliary constructors are optional. They are called `this`.
- Classes can be nested. Each outer class object has its own version of the inner class.

5.1 Simple Classes and Parameterless Methods

In its simplest form, a Scala class looks very much like its equivalent in Java or C++:

```
class Counter :  
    private var value = 0 // You must initialize the field  
    def increment() = // Methods are public by default  
        value += 1  
    def current = value
```

Note the colon after the class name. For greater clarity, I always add a space before such a colon, but that is not a requirement.



Note

If the name of a class consists of symbolic characters, enclose them in backticks to avoid a compiler warning:

```
class `` :  
    def now() = java.time.LocalTime.now()  
    ...
```

When you use the class, no backticks are necessary: `() .now()`.

You can also use the traditional syntax with braces:

```
class Counter {  
    private var value = 0  
    def increment() = {  
        value += 1  
    }
```

```
    def current = value  
}
```

Note

You can initialize a field with the special value `uninitialized`, defined in the `scala.compiletime` package. This causes initialization in the Java virtual machine with the default value (zero, `false`, or `null`). For clarity and brevity, I will always provide an explicit initial value.

In Scala, a class is not declared as `public`. A Scala source file can contain multiple top-level classes, and all of them have public visibility unless they are explicitly declared `private`.

To use this class, you construct objects and invoke methods in the usual way:

```
val myCounter = Counter() // Or new Counter()  
myCounter.increment()  
println(myCounter.current)
```

Note that the `increment` method has been defined with an empty parameter list, and `current` has been defined without parentheses. The methods must be called in the way that they were declared.

Which form should you use? It is considered good style to declare parameterless methods that change the object state with `()`. Parameterless methods that do not change the object state should be declared without `()`. Some programmers refer to the former as *mutator* methods, and the latter as *accessor* methods.

That's what we did in our example.

```
myCounter.increment() // A mutator method  
println(myCounter.current) // An accessor method
```

5.2 Properties with Getters and Setters

When writing a Java class, we don't like to use public fields:

```
public class Person { // This is Java
    public int age; // Frowned upon in Java
}
```

With a public field, anyone could write to `fred.age`, making Fred younger or older. That's why we prefer to use getter and setter methods:

```
public class Person { // This is Java
    private int age;
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
}
```

A getter/setter pair such as this one is often called a *property*. We say that the class `Person` has an `age` property.

Why is this any better? By itself, it isn't. Anyone can call `fred.setAge(21)`, keeping him forever twenty-one.

But if that becomes a problem, we can guard against it:

```
public void setAge(int newValue) { // This is Java
    if (newValue > age) age = newValue;
    // Can't get younger
}
```

Getters and setters are better than public fields because they let you start with simple get/set semantics and evolve them as needed.



Just because getters and setters are better than public fields doesn't mean they are always good. Often, it is plainly bad if every client can get or set bits and pieces of an object's state. In this section, I show you how to implement properties in Scala. It is up to you to choose wisely when a gettable/settable property is an appropriate design.

Scala provides getter and setter methods for every public field. Consider this example:

```
class Person :  
    var age = 0
```

Scala generates a class for the JVM with a *private* `age` field and getter and public setter methods.

In Scala, the getter and setter methods are called `age` and `age_=`. For example,

```
val fred = Person()  
fred.age = 21 // Calls fred.age_=(21)  
println(fred.age) // Calls the method fred.age()
```

In Scala, the getters and setters are not named `getXxx` and `setXxx`, but they fulfill the same purpose. If you need `getXxx` and `setXxx` methods for interoperability with Java, use the `@BeanProperty` annotation. See [Chapter 15](#) for details.



Note

To see these methods with your own eyes, compile the file containing the `Person` class and then look at the bytecode with `javap`:

```
$ scala3-compiler Person.scala  
$ javap -private Person  
Compiled from "Person.scala"  
public class Person {  
    private int age;  
    public Person();  
    public int age();  
    public void age_$eq(int);  
}
```

As you can see, the compiler created methods `age` and `age_$eq`. (The `=` symbol is translated to `$eq` because the JVM does not allow an `=` in a method name.)

You can redefine the getter and setter methods yourself. For example,

```
class Person :  
    private var privateAge = 0 // Make private and rename
```

```
def age = privateAge
def age_=(newValue: Int) =
    if newValue > privateAge then privateAge = newValue // Can't get
young
```

The user of your class still accesses `fred.age`, but now Fred can't get younger:

```
val fred = Person()
fred.age = 30
fred.age = 21
println(fred.age) // Prints 30
```



Bertrand Meyer, the inventor of the influential Eiffel language, formulated the *Uniform Access Principle* that states: “All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation.” In Scala, the caller of `fred.age` doesn’t know whether `age` is implemented through a field or a method. (Of course, in the JVM, the service is *always* implemented through a method, either synthesized or programmer-supplied.)

5.3 Properties with Only Getters

Sometimes you want a *read-only property* with a getter but no setter. If the value of the property never changes after the object has been constructed, use a `val` field:

```
class Message :
    val timeStamp = java.time.Instant.now
    ...
```

The Scala compiler produces a Java class with a private `final` field and a public getter method, but no setter.

Sometimes, however, you want a property that a client can't set at will, but that is mutated in some other way. The `Counter` class from [Section 5.1, “Simple Classes and Parameterless Methods,”](#) on page 65 is a good example. Conceptually, the counter has a `current` property that is updated when the `increment` method is called, but there is no setter for the property.

You can't implement such a property with a `val`—a `val` never changes. Instead, provide a private field and a property getter. That is what you saw in the `Counter` example:

```
class Counter :  
  private var value = 0  
  def increment() = value += 1  
  def current = value // No () in declaration
```

Note that there are no `()` in the definition of the getter method. Therefore, you *must* call the method without parentheses:

```
val n = myCounter.current // Calling myCounter.current() is a syntax  
error
```

To summarize, you have four choices for implementing properties:

1. `var foo`: Scala synthesizes a getter and a setter.
2. `val foo`: Scala synthesizes a getter.
3. You define methods `foo` and `foo_=`.
4. You define a method `foo`.



In some programming languages, you can declare *write-only* properties. Such a property has a setter but no getter. That is not possible in Scala.



When you see a public field in a Scala class, remember that it is not the same as a field in Java or C++. It is a private field *together with*

a public getter (for a `val` field) or a getter and a setter (for a `var` field).

5.4 Private Fields

If you define a private field, Scala does not normally produce getters and setters. But there are two exceptions.

A method can access the private fields of *all* objects of its class. For example,

```
class Counter :  
    private var value = 0  
    def increment() = value += 1  
    def isLess(other: Counter) = value < other.value  
    // Can access private field of other object
```

Accessing `other.value` is legal because `other` is also a `Counter` object.

When a private field is accessed through an object other than `this`, then private getters and setters are generated.

Moreover, there is a (rarely used) syntax for granting access rights to specific classes or packages. The `private[name]` qualifier states that only methods of the given enclosing class or package can access the given field.

The compiler will generate auxiliary getter and setter methods that allow the enclosing class or package to access the field. These methods will be public because the JVM does not have a fine-grained access control system, and they will have implementation-dependent names.

[Table 5–1](#) lists all field declarations that we discussed, and shows which methods are generated.

Table 5–1 Generated Methods for Fields

Scala Field	Generated Methods	When to Use
val/var name	public name name_= (var only)	To implement a property that is publicly accessible and backed by a field.
@BeanProperty val/var name	public name getName() name_= (var only) setName(...) (var only)	To interoperate with JavaBeans
private val/var name	private getters/setters generated if needed	To confine the field to the methods of this class, just like Java. Use private unless you really want a public property.
private[name] val/var name	implementation-dependent	To grant access to an enclosing class or package. Not commonly used.

5.5 Auxiliary Constructors

A Scala class can have as many constructors as you like. However, one constructor is more important than all the others, called the *primary constructor*. In addition, a class may have any number of *auxiliary constructors*.

We discuss auxiliary constructors first because they are similar to constructors in Java, C++, or Python.

The auxiliary constructors are called `this`. (In Java or C++, constructors have the same name as the class—which is not so convenient if you rename the class.)

Each auxiliary constructor *must* start with a call to another auxiliary constructor or the primary constructor.

Here is a class with two auxiliary constructors:

```
class Person :
    private var name = ""
    private var age = 0

    def this(name: String) = // An auxiliary constructor
```

```

this() // Calls primary constructor
this.name = name

def this(name: String, age: Int) = // Another auxiliary constructor
    this(name) // Calls previous auxiliary constructor
    this.age = age

```

We will look at the primary constructor in the next section. For now, it is sufficient to know that a class for which you don't define a primary constructor has a primary constructor with no arguments.

You can construct objects of this class in three ways:

```

val p1 = Person() // Primary constructor
val p2 = Person("Fred") // First auxiliary constructor
val p3 = Person("Fred", 42) // Second auxiliary constructor

```

You can optionally use the `new` keyword to invoke a constructor, but you don't have to.

5.6 The Primary Constructor

In Scala, every class has a primary constructor. The primary constructor is not defined with a `this` method. Instead, it is interwoven with the class definition.

The parameters of the primary constructor are placed *immediately after the class name*.

```

class Person(val name: String, val age: Int) :
    // Parameters of primary constructor in parentheses after class name
    ...

```

If a primary constructor parameter is declared with `val` or `var`, it turns into a field that is initialized with the construction parameter. In our example, `name` and `age` become fields of the `Person` class. A constructor call such as `Person("Fred", 42)` sets the `name` and `age` fields.

Half a line of Scala is the equivalent of seven lines of Java:

```

public class Person { // This is Java
    private String name;

```

```

private int age;
public Person(String name, int age) {
    this.name = name;
    this.age = age;
}
public String name() { return this.name; }
public int age() { return this.age; }
...
}

```

The primary constructor executes *all statements in the class definition*. For example, in the following class

```

class Person(val name: String, val age: Int) :
    println("Just constructed another person")
    def description = s"$name is $age years old"

```

the `println` statement is a part of the primary constructor. It is executed whenever an object is constructed.

This is useful when you need to configure a field during construction. For example:

```

class MyProg :
    private val props = java.util.Properties()
    props.load(new FileReader("myprog.properties"))
    // The statement above is a part of the primary constructor
    ...

```



If there are no parameters after the class name, then the class has a primary constructor with no parameters. That constructor simply executes all statements in the body of the class.



Tip

You can often eliminate auxiliary constructors by using default arguments in the primary constructor. For example:

```
class Person(val name: String = "", val age: Int = 0) :  
    ...
```

Primary constructor parameters can have any of the forms in [Table 5–1](#). For example,

```
class Person(val name: String, private var age: Int)
```

declares and initializes fields

```
val name: String  
private var age: Int
```

Construction parameters can also be regular method parameters, without `val` or `var`. They don't give rise to getters. How these parameters are processed depends on their usage inside the class.

```
class Person(firstName: String, lastName: String, age: Int) :  
    val name = firstName + " " + lastName  
    def description = s"$name is $age years old"
```

Since the `age` parameter is used inside a method, it becomes an object-private field.

However, the `firstName` and `lastName` parameters are not used in any method. After the primary constructor has completed, they are no longer required. Therefore, there is no need to turn them into fields

[Table 5–2](#) summarizes the fields and methods that are generated for different kinds of primary constructor parameters.

Table 5–2 Fields and Methods Generated for Primary Constructor Parameters

Primary Constructor Parameter	Generated Field/Methods
name: String	object-private field, or no field if no method name
private val/var name: String	private field, private getter/setter
val/var name: String	private field, public getter/setter
@BeanProperty val/var name: String	private field, public Scala and JavaBeans getters/setters

If you find the primary constructor notation confusing, you don't need to use it. Just provide one or more auxiliary constructors in the usual way, but remember to call `this()` if you don't chain to another auxiliary constructor.

However, many programmers like the concise syntax. Martin Odersky suggests to think about it this way: In Scala, classes take parameters, just like methods do.

Note

When you think of the primary constructor's parameters as class parameters, parameters without `val` or `var` become easier to understand. The scope of such a parameter is the entire class. Therefore, you can use the parameter in methods. If you do, it is the compiler's job to save it in a field.

Tip

The Scala designers think that *every keystroke is precious*, so they let you combine a class with its primary constructor. When reading a Scala class, you need to disentangle the two. For example, when you see

```
class Person(val name: String) :
  var age = 0
  def description = s"$name is $age years old"
```

mentally take this definition apart into a class definition:

```
class Person(val name: String) :  
    var age = 0  
    def description = s"$name is $age years old"
```

and a constructor definition:

```
class Person(val name: String) :  
    var age = 0  
    def description = s"$name is $age years old"
```



To make the primary constructor private, place the keyword `private` like this:

```
class Person private(val id: Int) :  
    ...
```

A class user must then use an auxiliary constructor to construct a `Person` object.

5.7 Nested Classes L1

In Scala, you can nest just about anything inside anything. You can define functions inside other functions, and classes inside other classes. Here is a simple example of the latter:

```
class Network :  
    class Member(val name: String) :  
        val contacts = ArrayBuffer[Member] ()  
  
    private val members = ArrayBuffer[Member] ()  
  
    def join(name: String) =  
        val m = Member(name)
```

```
members += m  
m
```

Consider two networks:

```
val chatter = Network()  
val myFace = Network()
```

In Scala, each *instance* has its own class `Member`, just like each instance has its own field `members`. That is, `chatter.Member` and `myFace.Member` are *different classes*. To construct a new inner object, you simply use the type name: `chatter.Member("Fred")`.



Note

This is different from Java, where an inner class belongs to the outer class.

In Java, the objects `chatter.new Member("Fred")` and `myFace.new Member("Wilma")` are both instances of a single class `Network$Member`.

In our network example, you can add a member within its own network:

```
val fred = chatter.join("Fred")  
val wilma = chatter.join("Wilma")  
fred.contacts += wilma // OK  
val barney = myFace.join("Barney") // Has type myFace.Member
```

However, trying to add across networks is a compile-time error:

```
fred.contacts += barney  
// No—can't add a myFace.Member to a buffer of chatter.Member  
elements
```

For networks of people, this behavior can be plausible. And it is certainly interesting to have types that depend on objects.

If you want to express the type “`Member` of *any* `Network`”, use a *type projection* `Network#Member`. For example,

```
class Network :  
    class Member(val name: String) :  
        val contacts = ArrayBuffer[Network#Member] ()  
    ...
```

The type `Network#Member` corresponds to an inner class in Java.

But then again, perhaps you just want to nest the classes to limit their scope, as one commonly does with static inner classes in Java or nested classes in C++. Then you can move the `Member` class to the `Network` companion object. (Companion objects are described in [Chapter 6](#).)

```
object Network :  
    class Member(val name: String) :  
        val contacts = ArrayBuffer[Member] ()  
  
class Network :  
    private val members = ArrayBuffer[Network.Member] ()  
    ...
```



Note

In a method of a nested class, you can access the `this` reference of the enclosing class as `EnclosingClass.this`:

```
class Network(val name: String) :  
    class Member(val name: String) :  
        val contacts = ArrayBuffer[Member] ()  
        def description = s"$name inside ${Network.this.name}"
```

Exercises

1. Improve the `Counter` class in [Section 5.1, “Simple Classes and Parameterless Methods,”](#) on page 65 so that it doesn’t turn negative at `Int.MaxValue`.
2. Write a class `BankAccount` with methods `deposit` and `withdraw`, and a read-only property `balance`.

3. Write a class `Time` with read-only properties `hours` and `minutes` and a method `before(other: Time): Boolean` that checks whether this time comes before the other. A `Time` object should be constructed as `Time(hrs, min)`, where `hrs` is in military time format (between 0 and 23).
4. Reimplement the `Time` class from the preceding exercise so that the internal representation is the number of minutes since midnight (between 0 and $24 \times 60 - 1$). *Do not* change the public interface. That is, client code should be unaffected by your change.
5. In the `Person` class of [Section 5.2, “Properties with Getters and Setters,”](#) on page 67, provide a primary constructor that turns negative ages to 0.
6. Write a class `Person` with a primary constructor that accepts a string containing a first name, a space, and a last name, such as `Person("Fred Smith")`. Supply read-only properties `firstName` and `lastName`. Should the primary constructor parameter be a `var`, a `val`, or a plain parameter? Why?
7. Make a class `Car` with read-only properties for manufacturer, model name, and model year, and a read-write property for the license plate. Supply four constructors. All require the manufacturer and model name. Optionally, model year and license plate can also be specified in the constructor. If not, the model year is set to `-1` and the license plate to the empty string. Which constructor are you choosing as the primary constructor? Why?
8. Reimplement the class of the preceding exercise in Java, JavaScript, Python, C#, or C++ (your choice). How much shorter is the Scala class?

9. Consider the class

```
class Employee(val name: String, var salary: Double) {  
    def this() { this("John Q. Public", 0.0) }  
}
```

Rewrite it to use explicit fields and a default primary constructor. Which form do you prefer? Why?

10. Implement the `equals` method for the `Member` class that is nested inside the `Network` class in [Section 5.7, “Nested Classes,”](#) on page 75. For two members to be equal, they need to be in the same network.

Chapter 6

Objects and Enumerations

Topics in This Chapter A1

- [6.1 Singletons](#)
- [6.2 Companion Objects](#)
- [6.3 Objects Extending a Class or Trait](#)
- [6.4 The `apply` Method](#)
- [6.5 Application Objects](#)
- [6.6 Enumerations](#)
- [Exercises](#)

In this short chapter, you will learn when to use the `object` construct in Scala. Use it when you need a class with a single instance, or when you want to find a home for miscellaneous values or functions. This chapter also covers enumerated types.

The key points of this chapter are:

- Use objects for singletons and utility methods.
- A class can have a companion object with the same name.
- Objects can extend classes or traits.

- The `apply` method of an object is usually used for constructing new instances of the companion class.
- If you don't provide your own `apply` method in the companion object, an `apply` method is provided for all constructors of a class.
- Instead of a `@main` annotated method, you can provide a `main(Array[String]): Unit` method in an object as a program starting point.
- The `enum` construct defines an enumeration. Enumeration objects can have state and methods.

6.1 Singletons

Scala has no static methods or fields. Instead, you use the `object` construct. An object defines a single instance of a class with the features that you want. For example,

```
object Accounts {
    private var lastNumber = 0
    def newUniqueNumber() =
        lastNumber += 1
        lastNumber
}
```

When you need a new unique account number in your application, call `Accounts.newUniqueNumber()`.

The constructor of an object is executed when the object is first used. In our example, the `Accounts` constructor is executed with the first call to `Accounts.newUniqueNumber()`. If an object is never used, its constructor is not executed.

An object can have essentially all the features of a class—it can even extend other classes or traits (see [Section 6.3, “Objects Extending a Class or Trait,”](#) on page 83). There is just one exception: You cannot provide constructor parameters.

Use an object in Scala whenever you would have used a singleton object in another programming language:

- As a home for utility functions or constants
 - When a single immutable instance can be shared efficiently
 - When a single instance is required to coordinate some service (the singleton design pattern)
-



Note

Many people view the singleton design pattern with disdain. Scala gives you the tools for both good and bad design, and it is up to you to use them wisely.

6.2 Companion Objects

In Java, JavaScript, or C++, you often have a class with both instance methods and static methods. In Scala, you can achieve this by having a class and a “companion” object of the same name. For example,

```
class Account :  
    val id = Account.newUniqueNumber()  
    private var balance = 0.0  
    def deposit(amount: Double) =  
        balance += amount  
    ...  
  
object Account : // The companion object  
    private var lastNumber = 0  
    private def newUniqueNumber() =  
        lastNumber += 1  
        lastNumber
```

The class and its companion object can access each other’s private features. They must be located in the *same source file*.

Note that the companion object's features are not in the scope of the class. For example, the `Account` class has to use `Account.newUniqueNumber()` and not just `newUniqueNumber()` to invoke the method of the companion object.

Tip

To define a companion object in the REPL, write both the class and the object in a text editor, and then paste both of them together into the REPL. If you define them separately, you will get an error message.

6.3 Objects Extending a Class or Trait

An `object` can extend a class and/or one or more traits. The result is an object of a class that extends the given class and/or traits, and in addition has all of the features specified in the object definition.

One useful application is to specify default objects that can be shared. For example, consider a class for undoable actions in a program.

```
abstract class UndoableAction(val description: String) :  
    def undo(): Unit  
    def redo(): Unit
```

A useful default is the “do nothing” action. Of course, we only need one of them.

```
object DoNothingAction extends UndoableAction("Do nothing") :  
    override def undo() = ()  
    override def redo() = ()
```

The `DoNothingAction` object can be shared across all places that need this default.

```
val actions = Map("open" -> DoNothingAction, "save" ->  
DoNothingAction)
```

```
// Open and save not yet implemented
```

6.4 The `apply` Method

The `apply` method is called for expressions of the form

Object(arg₁, ..., arg_N)

Typically, such an `apply` method returns an object of the companion class.

For example, the `Array` object defines `apply` methods that allow array creation with expressions such as

```
Array("Mary", "had", "a", "little", "lamb")
```

This call actually means:

```
Array.apply("Mary", "had", "a", "little", "lamb")
```

Whenever you define a Scala class, a companion object is automatically provided, with an `apply` method for every constructor. This is what enables construction without using the `new` operator.

For example, given the class

```
class Person(val name: String, val age: Int)
```

there is automatically a companion object `Person` and a method `Person.apply` so that you can call

```
val p = Person("Fred", 42)
val q = Person.apply("Wilma", 39)
```

These methods are called *constructor proxy methods*.

There is only one exception. If the class already has a companion object with at least one `apply` method, then no constructor proxy methods are provided.

You might want to declare your own `apply` method if you don't want to invoke a fixed constructor. The most common reason is to produce instances of a sub-type. For example, `Map.apply` produces maps of different classes:

```
val seasons = Map("Spring" -> 1, "Summer" -> 2, "Fall" -> 3,  
"Winter" -> 4)  
seasons.getClass // Yields class  
scala.collection.immutable.Map$Map4 val directions =  
  Map("Center" -> 0, "North" -> 1, "East" -> 2, "South" -> 3,  
"West" -> 4)  
directions.getClass // Yields class  
scala.collection.immutable.HashMap
```

6.5 Application Objects

If an object declares a method with name `main` and type `Array[String] => Unit`, then you can invoke that method from the command line. Consider the classic example, a file `Hello.scala` with this contents:

```
object Hello :  
  def main(args: Array[String]) =  
    println(s"Hello, ${args.mkString(" ")}!")
```

Now you can compile and run the class. The command line arguments are placed in the `args` parameter.

```
$ scalac Hello.scala  
$ scala Hello cruel world  
Hello, cruel world!
```

This mechanism is a little different than the one you saw in [Chapter 2](#), where the program entry point was a function with the `@main` annotation. Consider a file `Greeter.scala` with this contents:

```
@main def hello(args: String*) =  
  println(s"Hello, ${args.mkString(" ")}!")
```

The annotation produces an object named `hello` with a `public static void main(Array[String])` method. That method parses the command line arguments and calls the `hello` method, which is located in another class `Hello\$package\$`.

To invoke the program, you run the `hello` class:

```
$ scalac Greeter.scala  
$ scala hello cruel world  
Hello, cruel world!
```

6.6 Enumerations

An object has a single instance. An enumerated type has a finite number of instances:

```
enum TrafficLightColor :  
  case Red, Yellow, Green
```

Here we define a type `TrafficLightColor` with three instances, `TrafficLightColor.Red`, `TrafficLightColor.Yellow`, and `TrafficLightColor.Green`.

Every `enum` automatically has an `ordinal` method that yields an integer for each instance, numbering them in the order in which they were defined, starting at 0.

```
TrafficLightColor.Yellow.ordinal // Yields 1
```

The inverse is the `fromOrdinal` method of the companion object:

```
TrafficLightColor.fromOrdinal(1) // Yields  
TrafficLightColor.Yellow
```

The `values` method of the companion object yields an array of all enumerated values in the same order:

```
TrafficLightColor.values // Yields Array(Red, Yellow, Green)
```

The companion object has a `valueOf` method that obtains an instance by name:

```
TrafficLightColor.valueOf("Yellow")
```

This is the inverse of the `toString` method, which maps instances to their names:

```
TrafficLightColor.Yellow.toString // Yields "Yellow"
```

You can define your own methods:

```
enum TrafficLightColor :  
    case Red, Yellow, Green  
    def next = TrafficLightColor.fromOrdinal((ordinal + 2) % 3)
```

You can also add methods to the companion object:

```
object TrafficLightColor :  
    def random() =  
        TrafficLightColor.fromOrdinal(scala.util.Random.nextInt(3))
```

Instances can have state. Provide a constructor with the enumeration type, and then construct instances as follows:

```
enum TrafficLightColor(val description: String) :  
    case Red extends TrafficLightColor("Stop")  
    case Yellow extends TrafficLightColor("Hurry up")  
    case Green extends TrafficLightColor("Go")
```

The `extends` syntax reflects the fact that the enumeration instances are objects extending the enumeration type—see [Section 6.3, “Objects Extending a Class or Trait,”](#) on page 83.

You can deprecate an instance:

```
enum TrafficLightColor :  
    case Red, Yellow, Green  
    @deprecated("""https://99percentinvisible.org/article/stop-  
at-red-go-on-grue-\  
how-language-turned-traffic-lights-bleen-in-japan/""") case Blue
```

If you need to make an enumerated type compatible with Java enumerations, extend the `java.lang.Enum` class, as follows:

```
enum TrafficLightColor extends Enum[TrafficLightColor] :  
    case Red, Yellow, Green
```



In [Chapter 14](#), you will see “parameterized” `enum` that define class hierarchies, such as:

```
enum Amount :  
    case Nothing  
    case Dollar(value: Double)  
    case Currency(value: Double, unit: String)
```

This is equivalent to an abstract class `Amount` that is extended by an object `Nothing` and classes `Dollar` and `Currency`.

The `enum` types covered in this chapter are a special case, with only objects.

Exercises

1. Write an object `Conversions` with methods `inchesToCentimeters`, `gallonsToLiters`, and `milesToKilometers`.

2. The preceding problem wasn't very object-oriented. Provide a general superclass `UnitConversion` and define objects `InchesToCentimeters`, `GallonsToLiters`, and `MilesToKilometers` that extend it.
3. Define an `Origin` object that extends `java.awt.Point`. Why is this not actually a good idea? (Have a close look at the methods of the `Point` class.)
4. Define a `Point` class with a companion object so that you can construct `Point` instances as `Point(3, 4)`, without using `new`.
5. Write a Scala application, using the `App` trait, that prints its command-line arguments in reverse order, separated by spaces. For example, `scala Reverse Hello World` should print `World Hello`.
6. Write an enumeration describing the four playing card suits so that the `toString` method returns `, , ,` or `.`
7. Implement a function that checks whether a card suit value from the preceding exercise is red.
8. Write an enumeration describing the eight corners of the RGB color cube. As IDs, use the color values (for example, `0xff0000` for `Red`).

Chapter 7

Packages, Imports, and Exports

Topics in This Chapter A1

- [7.1 Packages](#)
- [7.2 Package Scope Nesting](#)
- [7.3 Chained Package Clauses](#)
- [7.4 Top-of-File Notation](#)
- [7.5 Package-Level Functions and Variables](#)
- [7.6 Package Visibility](#)
- [7.7 Imports](#)
- [7.8 Imports Can Be Anywhere](#)
- [7.9 Renaming and Hiding Members](#)
- [7.10 Implicit Imports](#)
- [7.11 Exports](#)
- [Exercises](#)

A package contains features with a common purpose. Large programs and libraries are typically organized into a multitude of packages.

Package names are unique but long. Using import statements, you can access the features of a package with a shorter name.

This chapter ends with a discussion of the `export` syntax, which is syntactically similar to import statements, but is used for delegation.

The key points of this chapter are:

- Packages nest just like inner classes.
- Package paths are *not* absolute.
- A chain `x.y.z` in a package clause leaves the intermediate packages `x` and `x.y` invisible.
- Package statements without braces at the top of the file extend to the entire file.
- A package can have functions and variables.
- Import statements can import packages, classes, and objects.
- Import statements can be anywhere.
- Import statements can rename and hide members.
- `java.lang`, `scala`, and `Predef` are always imported.
- Export statements provide a concise mechanism for declaring delegations, using the same syntax as import statements.

7.1 Packages

Packages in Scala fulfill the same purpose as packages in Java, modules in Python, or namespaces in C++: to manage names in a large program. For example, the name `Map` can occur in the packages `scala.collection.immutable` and `scala.collection.mutable` without conflict. To access either name, you can use the fully qualified `scala.collection.immutable.Map` or `scala.collection.mutable.Map`. Alternatively, use an `import` statement to provide a shorter alias—see [Section 7.7, “Imports,” on page 94](#).

To add items to a package, you can include them in nested package statements, such as:

```
package com :  
    package horstmann :  
        package people :  
            class Employee(name: String, var salary: Double) :  
                ...
```

Then the class name `Employee` can be accessed anywhere as `com.horstmann.people.Employee`.

Unlike an object or a class, a package can be defined in multiple files. The preceding code might be in a file `Employee.scala`, and a file `Manager.scala` might define another class `Manager` in the `com.horstmann.people` package.

Note

There is no enforced relationship between the directory of the source file and the package. You don't have to put `Employee.scala` and `Manager.scala` into a `com/horstmann/people` directory.

Conversely, you can contribute to more than one package in a single file. The file `People.scala` may contain

```
package com :  
    package horstmann :  
        package people :  
            class Person(val name: String) :  
                ...  
  
        package users :  
            class User(val username: String, password: String) :  
                ...
```

7.2 Package Scope Nesting

Scala packages nest just like all other scopes. You can access names from the enclosing scope. For example,

```
package com :  
  package horstmann :  
    package people :  
      class Employee(name: String, var salary: Double) :  
        ...  
        def giveRaise(rate: Double) =  
          salary += Math.percentOf(salary, rate)  
      object Math :  
        def percentOf(value: Double, rate: Double) = value * rate  
          / 100  
          ...
```

Note the `Math.percentOf` qualifier. The `Math` class was defined in the *parent* package. Everything in the parent package is in scope, and it is not necessary to use `com.horstmann.Math.percentOf`. (You could, though, if you prefer—after all, `com` is also in scope.)

This is more regular than Java, where packages cannot be nested.

There is a fly in the ointment, however. Some programmers like to take advantage of the fact that the `scala` package is always imported (see [Section 7.10, “Implicit Imports,”](#) on page 95). Therefore, it is possible to use `collection` instead of `scala.collection`:

```
package com :  
  package horstmann :  
    package people :  
      class Manager(name: String) :  
        val subordinates =  
          collection.mutable.ArrayBuffer[Employee]()  
          ...
```

However, suppose someone introduces a `com.horstmann.collection` package, perhaps later, and in a different file.

Then the `Manager` class no longer compiles. It looks for a `mutable` member inside the `com.horstmann.collection` package and doesn't find it. The `Manager` class intends the `collection` package in the top-level `scala` package, not whatever `collection` subpackage happened to be in some accessible scope.

In Java, this problem can't occur because package names are always *absolute*, starting at the root of the package hierarchy. But in Scala, package names are relative, just like inner class names. With inner classes, one doesn't usually run into problems because all the code is in one file, under control of whoever is in charge of that file. But packages are open-ended. Anyone can contribute to a package by adding a new file with the same package declaration.

The safest solution is to use absolute package names. They must start with the special identifier `_root_`, such as:

```
val subordinates =  
  _root_.scala.collection.mutable.ArrayBuffer[Employee]()
```



Most Scala programmers use complete paths for package names, without the `_root_` prefix. This is safe as long as everyone avoids names `scala`, `java`, `javax`, and top-level domains (`com`, `net`, and so on), for nested packages.

7.3 Chained Package Clauses

A package clause can contain a “chain,” or path segment, for example:

```
package com :  
  package horstmann.people :
```

```
// Members of com.horstmann are not visible here
class Executive :
    val subordinates =
        collection.mutable.ArrayBuffer[Manager]()
```

Such a clause limits the visible members. Now the `com.horstmann.collection` package is no longer accessible as `collection`.

7.4 Top-of-File Notation

Instead of the nested notation that we have used up to now, you can have a `package` clause at the top of the file, without a colon or braces. For example:

```
package com.horstmann.collection // No colon

class Group :
    ...
```

Everything in this file is inside the `com.horstmann.collection` package. This is the exact equivalent of the Java `package` statement.



The top-of-file notation is commonly used whenever all the code in the file belongs to the same package. It saves you from having to indent the package contents.

7.5 Package-Level Functions and Variables

You have seen that a package can contain classes. A package can also contain other type declarations such as traits, enumerations, and objects. For example, the following is valid:

```
package com.horstmann.people

val defaultName = "John Q. Public"
def employ(p: Person) = Employee(p.name, 0)
```

Behind the scenes, the package object gets compiled into a Java virtual machine class with static methods and fields, called *filename\$package.class*, inside the given package. If the example code is stored in `People.scala`, that would be a class `com.horstmann.people.People$package`.

Normally, the names of the Scala source files have no influence on the class files that the compiler generates. Each class, enumeration, object, or trait is compiled into a Java class file with the given package and class name. There may be helper classes (such as the classes for companion objects), but their names are derived from the name of the main feature.

But in the case of package-level functions and variables, the Scala compiler must construct a Java class to hold them. There can be multiple files with top-level definitions for the same package, and it would be too complex to merge them all into a single class file. Therefore, each source file gives rise to a separate class file. And those class files contain the source file name.

If you rename a source file, the name of the class file holding the package-level definitions changes, and you must recompile all files depending on them.

Therefore, it is a good idea to keep all top-level definitions of a package in a single file with a fixed name. The traditional choice is `package.scala`, placed in the subdirectory that matches the package name, such as `com/horstmann/people/package.scala`.

7.6 Package Visibility

In Java, a class member that isn't declared as `public`, `private`, or `protected` is visible in the package containing the class. In Scala, you can achieve the same effect with qualifiers enclosed in brackets. The following method is visible in its own package:

```
package com.horstmann.users

class User(username: String, password: String) :
    private[users] def longDescription = s"A user with name
$username and password $password"
    ...

```

You can extend the visibility to an enclosing package:

```
private[horstmann] def safeDescription = s"A user with name
$username and password ${"**" * password}
```

In the package `com.horstmann.users`, both methods are accessible. However, in the package `com.horstmann`, the first method is inaccessible.

7.7 Imports

Imports let you use short names instead of long ones. With the clause

```
import java.awt.Color
```

you can write `color` in your code instead of `java.awt.Color`.

That is the sole purpose of imports. If you don't mind long names, you'll never need them.

You can import all members of a package as

```
import java.awt.*
```



Caution

In Scala, `*` is a valid character for an identifier. You could define a package `com.horstmann.*.people`. Please don't. If someone else did, use `'*'` to import it.

You can also import all members of a class or object.

```
import java.awt.Color.*  
val c1 = RED // Color.RED  
val c2 = decode("#ff0000") // Color.decode
```

This is like `import static` in Java. Java programmers seem to live in fear of this variant, but in Scala it is commonly used.

Once you import a package, you can access its subpackages with shorter names. For example:

```
import java.awt.*  
  
val transform = geom.AffineTransform.getScaleInstance(0.5, 0.5)  
// java.awt.geom.AffineTransform
```

The `geom` package is a member of `java.awt`, and the import brings it into scope.

7.8 Imports Can Be Anywhere

In Scala, an `import` statement can be anywhere, not just at the top of a file. The scope of the `import` statement extends until the end of the enclosing block. For example,

```
class Group :  
    // Imports restricted to this class  
    import scala.collection.mutable.*  
    import com.horstmann.users.*  
    val members = ArrayBuffer[User]()  
    ...
```

Imports with restricted scope are very useful, particularly with wildcard imports. It is always a bit worrisome to import lots of names from different sources. In fact, some programmers dislike wildcard imports so much that

they never use them but let their IDE generate long lists of imported classes.

By restricting the imports to the scopes where they are needed, you can greatly reduce the potential for conflicts.

7.9 Renaming and Hiding Members

If you want to import more than one member from a package, use the *selector* syntax like this:

```
import java.awt.{Color, Font}
```

Use the `as` keyword to rename members:

```
import java.util.HashMap as JavaHashMap
import scala.collection.mutable.*
```

Now `JavaHashMap` is a `java.util.HashMap` and `plain HashMap` is a `scala.collection.mutable.HashMap`.

The `syntaxHashMap as _` hides a member instead of renaming it. This is useful if you import other members with the same name:

```
import java.util.{HashMap as _, *}
import scala.collection.mutable.*
```

Now `HashMap` unambiguously refers to `scala.collection.mutable.HashMap` since `java.util.HashMap` is hidden.

7.10 Implicit Imports

Every Scala program implicitly starts with

```
import java.lang.*
import scala.*
import Predef.*
```

The `java.lang` package is always imported. Next, the `scala` package is imported, but in a special way. Unlike all other imports, this one is allowed to override the preceding import. For example, `scala.StringBuilder` overrides `java.lang.StringBuilder` instead of conflicting with it.

Finally, the `Predef` object is imported. It contains commonly used types, implicit conversions, and utility methods. (The methods could equally well have been placed into the `scala` package, but `Predef` was introduced long before Scala had package-level functions.)

Since the `scala` package is imported by default, you never need to write package names that start with `scala`. For example,

```
collection.mutable.HashMap
```

is just as good as

```
scala.collection.mutable.HashMap
```

7.11 Exports

This chapter ends with the `export` statement, which, somewhat similar to `import`, provides aliases for certain features. However, `import` can be used with packages, classes, or objects, whereas `export` is only used with objects.

Let's move to a concrete example. A `ColoredPoint` has a color and a point. Now we'd like to get the color components and the point coordinates. Of course, we can delegate to the color methods and point fields:

```
import java.awt.*  
  
class ColoredPoint(val color: Color, val point: Point) :  
    def red = color.getRed()  
    def green = color.getGreen()  
    def blue = color.getBlue()  
    val x = point.x  
    val y = point.y
```

That delegation can get tedious, and it sometimes encourages programmers to reach for inheritance. Then you automatically inherit everything and don't have to implement delegations.

But of course, that inheritance may not be appropriate. Is a `ColoredPoint` a special kind of `Color`? A special kind of `Point`? There are philosophical and practical reasons why the answer might be "no".

A general software engineering principle is to favor composition over inheritance. The `export` feature helps to follow this guidance. Instead of manually delegating features, you declare which ones you want:

```
class ColoredPoint(val color: Color, val point: Point) :  
    export color.{getRed as red, getGreen as green, getBlue as  
blue}  
    export point.{x, y}
```

The syntax is just like with `import` statements. You enclose the selected features in braces and use arrows for renaming.

As with imports, you can export all but a subset of features:

```
export point.{  
    setLocation as _, translate as _, toString as _, hashCode as _,  
    equals as _, clone as _, *}
```

Exercises

1. Write an example program to demonstrate that

```
package com.horstmann.impatient
```

is not the same as

```
package com  
package horstmann  
package impatient
```

2. Write a puzzler that baffles your Scala friends, using a package `com` that isn't at the top level.
3. Write a package `random` with functions `nextInt(): Int`, `nextDouble(): Double`, and `setSeed(seed: Int): Unit`. To generate random numbers, use the linear congruential generator

$$next = (previous \times a + b) \bmod 2^n,$$

where $a = 1664525$, $b = 1013904223$, $n = 32$, and the initial value of *previous* is `seed`.

4. Make two source files, each contributing two classes and a top-level function to a given package. What class files are generated? In which directories? What happens if you move the source files to different directories? What happens if you rename the source files?
5. What is the meaning of `private[com] def giveRaise(rate: Double)`? Is it useful?
6. Write a program that copies all elements from a Java hash map into a Scala hash map. Use imports to rename both classes.
7. In the preceding exercise, move all imports into the innermost scope possible.
8. What is the effect of

```
import java.*  
import javax.*
```

Is this a good idea?

9. Write a program that imports the `java.lang.System` class, reads the user name from the `user.name` system property, reads a password from the `StdIn` object, and prints a message to the standard error stream if the password is not "secret". Otherwise, print a greeting to the standard output stream. Do not use any other imports, and do not use any qualified names (with dots).
10. Apart from `StringBuilder`, what other members of `java.lang` does the `scala` package override?

11. One common example of improper inheritance is a stack class that inherits from an `ArrayBuffer`. This is bad because the stack then inherits many methods that are not permitted on stacks. Use composition and `export` statements to define a `Stack` class that stores strings.
12. Pick an example where composition is preferred over inheritance and implement it in Scala, using the `export` syntax for method delegations.
13. An `export` statement can appear inside a class, object, or trait, or at the top-level. If it appears in a class, how does it differ from an `import`? What if it appears at the top-level?

Chapter 8

Inheritance

Topics in This Chapter A1

- 8.1 Extending a Class
- 8.2 Overriding Methods
- 8.3 Type Checks and Casts
- 8.4 Superclass Construction
- 8.5 Anonymous Subclasses
- 8.6 Abstract Classes
- 8.7 Abstract Fields
- 8.8 Overriding Fields
- 8.9 Open and Sealed Classes
- 8.10 Protected Fields and Methods
- 8.11 Construction Order
- 8.12 The Scala Inheritance Hierarchy
- 8.13 Object Equality L1
- 8.14 Multiversal Equality L2
- 8.15 Value Classes L2
- Exercises

In this chapter, you will learn the most important ways in which inheritance in Scala differs from inheritance in other programming languages. (I assume that

you are familiar with the general concept of inheritance.) The highlights are:

- The `extends` keyword denotes inheritance.
- You must use `override` when you override a method.
- A `final` class cannot be extended. A `final` method cannot be overridden.
- An `open` class is explicitly designed for being extended.
- Only the primary constructor can call the primary superclass constructor.
- You can override fields.
- You can define classes whose instances can only be compared with each other, or other suitable types.
- A value class wraps a single value without the overhead of a separate object.

In this chapter, we only discuss the case in which a class inherits from another class. See [Chapter 10](#) for inheriting *traits*—the Scala concept that generalizes Java interfaces.

8.1 Extending a Class

You form a subclass in Scala with the `extends` keyword:

```
class Employee extends Person :  
    var salary = 0.0  
    ...
```

In the body of the subclass, you specify fields and methods that are new to the subclass or that override methods in the superclass.

You can declare a class `final` so that it cannot be extended. You can also declare individual methods or fields `final` so that they cannot be overridden. (See [Section 8.8, “Overriding Fields,”](#) on page 106 for overriding fields.) Note that this is different from Java, where a `final` field is immutable, similar to `val` in Scala.

8.2 Overriding Methods

In Scala, you *must* use the `override` modifier when you override a method that isn’t abstract. (See [Section 8.6, “Abstract Classes,”](#) on page 105 for abstract methods.) For example,

```
class Person :  
  ...  
  override def toString = s"${getClass.getName} [name=$name]"
```

The `override` modifier can give useful error messages in a number of common situations, such as:

- When you misspell the name of the method that you are overriding
 - When you accidentally provide a wrong parameter type in the overriding method
 - When you introduce a new method in a superclass that clashes with a subclass method
-

Note

The last case is an instance of the *fragile base class problem* where a change in the superclass cannot be verified without looking at all the subclasses. Suppose programmer Alice defines a `Person` class, and, unbeknownst to Alice, programmer Bob defines a subclass `Student` with a method `id` yielding the student ID. Later, Alice also defines a method `id` that holds the person's national ID. When Bob picks up that change, something may break in Bob's program (but not in Alice's test cases) since `Student` objects now return unexpected IDs.

In Java, one is often advised to “solve” this problem by declaring all methods as `final` unless they are explicitly designed to be overridden. That sounds good in theory, but programmers hate it when they can't make even the most innocuous changes to a method (such as adding a logging call). That's why Java eventually introduced an optional `@Overrides` annotation.

To invoke a superclass method in Scala, you use the keyword `super`:

```
class Employee extends Person :  
  ...  
  override def toString = s"${super.toString} [salary=$salary]"
```

The call `super.toString` invokes the `toString` method of the superclass—that is, the `Person.toString` method.

8.3 Type Checks and Casts

To test whether an object belongs to a given class, use the `isInstanceOf` method. If the test succeeds, you can use the `asInstanceOf` method to convert a reference to a subclass reference:

```
if p.isInstanceOf[Employee] then  
    val s = p.asInstanceOf[Employee] // s has type Employee  
    ...
```

The `p.isInstanceOf[Employee]` test succeeds if `p` refers to an object of class `Employee` or its subclass (such as `Manager`).

If `p` is `null`, then `p.isInstanceOf[Employee]` returns `false` and `p.asInstanceOf[Employee]` returns `null`.

If `p` is not an `Employee`, then `p.asInstanceOf[Employee]` throws an exception.

If you want to test whether `p` refers to an `Employee` object, but not a subclass, use

```
if p.getClass == classOf[Employee] then ...
```

The `classOf` method is defined in the `scala.Predef` object that is always imported.

[Table 8–1](#) shows the correspondence between Scala and Java type checks and casts.

Table 8–1 Type Checks and Casts in Scala and Java

Scala	Java
<code>obj.isInstanceOf[C]</code>	<code>obj instanceof C</code>
<code>obj.asInstanceOf[C]</code>	<code>(C) obj</code>
<code>classOf[C]</code>	<code>C.class</code>

However, pattern matching is usually a better alternative to using type checks and casts. For example,

```
p match
  case s: Employee => ... // Process s as an Employee
  case _ => ... // p wasn't an Employee
```

See [Chapter 14](#) for more information.

8.4 Superclass Construction

Recall from [Chapter 5](#) that a class has one primary constructor and any number of auxiliary constructors, and that all auxiliary constructors must start with a call to a preceding auxiliary constructor or the primary constructor.

As a consequence, an auxiliary constructor can *never* invoke a superclass constructor directly.

The auxiliary constructors of the subclass eventually call the primary constructor of the subclass. Only the primary constructor can call a superclass constructor.

Recall that the primary constructor is intertwined with the class definition. The call to the superclass constructor is similarly intertwined. Here is an example:

```
class Employee(name: String, age: Int, var salary : Double) extends
  Person(name, age) :
  ...
```

This defines a subclass

```
class Employee(name: String, age: Int, var salary : Double) extends
  Person(name, age)
```

and a primary constructor that calls the superclass constructor

```
class Employee(name: String, age: Int, var salary : Double) extends
  Person(name, age)
```

Intertwining the class and the constructor makes for very concise code. You may find it helpful to think of the primary constructor parameters as parameters of the class. Here, the `Employee` class has three parameters: `name`, `age`, and `salary`, two of which it “passes” to the superclass.

In Java, the equivalent code is quite a bit more verbose:

```
public class Employee extends Person { // Java
    private double salary;
    public Employee(String name, int age, double salary) {
        super(name, age);
        this.salary = salary;
    }
}
```

Note

In a Scala constructor, you can never call `super(params)`, as you would in Java, to call the superclass constructor.

A Scala class can extend a Java class. Its primary constructor must invoke one of the constructors of the Java superclass. For example,

```
class ModernPrintWriter(p: Path, cs: Charset = StandardCharsets.UTF_8)
extends
  java.io.PrintWriter(Files.newBufferedWriter(p, cs))
```

8.5 Anonymous Subclasses

You construct an instance of an *anonymous* subclass by providing construction arguments and a block with overrides. For example, suppose we have the following superclass:

```
class Person(val name: String) :
  def greeting = s"Hello, my name is $name"
```

Here, we construct an object that belongs to an anonymous subclass of `Person`, overriding the `greeting` method:

```
val alien = new Person("Tweel") :
  override def greeting = "Greetings, Earthling!"
```



Caution

The `new` keyword is required to construct an instance of an anonymous class.

8.6 Abstract Classes

A class declared as `abstract` cannot be instantiated. This is usually done because one or more of its methods are not defined. For example,

```
abstract class Person(val name: String) :  
    def id: Int // No method body—this is an abstract method
```

Here we say that every person has an ID, but we don't know how to compute it. Each concrete subclass of `Person` needs to specify an `id` method. Note that you do not use the `abstract` keyword for an abstract method. You simply omit its body. A class with at least one abstract method *must* be declared `abstract`.

In a subclass, you need not use the `override` keyword when you define a method that was abstract in the superclass.

```
class Employee(name: String) extends Person(name) :  
    def id = name.hashCode // override keyword not required
```

8.7 Abstract Fields

In addition to abstract methods, a class can also have abstract fields. An abstract field is simply a field without an initial value. For example,

```
abstract class Person :  
    val id: Int  
        // No initializer—this is an abstract field with an abstract getter method  
    var name: String  
        // Another abstract field, with abstract getter and setter methods
```

This class defines abstract getter methods for the `id` and `name` fields, and an abstract setter for the `name` field. The generated Java class has *no fields*.

Concrete subclasses must provide concrete fields, for example:

```

class Employee(val id: Int) extends Person : // Subclass has concrete
  id property
    var name = "" // and concrete name property

```

As with methods, no `override` keyword is required in the subclass when you define a field that was abstract in the superclass.

You can always customize an abstract field by using an anonymous type:

```

val fred = new Person() {
  val id = 1729
  var name = "Fred"
}

```

8.8 Overriding Fields

Recall from [Chapter 5](#) that a field in Scala consists of a private field *and* accessor/mutator methods. You can override a `val` (or a parameterless `def`) with another `val` field of the same name. The subclass has a private field and a public getter, and the getter overrides the superclass getter (or method).

For example,

```

class Person(val name: String) :
  override def toString = s"${getClass.getName} [name=$name]"

class SecretAgent(codename: String) extends Person(codename) :
  override val name = "secret" // Don't want to reveal name...
  override val toString = "secret" // ... or class name

```

This example shows the mechanics, but it is rather artificial. A more common case is to override an abstract `def` with a `val`, like this:

```

abstract class User :
  def id: Int // Each user has an ID that is computed in some way

class Student(override val id: Int) extends User
  // A student ID is simply provided in the constructor

```

Note the following restrictions (see also [Table 8–2](#)):

- A `def` can only override another `def`.
- A `val` can only override another `val` or a parameterless `def`.
- A `var` can only override an abstract `var`.

Table 8–2 Overriding `val`, `def`, and `var`

	with <code>val</code>	with <code>def</code>	with <code>var</code>
Override <code>val</code>	<ul style="list-style-type: none"> • Subclass has a private field (with the same name as the superclass field—that’s OK). • Getter overrides the superclass getter. 	Error.	Error.
Override <code>def</code>	<ul style="list-style-type: none"> • Subclass has a private field. • Getter overrides the superclass method. 	Like in Java.	A <code>var</code> can override getter/setter pair. Overriding just a getter is an error.
Override <code>var</code>	Error.	Error.	Only if the super <code>var</code> is abstract.



Caution

In [Chapter 5](#), I said that it’s OK to use a `var` because you can always change your mind and reimplement it as a getter/setter pair. However, the programmers extending your class do not have that choice. They cannot override a `var` with a getter/setter pair. In other words, if you provide a `var`, all subclasses are stuck with it.

8.9 Open and Sealed Classes

Inheritance is a powerful feature, and it can be abused. Sometimes, programmers extend a class just because it is convenient to pick up some methods. [Chapter 7](#) discussed the `exports` feature that helps Scala programmers avoid this trap by making it easy to use composition and delegation instead of inheritance.

However, many classes are explicitly designed for inheritance. In Scala, you use the `open` keyword to mark those classes:

```
open class Person :
```

```
...
```

Starting with Scala 3.1, there will be a warning if you extend a non-`open` class outside the source file in which it is declared. To avoid the warning, the file containing the extending class must contain the following import statement:

```
import scala.language.adhocExtensions
```

This mechanism is a very modest nudge away from inheritance overuse.

Within a single file, there are no restrictions. Presumably the author of the file understands the superclass well enough to extend it.

If you extend a non-`open` class from another file and get a compiler warning, consider whether inheritance is appropriate in your situation. If you decide that it is, include the `adhocExtensions` import. It will signal to others that you either made a conscious choice, or you just wanted to shut up the compiler.



Note

It is a syntax error to declare a `final` class as `open`. Conversely, an `abstract` class ([Section 8.6, “Abstract Classes,” on page 105](#)) is automatically `open`.

A related concept is that of a `sealed` class. A sealed class has a fixed number of direct subclasses. They must all be declared in the same file.

```
// PostalRates.scala
sealed abstract class PostalRate :
    ...
class DomesticRate extends PostalRate :
    ...
class InternationalRate extends PostalRate :
    ...
```

Trying to extend a sealed class in another file is an error, not a warning.

Sealed classes are intended for pattern matching. If the compiler knows all subclasses of a class, it can verify that a match against subclasses is exhaustive. See [Chapter 14](#) for the details.

8.10 Protected Fields and Methods

As in Java or C++, you can declare a field or method as `protected`. Such a member is accessible from any subclass.

```
class Employee(name: String, age: Int, protected var salary: Double) :  
    ...  
  
class Manager(name: String, age: Int, salary: Double)  
    extends Employee(name, age, salary) :  
        def setSalary(newSalary: Double) = // A manager's salary can never  
        decrease  
            if (newSalary > salary) salary = newSalary  
  
        def outranks(other: Manager) =  
            salary > other.salary
```

Note that a `Manager` method can access the `salary` field in any `Manager` object.

Unlike in Java, a `protected` member is *not* visible throughout the package to which the class belongs. (If you want this visibility, you can use a package modifier—see [Chapter 7](#).)

8.11 Construction Order

When you override a `val` in a subclass *and* use the value in a superclass constructor, the resulting behavior is unintuitive.

Here is an example. A creature can sense a part of its environment. For simplicity, we assume the creature lives in a one-dimensional world, and the sensory data are represented as integers. A default creature can see ten units ahead.

```
class Creature :  
    val range: Int = 10
```

```
val env: Array[Int] = Array[Int](range)
```

Ants, however, are near-sighted:

```
class Ant extends Creature :  
    override val range = 2
```

Unfortunately, we now have a problem. The `range` value is used in the superclass constructor, and the superclass constructor runs *before* the subclass constructor. Specifically, here is what happens:

1. The `Ant` constructor calls the `Creature` constructor before doing its own construction.
2. The `Creature` constructor sets *its* `range` field to `10`.
3. The `Creature` constructor, in order to initialize the `env` array, calls the `range()` getter.
4. That method is overridden to yield the (as yet uninitialized) `range` field of the `Ant` class.
5. The `range` method returns `0`. (That is the initial value of all integer fields when an object is allocated.)
6. `env` is set to an array of length `0`.
7. The `Ant` constructor continues, setting its `range` field to `2`.

Even though it appears as if `range` is either `10` or `2`, `env` has been set to an array of length `0`. The moral is that you should not rely on the value of a `val` in the body of a constructor.

As a remedy, you can make `range` into a `lazy val` (see [Chapter 2](#)).



Note

At the root of the construction order problem lies a design decision of the Java language—namely, to allow the invocation of subclass methods in a superclass constructor. In C++, an object's virtual function table pointer is set to the table of the superclass when the superclass constructor executes. Afterwards, the pointer is set to the subclass table. Therefore, in C++, it is not possible to modify constructor behavior through overriding. The Java designers felt that this subtlety was

unnecessary, and the Java virtual machine does not adjust the virtual function table during construction.



Tip

You can have the compiler check for initialization errors such as the one in this section by compiling with the experimental `-Ysafe-init` flag.

8.12 The Scala Inheritance Hierarchy

Figure 8–1 shows the inheritance hierarchy of Scala classes. The classes that correspond to the primitive types in the Java virtual machine, as well as the type `Unit`, extend `AnyVal`. You can also define your own *value classes*—see [Section 8.15, “Value Classes,”](#) on page 115.

All other classes are subclasses of the `AnyRef` class. When compiling to the Java virtual machine, this is a synonym for the `java.lang.Object` class.

Both `AnyVal` and `AnyRef` extend the `Any` class, the root of the hierarchy.

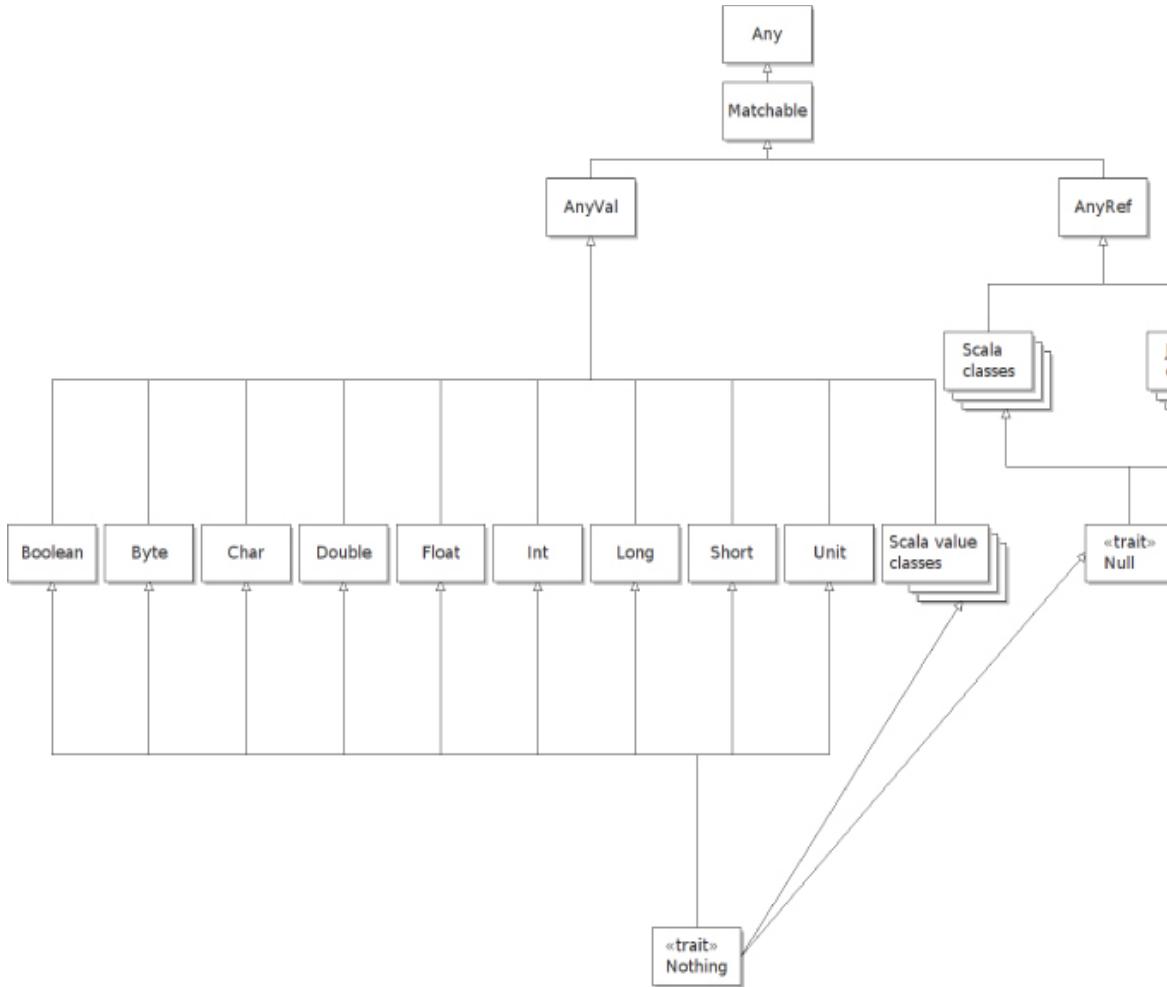


Figure 8–1 The inheritance hierarchy of Scala classes

The `Any` class defines methods `isInstanceOf`, `asInstanceOf`, and the methods for equality and hash codes that we will look at in [Section 8.13, “Object Equality,”](#) on page 113.

`AnyVal` does not add any methods. It is just a marker for value types.

The `AnyRef` class adds the monitor methods `wait` and `notify/notifyAll` from the `object` class. It also provides a `synchronized` method with a function parameter. That method is the equivalent of a `synchronized` block in Java. For example,

```
account.synchronized { account.balance += amount }
```



Note

Just like in Java, I suggest you stay away from `wait`, `notify`, and `synchronized` unless you have a good reason to use them instead of higher-level concurrency constructs.

At the other end of the hierarchy are the `Nothing` and `Null` types.

`Null` is the type whose sole instance is the value `null`. You can assign `null` to any reference, but not to one of the value types. For example, setting an `Int` to `null` is not possible. This is better than in Java, where it would be possible to set an `Integer` wrapper to `null`.

Note

With the experimental “explicit nulls” feature, the type hierarchy changes, and `Null` is no longer a subtype of the types extending `AnyRef`. Object references cannot be `Null`. If you want nullable values, you need to declare a union type `T | Null`. To opt in to this feature, compile with the `-Yexplicit-nulls` flag.

The `Nothing` type has no instances. It is occasionally useful for generic constructs. For example, the empty list `Nil` has type `List[Nothing]`, which is a subtype of `List[T]` for any `T`.

The `???` method is declared with return type `Nothing`. It never returns but instead throws a `NotImplementedError` when invoked. You can use it for methods that you still need to implement:

```
class Person(val name: String) :  
    def description: String = ???
```

The `Person` class compiles since `Nothing` is a subtype of every type. You can start using the class, so long as you don’t call the `description` method.

The `Nothing` type is not at all the same as `void` in Java or C++. In Scala, `void` is represented by the `Unit` type, the type with the sole value `()`.

Caution

`Unit` is not a supertype of any other type. However, a value of any type can be *replaced* by a `()`. Consider this example:

```
def showAny(o: Any) = println(s"${o.getClass.getName}: $o")
def showUnit(o: Unit) = println(s"${o.getClass.getName}: $o")
showAny("Hello") // Yields "java.lang.String: Hello"
showUnit("Hello") // Yields "void: ()"
// "Hello" is replaced with () (with a warning)
```



Caution

When a method has a parameter of type `Any` or `AnyRef`, and it is called with multiple arguments, then the arguments are placed in a tuple:

```
showAny(3) // Prints class java.lang.Integer: 3
showAny(3, 4, 5) // Prints class scala.Tuple3: (3,4,5)
```

8.13 Object Equality L1

You can override the `equals` method to provide a natural notion of equality for the objects of your classes.

Consider the class

```
class Item(val description: String, val price: Double) :
  ...
```

You might want to consider two items equal if they have the same description and price. Here is an appropriate `equals` method:

```
final override def equals(other: Any) =
  other.isInstanceOf[Item] && {
    val that = other.asInstanceOf[Item]
    description == that.description && price == that.price
  }
```

Or better, use pattern matching:

```
final override def equals(other: Any) = other match
  case that: Item => description == that.description && price ==
    that.price
  case _ => false
```

Tip

Generally, it is very difficult to correctly extend equality in a subclass. The problem is symmetry. You want `a.equals(b)` to have the same result as `b.equals(a)`, even when `b` belongs to a subclass. Therefore, an `equals` method should usually be declared as `final`.

Caution

Be sure to define the `equals` method with parameter type `Any`. The following would be wrong:

```
final def equals(other: Item) = ... // Don't!
```

This is a different method which does not override the `equals` method of `Any`.

When you define `equals`, remember to define `hashCode` as well. The hash code should be computed only from the fields that you use in the equality check, so that equal objects have the same hash code. In the `Item` example, combine the hash codes of the fields.

```
final override def hashCode = (description, price).##
```

The `##` method is a null-safe version of the `hashCode` method that yields `0` for `null` instead of throwing an exception.

Caution

Do not supply your own `==` method instead of `equals`. You can't override the `==` method defined in `Any`, but you can supply a different one with an

`Item` argument:

```
final def ==(other: Item) = // Don't supply == instead of equals!
  description == other.description && price == other.price
```

This method will be invoked when you compare two `Item` objects with `==`. But other classes, such as `scala.collection.mutable.HashSet`, use `equals` to compare elements, for compatibility with Java objects. Your `==` method will not get called!



Tip

You are not compelled to override `equals` and `hashCode`. For many classes, it is appropriate to consider distinct objects unequal. For example, if you have two distinct input streams or radio buttons, you will never consider them equal.

Unlike in Java, don't call the `equals` method directly. Simply use the `==` operator. For reference types, it calls `equals` after doing the appropriate check for `null` operands.

8.14 Multiversal Equality L2

In Java, the `equals` method is *universal*. It can be invoked to compare objects of any class.

That sounds nice, but it limits the ability of the compiler to find errors. In Scala, the `==` operator can be made to fail at compile-time when its arguments could not possibly be comparable.

There are two mechanisms for opting into this *multiversal* equality.

If you want to ensure that all equality checks use multiversal equality, add the import:

```
import scala.language.strictEquality
```

Alternatively, you can activate multiversal equality selectively. A class can declare that it wants to disallow comparison with instances of other classes:

```
class Item(val description: String, val price: Double) derives
CanEqual :
  ...
  ...
```

The `derives` keyword is a general mechanism that you will see in [Chapter 20](#).

Now it is impossible to compare `Item` instances with unrelated objects:

```
Item("Blackwell toaster", 29.95) == Product("Blackwell toaster") //  
Compile-time error
```

Even with multiversal equality, you can implement equality checks between different classes. This is not something that you should attempt in general, but it happens with a number of classes in the Scala library. You can check for equality between any two sequences or sets (subtypes of `scala.collection.Seq` and `scala.collection.Set`), provided the element types can also be equal.

You can test for equality between any numeric primitive types, or between any primitive types and their wrapper types. Finally, any reference type can be compared for equality with `null`.

For backwards compatibility, mixed equality checks are still permitted when neither operand opts in to multiversal equality.

8.15 Value Classes L2

Some classes have a single field, such as the wrapper classes for primitive types, and the “rich” or “ops” wrappers that Scala uses to add methods to existing types. It is inefficient to allocate a new object that holds just one value. *Value classes* allow you to define classes that are “inlined,” so that the single field is used directly.

A value class has these properties:

1. The class extends `AnyVal`.
2. Its primary constructor has exactly one parameter, which is a `val`, and no body.
3. The class has no other fields or constructors.
4. The automatically provided `equals` and `hashCode` methods compare and hash the underlying value.

As an example, let us define a value class that wraps a “military time” value:

```
class MilTime(val time: Int) extends AnyVal :  
    def minutes = time % 100  
    def hours = time / 100  
    override def toString = f"$time%04d"
```

When you construct a `MilTime(1230)`, the compiler doesn't allocate a new object. Instead, it uses the underlying value, the integer 1230. You can invoke the `minutes` and `hours` methods on the value:

```
val lunch = MilTime(1230)  
println(lunch.hours) // OK
```

Just as importantly, you cannot invoke `Int` methods:

```
println(lunch * 2) // Error
```

To guarantee proper initialization, make the primary constructor private and provide a factory method in the companion object:

```
class MilTime private(val time: Int) extends AnyVal :  
    ...  
object MilTime :  
    def apply(t: Int) =  
        if 0 <= t && t < 2400 && t % 100 < 60 then MilTime(t)  
        else throw IllegalArgumentException()
```



Caution

In some programming languages, value types are any types that are allocated on the runtime stack, including structured types with multiple fields. In Scala, a value class can only have one field.



Note

If you want a value class to implement a trait (see [Chapter 10](#)), the trait must explicitly extend `Any`, and it may not have fields. Such traits are called *universal traits*.

Opaque types are an alternative to value types. An opaque type alias lets you give a name to an existing type, so that the original type cannot be used. For example, `MilTime` can be an opaque type alias to `Int`. In [Chapter 18](#), you will see how to declare opaque types and define their behavior with “extension methods”.

In the not too distant future, the Java virtual machine will support native value types that can have multiple fields. It is expected that Scala value will evolve to align with JVM value types.

Exercises

1. Extend the following `BankAccount` class to a `CheckingAccount` class that charges \$1 for every deposit and withdrawal.

```
class BankAccount(initialBalance: Double) :  
    private var balance = initialBalance  
    def currentBalance = balance  
    def deposit(amount: Double) = { balance += amount; balance }  
    def withdraw(amount: Double) = { balance -= amount; balance }
```

2. Extend the `BankAccount` class of the preceding exercise into a class `SavingsAccount` that earns interest every month (when a method `earnMonthlyInterest` is called) and has three free deposits or withdrawals every month. Reset the transaction count in the `earnMonthlyInterest` method.
3. Consult your favorite Java, Python, or C++ textbook which is sure to have an example of a toy inheritance hierarchy, perhaps involving employees, pets, graphical shapes, or the like. Implement the example in Scala.
4. Define an abstract class `Item` with methods `price` and `description`. A `SimpleItem` is an item whose price and description are specified in the constructor. Take advantage of the fact that a `val` can override a `def`. A `Bundle` is an item that contains other items. Its price is the sum of the prices in the bundle. Also provide a mechanism for adding items to the bundle and a suitable `description` method.
5. Design a class `Point` whose `x` and `y` coordinate values can be provided in a constructor. Provide a subclass `LabeledPoint` whose constructor takes a label value and `x` and `y` coordinates, such as

```
LabeledPoint("Black Thursday", 1929, 230.07)
```

6. Define an abstract class `Shape` with an abstract method `centerPoint` and subclasses `Rectangle` and `Circle`. Provide appropriate constructors for the subclasses and override the `centerPoint` method in each subclass.
7. Provide a class `Square` that extends `java.awt.Rectangle` and has three constructors: one that constructs a square with a given corner point and width, one that constructs a square with corner `(0, 0)` and a given width, and one that constructs a square with corner `(0, 0)` and width 0.
8. Compile the `Person` and `SecretAgent` classes in [Section 8.8, “Overriding Fields,”](#) on page 106 and analyze the class files with `javap`. How many `name` fields are there? How many `name` getter methods are there? What do they get? (Hint: Use the `-c` and `-private` options.)
9. In the `Creature` class of [Section 8.11, “Construction Order,”](#) on page 109, replace `val range` with a `def`. What happens when you also use a `def` in the `Ant` subclass? What happens when you use a `val` in the subclass? Why?
10. The file `scala/collection/immutable/Stack.scala` contains the definition

```
class Stack[A] protected (protected val elems: List[A])
```

Explain the meanings of the `protected` keywords. (Hint: Review the discussion of private constructors in [Chapter 5.](#))

11. Write a class `Circle` with a center and a radius. Add `equals` and `hashCode` methods. Activate multiversal equality.
12. Define a value class `Point` that packs integer `x` and `y` coordinates into a `Long` (which you should make `private`).

Chapter 9

Files and Regular Expressions

Topics in This Chapter A1

- 9.1 Reading Lines
- 9.2 Reading Characters
- 9.3 Reading Tokens and Numbers
- 9.4 Reading from URLs and Other Sources
- 9.5 Writing Files
- 9.6 Visiting Directories
- 9.7 Serialization
- 9.8 Process Control A2
- 9.9 Regular Expressions
- 9.10 Regular Expression Groups
- Exercises

This chapter interrupts the treatment of the Scala language to give you some tools from the Scala libraries. You will learn how to carry out common file processing tasks, such as reading all lines or words from a file, and to work with regular expressions.

This interlude has useful information for projects that you can embark on with your current knowledge of Scala. Of course, if you prefer, skip the chapter until you need it and move on to more information about the Scala language.

Chapter highlights:

- `Source.fromFile(...).getLines.toArray` yields all lines of a file.
- `Source.fromFile(...).mkString` yields the file contents as a string.
- To convert a string into a number, use the `toInt` or `toDouble` method.
- Use the Java `PrintWriter` to write text files.
- `"regex".r` is a `Regex` object.
- Use `"""..."""` if your regular expression contains backslashes or quotes.
- If a regex pattern has groups, you can extract their contents using the syntax for `regex(var1, ..., varn) <- string`.

9.1 Reading Lines

To read all lines from a file, call the `getLines` method on a `scala.io.Source` object:

```
import scala.io.Source
val filename = "/usr/share/dict/words"
var source = Source.fromFile(filename, "UTF-8")
// You can omit the encoding if you know that the file uses
// the default platform encoding
var lineIterator = source.getLines
```

The result is an iterator (see [Chapter 13](#)). You can use it to process the lines one at a time:

```
for l <- lineIterator do
  process(l)
```

Or you can put the lines into an array or array buffer by applying the `toArray` or `toBuffer` method to the iterator:

```
val lines = source.getLines.toArray
```

Sometimes, you just want to read an entire file into a string. That's even simpler:

```
var contents = source.mkString
```



Caution

Call `close` when you are done using the `Source` object.

When the `Source` class was created, the Java API for file processing was very limited. Java has caught up, and you may want to use the `java.nio.file.Files` class instead:

```
import java.nio.file.{Files, Path}
import java.nio.charset.StandardCharsets.UTF_8
contents = Files.readString(Path.of(filename), UTF_8)
```

To read all lines with the `Files.lines` method, convert the Java stream to Scala:

```
import scala.jdk.StreamConverters./*
val lineBuffer = Files.lines(Path.of(filename),
    UTF_8).toScala(Buffer)
lineIterator = Files.lines(Path.of(filename),
    UTF_8).toScala(Iterator)
```

9.2 Reading Characters

To read individual characters from a file, you can use a `Source` object directly as an iterator since the `Source` class extends `Iterator[Char]`:

```
for c <- source do process(c)
```

If you want to be able to peek at a character without consuming it (like `istream::peek` in C++ or a `PushbackInputStreamReader` in Java), call the `buffered` method on the `source` object. Then you can peek at the next input character with the `head` method without consuming it.

```
source = Source.fromFile("myfile.txt", "UTF-8")
val iter = source.buffered
while iter.hasNext do
    if isNice(iter.head) then
        process(iter)
    else
        iter.next
source.close()
```

9.3 Reading Tokens and Numbers

Here is a quick-and-dirty way of reading all whitespace-separated tokens in a source:

```
val tokens = source.mkString.split("\\s+")
```

To convert a string into a number, use the `toInt` or `toDouble` method. For example, if you have a file containing floating-point numbers, you can read them all into an array by

```
val numbers = for w <- tokens yield w.toDouble
```



Tip
Remember—you can always use the `java.util.Scanner` class to process a file that contains a mixture of text and numbers.

Finally, note that you can read numbers from `scala.io.StdIn`:

```
print("How old are you? ")
val age = StdIn.readInt()
// Or use readDouble or readLong
```



Caution

These methods assume that the next input line contains a single number, without leading or trailing whitespace. Otherwise, a `NumberFormatException` occurs.

9.4 Reading from URLs and Other Sources

The `Source` object has methods to read from sources other than files:

```
val source1 = Source.fromURL("https://horstmann.com/index.html",
"UTF-8")
val source2 = Source.fromString("Hello, World!")
// Reads from the given string—useful for debugging
val source3 = Source.stdin
// Reads from standard input
```



Caution

When you read from a URL, you need to know the character set in advance, perhaps from an HTTP header. See www.w3.org/International/Ocharset for more information.

Scala has no provision for reading binary files. You'll need to use the Java library. Here is how you can read a file into a byte array:

```
val bytes = Files.readAllBytes(Path.of(filename)); // An  
Array[Byte]
```

9.5 Writing Files

Scala has no built-in support for writing files. To write a text file, use a `java.io.PrintWriter`, for example:

```
val out = PrintWriter(filename)  
for i <- 1 to 100 do out.println(i)
```

You can also write formatted output:

```
val quantity = 10  
val price = 29.95  
out.printf("%6d %10.2f%n", quantity, price)
```

Remember to close the writer:

```
out.close()
```

9.6 Visiting Directories

There are no “official” Scala classes for visiting all files in a directory, or for recursively traversing directories.

The simplest approach is to use the `Files.list` and `Files.walk` methods of the `java.nio.file` package. The `list` method only visits the children of a directory, and the `walk` method visits all descendants. These methods yield Java streams of `Path` objects. You can visit them as follows:

```
import java.nio.file.*  
import scala.jdk.StreamConverters.*  
val dirname = "/home"  
val entries = Files.list(Paths.get(dirname)) // or Files.walk  
try
```

```
for p <- entries.toScala(Iterator) do
    process(p)
finally
    entries.close()
```

9.7 Serialization

In Java, serialization is used to transmit objects to other virtual machines or for short-term storage. (For long-term storage, serialization can be awkward—it is tedious to deal with different object versions as classes evolve over time.)

Here is how you declare a serializable class in Java and Scala.

Java:

```
public class Person implements java.io.Serializable { // This is
Java
    private static final long serialVersionUID = 42L;
    private String name;
    ...
}
```

Scala:

```
@SerialVersionUID(42L) class Person(val name: String) extends
Serializable
```

The `Serializable` trait is defined in the `scala` package and does not require an import.



Note

You can omit the `@SerialVersionUID` annotation if you are OK with the default ID.

Serialize and deserialize objects in the usual way:

```
val fred = Person("Fred")
val out = ObjectOutputStream(FileOutputStream("/tmp/test.ser"))
out.writeObject(fred)
out.close()
val in = ObjectInputStream(FileInputStream("/tmp/test.ser"))
val savedFred = in.readObject().asInstanceOf[Person]
```

The Scala collections are serializable, so you can have them as members of your serializable classes:

```
class Person extends Serializable :
    private val friends = ArrayBuffer[Person] () // OK—
    ArrayBuffer is serializable
    ...
```

9.8 Process Control A2

Traditionally, programmers use shell scripts to carry out mundane processing tasks, such as moving files from one place to another, or combining a set of files. The shell language makes it easy to specify subsets of files and to pipe the output of one program into the input of another. However, as programming languages, most shell languages leave much to be desired.

Scala was designed to scale from humble scripting tasks to massive programs. The `scala.sys.process` package provides utilities to interact with shell programs. You can write your shell scripts in Scala, with all the power that the Scala language puts at your disposal.

Here is a simple example:

```
import scala.sys.process.*
"ls -al ..".!
```

As a result, the `ls -al ..` command is executed, showing all files in the parent directory. The result is printed to standard output.

The `scala.sys.process` package contains an implicit conversion from strings to `ProcessBuilder` objects. The `!` method *executes* the `ProcessBuilder` object.

The result of the `!` method is the exit code of the executed program: `0` if the program was successful, or a nonzero failure indicator otherwise.

If you use `!!` instead of `!`, the output is returned as a string:

```
val result = "ls -al /"!!
```



The `!` and `!!` operators were originally intended to be used as postfix operators without the method invocation syntax:

```
"ls -al /" !!
```

However, as you will see in [Chapter 11](#), the postfix syntax is being deprecated since it can lead to parsing errors.

You can pipe the output of one program into the input of another, using the `#|` method:

```
("ls -al /" #| "grep u") .!
```



As you can see, the process library uses the commands of the underlying operating system. Here, I use `bash` commands because `bash` is available on Linux, Mac OS X, and Windows.

To redirect the output to a file, use the `#>` method:

```
("ls -al /" #> File("/tmp/filelist.txt")) .!
```

To append to a file, use #>> instead:

```
("ls -al /etc" #>> File("/tmp/filelist.txt")) .!
```

To redirect input from a file, use #<:

```
("grep u" #< File("/tmp/filelist.txt")) .!
```

You can also redirect input from a URL:

```
("grep Scala" #< URL("http://horstmann.com/index.html")) .!
```

You can combine processes with `p #&& q` (execute `q` if `p` was successful) and `p #|| q` (execute `q` if `p` was unsuccessful). But frankly, Scala is better at control flow than the shell, so why not implement the control flow in Scala?



The process library uses the familiar shell operators `| > >> < &&` `||`, but it prefixes them with a `#` so that they all have the same precedence.

If you need to run a process in a different directory, or with different environment variables, construct a `ProcessBuilder` with the `apply` method of the `Process` object. Supply the command, the starting directory, and a sequence of `(name, value)` pairs for environment settings:

```
val p = Process(cmd, File(dirName), ("LC_ALL", myLocale))
```

Then execute it with the `!` method:

```
("echo 42" #| p) .!
```

When executing a process command that generates a large amount of output, you can read the output lazily:

```
val result = "ls -al /.lazyLines // Yields a LazyList[String]
```

See [Chapter 13](#) how to process a lazy list.

Note

If you want to use Scala for shell scripts in a UNIX/Linux/MacOS environment, start your script files like this:

```
#!/bin/sh exec  
scala "$0" "$@"  
#!  
Scala commands
```

Note

You can also run Scala scripts from Java programs with the scripting integration of the `javax.script` package. To get a script engine, call

```
ScriptEngine engine =  
    new ScriptEngineManager().getEngineByName("scala") //  
This is Java
```

You need the Scala compiler on the class path. If you use Coursier, you can get the class path as

```
coursier fetch -p org.scala-lang:scala3-compiler_3:3.2.0
```

9.9 Regular Expressions

When you process input, you often want to use regular expressions to analyze it. The `scala.util.matching.Regex` class makes this simple. To

construct a `Regex` object, use the `r` method of the `String` class:

```
val numPattern = "[0-9]+".r
```

If the regular expression contains backslashes or quotation marks, then it is a good idea to use the “raw” string syntax, `"""..."""`. For example:

```
val wsnumwsPattern = """\s+[0-9]+\s+""".r  
// A bit easier to read than "\\\s+[0-9]+\\\s+".r
```

The `matches` method tests whether a regular expression matches a string:

```
if numPattern.matches(input) then  
  val n = input.toInt  
  ...
```

The entire input must match. To find out whether the string *contains* a match, you turn the regular expression into *unanchored* mode:

```
if numPattern.unanchored.matches(input) then  
  println("There is a number here somewhere")
```

The `findAllIn` method returns an `Iterator[String]` through all matches. Since you are unlikely to have many matches, you can simply collect the results:

```
input = "99 bottles, 98 bottles"  
numPattern.findAllIn(input).toArray // Yields Array(99, 98)
```

Note that you don’t need to call `unanchored`.

To get more information about the matches, call `findAllMatchIn` to get an `Iterator[Match]`. Each `Match` object describes the current match. Use the following methods for the match details:

- `start, end`: The starting and ending index of the matching substring
- `matched`: The matched substring
- `before, after`: The substrings before or after the match

For example:

```
for m <- numPattern.findAllMatchIn(input) do
  println(s"${m.start} ${m.end}")
```

To find the first match in a string, use `findFirstIn` or `findFirstMatchIn`. You get an `Option[String]` or `Option[Match]`.

```
val firstMatch = wsnumwsPattern.findFirstIn("99 bottles, 98
bottles")
// Some(" 98 ")
```

You can replace the first match, all matches, or some matches. In the latter case, supply a function `Match => Option[String]`. If the function returns `Some(str)`, the match is replaced with `str`.

```
numPattern.replaceFirstIn("99 bottles, 98 bottles", "XX")
// "XX bottles, 98 bottles"
numPattern.replaceAllIn("99 bottles, 98 bottles", "XX")
// "XX bottles, XX bottles"
numPattern.replaceSomeIn("99 bottles, 98 bottles",
  m => if m.matched.toInt % 2 == 0 then Some("XX") else None)
// "99 bottles, XX bottles"
```

Here is a more useful application of the `replaceSomeIn` method. We want to replace placeholders `$0`, `$1`, and so on, in a message string with values from an argument sequence. Make a pattern for the variable with a group for the index, and then map the group to the sequence element.

```
val varPattern = """\$[0-9]+""".r
def format(message: String, vars: String*) =
  varPattern.replaceSomeIn(message, m => vars.lift(
    m.matched.tail.toInt))
format("At $1, there was $2 on $0.",
  "planet 7", "12:30 pm", "a disturbance of the force")
// At 12:30 pm, there was a disturbance of the force on planet
```

The `lift` method turns a `Seq[String]` into a function. The expression `vars.lift(i)` is `Some(vars(i))` if `i` is a valid index or `None` if it is not.

9.10 Regular Expression Groups

Groups are useful to get subexpressions of regular expressions. Add parentheses around the subexpressions that you want to extract, for example:

```
val numitemPattern = "([0-9]+)([a-z]+)".r
```

You can get the group contents from a `Match` object. If `m` is a `Match` object, then `m.group(i)` is the `i`th group. The start and end positions of these substrings in the original string are `m.start(i)`, and `m.end(i)`.

```
for m <- numitemPattern.findAllMatchIn("99 bottles, 98 bottles")
do
  println(m.group(1)) // Prints 99 and 98
```



Caution

The `Match` class has methods for retrieving groups by name. However, this does *not* work with group names inside regular expressions, such as `"(?<num>[0-9]+) (?<item>[a-z]+)".r`. Instead, one needs to supply names to the `r` method: `"([0-9]+)([a-z]+)".r("num", "item")`

There is another convenient way of extracting group matches. Use a regular expression variable as an “extractor” (see [Chapter 14](#)), like this:

```
val numitemPattern(num, item) = "99 bottles"
// Sets num to "99", item to "bottles"
```

When you use a pattern as an extractor, it must match the string from which you extract the matches, and there must be a group for each variable.

If you are not sure whether there is a match, use

```
str match
  case numitemPattern(num, item) => ...
```

To extract groups from multiple matches, you can use a `for` statement like this:

```
for numitemPattern(num, item) <- numitemPattern.findAllIn("99
bottles, 98 bottles") do
  process(num, item)
```

Exercises

1. Write a Scala code snippet that reverses the lines in a file (making the last line the first one, and so on).
2. Write a Scala program that reads a file with tabs, replaces each tab with spaces so that tab stops are at n -column boundaries, and writes the result to the same file.
3. Write a Scala code snippet that reads a file and prints all words with more than 12 characters to the console. Extra credit if you can do this in a single line.
4. Write a Scala program that reads a text file containing only floating-point numbers. Print the sum, average, maximum, and minimum of the numbers in the file.
5. Write a Scala program that writes the powers of 2 and their reciprocals to a file, with the exponent ranging from 0 to 20. Line up the columns:

1	1
2	0.5
4	0.25
...	...

6. Make a regular expression searching for quoted strings "like this, maybe with \" or \\\" in a source file. Write a Scala program that prints out all such strings.
7. Write a Scala program that reads a text file and prints all tokens in the file that are *not* floating-point numbers. Use a regular expression.
8. Write a Scala program that prints the `src` attributes of all `img` tags of a web page. Use regular expressions and groups.
9. Write a Scala program that counts how many files with `.class` extension are in a given directory and its subdirectories.
10. Expand the example in [Section 9.7, “Serialization,”](#) on page 123. Construct a few `Person` objects, make some of them friends of others, and save an `Array[Person]` to a file. Read the array back in and verify that the friend relations are intact.

Chapter 10

Traits

Topics in This Chapter **L1**

- 10.1 Why No Multiple Inheritance?
- 10.2 Traits as Interfaces
- 10.3 Traits with Concrete Methods
- 10.4 Traits for Rich Interfaces
- 10.5 Objects with Traits
- 10.6 Layered Traits
- 10.7 Overriding Abstract Methods in Traits
- 10.8 Concrete Fields in Traits
- 10.9 Abstract Fields in Traits
- 10.10 Trait Construction Order
- 10.11 Trait Constructors with Parameters
- 10.12 Traits Extending Classes
- 10.13 What Happens under the Hood
- 10.14 Transparent Traits **L2**
- 10.15 Self Types **L2**
- Exercises

In this chapter, you will learn how to work with traits. A class extends one or more traits in order to take advantage of the services that the traits provide. A trait may require implementing classes to support certain features. However, unlike Java interfaces, Scala traits can supply state and behavior for these features, which makes them far more useful.

Key points of this chapter:

- A class can implement any number of traits.
- Traits can require implementing classes to have certain fields, methods, or superclasses.
- Unlike Java interfaces, a Scala trait can provide implementations of methods and fields.
- When you layer multiple traits, the order matters—the trait whose methods execute *first* goes to the back.
- Traits are compiled into Java interfaces. A class implementing traits is compiled into a Java class with all the methods and fields of its traits.
- Use a self type declaration to indicate that a trait requires another type.

10.1 Why No Multiple Inheritance?

Scala, like Java, does not allow a class to inherit from multiple superclasses. At first, this seems like an unfortunate restriction. Why shouldn't a class extend multiple classes? Some programming languages, in particular C++, allow multiple inheritance—but at a surprisingly high cost.

Multiple inheritance works fine when you combine classes that have *nothing in common*. But if these classes have common methods or fields, thorny issues come up. Here is a typical example. A teaching assistant is a student and also an employee:

```
class Student :  
    def id: String = ...  
    ...  
  
class Employee :
```

```
def id: String = ...  
...
```

Suppose we could have

```
class TeachingAssistant extends Student, Employee // Not actual  
Scala code
```

Unfortunately, this `TeachingAssistant` class inherits *two* `id` methods. What should `myTA.id` return? The student ID? The employee ID? Both? (In C++, you need to redefine the `id` method to clarify what you want.)

Next, suppose that both `Student` and `Employee` extend a common superclass `Person`:

```
class Person :  
    var name: String = null  
  
class Student extends Person :  
    ...  
  
class Employee extends Person :  
    ...
```

This leads to the *diamond inheritance* problem (see [Figure 10–1](#)). We only want one `name` field inside a `TeachingAssistant`, not two. How do the fields get merged? How does the field get constructed? In C++, you use “virtual base classes,” a complex and brittle feature, to address this issue.

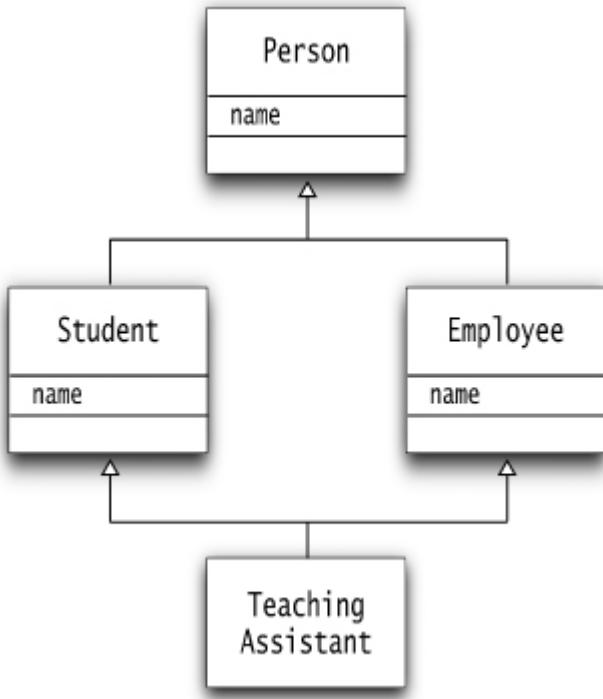


Figure 10–1 Diamond inheritance must merge common fields.

Java designers were so concerned about these complexities that they took a very restrictive approach. A class can extend only one superclass; it can implement any number of *interfaces*, but interfaces can have only abstract, static, or default methods, and no fields.

Java default methods are very limited. They can call other interface methods, but they cannot make use of object state. It is therefore common in Java to provide both an interface and an abstract base class, but that just kicks the can down the road. What if you need to extend two of those abstract base classes?

Scala has *traits* instead of interfaces. A trait can have abstract and concrete methods, as well as state. In fact, a trait can do everything a class does. There are just three differences between classes and traits:

- You cannot instantiate a trait.
- In trait methods, calls of the form `super.someMethod` are dynamically resolved.
- Traits cannot have auxiliary constructors.

You will see in the following sections how Scala deals with the perils of conflicting features from multiple traits.

10.2 Traits as Interfaces

Let's start with the simplest case. A Scala trait can work exactly like a Java interface, declaring one or more abstract methods. For example:

```
trait Logger :  
    def log(msg: String) : Unit // An abstract method
```

Note that you need not declare the method as `abstract`—an unimplemented method in a trait is automatically abstract.

A subclass can provide an implementation:

```
class ConsoleLogger extends Logger : // Use extends, not  
implements  
    def log(msg: String) = println(msg) // No override needed
```

You need not supply the `override` keyword when overriding an abstract method of a trait.



Note

Scala doesn't have a special keyword for implementing a trait. You use the same keyword `extends` for forming a subtype of a class or a trait.

If you need more than one trait, add the others using commas:

```
class FileLogger extends Logger, AutoCloseable, Appendable :  
    ...
```

Note the `AutoCloseable` and `Appendable` interfaces from the Java library. All Java interfaces can be used as Scala traits.

As in Java, a Scala class can have only one superclass but any number of traits.

 **Note**

You can use the `with` keyword instead of commas:

```
class FileLogger extends Logger with AutoCloseable with  
Appendable
```

10.3 Traits with Concrete Methods

In Scala, the methods of a trait need not be abstract. For example, we can make our `ConsoleLogger` into a trait:

```
trait ConsoleLogger extends Logger :  
    def log(msg: String) = println(msg)
```

The `ConsoleLogger` trait provides a method *with an implementation*—in this case, one that prints the logging message on the console.

Here is an example of using this trait:

```
class Account :  
    protected var balance = 0.0  
  
class ConsoleLoggedAccount extends Account, ConsoleLogger :  
    def withdraw(amount: Double) =  
        if amount > balance then log("Insufficient funds")  
        else balance -= amount  
    ...
```

Note how the `ConsoleLoggedAccount` picks up a concrete implementation from the `ConsoleLogger` trait. In Java, this is also possible by using default methods in interfaces.

In Scala (and other programming languages that allow this), we say that the `ConsoleLogger` functionality is “mixed in” with the `ConsoleLoggedAccount` class.

 **Note**

Supposedly, the “mix in” term comes from the world of ice cream. In the ice cream parlor parlance, a “mix in” is an additive that is kneaded into a scoop of ice cream before dispensing it to the customer—a practice that may be delicious or disgusting depending on your point of view.

10.4 Traits for Rich Interfaces

A trait can have many utility methods that depend on a few abstract ones. One example is the Scala `Iterator` trait that defines dozens of methods in terms of the abstract `next` and `hasNext` methods.

Let us enrich our rather anemic logging API. Usually, a logging API lets you specify a level for each log message to distinguish informational messages from warnings or errors. We can easily add this capability without forcing any policy for the destination of logging messages.

```
trait Logger :  
    def log(msg: String) : Unit  
    def info(msg: String) = log(s"INFO: $msg")  
    def warn(msg: String) = log(s"WARN: $msg")  
    def severe(msg: String) = log(s"SEVERE: $msg")
```

Note the combination of abstract and concrete methods.

A class that uses the `Logger` trait can now call any of these logging messages. For example, this class uses the `severe` method:

```
class ConsoleLoggedAccount extends Account, ConsoleLogger :  
    def withdraw(amount: Double) =
```

```
if amount > balance then severe("Insufficient funds")
else balance -= amount
...
```

This use of concrete and abstract methods in a trait is very common in Scala. In Java, you can achieve the same with default methods.

10.5 Objects with Traits

You can add a trait to an individual object when you construct it. Let's first define this class:

```
abstract class LoggedAccount extends Account, Logger :
  def withdraw(amount: Double) =
    if amount > balance then log("Insufficient funds")
    else balance -= amount
```

This class is abstract since it can't yet do any logging, which might seem pointless. But you can "mix in" a concrete logger trait when constructing an object.

Let's assume the following concrete trait:

```
trait ConsoleLogger extends Logger :
  def log(msg: String) = println(msg)
```

Here is how you can construct an object:

```
val acct = new LoggedAccount() with ConsoleLogger
```



Caution

Note that you need the `new` keyword to construct an object that mixes in a trait. (With `new`, you don't need empty parentheses to invoke the no-argument constructor of the class, but I am adding them for consistency.)

You also need to use the `with` keyword, not a comma, before each trait.

When calling `log` on the `acct` object, the `log` method of the `ConsoleLogger` trait executes.

Of course, another object can add in a different concrete trait:

```
val acct2 = new LoggedAccount() with FileLogger
```

10.6 Layered Traits

You can add, to a class or an object, multiple traits that invoke each other starting with the *last one*. This is useful when you need to transform a value in stages.

Here is a simple example. We may want to add a timestamp to all logging messages.

```
trait TimestampLogger extends ConsoleLogger :  
    override def log(msg: String) =  
        super.log(s"${java.time.Instant.now()} $msg")
```

Also, suppose we want to truncate overly chatty log messages like this:

```
trait ShortLogger extends ConsoleLogger :  
    override def log(msg: String) =  
        super.log(  
            if msg.length <= 15 then msg  
            else s"${msg.substring(0, 14)}...")
```

Note that each of the `log` methods passes a modified message to `super.log`.

With traits, `super.log` does *not* have the same meaning as it does with classes. Instead, `super.log` calls the `log` method of another trait, which depends on the order in which the traits are added.

To see how the order matters, compare the following two examples:

```
val acct1 = new LoggedAccount() with TimestampLogger with
ShortLogger
val acct2 = new LoggedAccount() with ShortLogger with
TimestampLogger
```

If we overdraw `acct1`, we get a message

```
2021-09-30T10:32:46.309584537Z Insufficient f...
```

As you can see, the `shortLogger`'s `log` method was called first, and its call to `super.log` called the `TimestampLogger`.

However, overdrawing `acct2` yields

```
2021-09-30T10:...
```

Here, the `TimestampLogger` appeared last in the list of traits. Its `log` message was called first, and the result was subsequently shortened.

For simple mixin sequences, the “back to front” rule will give you the right intuition. See [Section 10.10, “Trait Construction Order,”](#) on page 140 for the gory details that arise when the traits form a more complex graph.

Note

With traits, you cannot tell from the source code which method is invoked by `super.someMethod`. The exact method depends on the ordering of the traits in the object or class that uses them. This makes `super` far more flexible than in plain old inheritance.

Note

If you want to control which trait’s method is invoked, you can specify it in brackets: `super[ConsoleLogger].log(...)`. The specified type must be an immediate supertype; you can’t access traits or classes that are further away in the inheritance hierarchy.

10.7 Overriding Abstract Methods in Traits

In the preceding section, the `TimestampLogger` and `ShortLogger` traits extended `ConsoleLogger`. Let's make them extend our `Logger` trait instead, where we provide *no implementation* to the `log` method.

```
trait Logger :  
    def log(msg: String) : Unit // This method is abstract
```

Then, the `TimestampLogger` class no longer compiles.

```
trait TimestampLogger extends Logger :  
    override def log(msg: String) = // Overrides an abstract method  
        super.log(s"${java.time.Instant.now()} $msg") // Is  
        super.log defined?
```

The compiler flags the call to `super.log` as an error.

Under normal inheritance rules, this call could never be correct—the `Logger.log` method has no implementation. But actually, as you saw in the preceding section, there is no way of knowing which `log` method is actually being called—it depends on the order in which traits are mixed in.

Scala takes the position that `TimestampLogger.log` is still abstract—it requires a concrete `log` method to be mixed in. You therefore need to tag the method with the `abstract` keyword *and* the `override` keyword, like this:

```
abstract override def log(msg: String) =  
    super.log(s"${java.time.Instant.now()} $msg")
```

10.8 Concrete Fields in Traits

A field in a trait can be concrete or abstract. If you supply an initial value, the field is concrete.

```

trait ShortLogger extends Logger {
    val maxLength = 15 // A concrete field
    abstract override def log(msg: String) =
        super.log(
            if msg.length <= maxLength then msg
            else s"${msg.substring(0, maxLength - 1)}...")
}

```

A class that mixes in this trait acquires a `maxLength` field. In general, a class gets a field for each concrete field in one of its traits. These fields are not inherited; they are simply added to the subclass. Let us look at the process more closely, with a `SavingsAccount` class that has a field to store the interest rate:

```

class SavingsAccount extends Account, ConsoleLogger, ShortLogger
{
    var interest = 0.0
    ...
}

```

The superclass has a field:

```

class Account {
    protected var balance = 0.0
    ...
}

```

A `SavingsAccount` object is made up of the fields of its superclasses, together with the fields in the subclass. In [Figure 10–2](#), you can see that the `balance` field is contributed by the `Account` superclass, and the `interest` field by the subclass.

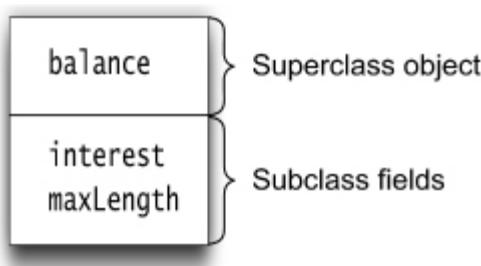


Figure 10–2 Fields from a trait are placed in the subclass.

In the JVM, a class can only extend one superclass, so the trait fields can't be picked up in the same way. Instead, the Scala compiler adds the `maxLength` field to the `SavingsAccount` class, together with the `interest` field.



Caution

When you extend a class and then change the superclass, the subclass doesn't have to be recompiled because the virtual machine understands inheritance. But when a trait changes, all classes that mix in that trait must be recompiled.

You can think of concrete trait fields as “assembly instructions” for the classes that use the trait. Any such fields become fields of the class.

10.9 Abstract Fields in Traits

An uninitialized field in a trait is abstract and must be overridden in a concrete subclass.

For example, the following `maxLength` field is abstract:

```
trait ShortLogger extends Logger :  
    val maxLength: Int // An abstract field  
    abstract override def log(msg: String) =  
        super.log()  
        if msg.length <= maxLength then msg  
        else s"${msg.substring(0, maxLength - 1)}..."  
    // The maxLength field is used in the implementation
```

When you use this trait in a concrete class, you must supply the `maxLength` field:

```
class ShortLoggedAccount extends LoggedAccount, ConsoleLogger,  
    ShortLogger :
```

```
val maxLength = 20 // No override necessary
```

Now all logging messages are truncated after 20 characters.

This way of supplying values for trait parameters is particularly handy when you construct objects on the fly. You can truncate the messages in an instance as follows:

```
val acct = new LoggedAccount() with ConsoleLogger with
ShortLogger :
    val maxLength = 15
```

10.10 Trait Construction Order

Just like classes, traits can have primary constructors. Let's defer constructor parameters until the next section. In the absence of parameters, the primary constructor consists of field initializations and other statements in the trait's body. For example,

```
trait FileLogger extends Logger :
    println("Constructing FileLogger") // Constructor code
    private val out = PrintWriter("/tmp/log.txt") // Constructor
    code
    def log(msg: String) =
        out.println(msg)
        out.flush()
```

The trait's primary constructor is executed during construction of any object incorporating the trait.

Constructors execute in the following order:

1. The superclass constructor is called first.
2. Trait constructors are executed after the superclass constructor but before the class constructor.
3. Traits are constructed left-to-right.
4. Within each trait, the parents get constructed first.

5. If multiple traits share a common parent, and that parent has already been constructed, it is not constructed again.
6. After all traits are constructed, the subclass is constructed.

For example, consider this class:

```
class FileLoggedAccount extends Account, FileLogger,  
TimestampLogger
```

The constructors execute in the following order:

1. `Account` (the superclass).
 2. `Logger` (the parent of the first trait).
 3. `FileLogger` (the first trait).
 4. `TimestampLogger` (the second trait). Note that its `Logger` parent has already been constructed.
 5. `FileLoggedAccount` (the class).
-



Note

The constructor ordering is the reverse of the *linearization* of the class. The linearization is a technical specification of all supertypes of a type. It is defined by the rule:

If C extends C_1, C_2, \dots, C_n , then $\text{lin}(C) = C \gg \text{lin}(C_n) \gg \dots \gg \text{lin}(C_2) \gg \text{lin}(C_1)$

Here, \gg means “concatenate and remove duplicates, with the right winning out.” For example,

$\text{lin}(\text{FileLoggedAccount})$

```
= FileLoggedAccount >> lin(TimestampLogger) >> lin(FileLogger)  
>> lin(Account)
```

```

= FileLoggedAccount » (TimestampLogger » Logger) »
(FileLogger » Logger) »
lin(Account)

= FileLoggedAccount » TimestampLogger » FileLogger » Logger
» Account.

```

(For simplicity, I omitted the types `AnyRef`, and `Any` that are at the end of any linearization.)

The linearization gives the order in which `super` is resolved in a trait. For example, calling `super` in a `TimestampLogger` invokes the `FileLogger` method.

10.11 Trait Constructors with Parameters

In the preceding section, we looked at trait constructors without parameters. You saw how a given trait is constructed exactly once. Let's turn to trait constructors with parameters. For a file logger, one would like to specify the log file:

```

trait FileLogger(filename: String) extends Logger :
  private val out = PrintWriter(filename)
  def log(msg: String) =
    out.println(msg)
    out.flush()

```

Then you pass the file name when mixing in the file logger:

```
val acct = new LoggedAccount() with FileLogger("/tmp/log.txt")
```

Of course, it must be guaranteed that the trait is initialized exactly once. To ensure this, there are three simple rules:

1. A class must initialize any uninitialized trait that it extends.
2. A class cannot initialize a trait that a superclass already initialized.

3. A trait cannot initialize another trait.

Let us go through these rules with some examples. First, consider a class extending a parameterized trait. It must provide an argument. For example, the following would be illegal:

```
class FileLoggedAccount extends LoggedAccount, FileLogger
    // Error—no arguments for FileLogger constructor
```

The remedy is to provide an argument:

```
class FileLoggedAccount(filename: String) extends LoggedAccount,
    FileLogger(filename)
```

You cannot initialize a trait that was already initialized by a superclass. This isn't a common issue, so here is a contrived example:

```
class TmpLoggedAccount extends Account,
    FileLogger("/tmp/log.txt")
class FileLoggedAccount(filename) extends TmpLoggedAccount,
    FileLogger(filename)
    // Error—FileLogger already initialized
```

Finally, a trait extending a parameterized trait cannot pass initialization arguments.

```
trait TimestampFileLogger extends FileLogger("/tmp/log.txt") :
    // Error—a trait cannot call the constructor of another trait
```

Instead, drop the constructor parameter:

```
trait TimestampFileLogger extends FileLogger :
    override def log(msg: String) =
        super.log(s"${java.time.Instant.now()} $msg")
```

The initialization must happen in each class using a `TimestampFileLogger`:

```
val acct2 = new LoggedAccount() with TimestampFileLogger with  
FileLogger("/tmp/log.txt")
```

10.12 Traits Extending Classes

As you have seen, a trait can extend another trait, and it is common to have a hierarchy of traits. Less commonly, a trait can also extend a class. That class becomes a superclass of any class mixing in the trait.

Here is an example. The `LoggedException` trait extends the `Exception` class:

```
trait LoggedException extends Exception, ConsoleLogger :  
    override def log(msg: String) = super.log(s"${getMessage() }  
$msg")
```

A `LoggedException` has a `log` method to log the exception's message. Note that the `log` method calls the `getMessage` method that is inherited from the `Exception` superclass.

Now let's form a class that mixes in this trait:

```
class UnhappyException extends LoggedException : // This class  
extends a trait  
    override def getMessage() = "arggh!"
```

The superclass of the trait becomes the superclass of our class (see [Figure 10–3](#)).

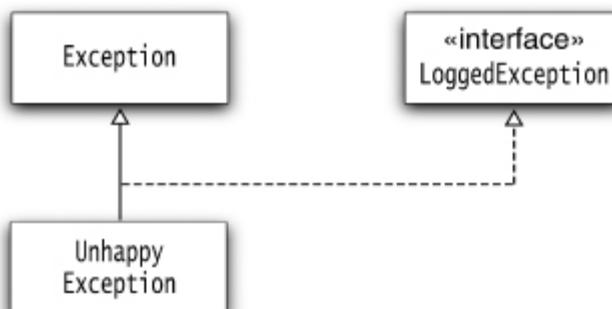


Figure 10–3 The superclass of a trait becomes the superclass of any class mixing in the trait.

What if our class already extends another class? That's OK, as long as it's a subclass of the trait's superclass. For example,

```
class UnhappyIOException extends IOException, LoggedException
```

Here `UnhappyIOException` extends `IOException`, which already extends `Exception`. When mixing in the trait, its superclass is already present, and there is no need to add it.

However, if our class extends an unrelated class, then it is not possible to mix in the trait. For example, you cannot form the following class:

```
class UnhappyFrame extends javax.swing.JFrame, LoggedException
// Error: Unrelated superclasses
```

It would be impossible to add both `JFrame` and `Exception` as superclasses.

10.13 What Happens under the Hood

Scala translates traits into interfaces of the JVM. You are not required to know how this is done, but you may find it helpful for understanding how traits work.

A trait that has only abstract methods is simply turned into a Java interface. For example,

```
trait Logger :
  def log(msg: String) : Unit
```

turns into

```
public interface Logger { // Generated Java interface
  void log(String msg);
}
```

Trait methods become default methods. For example,

```
trait ConsoleLogger :  
    def log(msg: String) = println(msg)
```

becomes

```
public interface ConsoleLogger {  
    default void log(String msg) { ... }  
}
```

If the trait has fields, the Java interface has getter and setter methods.

```
trait ShortLogger extends ConsoleLogger :  
    val maxLength = 15 // A concrete field  
    ...
```

is translated to

```
public interface ShortLogger extends Logger {  
    int maxLength();  
    void some_prefix$maxLength_$eq(int);  
    default void log(String msg) { ... } // Calls maxLength()  
    default void $init$() { some_prefix$maxLength_$eq(15); }  
}
```

Of course, the interface can't have any fields, and the getter and setter methods are unimplemented. The getter is called when the field value is needed.

The setter is needed to initialize the field. This happens in the `$init$` method.

When the trait is mixed into a class, the class gets a `maxLength` field, and the getter and setter are defined to get and set that field. The constructors of the class invokes the `$init$` method of the trait. For example,

```
class ShortLoggedAccount extends Account, ShortLogger
```

turns into

```
public class ShortLoggedAccount extends Account implements
ShortLogger {
    private int maxLength;
    public int maxLength() { return maxLength; }
    public void some_prefix$maxLength_$eq(int arg) { maxLength =
arg; }
    public ShortLoggedAccount() {
        super();
        ShortLogger.$init$();
    }
    ...
}
```

If a trait extends a superclass, the trait still turns into an interface. Of course, a class mixing in the trait extends the superclass.

As an example, consider the following trait:

```
trait LoggedException extends Exception, ConsoleLogger :
    override def log(msg: String) = super.log(s"${getMessage()} "
$msg")
```

It becomes a Java interface. The superclass is nowhere to be seen.

```
public interface LoggedException extends ConsoleLogger {
    public void log();
}
```

When the trait is mixed into a class, then the class extends the trait's superclass. For example,

```
class UnhappyException extends LoggedException :
    override def getMessage() = "arggh!"
```

becomes

```
public class UnhappyException extends Exception implements
```

```
LoggedException
```

10.14 Transparent Traits L2

Consider this inheritance hierarchy:

```
class Person
class Employee extends Person, Serializable, Cloneable
class Contractor extends Person, Serializable, Cloneable
```

When you declare

```
val p = if scala.math.random() < 0.5 then Employee() else
Contractor()
```

you probably expect `p` to have type `Person`. Actually, the type is `Person & Cloneable`. That actually makes sense: both `Employee` and `Contractor` are subtypes of `Cloneable`.

Why isn't the inferred type `Person & Serializable & Cloneable`? The `Serializable` trait is marked as *transparent* so that it is not used for type inference. Other transparent traits include `Product` and `Comparable`.

In the unlikely situation that you want to declare another trait as transparent, here is how to do it:

```
transparent trait Logged
```

10.15 Self Types L2

A trait can require that it is mixed into a class that extends another type. You achieve this with a *self type* declaration, which has the following unlovable syntax:

```
this: Type =>
```

In the following example, the `LoggedException` trait can only be mixed into a class that extends `Exception`:

```
trait LoggedException extends Logger :  
  this: Exception =>  
    def log(): Unit = log(getMessage())  
    // OK to call getMessage because this is an Exception
```

If you try to mix the trait into a class that doesn't conform to the self type, an error occurs:

```
val f = new Account() with LoggedException  
// Error: Account isn't a subtype of Exception, the self type of  
LoggedException
```

A trait with a self type is similar to a trait with a supertype. In both cases, it is ensured that a type is present in a class that mixes in the trait. However, self types can handle circular dependencies between traits. This can happen if you have two traits that need each other.



Caution

Self types do not automatically inherit. If you define

```
trait MonitoredException extends LoggedException
```

you get an error that `MonitoredException` doesn't supply `Exception`. In this situation, you need to repeat the self type:

```
trait MonitoredException extends LoggedException {  
  this: Exception =>
```

To require multiple types, use an intersection type:

```
this: T & U & ... =>
```

 **Note**

If you give a name other than `this` to the variable in the self type declaration, then it can be used in subtypes by that name. For example,

```
trait Group :  
    outer: Network =>  
        class Member :  
            ...
```

Inside `Member`, you can refer to the `this` reference of `Group` as `outer`. By itself, that is not an important benefit since you could introduce the name as follows:

```
trait Group :  
    val self: this.type = this  
        class Member :  
            ...
```

Exercises

1. The `java.awt.Rectangle` class has useful methods `translate` and `grow` that are unfortunately absent from classes such as `java.awt.geom.Ellipse2D`. In Scala, you can fix this problem. Define a trait `RectangleLike` with concrete methods `translate` and `grow`. Provide any abstract methods that you need for the implementation, so that you can mix in the trait like this:

```
val egg = java.awt.geom.Ellipse2D.Double(5, 10, 20, 30) with  
    RectangleLike  
egg.translate(10, -10)  
egg.grow(10, 20)
```

2. Define a class `OrderedPoint` by mixing `scala.math.Ordered[Point]` into `java.awt.Point`. Use lexicographic ordering, i.e. $(x, y) < (x', y')$ if $x < x'$ or $x = x'$ and $y < y'$.
3. Look at the `BitSet` class, and make a diagram of all its superclasses and traits. Ignore the type parameters (everything inside the `[...]`). Then give the linearization of the traits.
4. Provide a `CryptoLogger` trait that encrypts the log messages with the Caesar cipher. The key should be 3 by default, but it should be overridable by the user. Provide usage examples with the default key and a key of -3.
5. The JavaBeans specification has the notion of a *property change listener*, a standardized way for beans to communicate changes in their properties. The `PropertyChangeSupport` class is provided as a convenience superclass for any bean that wishes to support property change listeners. Unfortunately, a class that already has another superclass—such as `JComponent`—must reimplement the methods. Reimplement `PropertyChangeSupport` as a trait, and mix it into the `java.awt.Point` class.
6. In the Java AWT library, we have a class `Container`, a subclass of `Component` that collects multiple components. For example, a `Button` is a `Component`, but a `Panel` is a `Container`. That's the composite pattern at work. Swing has `JComponent` and `JButton`, but if you look closely, you will notice something strange. `JComponent` extends `Container`, even though it makes no sense to add other components to, say, a `JButton`. Ideally, the Swing designers would have preferred the design in [Figure 10–4](#).

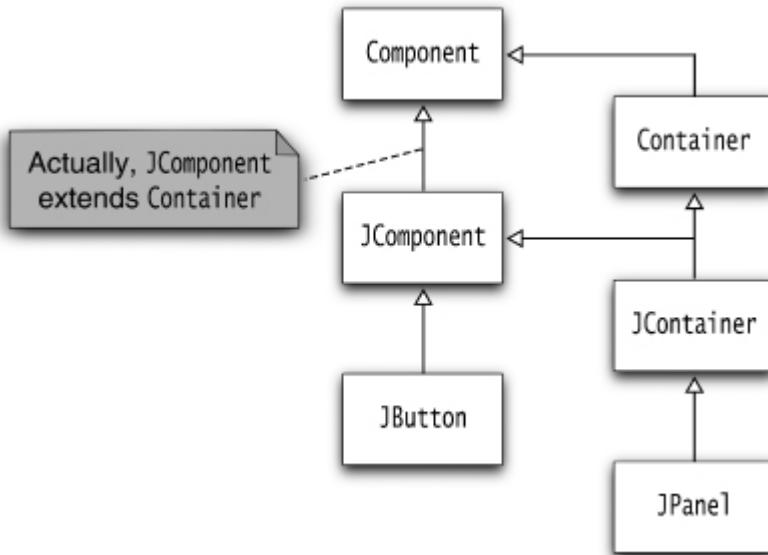


Figure 10–4 A better design for Swing containers

But that's not possible in Java. Explain why not. How could the design be executed in Scala with traits?

7. Construct an example where a class needs to be recompiled when one of the mixins changes. Start with `class ConsoleLoggedAccount extends Account, ConsoleLogger`. Put each class and trait in a separate source file. Add a field to `Account`. In your main method (also in a separate source file), construct a `ConsoleLoggedAccount` and access the new field. Recompile all files *except for* `ConsoleLoggedAccount` and verify that the program works. Now add a field to `ConsoleLogger` and access it in your main method. Again, recompile all files *except for* `ConsoleLoggedAccount`. What happens? Why?

8. There are dozens of Scala trait tutorials with silly examples of barking dogs or philosophizing frogs. Reading through contrived hierarchies can be tedious and not very helpful, but designing your own is very illuminating. Make your own silly trait hierarchy example that demonstrates layered traits, concrete and abstract methods, concrete and abstract fields, and trait parameters.

9. In the `java.io` library, you add buffering to an input stream with a `BufferedInputStream` decorator. Reimplement buffering as a trait. For simplicity, override the `read` method.

Chapter 11

Operators

Topics in This Chapter **L1**

- 11.1 Identifiers
- 11.2 Infix Operators
- 11.3 Unary Operators
- 11.4 Assignment Operators
- 11.5 Precedence
- 11.6 Associativity
- 11.7 The `apply` and `update` Methods
- 11.8 The `unapply` Method **L2**
- 11.9 The `unapplySeq` Method **L2**
- 11.10 Alternative Forms of the `unapply` and `unapplySeq` Methods **L3**
- 11.11 Dynamic Invocation **L2**
- 11.12 Typesafe Selection and Application **L2**
- Exercises

This chapter covers in detail implementing your own *operators*—methods with the same syntax as the familiar mathematical operators. Operators are often used to build *domain-specific languages*—minilanguages embedded

inside Scala. *Implicit conversions* (type conversion functions that are applied automatically) are another tool facilitating the creation of domain-specific languages. This chapter also discusses the special methods `apply`, `update`, and `unapply`. We end the chapter with a discussion of *dynamic invocations*—method calls that can be intercepted at runtime, so that arbitrary actions can occur depending on the method names and arguments.

The key points of this chapter are:

- Identifiers contain either alphanumeric or operator characters.
- Unary and binary operators are method calls.
- Operator precedence depends on the first character, associativity on the last.
- The `apply` and `update` methods are called when evaluating `expr(args)`.
- Extractors extract tuples or sequences of values from an input.
- Types extending the `Dynamic` trait can inspect the names of methods and arguments at runtime. **L2**

11.1 Identifiers

The names of variables, functions, classes, and so on are collectively called *identifiers*. In Scala, you have more choices for forming identifiers than in most other programming languages. Of course, you can follow the time-honored pattern: sequences of alphanumeric characters, starting with an alphabetic character or an underscore, such as `input1` or `next_token`.

Unicode characters are allowed. For example, `quantité` or `ποσό` are valid identifiers.

In addition, you can use *operator characters* in identifiers:

- The ASCII characters `! # % & * + - / : < = > ? @ \ ^ | ~` that are not letters, digits, underscore, the `..`; punctuation marks, parentheses `()` `[] {}`, or quotation marks `' ` "`.
- Unicode mathematical symbols or other symbols from the Unicode categories Sm and So.

For example, `**` and `\sqrt` are valid identifiers. With the definition

```
val √ = scala.math.sqrt
```

you can write `\sqrt(2)` to compute a square root. This may be a good idea, provided one's programming environment makes it easy to type the symbol.



Note

The identifiers `@` `#` `:` `=` `_` `=>` `<-` `<:` `<%` `>:` `⇒` `←` are reserved in the specification, and you cannot redefine them.

You can also form identifiers from alphanumerical characters, followed by an underscore, and then a sequence of operator characters, such as

```
val happy_birthday_!!! = "Bonne anniversaire!!!"
```

This is probably not a good idea.

Finally, you can include just about any sequence of characters in backquotes. For example,

```
val `val` = 42
```

That example is silly, but backquotes can sometimes be an “escape hatch.” For example, in Scala, `yield` is a reserved word, but you may need to access a Java method of the same name. Backquotes to the rescue: `Thread.`yield`()`.

11.2 Infix Operators

You can write

a *identifier* b

where *identifier* denotes a method with two parameters (one implicit, one explicit). For example, the expression

1 to 10

is actually a method call

1.to(10)

This is called an *infix* expression because the operator is between the arguments. The operator can contain letters, as in `to`, or it can contain operator characters—for example,

1 -> 10

is a method call

1 .->(10)

To define an operator in your own class, simply define a method whose name is that of the desired operator. For example, here is a `Fraction` class that multiplies two fractions according to the law

$$(n_1 / d_1) \times (n_2 / d_2) = (n_1 n_2 / d_1 d_2)$$

```
class Fraction(n: Int, d: Int) :  
    private val num = ...  
    private val den = ...  
    ...  
    def *(other: Fraction) = Fraction(num * other.num, den *  
other.den)
```

If you want to call a symbolic operator from Java, use the `@targetName` annotation to give it an alphanumeric name:

```
@targetName("multiply") def *(other: Fraction) =  
Fraction(num * other.num, den * other.den)
```

In Scala code, you use `f * g`, but in Java, you use `f.multiply(g)`.

In order to use a method with an alphanumeric name as an infix operator, use the `infix` modifier:

```
infix def times(other: Fraction) = Fraction(num * other.num, den  
* other.den)
```

Now you can call `f times g` in Scala.

Note

You can use a method with an alphanumeric name with infix syntax whenever it is followed by an opening brace.

```
f repeat { "Hello" }
```

The method need not be declared as `infix`.

Caution

It is possible to declare an infix operator with multiple arguments. The `+=` operator for mutable collections has such a form:

```
val smallPrimes = ArrayBuffer[Int]()  
smallPrimes += 2 // binary infix operator, invokes +=(Int)  
smallPrimes += (3, 5) multiple arguments, invokes +=(Int,  
Int, Int*)
```

This sounded a good idea at the time, but it gives grief with tuples. Consider:

```
val twinPrimes = ArrayBuffer[(Int, Int)]()  
twinPrimes += (11, 13) // Error
```

Surely this should add the tuple `(11, 13)` to the buffer of tuples, but it triggers the multi-argument infix syntax and fails, since `11`

and `13` are not tuples.

At some point, infix operators with multiple arguments may be removed. In the meantime, you have to call

```
twinPrimes += ((11, 13))
```

11.3 Unary Operators

Infix operators are binary operators—they have two parameters. An operator with one parameter is called a unary operator.

The four operators `+`, `-`, `!`, `~` are allowed as *prefix* operators, appearing before their arguments. They are converted into calls to methods with the name `unary_operator`. For example,

```
-a
```

means the same as `a.unary_-`.

If a unary operator follows its argument, it is a *postfix* operator. For example, the expression `42 toString` is the same as `42.toString`.

However, postfix operators can lead to parsing errors. For example, the code

```
val result = 42 toString  
println(result)
```

yields the error message “Recursive value `result` needs type”. Since parsing precedes type inference and overload resolution, the compiler does not yet know that `toString` is a unary method. Instead, the code is parsed as `val result = 42.toString(println(result))`.

For that reason, Scala now discourages the use of postfix operators. If you really want to use them, you must use the compiler option `-language:postfixOps` or add the clause

```
import scala.language.postfixOps
```

11.4 Assignment Operators

An assignment operator has the form *operator*=, and the expression

a *operator*= b

means the same as

a = a *operator* b

For example, a += b is equivalent to a = a + b.

There are a few technical details.

- <=, >=, and != are not assignment operators.
- An operator starting with an = is never an assignment operator (==, ===, /=, and so on).
- If a has a method called *operator*=, then that method is called directly.

11.5 Precedence

When you have two or more operators in a row without parentheses, the ones with higher *precedence* are executed first. For example, in the expression

1 + 2 * 3

the * operator is evaluated first.

In most languages, there is a fixed set of operators, and the language standard decrees which have precedence over which. Scala can have arbitrary operators, so it uses a scheme that works for all operators, while also giving the familiar precedence order to the standard ones.

Except for assignment operators, the precedence is determined by the *first character* of the operator.

Highest precedence: An operator character other than those below

* / %

+ - :

< >

! =

&

^

|

A character that is not an operator character

Lowest precedence: Assignment operators

Characters in the same row yield operators with the same precedence. For example, `+` and `->` have the same precedence.

Postfix operators have lower precedence than infix operators:

a *infixOp* b*postfixOp*

is the same as

(a *infixOp* b) *postfixOp*

11.6 Associativity

When you have a sequence of operators of the same precedence, the *associativity* determines whether they are evaluated left-to-right or right-to-left. For example, in the expression $17 - 2 - 9$, one computes $(17 - 2) - 9$. The `-` operator is *left-associative*.

In Scala, all operators are left-associative except for

- operators that end in a colon (`:`)
- assignment operators

In particular, the `::` operator for constructing lists is right-associative. For example,

```
1 :: 2 :: Nil
```

means

```
1 :: (2 :: Nil)
```

This is as it should be—we first need to form the list containing `2`, and that list becomes the tail of the list whose head is `1`.

A right-associative binary operator is a method of its second argument. For example,

```
2 :: Nil
```

means

```
Nil.::(2)
```

11.7 The `apply` and `update` Methods

Scala lets you extend the function call syntax

```
f(arg1, arg2, ...)
```

to values other than functions. If `f` is not a function or method, then this expression is equivalent to the call

```
f.apply(arg1, arg2, ...)
```

unless it occurs to the left of an assignment. The expression

```
f(arg1, arg2, ...) = value
```

corresponds to the call

```
f.update(arg1, arg2, ..., value)
```

This mechanism is used in arrays and maps. For example,

```
val scores = scala.collection.mutable.HashMap[String, Int]()
scores("Bob") = 100 // Calls scores.update("Bob", 100)
val bobsScore = scores("Bob") // Calls scores.apply("Bob")
```



Note

As you have already seen in [Chapter 5](#), the companion object of every class has an `apply` method that calls the primary constructor.

For example, `Fraction(3, 4)` calls the `Fraction.apply` method, which returns new `Fraction(3, 4)`.

11.8 The `unapply` Method L2

An `apply` method takes construction parameters and turns them into an object. An `unapply` method does the opposite. It takes an object and extracts values from it—usually the values from which the object was, or could be, constructed.

Consider the `Fraction` class from [Section 11.2, “Infix Operators,”](#) on page 152. A call such as `Fraction(3, 4)` calls the `Fraction.apply` method which makes a fraction from a numerator and denominator. An `unapply` method does the opposite and extracts the numerator and denominator from a fraction.

One way to invoke an extractor is in a variable declaration. Here is an example:

```
val Fraction(n, d) = Fraction(3, 4) * Fraction(2, 5)
// n, d are initialized with the numerator and denominator of the
result
```

This statement declares two variables `n` and `d`, both of type `Int`, and *not* a `Fraction`. The variables are initialized with the values that are extracted from the right hand side.

More commonly, extractors are invoked in pattern matches such as the following:

```
val value = f match
  case Fraction(a, b) => a.toDouble / b // a, b are bound to the
numerator and denominator
  case _ => Double.NaN
```

The details of implementing `unapply` are somewhat tedious. You may want to skip them until after reading [Chapter 14](#).

Since a pattern match can fail, the `unapply` method returns an `Option`. Upon success, the `Option` contains a tuple holding the extracted values. In our case, we return an `Option[(Int, Int)]`.

```
object Fraction :  
    def unapply(input: Fraction) =  
        if input.den == 0 then None else Some((input.num,  
input.den))
```

This method returns `None` when the fraction is malformed (with a zero denominator), indicating no match.

A statement

```
val Fraction(a, b) = f
```

leads to the method call

```
Fraction.unapply(f)
```

If the method returns `None`, a `MatchError` is thrown. Otherwise, the variables `a` and `b` are set to the components of the returned tuple.

Note that neither the `Fraction.apply` method nor the `Fraction` constructor are called. However, the intent is to initialize `a` and `b` so that they would yield `f` if they were passed to `Fraction.apply`. This is sometimes called *destructuring*. In that sense, `unapply` is the inverse of `apply`.

It is not a requirement for the `apply` and `unapply` methods to be inverses of one another. However, that is not a requirement. You can use extractors to extract information from an object of any type.

For example, suppose you want to extract first and last names from a string:

```
val author = "Cay Horstmann"  
val Name(first, last) = author // Calls Name.unapply(author)
```

Provide an object `Name` with an `unapply` method that returns an `Option[(String, String)]`. If the match succeeds, return a pair with the first and last name. Otherwise, return `None`.

```
object Name :  
    def unapply(input: String) =  
        val pos = input.indexOf(" ")  
        if pos >= 0 then Some((input.substring(0, pos),  
input.substring(pos + 1)))  
        else None
```

Note

In this example, there is no `Name` class. The `Name` object is an extractor for `String` objects.

Note

The `unapply` methods in this section return an `Option` of a tuple. It is possible to return other types. See [Chapter 14](#) for the details.

11.9 The `unapplySeq` Method [L2]

The `unapply` method extracts a fixed number of values. To extract an arbitrary number of values, the method needs to be called `unapplySeq`. In the simplest case, the method returns an `Option[Seq[T]]`, where `T` is the type of the extracted values. For example, a `Name` extractor can produce a sequence of the name's components:

```
object Name :  
    def unapplySeq(input: String): Option[Seq[String]] =  
        if input.strip == "" then None else  
        Some(input.strip.split(",?\\s+").toSeq)
```

Now you can extract any number name components:

```
val Name(first, middle, last, rest*) = "John D. Rockefeller IV,  
B.A."
```

The `rest` variable is set to a `Seq[String]`.



Caution

Do not supply both an `unapply` and an `unapplySeq` method with the same argument types.

11.10 Alternative Forms of the `unapply` and `unapplySeq` Methods L3

In [Section 11.8, “The `unapplyMethod`,”](#) on page 157, you saw how to implement an `unapply` method that returns an `Option` of a tuple:

```
object Fraction :  
  def unapply(input: Fraction) =  
    if input.den == 0 then None else Some((input.num,  
    input.den))
```

But the return type of `unapply` is quite a bit more flexible.

- You don’t need an `Option` if the match never fails.
- Instead of an `Option`, you can use any type with methods `isEmpty` and `get`.
- Instead of a tuple, you can use any subtype of `Product` with methods `_1`, `_2`, ..., `_n`.
- To extract a single value, you don’t need a tuple or `Product`.
- Return a `Boolean` to have the match succeed or fail without extracting a value.

In the preceding example, the `Fraction.unapply` method has return type `option[(Int, Int)]`. Fractions with zero denominators return `None`. To have the match succeed in all cases, simply return a tuple without wrapping it into an `Option`:

```
object Fraction :  
    def unapply(input: Fraction) = (input.num, input.den)
```

Here is an example of an extractor that produces a single value:

```
object Number :  
    def unapply(input: String): Option[Int] =  
        try  
            Some(input.strip.toInt)  
        catch  
            case ex: NumberFormatException => None
```

With this extractor, you can extract a number from a string:

```
val Number(n) = "1729"
```

An extractor returning `Boolean` tests its input without extracting any value. Here is such a test extactor:

```
object IsCompound :  
    def unapply(input: String) = input.contains(" ")
```

You can use this extractor to add a test to a pattern:

```
author match  
    case Name(first, last @ IsCompound()) => ...  
        // Matches if the last name is compound, such as van der Linden  
    case Name(first, last) => ...
```

Finally, the return type of `unapplySeq` can be more general than an `Option[Seq[T]]`:

- An `Option` isn't needed if the match never fails.

- Any type with methods `isEmpty` and `get` can be used instead of `Option`
- Any type with methods `apply`, `drop`, `toSeq`, and either `length` or `lengthCompare` can be used instead of `Seq`

11.11 Dynamic Invocation L2

Scala is a strongly typed language that reports type errors at compile time rather than at runtime. If you have an expression `x.f(args)`, and your program compiles, then you know for sure that `x` has a method `f` that can accept the given arguments. However, there are situations where it is desirable to define methods in a running program. This is common with object-relational mappers in dynamic languages such as Ruby or JavaScript. Objects that represent database tables have methods `findByName`, `findById`, and so on, with the method names matching the table columns. For database entities, the column names can be used to get and set fields, such as `person.lastName = "Doe"`.

In Scala, you can do this too. If a type extends the trait `scala.Dynamic`, then method calls, getters, and setters are rewritten as calls to special methods that can inspect the name of the original call and the parameters, and then take arbitrary actions.



Note

Dynamic types are an “exotic” feature, and the compiler wants your explicit consent when you implement such a type. You do that by adding the `import` statement

```
import scala.language.dynamics
```

Users of such types do not need to provide the `import` statement.

Here are the details of the rewriting. Consider `obj.name`, where `obj` belongs to a class that’s a subtype of `Dynamic`. Here is what the Scala compiler does with it.

1. If `name` is a known method or field of `obj`, it is processed in the usual way.
2. If `obj.name` is followed by `(arg1, arg2, ...)`,
 - a. If none of the arguments are named (of the form `name=arg`), pass the arguments on to `applyDynamic`:

```
obj.applyDynamic("name") (arg1, arg2, ...)
```

- b. If at least one of the arguments is named, pass the name/value pairs on to `applyDynamicNamed`:

```
obj.applyDynamicNamed("name") ((name1, arg1), (name2, arg2), ...)
```

Here, `name1`, `name2`, and so on are strings with the argument names, or `" "` for unnamed arguments.

3. If `obj.name` is to the left of an `=`, call

```
obj.updateDynamic("name") (rightHandSide)
```

4. Otherwise call

```
obj.selectDynamic("sel")
```



Note

The calls to `updateDynamic`, `applyDynamic`, and `applyDynamicNamed` are “curried”—they have two sets of parentheses, one for the selector name and one for the arguments. This construct is explained in [Chapter 12](#).

Let’s look at a few examples. Suppose `person` is an instance of a type extending `Dynamic`. A statement

```
person.lastName = "Doe"
```

is replaced with a call

```
person.updateDynamic("lastName") ("Doe")
```

The `Person` class must have such a method:

```
class Person :  
  ...  
  def updateDynamic(field: String) (newValue: String) = ...
```

It is then up to you to implement the `updateDynamic` method. For example, if you are implementing an object-relational mapper, you might update the cached entity and mark it as changed, so that it can be persisted in the database.

Conversely, a statement

```
val name = person.lastName
```

turns into

```
val name = name.selectDynamic("lastName")
```

The `selectDynamic` method would simply look up the field value.

Method calls are translated to calls of the `applyDynamic` or `applyDynamicNamed` method. The latter is used for calls with named parameters. For example,

```
val does = people.findByLastName("Doe")
```

becomes

```
val does = people.applyDynamic("findByLastName") ("Doe")
```

and

```
val johnDoes = people.find(lastName = "Doe", firstName = "John")
```

becomes

```
val johnDoes = people.applyDynamicNamed("find")(("lastName",  
"Doe"),  
("firstName", "John"))
```

It is then up to you to implement `applyDynamic` and `applyDynamicNamed` as calls that retrieve the matching objects.

Here is a concrete example. Suppose we want to be able to dynamically look up and set elements of a `java.util.Properties` instance, using the dot notation:

```
val sysProps = DynamicProps(System.getProperties)  
sysProps.username = "Fred" // Sets the "username" property to  
"Fred"  
val home = sysProps.java_home // Gets the "java.home" property
```

For simplicity, we replace periods in the property name with underscores. ([Exercise 13](#) on page 166 shows how to keep the periods.)

The `DynamicProps` class extends the `Dynamic` trait and implements the `updateDynamic` and `selectDynamic` methods:

```
class DynamicProps(val props: java.util.Properties) extends  
Dynamic :  
  def updateDynamic(name: String) (value: String) =  
    props.setProperty(name.replaceAll("_", "."), value)  
  def selectDynamic(name: String) =  
    props.getProperty(name.replaceAll("_", ".") )
```

As an additional enhancement, let us use the `add` method to add key/value pairs in bulk, using named arguments:

```
sysProps.add(username="Fred", password="Secret")
```

Then we need to supply the `applyDynamicNamed` method in the `DynamicProps` class. Note that the name of the method is fixed. We are only interested in arbitrary parameter names.

```

def applyDynamicNamed(name: String)(args: (String, String)*) =
  if name != "add" then throw IllegalArgumentException()
  for ((k, v) <- args)
    props.setProperty(k.replaceAll("_", ".") , v)

```

These examples are only meant to illustrate the mechanism. Is it really that useful to use the dot notation for map access? Like operator overloading, dynamic invocation is a feature that is best used with restraint.

11.12 Typesafe Selection and Application

L2

In the preceding section, you saw how to resolve selections `obj.selector` and method calls `obj.method(args)` dynamically. However, that approach is not typesafe. If `selector` or `method` are not appropriate, a run time error occurs. In this section, you will learn how to detect invalid selections and invocations at compile time.

Instead of the `Dynamic` trait, you use the `Selectable` trait. It has methods

```

def selectDynamic(name: String): Any
def applyDynamic(name: String)(args: Any*): Any

```

The key difference is that you specify the selectors and methods that can be applied.

Let us first look at selection. Suppose we have objects with properties from a cache of database records, or JSON values, or, to keep the example simple, from a map. Then we can select a property with this class:

```

class Props(props: Map[String, Any]) extends Selectable :
  def selectDynamic(name: String) = props(name)

```

Now let's move on to the typesafe part. Suppose we want to work with invoice items. Define a type with the valid properties, like this:

```
type Item = Props {  
    val description: String  
    val price: Double  
}
```

This is an example of a *structural type*—see [Chapter 18](#) for details.

Construct instances as follows:

```
val toaster = Props(Map("description" -> "Blackwell Toaster",  
"price" -> 29.95)).asInstanceOf[Item]
```

When you call

```
toaster.price
```

the `selectDynamic` method is invoked. This is no different than in the preceding section.

However, a call `toaster.brand` is a *compile-time error* since `brand` is not one of the listed selectors in the `Item` type.

Next, let's turn to methods. We want to write a library for making REST calls with Scala syntax. Calls such as

```
val buyer = myShoppingService.customer(id)  
shoppingCart += myShoppingService.item(id)
```

should, behind the scenes, invoke REST requests <http://myserver.com/customer/id> and <http://myserver.com/item/id>, yielding the JSON responses.

And we want this to be typesafe. A call to a nonexistent REST service should fail at compile time.

First, make a generic class

```
class Request(baseUrl: String) extends Selectable :  
    def applyDynamic(name: String)(args: Any*) : Any =
```

```
val url = s"$baseUrl/$name/${args(0)}"  
scala.io.Source.fromURL(url).mkString
```

Then constrain to the services that you know to be actually supported.

I don't have access to a shopping service, but I have a service that produces random nouns and adjectives:

```
type RandomService = Request {  
    def nouns(qty: Int) : String  
    def adjectives(qty: Int) : String  
}
```

Construct an instance:

```
val myRandomService = new  
Request("https://horstmann.com/random").asInstanceOf[RandomService]
```

Now a call

```
myRandomService.nouns(5)
```

is translated to a call

```
myRandomService.updateDynamic("nouns")(5)
```

However, if the method name is neither `nouns` nor `adjectives`, a compiler error occurs.

Exercises

1. According to the precedence rules, how are `3 + 4 -> 5` and `3 -> 4 + 5` evaluated?
2. The `BigInt` class has a `pow` method, not an operator. Why didn't the Scala library designers choose `**` (as in Fortran) or `^` (as in Pascal) for a power operator?

3. Implement the `Fraction` class with operations `+` `-` `*` `/`. Normalize fractions, for example, turning $15/-6$ into $-5/2$. Divide by the greatest common divisor, like this:

```
class Fraction(n: Int, d: Int) :  
    private val num: Int = if d == 0 then 1 else n * sign(d) /  
    gcd(n, d);  
    private val den: Int = if d == 0 then 0 else d * sign(d) /  
    gcd(n, d);  
    override def toString = s"$num/$den"  
    def sign(a: Int) = if a > 0 then 1 else if a < 0 then -1  
    else 0  
    def gcd(a: Int, b: Int): Int = if b == 0 then abs(a) else  
    gcd(b, a % b)  
    ...
```

4. Implement a class `Money` with fields for dollars and cents. Supply `+`, `-` operators as well as comparison operators `==` and `<`. For example, `Money(1, 75) + Money(0, 50) == Money(2, 25)` should be `true`. Should you also supply `*` and `/` operators? Why or why not?

5. Provide operators that construct an HTML table. For example,

```
Table() | "Java" | "Scala" || "Gosling" | "Odersky" || "JVM" |  
"JVM, .NET"
```

should produce

```
<table><tr><td>Java</td><td>Scala</td></tr><tr><td>Gosling...
```

6. Provide a class `ASCIIArt` whose objects contain figures such as

```
/\_\_/\_\_  
( ' ' )  
( - - )  
| | |  
(__|__)
```

Supply operators for combining two ASCII Art figures horizontally

```
/\_/\ \ -----  
( ' ' ) / Hello \  
( - ) < Scala |  
| | | \ Coder /  
(__|__)\ -----
```

or vertically. Choose operators with appropriate precedence.

7. Implement a class `BitSequence` that stores a sequence of 64 bits packed in a `Long` value. Supply `apply` and `update` operators to get and set an individual bit.
8. Provide a class `Matrix`. Choose whether you want to implement 2×2 matrices, square matrices of any size, or $m \times n$ matrices. Supply operations `+` and `*`. The latter should also work with scalars, for example, `mat * 2`. A single element should be accessible as `mat(row, col)`.
9. Define an object `PathComponent`s with an `unapply` operation class that extracts the directory path and file name from an `java.nio.file.Path`. For example, the file `/home/cay/readme.txt` has directory path `/home/cay` and file name `readme.txt`.
10. Modify the `PathComponent` object of the preceding exercise to instead define an `unapplySeq` operation that extracts all path segments. For example, for the file `/home/cay/readme.txt`, you should produce a sequence of three segments: `home`, `cay`, and `readme.txt`.
11. Show that the return type of `unapply` can be an arbitrary subtype of `Product`. Make your own concrete type `MyProduct` that extends `Product` and defines methods `_1`, `_2`. Then define an `unapply` method returning a `MyProduct` instance. What happens if you don't define `_1`?
12. Provide an extractor for strings where the first component is a title from an appropriate enumeration, and the remainder is a sequence of name components. For example, when called with `"Dr. Peter van der Linden"`, the result would be `Some(Title.DR, Seq("Peter", "van", "der", "Linden"))`. Show how the values can be obtained through destructuring or in a `match` expression.

13. Improve the dynamic property selector in [Section 11.11, “Dynamic Invocation,”](#) on page 160 so that one doesn’t have to use underscores. For example, `sysProps.java.home` should select the property with key `"java.home"`. Use a helper class, also extending `Dynamic`, that contains partially completed paths.

14. Define a class `XMLElement` that models an XML element with a name, attributes, and child elements. Using dynamic selection and method calls, make it possible to select paths such as `rootElement.html.body.ul(id="42").li`, which should return all `li` elements inside `ul` with `id` attribute `42` inside `body` inside `html`.

15. Provide an `XMLBuilder` class for dynamically building XML elements, as `builder.ul(id="42", style="list-style: lower-alpha;")`, where the method name becomes the element name and the named arguments become the attributes. Come up with a convenient way of building nested elements.

16. [Section 11.12, “Typesafe Selection and Application,”](#) on page 163 describes a `Request` class that makes requests to `https://$server/$name/$arg`. However, the URLs of real REST APIs aren’t always that regular. Consider <https://www.thecocktailldb.com/api.php>. Change the `Request` class so that it receives a map from method names to template strings, where the `$` character is replaced with the argument value.

For this service, the following should work:

```
val cocktailServiceTemplate = Map(
    "cocktail" ->
        "https://www.thecocktailldb.com/api/json/v1/1/search.php?s=$",
    "ingredient" ->
        "https://www.thecocktailldb.com/api/json/v1/1/search.php?i=$")
val cocktailService =
    Request(cocktailServiceTemplate).asInstanceOf[CocktailService]
```

Now the call

```
cocktailService.cocktail("Negroni")
```

should invoke

```
https://www.thecocktaildb.com/api/json/v1/1/search.php?  
s=Negroni
```

Chapter 12

Higher-Order Functions

Topics in This Chapter **L1**

- [12.1 Functions as Values](#)
- [12.2 Anonymous Functions](#)
- [12.3 Parameters That Are Functions](#)
- [12.4 Parameter Inference](#)
- [12.5 Useful Higher-Order Functions](#)
- [12.6 Closures](#)
- [12.7 Interoperability with Lambda Expressions](#)
- [12.8 Currying](#)
- [12.9 Methods for Composing, Currying, and Tupling](#)
- [12.10 Control Abstractions](#)
- [12.11 The `return` Expression](#)
- [Exercises](#)

Scala mixes object orientation with functional features. In a functional programming language, functions are first-class citizens that can be passed around and manipulated just like any other data types. This is very useful whenever you want to pass some action detail to an algorithm. In a

functional language, you just wrap that detail into a function that you pass as a parameter. In this chapter, you will see how to be productive with functions that use or return functions.

Highlights of the chapter include:

- Functions are “first-class citizens” in Scala, just like numbers.
- You can create anonymous functions, usually to give them to other functions.
- A function argument specifies behavior that should be executed later.
- Many collection methods take function parameters, applying a function to the values of the collection.
- There are syntax shortcuts that allow you to express function parameters in a way that is short and easy to read.
- You can create functions that operate on blocks of code and look much like the built-in control statements.

12.1 Functions as Values

In Scala, a function is a first-class citizen, just like a number. You can store a function in a variable:

```
import scala.math.*  
val num = 3.14  
val fun = ceil
```

This code sets `num` to `3.14` and `fun` to the `ceil` function.

When you try this code in the REPL, the type of `num` is, not surprisingly, `Double`. The type of `fun` is reported as `(Double) => Double`—that is, a function receiving and returning a `Double`.

What can you do with a function? Two things:

- Call it.
- Pass it around, by storing it in a variable or giving it to a function as a parameter.

Here is how to call the function stored in `fun`:

```
fun(num) // 4.0
```

As you can see, the normal function call syntax is used. The only difference is that `fun` is a *variable containing a function*, not a fixed function.

Here is how you can give `fun` to another function:

```
Array(3.14, 1.42, 2.0).map(fun) // Array(4.0, 2.0, 2.0)
```

The `map` method accepts a function, applies it to all values in an array, and returns an array with the function values. In this chapter, you will see many other methods that accept functions as parameters.

12.2 Anonymous Functions

In Scala, you don't have to give a name to each function, just like you don't have to give a name to each number. Here is an *anonymous function*:

```
(x: Double) => 3 * x
```

This function multiplies its argument by 3.

Of course, you can store this function in a variable:

```
val triple = (x: Double) => 3 * x
```

That's just as if you had used a `def`:

```
def triple(x: Double) = 3 * x
```

But you don't have to name the function. You can just pass it to another function:

```
Array(3.14, 1.42, 2.0).map((x: Double) => 3 * x)  
// Array(9.42, 4.26, 6.0)
```

Here, we tell the `map` method: "Multiply each element by 3."

 **Note**

If you prefer, you can enclose the function argument in braces instead of parentheses, for example:

```
Array(3.14, 1.42, 2.0).map{ (x: Double) => 3 * x }
```

This is more common when a method is used in infix notation (without the dot).

```
Array(3.14, 1.42, 2.0) map { (x: Double) => 3 * x }
```

 **Note**

As noted in [Chapter 2](#), some people consider anything that is declared with `def` to be a method. However, in this book, we use a conceptually cleaner mental model. Methods are invoked with a `this` argument. They are members of classes, traits, or objects. Top-level and block-level `def` statements declare functions.

In those cases, you cannot tell whether `def` declares anything other than a function. The *expression* `triple` is indistinguishable from the function `(x: Double) => 3 * x`. (This conversion is called `eta` expansion, using terminology from the lambda calculus.)

 **Caution**

There is only one exception to eta expansion—when there are no parameters:

```
def heads() = scala.math.random() < 0.5
```

For arcane reasons, `heads` is a syntax error. The function is `heads_`, and it is invoked as `heads()`.

12.3 Parameters That Are Functions

In this section, you will see how to implement a function that takes another function as a parameter. Here is an example:

```
def valueAtOneQuarter(f: (Double) => Double) = f(0.25)
```

Note that the parameter can be *any* function receiving and returning a Double. The `valueAtOneQuarter` function computes the value of that function at 0.25.

For example,

```
valueAtOneQuarter(ceil) // 1.0  
valueAtOneQuarter(sqrt) // 0.5 (because 0.5 × 0.5 = 0.25)
```

What is the type of `valueAtOneQuarter`? It is a function with one parameter, so its type is written as

$(\text{parameterType}) \Rightarrow \text{resultType}$

The `resultType` is clearly `Double`, and the `parameterType` is already given in the function header as `(Double) => Double`. Therefore, the type of `valueAtOneQuarter` is

```
((Double) => Double) => Double
```

Since `valueAtOneQuarter` is a function that receives a function, it is called a *higher-order function*.

A higher-order function can also *produce a function*. Here is a simple example:

```
def mulBy(factor : Double) = (x : Double) => factor * x
```

For example, `mulBy(3)` returns the function `(x : Double) => 3 * x` which you have seen in the preceding section. The power of `mulBy` is that it can deliver functions that multiply by any amount:

```
val quintuple = mulBy(5)
quintuple(20) // 100
```

The `mulBy` function has a parameter of type `Double`, and it returns a function of type `(Double) => Double`. Therefore, its type is

```
(Double) => ((Double) => Double)
```

12.4 Parameter Inference

When you pass an anonymous function to another function or method, Scala helps you out by deducing types when possible. For example, you don't have to write

```
valueAtOneQuarter((x: Double) => 3 * x) // 0.75
```

Since the `valueAtOneQuarter` method knows that you will pass in a `(Double) => Double` function, you can just write

```
valueAtOneQuarter(x => 3 * x)
```

As a special bonus, for a function that has just one parameter, you can omit the `()` around the parameter:

```
valueAtOneQuarter(x => 3 * x)
```

It gets better. If a parameter occurs only once on the right-hand side of the `=>`, you can replace it with an underscore:

```
valueAtOneQuarter(3 * _)
```

This is the ultimate in comfort, and it is also pretty easy to read: a function that multiplies something by 3.

Keep in mind that these shortcuts only work when the parameter types are known.

```
val fun = 3 * _ // Error: Can't infer types
```

You can specify a type for the anonymous parameter or for the variable:

```
3 * (_: Double) // OK  
val fun: (Double) => Double = 3 * _ // OK because we specified the  
type for fun
```

Of course, the last definition is contrived. But it shows what happens when a function is passed to a parameter (which has just such a type).



Tip
Specifying the type of `_` is useful for turning methods into functions. For example, `(_: String).length` is a function `String => Int`, and `(_: String).substring(_:_Int, _: Int)` is a function `(String, Int, Int) => String`.

12.5 Useful Higher-Order Functions

A good way of becoming comfortable with higher-order functions is to practice with some common (and obviously useful) methods in the Scala collections library that take function parameters.

You have seen `map`, which applies a function to all elements of a collection and returns the result. Here is a quick way of producing a collection containing `0.1, 0.2, ..., 0.9`:

```
(1 to 9).map(0.1 * _)
```



Note
There is a general principle at work. If you want a sequence of values, see if you can transform it from a simpler one.

Try this to print a triangle:

```
(1 to 9).map(" *" * _).foreach(println _)
```

The result is

```
*  
**  
***  
****  
*****  
*****  
*****  
*****  
*****  
*****
```

Here, we also use `foreach`, which is like `map` except that its function doesn't return a value. The `foreach` method simply applies the function to each argument.

The `filter` method yields all elements that match a particular condition. For example, here's how to get only the even numbers in a sequence:

```
(1 to 9).filter(_ % 2 == 0) // 2, 4, 6, 8
```

Of course, that's not the most efficient way of getting this result 😊.

The `reduceLeft` method takes a *binary* function—that is, a function with two parameters—and applies it to all elements of a sequence, going from left to right. For example,

```
(1 to 9).reduceLeft(_ * _)
```

is

```
1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9
```

or, strictly speaking,

```
(...((1 * 2) * 3) * ... * 9)
```

Note the compact form of the multiplication function: `_ * _`. Each underscore denotes a separate parameter.

You also need a binary function for sorting. For example,

```
"Mary had a little lamb".split(" ").sortWith(_.length < _.length)
```

yields an array that is sorted by increasing length: `Array("a", "had", "Mary", "lamb", "little")`.

12.6 Closures

In Scala, you can define a function inside any scope: in a package, in a class, or even inside another function or method. In the body of a function, you can access any variables from an enclosing scope. That may not sound so remarkable, but note that your function may be called when the variable is *no longer in scope*.

Here is an example: the `mulBy` function from [Section 12.3, “Parameters That Are Functions,” on page 171](#).

```
def mulBy(factor : Double) = (x : Double) => factor * x
```

Consider these calls:

```
val triple = mulBy(3)
val half = mulBy(0.5)
println(s"${triple(14)} ${half(14)}") // Prints 42 7
```

Let's look at them in slow motion.

1. The first call to `mulBy` sets the parameter variable `factor` to 3. That variable is referenced in the body of the function `(x : Double) => factor * x`, which is stored in `triple`. Then the parameter variable `factor` is popped off the runtime stack.
2. Next, `mulBy` is called again, now with `factor` set to 0.5. That variable is referenced in the body of the function `(x : Double) => factor * x`, which is stored in `half`.

Each of the returned functions has its own setting for `factor`.

Such a function is called a *closure*. A closure consists of code together with the definitions of any nonlocal variables that the code uses.

These functions are actually implemented as objects of a class, with an instance variable `factor` and an `apply` method that contains the body of the function.

It doesn't really matter how a closure is implemented. It is the job of the Scala compiler to ensure that your functions can access nonlocal variables.

 **Note**

Closures aren't difficult or surprising if they are a natural part of the language. Many modern languages, such as JavaScript, Ruby, and Python, support closures. Java, as of version 8, has closures in the form of lambda expressions.

12.7 Interoperability with Lambda Expressions

In Scala, you pass a function as a parameter whenever you want to tell another function what action to carry out. In Java, you use a lambda expression:

```
var button = new JButton("Increment"); // This is Java  
button.addActionListener(event -> counter++);
```

In order to pass a lambda expression, the parameter type must be a “functional interface”—that is, any Java interface with a single abstract method.

You can pass a Scala function to a Java functional interface:

```
val button = JButton("Increment")
```

```
button.addActionListener(event => counter += 1)
```

Note that the conversion from a Scala function to a Java functional interface only works for *function literals*, not for variables holding functions. The following does not work:

```
val listener = (event: ActionEvent) => println(counter)
button.addActionListener(listener)
// Cannot convert a nonliteral function to a Java functional interface
```

The simplest remedy is to declare the variable holding the function as a Java functional interface:

```
val listener: ActionListener = event => println(counter)
button.addActionListener(listener) // Ok
```

Alternatively, you can turn a function variable into a literal expression:

```
val exit = (event: ActionEvent) => if counter > 9 then
System.exit(0)
button.addActionListener(exit(_))
```

12.8 Currying

Currying (named after logician Haskell Brooks Curry) is the process of turning a function that takes two arguments into a function that takes one argument. That function returns a function that consumes the second argument.

Huh? Let's look at an example. This function takes two arguments:

```
val mul = (x: Int, y: Int) => x * y
```

This function takes one argument, yielding a function that takes one argument:

```
val mulOneAtATime = (x: Int) => ((y: Int) => x * y)
```

To multiply two numbers, you call

```
mulOneAtATime(6)(7)
```

Strictly speaking, the result of `mulOneAtATime(6)` is the function `(y: Int) => 6 * y`. That function is applied to `7`, yielding `42`.

When you use `def`, there is a shortcut for defining such curried methods in Scala:

```
def divOneAtATime(x: Int)(y: Int) = x / y
```

As you can see, multiple parameters are just a frill, not an essential feature of a programming language. That's an amusing theoretical insight, but it has one practical use in Scala. Sometimes, you can use currying for a method parameter so that the type inferencer has more information.

Here is a typical example. The `corresponds` method can compare whether two sequences are the same under some comparison criterion. For example,

```
val a = Array("Mary", "had", "a", "little", "lamb")
val b = Array(4, 3, 1, 6, 5)
a.corresponds(b)(_.length == _)
```

Note that the function `_.length == _` is passed as a curried parameter, in a separate set of `(...)`. When you look into the Scaladoc, you will see that `corresponds` is declared as

```
def corresponds[B](that: Seq[B])(p: (A, B) => Boolean): Boolean
```

The `that` sequence and the predicate function `p` are separate curried parameters. The type inferencer can figure out what `B` is from the type of `that`, and then it can use that information when analyzing the function `that` is passed for `p`.

In our example, `that` is an `Int` sequence. Therefore, the predicate is expected to have type `(String, Int) => Boolean`. With that information, the compiler can accept `_.length == _` as a shortcut for `(a: String, b: Int) => a.length == b`.

12.9 Methods for Composing, Currying, and Tupling

A unary function is a function with one parameter. All unary functions are instances of the `Function1` trait. That trait defines a method for composing two unary functions; that is, executing one and then passing the result to the other.

Suppose we want to be sure that we don't compute square roots of negative values. Then we can compose the square root function with the absolute value function:

```
val sqrt = scala.math.sqrt
val fabs: Double => Double = scala.math.abs
    // Need the type because abs is overloaded
val f = sqrt compose fabs // Infix notation; could also write
  sqrt.compose(fabs)
f(-9) // Yields sqrt(fabs(-9)) or 3
```

Note that the second function is executed first, because functions are applied right to left. If you find that unnatural, use the `andThen` method instead:

```
val g = fabs andThen sqrt // The same as sqrt compose fabs
```

Functions with more than one parameter have a `curried` method, producing the curried version of a function.

```
val fmax : (Double, Double) => Double = scala.math.max
    // Need the type because max is overloaded
fmax.curried // Has type Double => Double => Double
val h = fmax.curried(0)
```

The function `h` has one parameter, and `h(x) = fmax.curried(0)(x) = fmax(0, x)`. Positive arguments are returned, and negative arguments yield zero.

The `tupled` method turns a function with more than one parameter into a unary function receiving a tuple.

```
val k = mul.tupled // Has type ((Int, Int)) => Int
```

When calling `k`, you supply a pair, whose components are passed to `mul`. For example, `k((6, 7))` is 42.

This sounds abstract, but here is a practical use. Let's say you have two arrays, holding the first and second arguments.

```
val xs = Array(1, 7, 2, 9)
val ys = Array(1000, 100, 10, 1)
```

Now we want to pass corresponding elements to a binary function such as `mul`. An elegant solution is to zip the arrays, and then apply the tupled function to the pairs:

```
xs.zip(ys).map(mul.tupled) // Yields Array(1000, 700, 20, 9)
```



Note

If you pass a function *literal*, then you don't need to call `tupled`. The elements of the zipped array are untupled, and the components are passed to the function:

```
xs.zip(ys).map(_ * _) // The tuple components are passed to
the function
```

12.10 Control Abstractions

In Scala, one can model a sequence of statements as a function with no parameters or return value. For example, here is a function that runs some code in a thread:

```

def runInThread(block: () => Unit) =
  (new Thread :
    override def run() = block()
  ).start()

```

The code is given as a function of type `() => Unit`. However, when you call this function, you need to supply an unsightly `() =>`:

```

runInThread { () => println("Hi"); Thread.sleep(10000);
  println("Bye") }

```

To avoid the `() =>` in the call, use the *call by name* notation: Omit the `()`, but not the `=>`, in the parameter declaration and in the call to the parameter function:

```

def runInThread(block: => Unit) =
  (new Thread :
    override def run() = block
  ).start()

```

Then the call becomes simply

```

runInThread { println("Hi"); Thread.sleep(10000); println("Bye") }

```

This looks pretty nice. Scala programmers can build *control abstractions*: functions that look like language keywords. For example, we can implement a function that is used *exactly* as a `while` statement. Or, we can innovate a bit and define an `until` statement that works like `while`, but with an inverted condition:

```

def until(condition: => Boolean)(block: => Unit): Unit =
  if !condition then
    block
    until(condition)(block)

```

Here is how you use `until`:

```
var x = 10
until (x == 0) {
    x -= 1
    println(x)
}
```

The technical term for such a function parameter is a *call-by-name* parameter. Unlike a regular (or call-by-value) parameter, the parameter expression is *not* evaluated when the function is called. After all, we don't want `x == 0` to evaluate to `false` in the call to `until`. Instead, the expression becomes the body of a function with no arguments. That function is passed as a parameter.

Look carefully at the `until` function definition. Note that it is curried: It first consumes the `condition`, then the `block` as a second parameter. Without currying, the call would look like this:

```
until(x == 0, { ... })
```

which wouldn't be as pretty.

12.11 The `return` Expression

In Scala, you don't use a `return` statement to return function values. The return value of a function is simply the value of the function body.

However, you can use `return` to return a value from an anonymous function to an enclosing named function. This is useful in control abstractions. For example, consider this function:

```
def indexOf(str: String, ch: Char): Int =
    var i = 0
    until (i == str.length) {
        if str(i) == ch then return i
        i += 1
    }
    -1
```

Here, the anonymous function `{ if str(i) == ch then return i; i += 1 }` is passed to `until`. When the `return` expression is executed, the enclosing named function `indexOf` terminates and returns the given value.

If you use `return` inside a named function, you need to specify its return type. For example, in the `indexOf` function above, the compiler was not able to infer that it returns an `Int`.

The control flow is achieved with a special exception that is thrown by the `return` expression in the anonymous function, passed out of the `until` function, and caught in the `indexOf` function.



Caution

If the exception is caught in a `try` block, before it is delivered to the named function, then the value will not be returned.

Exercises

1. Write a function `values(fun: (Int) => Int, low: Int, high: Int)` that yields a collection of function inputs and outputs in a given range. For example, `values(x => x * x, -5, 5)` should produce a collection of pairs `(-5, 25), (-4, 16), (-3, 9), ..., (5, 25)`.
2. How do you get the largest element of an array with `reduceLeft`?
3. Implement the factorial function using `to` and `reduceLeft`, without a loop or recursion.
4. The previous implementation needed a special case when $n < 1$. Show how you can avoid this with `foldLeft`. (Look at the Scaladoc for `foldLeft`. It's like `reduceLeft`, except that the first value in the chain of combined values is supplied in the call.)
5. Write a function `largest(fun: (Int) => Int, inputs: Seq[Int])` that yields the largest value of a function within a given sequence of inputs. For example, `largest(x => 10 * x - x * x, 1 to 10)` should return `25`. Don't use a loop or recursion.

6. Modify the previous function to return the *input* at which the output is largest. For example, `largestAt(x => 10 * x - x * x, 1 to 10)` should return 5. Don't use a loop or recursion.

7. Write a function that composes two functions of type `Double => Option[Double]`, yielding another function of the same type. The composition should yield `None` if either function does. For example,

```
def f(x: Double) = if x != 1 then Some(1 / (x - 1)) else None
def g(x: Double) = if x >= 0 then Some(sqrt(x)) else None
val h = compose(g, f) // h(x) should be g(f(x))
```

Then `h(2)` is `Some(1)`, and `h(1)` and `h(0)` are `None`.

8. [Section 12.9](#), “Methods for Composing, Currying, and Tupling,” on page 177 covers the `composing`, `currying`, and `tupling` methods that all functions have. Implement these from scratch, as functions that operate on integer functions. For example, `tupling(mul)` returns a function `((Int, Int)) => Int`, and `tupling(mul)((6, 7))` is 42.

9. In [Section 12.8](#), “Currying,” on page 176, you saw the `corresponds` method used with two arrays of strings. Make a call to `corresponds` that checks whether the elements in an array of strings have the lengths given in an array of integers.

10. Implement `corresponds` without currying. Then try the call from the preceding exercise. What problem do you encounter?

11. Implement an `unless` control abstraction that works just like `if`, but with an inverted condition. Does the first parameter need to be a call-by-name parameter? Do you need currying?

Chapter 13

Collections

Topics in This Chapter A2

- 13.1 The Main Collections Traits
- 13.2 Mutable and Immutable Collections
- 13.3 Sequences
- 13.4 Lists
- 13.5 Sets
- 13.6 Operators for Adding or Removing Elements
- 13.7 Common Methods
- 13.8 Mapping a Function
- 13.9 Reducing, Folding, and Scanning A3
- 13.10 Zipping
- 13.11 Iterators
- 13.12 Lazy Lists A3
- 13.13 Interoperability with Java Collections
- Exercises

In this chapter, you will learn about the Scala collections library from a library user's point of view. In addition to arrays and maps, which you have already encountered, you will see other useful collection types. There are many methods

that can be applied to collections, and this chapter presents them in an orderly way.

The key points of this chapter are:

- All collections extend the `Iterable` trait.
- The three major categories of collections are sequences, sets, and maps.
- Scala has mutable and immutable versions of most collections.
- A Scala list is either empty, or it has a head and a tail which is again a list.
- Sets are unordered collections.
- Use a `LinkedHashSet` to retain the insertion order or a `SortedSet` to iterate in sorted order.
- `+` adds an element to an unordered collection; `:+` and `:+` prepend or append to a sequence; `++` concatenates two collections; `-` and `--` remove elements.
- The `Iterable` and `Seq` traits have dozens of useful methods for common operations. Check them out before writing tedious loops.
- Mapping, folding, and zipping are useful techniques for applying a function or operation to the elements of a collection.
- You can consider an `Option` to be a collection of size 0 or 1.
- A lazy list is evaluated on demand and can hold an unbounded number of elements.

13.1 The Main Collections Traits

Figure 13–1 shows the most important traits that make up the Scala collections hierarchy.

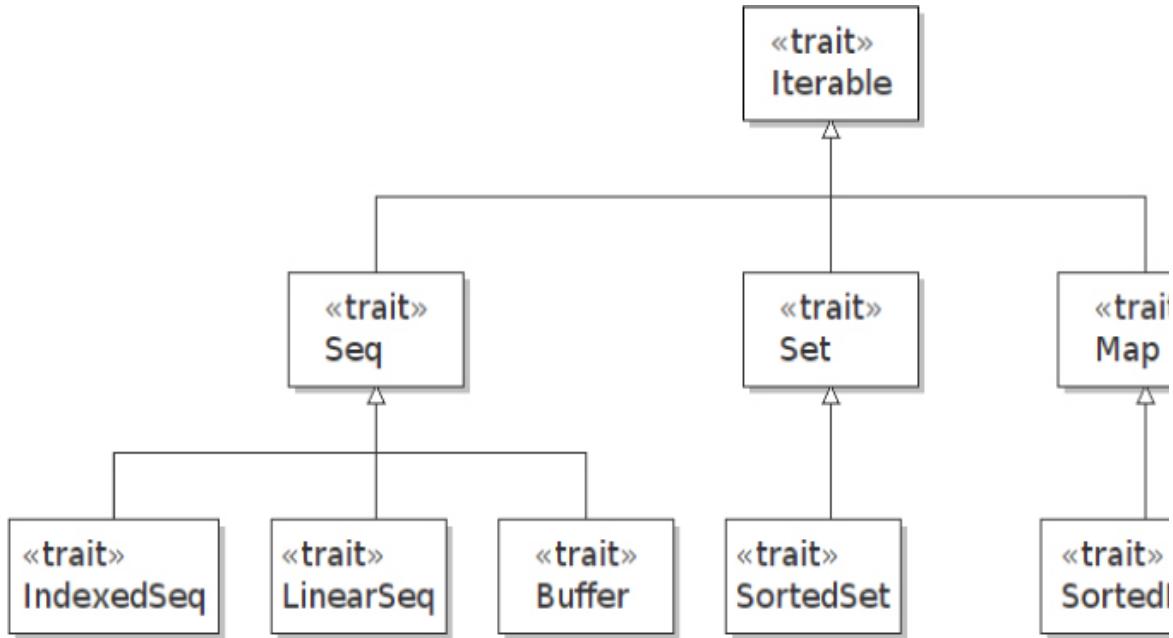


Figure 13–1 Key traits in the Scala collections hierarchy

An `Iterable` is any collection that can yield an `Iterator` with which you can access all elements in the collection:

```

val coll = ... // some Iterable
val iter = coll.iterator
while iter.hasNext do
  process(iter.next())
  
```

In a `Seq`, the elements are ordered, and the iterator visits the elements in order. An `IndexedSeq` allows fast random access through an integer index. For example, an `ArrayBuffer` is indexed but a linked list is not. A `LinearSeq` allows fast head and tail operations. A `Buffer` allows fast insertion and removal at the end. An `ArrayBuffer` is both a `Buffer` and, as already mentioned, an `IndexedSeq`.

A `Set` is an unordered collection of values. In a `SortedSet`, elements are always visited in sorted order.

A `Map` is a set of `(key, value)` pairs. A `SortedMap` visits the entries as sorted by the keys. See [Chapter 4](#) for more information.

This hierarchy is similar to that in Java, with a couple of welcome improvements:

1. Maps are a part of the hierarchy and not a separate hierarchy.
2. `IndexedSeq` is the supertype of arrays but not of lists, allowing you to tell the two apart.

 **Note**

In Java, both `ArrayList` and `LinkedList` implement a common `List` interface, making it difficult to write efficient code when random access is preferred, for example when searching in a sorted sequence. This was a flawed design decision in the original Java collections framework. In a later version, a marker interface `RandomAccess` was added to deal with this problem.

Each Scala collection trait has a companion object with an `apply` method for constructing an instance of the collection. For example,

```
Iterable(0xFF, 0xFF00, 0xFF0000)
Set(Color.RED, Color.GREEN, Color.BLUE)
Map(Color.RED -> 0xFF0000, Color.GREEN -> 0xFF00, Color.BLUE -> 0xFF)
SortedSet("Hello", "World")
```

This is called the “uniform creation principle.”

There are methods `toSeq`, `toSet`, `toMap`, and so on, as well as a generic `to` method, that you can use to translate between collection types.

```
val coll = Seq(1, 1, 2, 3, 5, 8, 13)
val set = coll.toSet
val buffer = coll.to(ArrayBuffer)
```

 **Note**

You can use the `==` operator to compare any sequence, set, or map with another collection of the same kind. For example, `Seq(1, 2, 3) == (1 to 3)` yields `true`. But comparing different kinds, for example, `Seq(1, 2, 3) == Set(1, 2, 3)` is a compilation error. In that case, use the `sameElements` method.

13.2 Mutable and Immutable Collections

Scala supports both mutable and immutable collections. An immutable collection can never change, so you can safely share a reference to it, even in a multi-threaded program. For example, there is a `scala.collection.mutable.Map` and a `scala.collection.immutable.Map`. Both have a common supertype `scala.collection.Map` (which, of course, contains no mutation operations).

Note

When you have a reference to a `scala.collection.immutable.Map`, you know that *nobody* can change the map. If you have a `scala.collection.Map`, then *you* can't change it, but someone else might.

Scala gives a preference to immutable collections. The `scala` package and the `Predef` object, which are always imported, have type aliases `Seq`, `IndexedSeq`, `List`, `Set`, and `Map` that refer to the immutable traits. For example, `Predef.Map` is the same as `scala.collection.immutable.Map`.

Tip

With the statement

```
import scala.collection.mutable
```

you can get an immutable map as `Map` and a mutable one as `mutable.Map`.

If you had no prior experience with immutable collections, you may wonder how you can do useful work with them. The key is that you can create new collections out of old ones. For example, if `numbers` is an immutable set, then

```
numbers + 9
```

is a new set containing the `numbers` together with `9`. If `9` was already in the set, you just get a reference to the old set. This is particularly natural in recursive computations. For example, here we compute the set of all digits of an integer:

```

def digits(n: Int): Set[Int] =
  if n < 0 then digits(-n)
  else if n < 10 then Set(n)
  else digits(n / 10) + (n % 10)

```

This method starts out with a set containing a single digit. At each step, another digit is added. However, adding the digit doesn't mutate a set. Instead, in each step, a new set is constructed.



In addition to the `scala.collection.immutable` and `scala.collection.mutable` package, there is a `scala.collection.concurrent` package. It has a `Map` trait with methods for atomic processing: `putIfAbsent`, `remove`, `replace`, `getOrElseUpdate`, `updateWith`. A `TrieMap` implementation is provided. You can also adapt a Java `ConcurrentHashMap` to this trait—see [Section 13.13, “Interoperability with Java Collections,”](#) on page 204.

13.3 Sequences

[Figure 13–2](#) shows the most important immutable sequences.

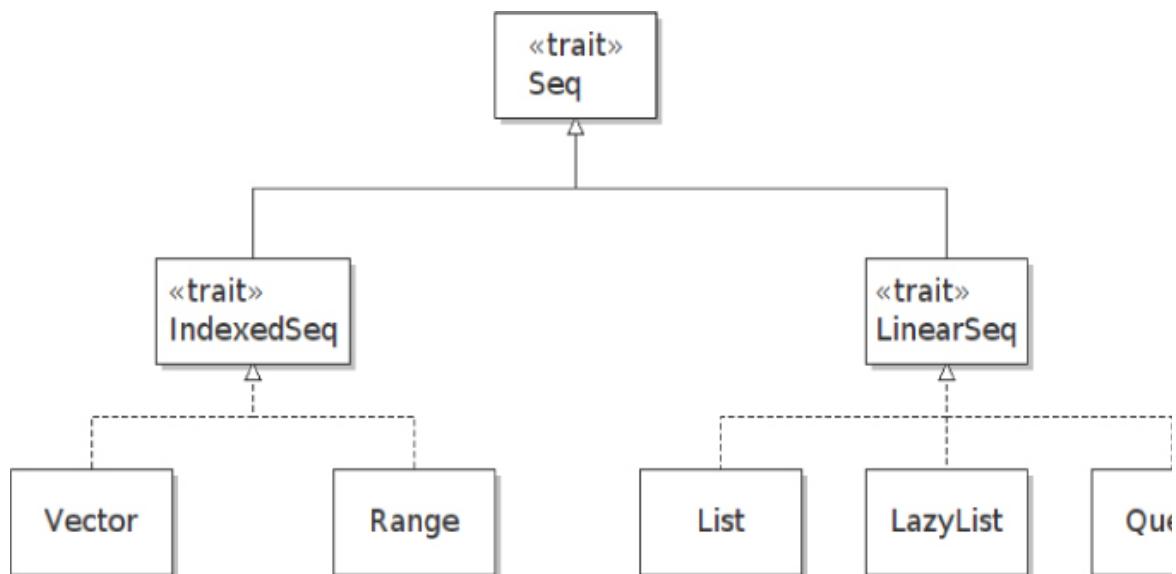


Figure 13–2 Immutable sequences

A `Vector` is the immutable equivalent of an `ArrayBuffer`: an indexed sequence with fast random access. Vectors are implemented as trees where each node has up to 32 children. For a vector with one million elements, one needs four layers of nodes. (Since $10^3 \gg 2^{10}$, $10^6 \gg 32^4$.) Accessing an element in such a list will take 4 hops, whereas in a linked list it would take an average of 500,000.

A `Range` represents an integer sequence, such as `0,1,2,3,4,5,6,7,8,9` or `10,20,30`. Of course a `Range` object doesn't store all sequence values but only the start, end, and increment. You construct `Range` objects with the `to` and `until` methods, as described in [Chapter 2](#).

We discuss lists in the next section, and lazy lists in [Section 13.12, “Lazy Lists,”](#) on page 202.

See [Figure 13–3](#) for the most useful mutable sequences.

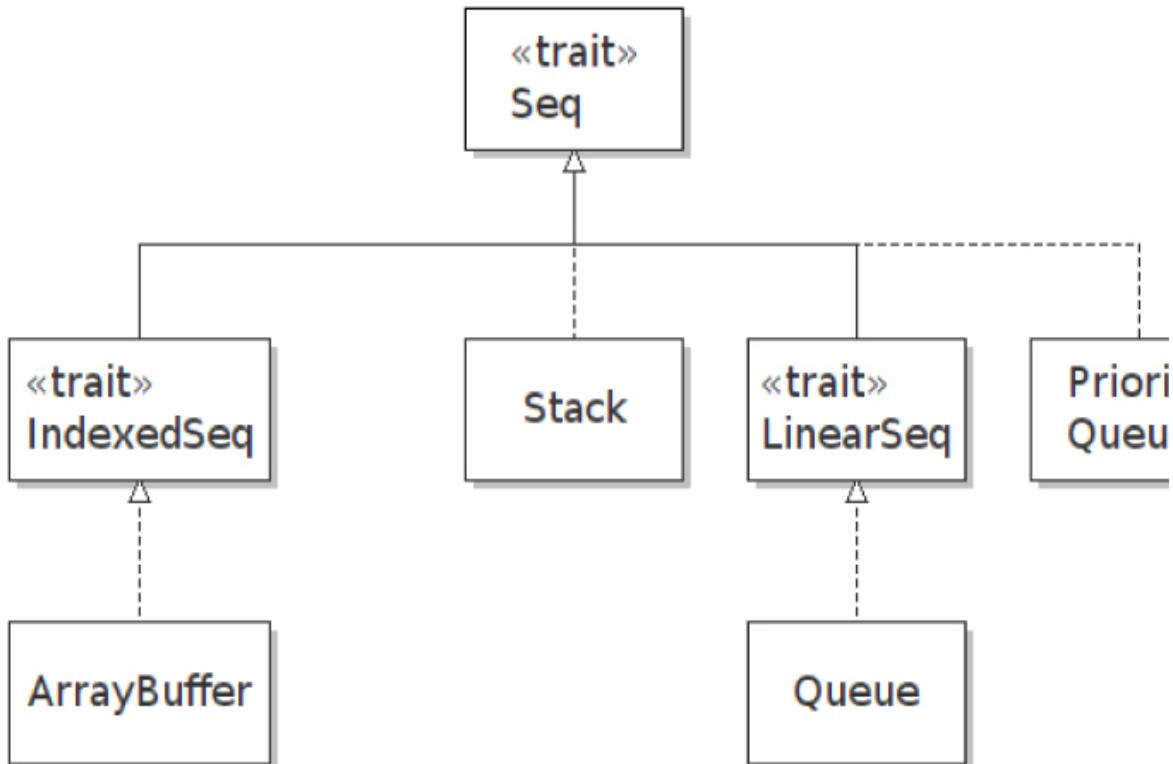


Figure 13–3 Mutable sequences

We discussed array buffers in [Chapter 3](#). Stacks, queues, and priority queues are standard data structures that are useful for implementing certain algorithms. If you are familiar with these structures, the Scala implementations won't surprise you.

13.4 Lists

In Scala, a list is either `Nil` (that is, empty) or an object with a `head` element and a `tail` that is again a list. For example, consider the list

```
val digits = List(4, 2)
```

The value of `digits.head` is 4, and `digits.tail` is `List(2)`. Moreover, `digits.tail.head` is 2 and `digits.tail.tail` is `Nil`.

The `::` operator makes a new list from a given head and tail. For example,

```
9 :: List(4, 2)
```

is `List(9, 4, 2)`. You can also write that list as

```
9 :: 4 :: 2 :: Nil
```

Note that `::` is right-associative. With the `::` operator, lists are constructed from the end:

```
9 :: (4 :: (2 :: Nil))
```



Note

There is a similar right-associative operator to build tuples:

```
1 *: 2.0 *: "three" *: EmptyTuple
```

yields the tuple `(1, 2.0, "three")`.

In Java or C++, one uses an iterator to traverse a linked list. You can do this in Scala as well, but it is often more natural to use recursion. For example, the following function computes the sum of all elements in a linked list of integers:

```
def sum(lst: List[Int]): Int =
  if lst == Nil then 0 else lst.head + sum(lst.tail)
```

Or, if you prefer, you can use pattern matching:

```
def sum(lst: List[Int]): Int = lst match
  case Nil => 0
  case h :: t => h + sum(t) // h is lst.head, t is lst.tail
```

Note the `::` operator in the second pattern. It “destructures” the list into head and tail.

Note

Recursion works so naturally because the tail of a list is again a list.

Of course, for this particular example, you do not need to use recursion at all. The Scala library already has a `sum` method:

```
List(9, 4, 2).sum // Yields 15
```

If you want to mutate list elements in place, you can use a `ListBuffer`, a data structure that is backed by a linked list with references to the first and last node. This makes it efficient to add or remove elements at either end of the list.

However, adding or removing elements in the middle is not efficient. For example, suppose you want to remove every second element of a mutable list. With a Java `LinkedList`, you use an iterator and call `remove` after every second call to `next`. There is no analogous operation on a `ListBuffer`. Of course, removing multiple elements by their index positions is very inefficient in a linked list. Your best bet is to generate a new list with the result (see [Exercise 3](#) on page 205).

13.5 Sets

A set is a collection of distinct elements. Trying to add an existing element has no effect. For example,

```
Set(2, 0, 1) + 1
```

is the same as `Set(2, 0, 1)`.

Unlike lists, sets do not retain the order in which elements are inserted. By default, sets are implemented as *hash sets* in which elements are organized by the

value of the `hashCode` method. (In Scala, as in Java, every object has a `hashCode` method.)

For example, if you iterate over

```
Set(1, 2, 3, 4, 5, 6)
```

the elements are visited in the order

```
5 1 6 2 3 4
```

You may wonder why sets don't retain the element order. It turns out that you can find elements much faster if you allow sets to reorder their elements. Finding an element in a hash set is *much* faster than in an array or list.

A *linked hash set* remembers the order in which elements were inserted. It keeps a linked list for this purpose. For example,

```
val weekdays = scala.collection.mutable.LinkedHashSet("Mo", "Tu",
"We", "Th", "Fr")
```

If you want to iterate over elements in sorted order, use a *sorted set*:

```
val numbers = scala.collection.mutable.SortedSet(1, 2, 3, 4, 5, 6)
```

A *bit set* is an implementation of a set of non-negative integers as a sequence of bits. The i th bit is 1 if i is present in the set. This is an efficient implementation as long as the maximum element is not too large. Scala provides both mutable and immutable `BitSet` classes.

The `contains` method checks whether a set contains a given value. The `subsetOf` method checks whether all elements of a set are contained in another set.

```
val digits = Set(1, 7, 2, 9)
digits.contains(0) // false
Set(1, 2).subsetOf(digits) // true
```

The `union`, `intersect`, `and` `diff` methods carry out the usual set operations. If you prefer, you can write them as `|`, `&`, and `&~`. You can also write `union` as `++` and `difference` as `--`. For example, if we have the set

```
val primes = Set(2, 3, 5, 7)
```

then `digits.union(primes)` is `Set(1, 2, 3, 5, 7, 9)`, `digits & primes` is `Set(2, 7)`, and `digits -- primes` is `Set(1, 9)`.

13.6 Operators for Adding or Removing Elements

When you want to add or remove an element, or a number of elements, the operators to use depend on the collection type. [Table 13–1](#) provides a summary.

Table 13–1 Operators for Adding and Removing Elements

Operator	Description	Collection Type
<code>coll :+ elem</code> <code>elem +: coll</code>	A collection of the same type as <code>coll</code> to which <code>elem</code> has been appended or prepended.	Seq
<code>coll + elem</code> <code>coll - elem</code>	A collection of the same type as <code>coll</code> with the given element added or removed.	Immutable Set, Map
<code>coll ++ coll2</code>	A collection of the same type as <code>coll</code> , containing the elements of both collections.	Iterable
<code>coll -- coll2</code>	A collection of the same type as <code>coll</code> from which the elements of <code>coll2</code> have been removed. (For sequences, use <code>diff.</code>)	Immutable Set, Map
<code>elem :: lst</code> <code>lst2 :::: lst</code>	A list with the element or given list prepended to <code>lst</code> . Same as <code>+:</code> and <code>++:</code>	List
<code>set set2</code> <code>set & set2</code> <code>set &~ set2</code>	Set union, intersection, difference. <code> </code> is the same as <code>++</code> , and <code>&~</code> is the same as <code>--</code> .	Set
<code>coll += elem</code> <code>coll ++= coll2</code> <code>coll -= elem</code> <code>coll --= coll2</code>	Modifies <code>coll</code> by adding or removing the given elements.	Mutable collection
<code>elem +=: coll</code> <code>coll2 ++=: coll</code>	Modifies <code>coll</code> by prepending the given element or collection, alias for <code>prepend</code> .	Buffer

Note that `+` is used for adding an element to an unordered immutable collection, while `+:` and `:+` add an element to the beginning or end of an ordered collection.

```
Vector(1, 2, 3) :+ 5 // Yields Vector(1, 2, 3, 5)  
0 +: 1 +: Vector(1, 2, 3) // Yields Vector(0, 1, 1, 2, 3)
```

Note that `+:`, like all operators ending in a colon, is right-associative, and that it is a method of the right operand.

These operators return new collections (of the same type as the original ones) without modifying the original. Mutable collections have a `+=` operator that mutates the left-hand side. For example,

```
val numberBuffer = ArrayBuffer(1, 2, 3)  
numberBuffer += 5 // Adds 5 to numberBuffer
```

With an immutable collection, you can use `+=` or `:+=` with a `var`, like this:

```
var numberSet = Set(1, 2, 3)  
numberSet += 5 // Sets numberSet to the immutable set numberSet + 5  
var numberVector = Vector(1, 2, 3)  
numberVector :+= 5 // += does not work since vectors don't have a +  
operator
```

To remove an element, use the `-` operator:

```
Set(1, 2, 3) - 2 // Yields Set(1, 3)
```

You can add multiple elements with the `++` operator:

```
coll ++ coll2
```

yields a collection of the same type as `coll` that contains both `coll` and `coll2`. Similarly, the `--` operator removes multiple elements.



For lists, you can use `+:` instead of `::` for consistency, with one exception: Pattern matching (`case h :: t`) does *not* work with the `+:` operator.



Caution

The `+` and `-` operators are deprecated for mutable collections, as well as collections of unknown mutability (such as `scala.collection.Set`). They don't mutate the collection but compute new collections with elements added or removed. If you really want to do this with a (potentially) mutable collection, use the `++` and `--` operators.



Caution

The operators `coll += (e1, e2, ...)` and `coll -= (e1, e2, ...)` with multiple arguments are deprecated, since infix operators with more than two arguments are now discouraged.

As you can see, Scala provides many operators for adding and removing elements. Here is a cheat sheet:

1. Append `(:+)` or prepend `(+::)` to a sequence.
2. Add `(+)` to an unordered collection.
3. Remove with the `-` operator.
4. Use `++` and `--` for bulk add and remove.
5. Mutations are `+=`, `++=`, `--`, `---`.
6. For lists, many Scala programmers prefer the `::` and `:::` operators to `:+` and `++`.
7. Stay away from deprecated `+= (e1, e2, ...)`, `-= (e1, e2, ...)`, `++:` and enigmatic `+:=`, `++=::`.

13.7 Common Methods

[Table 13–2](#) gives a brief overview of the most important methods of the `Iterable` trait, sorted by functionality.

Table 13–2 Important Methods of the `Iterable` Trait

Methods	Description
<code>head, last, headOption, lastOption</code>	Returns the first or last element; or, that element an Option.
<code>tail, init</code>	Returns everything but the first or last element.
<code>length, isEmpty</code>	Returns the length, or true if the length is zero.
<code>map(f), flatMap(f), foreach(f), mapInPlace(f).collect(pf)</code>	Applies a function to all elements; see Section 13
<code>reduceLeft(op), reduceRight(op), foldLeft(init)(op), foldRight(init)(op)</code>	Applies a binary operation to all elements in a given order; see Section 13.9.
<code>reduce(op), fold(init)(op), aggregate(init)(op, combineOp)</code>	Applies a binary operation to all elements in arbitrary order.
<code>sum, product, max, min, maxBy(f), minBy(f), maxOption, minOption, maxByOption(f), minByOption(f)</code>	Returns the sum or product (provided the element type can be implicitly converted to the Numeric trait) or the maximum or minimum. The <code>max</code> and <code>min</code> functions require that the element type has an <code>Ordering</code> . The <code>maxBy</code> and <code>minBy</code> functions measure the elements by applying the function <code>f</code> . The methods ending in <code>Option</code> return <code>Option</code> so that they can be safely applied to empty collections.
<code>count(pred), forall(pred), exists(pred)</code>	Returns the count of elements fulfilling the predicate; true if all elements do, or at least one element does.
<code>filter(pred), filterNot(pred), partition(pred)</code>	Returns all elements fulfilling or not fulfilling the predicate; the pair of both.
<code>takeWhile(pred), dropWhile(pred), span(pred)</code>	Returns the first elements fulfilling <code>pred</code> ; all but the last elements; the pair of both.
<code>take(n), drop(n), splitAt(n)</code>	Returns the first <code>n</code> elements; everything but the first <code>n</code> elements; the pair of both.
<code>takeRight(n), dropRight(n)</code>	Returns the last <code>n</code> elements; everything but the last <code>n</code> elements.
<code>slice(from, to), view(from, to)</code>	Returns the elements in the range <code>from</code> until <code>to</code> , or a view thereto.

The `Seq` trait adds several methods to the `Iterable` trait. [Table 13–3](#) shows the most important ones.

Table 13–3 Important Methods of the `Seq` Trait

Methods	Description
<code>coll(k)</code> (i.e., <code>coll.apply(k)</code>)	The k th sequence element.
<code>contains(elem)</code> , <code>containsSlice(seq)</code> , <code>startsWith(seq)</code> , <code>endsWith(seq)</code>	Returns true if this sequence contains the given elem or sequence; if it starts or ends with the given sequence.
<code>indexOf(elem)</code> , <code>lastIndexOf(elem)</code> , <code>indexOfSlice(seq)</code> , <code>lastIndexOfSlice(seq)</code>	Returns the index of the first or last occurrence of given element or element sequence.
<code>indexWhere(pred)</code>	Returns the index of the first element fulfilling pred
<code>prefixLength(pred)</code> , <code>segmentLength(pred, n)</code>	Returns the length of the longest sequence of elements fulfilling pred, starting with 0 or n.
<code>padTo(n, fill)</code>	Returns a copy of this sequence, with fill appended until the length is n.
<code>intersect(seq)</code> , <code>diff(seq)</code>	Returns the “multiset” intersection or difference of sequences. For example, if a contains five 1s and b contains two, then a <code>intersect</code> b contains two (the smaller count), and a <code>diff</code> b contains three (the difference).
<code>reverse</code>	The reverse of this sequence.
<code>sorted</code> , <code>sortWith(less)</code> , <code>sortBy(f)</code>	The sequence sorted using the element ordering, a binary less function, or a function f that maps each element to an ordered type.
<code>permutations</code> , <code>combinations(n)</code>	Returns an iterator over all permutations or combinations (subsequences of length n).



Note

These methods never mutate a collection. If their result is a collection, the type is of the same type as the original, or as close as possible to it. (With types such as `Range` and `BitSet`, it can happen that the result is a more general sequence or set.) This is sometimes called the “uniform return type” principle.

13.8 Mapping a Function

You often want to transform all elements of a collection. The `map` method applies a function to a collection and yields a collection of the results. For example, given a list of strings

```
val names = List("Peter", "Paul", "Mary")
```

you get a list of the uppercased strings as

```
names.map(_.toUpperCase) // List("PETER", "PAUL", "MARY")
```

This is exactly the same as

```
for (n <- names) yield n.toUpperCase
```

If the function yields a collection instead of a single value, you may want to concatenate all results. In that case, use `flatMap`. For example, consider

```
def ulcase(s: String) = Vector(s.toUpperCase, s.toLowerCase)
```

Then `names.map(ulcase)` is

```
List(Vector("PETER", "peter"), Vector("PAUL", "paul"), Vector("MARY", "mary"))
```

but `names.flatMap(ulcase)` is

```
List("PETER", "peter", "PAUL", "paul", "MARY", "mary")
```



Tip

If you use `flatMap` with a function that returns an `Option`, the resulting collection contains all values v for which the function returns `Some(v)`.

For example, given a list of keys and a map, here is a list the matching values that are actually present:

```
val scores = Map("Alice" -> 10, "Bob" -> 3, "Cindy" -> 8)
val keys = Array("Alice", "Cindy", "Eloïse")
keys.flatMap(k => scores.get(k)) // Array(10, 8)
```

Note

The `map` and `flatMap` methods are important because they are used for translating `for` expressions. For example, the expression

```
for (i <- 1 to 10) yield i * i
```

is translated to

```
(1 to 10).map(i => i * i)
```

and

```
for (i <- 1 to 10; j <- 1 to i) yield i * j
```

becomes

```
(1 to 10).flatMap(i => (1 to i).map(j => i * j))
```

Why `flatMap`? See [Exercise 9](#) on page 206.

The `mapInPlace` method is the in-place equivalent of `map`. It applies to mutable collections, and replaces each element with the result of a function. For example, the following code changes all buffer elements to uppercase:

```
val buffer = ArrayBuffer("Peter", "Paul", "Mary")
buffer.mapInPlace(_.toUpperCase)
```

If you just want to apply a function for its side effect and don't care about the function values, use `foreach`:

```
names.foreach(println)
```

The `collect` method works with *partial functions*—functions that may not be defined for all inputs. It yields a collection of all function values of the arguments on which it is defined. For example,

```
"-3+4".collect({ case '+' => 1 ; case '-' => -1 }) // Vector(-1, 1)
```

The `groupBy` method yields a map whose keys are the function values, and whose values are the collections of elements whose function value is the given key. For example,

```
val words = ...
val map = words.groupBy(_.substring(0, 1).toUpperCase)
```

builds a map that maps "A" to all words starting with A, and so on.

13.9 Reducing, Folding, and Scanning A3

The `map` method applies a unary function to all elements of a collection. The methods that we discuss in this section combine elements with a *binary* function. The call `c.reduceLeft(op)` applies `op` to successive elements, like this:

```
.  
. .  
.  
op  
/ \  
op coll(3)  
/ \  
op coll(2)  
/ \  
coll(0) coll(1)
```

For example,

```
List(1, 7, 2, 9).reduceLeft(_ - _)
```

is

```
-  
 / \  
 - 9  
 / \  
 - 2  
 / \  
1 7
```

or

$$((1 - 7) - 2) - 9 = 1 - 7 - 2 - 9 = -17$$

The `reduceRight` method does the same, but it starts at the end of the collection. For example,

```
List(1, 7, 2, 9).reduceRight(_ - _)
```

is

$$1 - (7 - (2 - 9)) = 1 - 7 + 2 - 9 = -13$$



Note

Reducing assumes that the collection has at least one element. To avoid an exception for potentially empty collections, use `reduceLeftOption` and `reduceRightOption`.

Often, it is useful to start the computation with an initial element other than the initial element of a collection. The call `coll.foldLeft(init)(op)` computes

```
.  
. .  
op  
/ \  
op  coll(2)  
/ \  
.
```

```
op    coll(1)  
 / \  
init coll(0)
```

For example,

```
List(1, 7, 2, 9).foldLeft(0)(_ - _)
```

is

$$0 - 1 - 7 - 2 - 9 = -19$$



Note

The initial value and the operator are separate “curried” parameters so that Scala can use the type of the initial value for type inference in the operator. For example, in `List(1, 7, 2, 9).foldLeft("")(_ + _)`, the initial value is a string, so the operator must be a function `(String, Int) => String`.

There is a `foldRight` variant as well, computing

```

    op
   / \
coll(n-3)  op
   / \
coll(n-2)  op
   / \
coll(n-1) init

```

These examples don't seem to be very useful. Of course, `coll.reduceLeft(_ + _)` or `coll.foldLeft(0)(_ + _)` computes the sum, but you can get that directly with `coll.sum`.

Folding is sometimes attractive as a replacement for a loop. Suppose, for example, we want to count the frequencies of the letters in a string. One way is to

visit each letter and update a mutable map.

```
val freq = scala.collection.mutable.Map[Char, Int]()
for (c <- "Mississippi") freq(c) = freq.getOrElse(c, 0) + 1
// Now freq is Map('i' -> 4, 'M' -> 1, 's' -> 4, 'p' -> 2)
```

Here is another way of thinking about this process. At each step, combine the frequency map and the newly encountered letter, yielding a new frequency map. That's a fold:

```
          op
        /   \
op 's'
  /   \
op  'i'
  /   \
empty map 'M'
```

What is `op`? The left operand is the partially filled map, and the right operand is the new letter. The result is the augmented map. It becomes the input to the next call to `op`, and at the end, the result is a map with all counts. The code is

```
"Mississippi".foldLeft(Map[Char, Int]())((m, c) => m + (c ->
(m.getOrElse(c, 0) + 1)))
```

Note that this is an immutable map. We compute a new map at each step.



It is possible to replace any `while` loop with a fold. Build a data structure that combines all variables updated in the loop, and define an operation that implements one step through the loop. I am not saying that this is always a good idea, but you may find it interesting that loops and mutations can be eliminated in this way.

Finally, the `scanLeft` and `scanRight` methods combine folding and mapping. You get a collection of all intermediate results. For example,

```
(1 to 10).scanLeft(0)(_ + _)
```

yields all partial sums:

```
Vector(0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55)
```

13.10 Zipping

The methods of the preceding section apply an operation to adjacent elements in the same collection. Sometimes, you have two collections, and you want to combine corresponding elements. For example, suppose you have a list of product prices and corresponding quantities:

```
val prices = List(5.0, 20.0, 9.95)
val quantities = List(10, 2, 1)
```

The `zip` method lets you combine them into a list of pairs. For example,

```
prices.zip(quantities)  
is a List[(Double, Int)]:
```

```
List[(Double, Int)] = List((5.0, 10), (20.0, 2), (9.95, 1))
```

The method is called “zip” because it combines the two collections like the teeth of a zipper.

Now it is easy to apply a function to each pair.

```
prices.zip(quantities).map(_ * _)
```

It is perhaps surprising that you can provide a function of type `(Double, Int) => Double` when the collection contains `(Double, Int)` pairs. Section 7, “,” on page 206 explores how this “tupling” works.

The result is a list of prices:

```
List(50.0, 40.0, 9.95)
```

The total price of all items is then

```
prices.zip(quantities).map(_ * _).sum
```

If one collection is shorter than the other, the result has as many pairs as the shorter collection. For example,

```
List(5.0, 20.0, 9.95).zip(List(10, 2))
```

is

```
List((5.0, 10), (20.0, 2))
```

The `zipAll` method lets you specify defaults for the shorter list:

```
List(5.0, 20.0, 9.95).zipAll(List(10, 2), 0.0, 1)
```

is

```
List((5.0, 10), (20.0, 2), (9.95, 1))
```

The `zipWithIndex` method returns a list of pairs where the second component is the index of each element. For example,

```
"Scala".zipWithIndex
```

is

```
Vector(('S', 0), ('c', 1), ('a', 2), ('l', 3), ('a', 4))
```

This can be useful if you want to compute the index of an element with a certain property. For example,

```
"Scala".zipWithIndex.max
```

is ('l', 3), giving you both the maximum and the position where it occurs.

13.11 Iterators

You can obtain an iterator from a collection with the `iterator` method. This isn't as common as in Java or C++ because you can usually get what you need more

easily with one of the methods from the preceding sections.

However, iterators are useful for collections that are expensive to construct fully. For example, `Source.fromFile` yields an iterator because it might not be efficient to read an entire file into memory. There are a few `Iterable` methods that yield an iterator, such as `permutations`, `grouped` or `sliding`.

When you have an iterator, you can iterate over the elements with the `next` and `hasNext` methods and stop when you have seen enough:

```
val iter = ... // some Iterator
var done = false
while !done && iter.hasNext do
  done = process(iter.next())
```

Be aware that the iterator operations move the iterator. There is no way to reset it to the start of the iteration.

If the stopping condition is simple, you can avoid the loop. The `Iterator` class has many methods that work identically to the methods on collections. In particular, all `Iterable` methods listed in [Section 13.7, “Common Methods,”](#) on page 193 are available, except for `head`, `headOption`, `last`, `lastOption`, `tail`, `init`, `takeRight`, and `dropRight`. After calling a method such as `map`, `filter`, `count`, `sum`, or even `length`, the iterator is at the end of the collection, and you can’t use it again. With other methods, such as `find` or `take`, the iterator is past the found element or the taken ones.

```
val result = iter.take(500).toList
val result2 = iter.takeWhile(isNice).toList // isNice is some function
returning Boolean.
```

Sometimes, you want to be able to look at the next element before deciding whether to consume it. In that case, use the `buffered` method to turn an `Iterator` into a `BufferedIterator`. The `head` method yields the next element without advancing the iterator.

```
val iter = scala.io.Source.fromFile(filename).buffered
while iter.hasNext && iter.head == '#' do
  while iter.hasNext && iter.head != '\n' do
    iter.next
  // Now iter points to the first line not starting with #
```



Tip

If you find it too tedious to work with an iterator, and it doesn't produce a huge number of elements, just dump the elements into a collection by calling `toSeq`, `toArray`, `toBuffer`, `toSet`, or `toMap` to copy the values into a collection.

13.12 Lazy Lists A3

In the preceding section, you saw that an iterator is a “lazy” alternative to a collection. You get the elements as you need them. If you don’t need any more elements, you don’t pay for the expense of computing the remaining ones.

However, iterators are fragile. Each call to `next` mutates the iterator. *Lazy lists* offer an immutable alternative. A lazy list is an immutable list in which the elements are computed lazily—that is, only when you ask for them.

Here is a typical example:

```
def numsFrom(n: BigInt): LazyList[BigInt] = { log(n) ; n } #:::  
  numsFrom(n + 1)
```

The `#::` operator is like the `::` operator for lists, but it constructs a lazy list.

You couldn’t do this with a regular list. You would get a stack overflow when the `numsFrom` function calls it self recursively. With a lazy list, the expressions to the left and right of `#::` are only executed when needed.

When you call

```
val tenOrMore = numsFrom(10)
```

you get a lazy list that is displayed as

```
LazyList(<not computed>)
```

The elements are unevaluated, and the log shows nothing. If you call

```
tenOrMore.tail.tail.tail.head
```

the result is 13 and the log shows

```
10 11 12 13
```

The lazy list caches the evaluated head values. If you call

```
tenOrMore.tail.tail.tail.head
```

one more time, you get the result without another call to `log`.

When you apply a method to a lazy list that yields another list, such as `map` or `filter`, the result is also lazy. For example,

```
val squares = numsFrom(1).map(x => x * x)
```

yields

```
LazyList(<not computed>)
```

You have to call `squares.head` to force evaluation of the first entry.

If you want to get more than one answer, you can invoke `take` followed by `force`, which forces evaluation of all values. For example, `squares.take(5).force` produces `LazyList(1, 4, 9, 16, 25)`.

Of course, you don't want to call

```
squares.force // No!
```

That call would attempt to evaluate all members of an infinite list, eventually causing an `OutOfMemoryError`.

You can construct a lazy list from an iterator. For example, the `Source.getLines` method returns an `Iterator[String]`. With that iterator, you can only visit the lines once. A lazy list caches the visited lines so you can revisit them:

```
val words =
  Source.fromFile("/usr/share/dict/words").getLines.to(LazyList)
```

You just saw that methods such as `map` and `filter` are computed on demand when applied to a lazy list. You can get a similar effect with other collections by applying the `view` method. For example,

```
val palindromicSquares = (1 to 1000000).view
  .map(x => x * x)
```

```
.filter(x => x.toString == x.toString.reverse)
```

yields a collection that is unevaluated. When you call

```
palindromicSquares.take(10).mkString(",")
```

then enough squares are generated until ten palindromes have been found, and then the computation stops.

Unlike lazy lists, views do not cache any values. If you call `palindromicSquares.take(10).mkString(",")` again, the computation starts over.

13.13 Interoperability with Java Collections

At times you may need to use a Java collection, and you will likely miss the rich set of methods that you get with Scala collections. Conversely, you may want to build up a Scala collection and then pass it to Java code. The `CollectionConverters` object provides a set of conversions between Scala and Java collections.

For example,

```
import scala.jdk.CollectionConverters.*
```

[Table 13–4](#) shows the conversions from Scala to Java collections.

Table 13–4 Conversions between Scala Collections and Java Collections

Conversion Function	Type in <code>scala.collection</code>	Type (generally in <code>java.util</code>)
<code>asJava/asScala</code>	<code>Iterable</code>	<code>java.lang.Iterable</code>
<code>asJavaCollection/asScala</code>	<code>Iterable</code>	<code>Collection</code>
<code>asJava/asScala</code>	<code>Iterator</code>	<code>Iterator</code>
<code>asJavaEnumeration/asScala</code>	<code>Iterator</code>	<code>Enumeration</code>
<code>asJava/asScala</code>	<code>mutable.Buffer</code>	<code>List</code>
<code>asJava</code>	<code>Seq, mutable.Seq</code>	<code>List</code>
<code>asJava/asScala</code>	<code>mutable.Set</code>	<code>Set</code>
<code>asJava</code>	<code>Set</code>	<code>Set</code>
<code>asJava/asScala</code>	<code>mutable.Map</code>	<code>Map</code>
<code>asJava</code>	<code>Map</code>	<code>Map</code>
<code>asJavaDictionary/asScala</code>	<code>Map</code>	<code>Dictionary</code>
<code>asJava/asScala</code>	<code>concurrent.Map</code>	<code>concurrent.ConcurrentMap</code>

Note that the conversions yield wrappers that let you use the target interface to access the original type. For example, if you use

```
val props = System.getProperties.asScala
```

then `props` is a wrapper whose methods call the methods of the underlying Java object. If you call

```
props("com.horstmann.scala") = "impatient"
```

then the wrapper calls `put("com.horstmann.scala", "impatient")` on the underlying `Properties` object.

To convert from a Scala collection to a Java stream (sequential or parallel), start with the statement

```
import scala.jdk.StreamConverters.*
```

Then use one of the following methods:

- `asJavaSeqStream`, `asJavaParStream` for sequences, maps, or strings
- `asJavaParKeyStream`, `asJavaParValueStream` for maps
- `asJavaSeqCodePointStream`, `asJavaParCodePointStream`,
`asJavaSeqCharStream`, `asJavaParCharStream` for strings

If the element type is a primitive type, the resulting Java stream is a `DoubleStream`, `IntStream`, or `LongStream`.

To convert from a Java Stream to a Scala collection, use the `toScala` method and pass the collection type.

```
val lineBuffer = Files.lines(Path.of(filename)).toScala[Buffer]
```

Exercises

1. Write a function that, given a string, produces a map of the indexes of all characters. For example, `indexes("Mississippi")` should return a map associating '`M`' with the set `{0}`, '`i`' with the set `{1, 4, 7, 10}`, and so on. Use a mutable map of characters to mutable sets. How can you ensure that the set is sorted?
2. Repeat the preceding exercise, using an immutable map of characters to lists.
3. Write a function that removes every second element from a `ListBuffer`. Try it two ways. Call `remove(i)` for all even `i` starting at the end of the list. Copy every second element to a new list. Compare the performance.
4. Write a function that receives a collection of strings and a map from strings to integers. Return a collection of integers that are values of the map corresponding to one of the strings in the collection. For example, given `Array("Tom", "Fred", "Harry")` and `Map("Tom" -> 3, "Dick" -> 4, "Harry" -> 5)`, return `Array(3, 5)`. Hint: Use `flatMap` to combine the option values returned by `get`.
5. Implement a function that works just like `mkString`, using `reduceLeft`.
6. Given a list of integers `lst`, what is `lst.foldRight(List[Int]())(_ :: _)`? `lst.foldLeft(List[Int]())(_ :+ _)`? How can you modify one of them to reverse the list?
7. In [Section 13.10, “Zipping,”](#) on page 200, it is not obvious why the expression `prices.zip(quantities).map(_ * _)` works. Shouldn’t that be

`prices.zip(quantities).map(t => t(0) * t(1))` After all, the parameter of `map` is a function that takes a single parameter. Verify this by passing a function

```
((Double, Int)) => Double
```

The Scala compiler converts the expression `_ * _` to a function with a single tuple parameter. Explore in which contexts this happens. Does it work for function literals with explicit parameters `(x: Double, y: Int) => x * y`? Can you assign a literal to a variable of type `((Double, Int)) => Double`?

Conversely, what happens when you pass a variable `f` of type `((Double, Int)) => Double`, i.e. `prices.zip(quantities).map(f)`? How can you fix that? (Hint: `tupled`.) Does `tupled` work with function literals?

8. Write a function that turns an array of `Double` values into a two-dimensional array. Pass the number of columns as a parameter. For example, with `Array(1, 2, 3, 4, 5, 6)` and three columns, return `Array(Array(1, 2, 3), Array(4, 5, 6))`. Use the `grouped` method.

9. The Scala compiler transforms a `for/yield` expression

```
for (i <- 1 to 10; j <- 1 to i) yield i * j
```

to invocations of `flatMap` and `map`, like this:

```
(1 to 10).flatMap(i => (1 to i).map(j => i * j))
```

Explain the use of `flatMap`. Hint: What is `(1 to i).map(j => i * j)` when `i` is 1, 2, 3?

What happens when there are three generators in the `for/yield` expression?

10. Write a function that computes the sum of the non-`None` values in a `List[Option[Int]]`. Hint: `flatten`.

11. The method `java.util.TimeZone.getAvailableIDs` yields time zones such as `Africa/Cairo` and `Asia/Chungking`. Which continent has the most time zones? Hint: `groupBy`.

12. Produce a lazy list of random numbers. Hint: `continually`.

13. A *fixed point* of a function `f` is an argument `x` so that `f(x) == x`. Sometimes, it is possible to find fix points by starting with a value `a` and then computing `f(a)`, `f(f(a))`, `f(f(f(a)))`, and so on, until the sequence

converges to a fix point x . I happen to know this from personal experience. Bored in math class, I kept hitting the cosine key of my calculator in radian mode, and pretty soon got a value for which $\cos(x) == x$.

Produce a lazy list of iterated function applications and search for identical consecutive values. Hint: `sliding(2)`.

Chapter 14

Pattern Matching

Topics in This Chapter A2

- 14.1 A Better Switch
- 14.2 Guards
- 14.3 Variables in Patterns
- 14.4 Type Patterns
- 14.5 The `Matchable` Trait
- 14.6 Matching Arrays, Lists, and Tuples
- 14.7 Extractors
- 14.8 Patterns in Variable Declarations
- 14.9 Patterns in `for` Expressions
- 14.10 Case Classes
- 14.11 Matching Nested Structures
- 14.12 Sealed Classes
- 14.13 Parameterized Enumerations
- 14.14 Partial Functions A3
- 14.15 Infix Notation in `case` Clauses L2
- Exercises

Pattern matching is a powerful mechanism that has a number of applications: `switch` statements, type inquiry, and “destructuring” (getting at the parts of complex expressions). This chapter covers the many forms of the `match` expression, as well as the `case` syntax in partial functions and `for` loops. You will learn about `case` classes that are particularly adapted for use with pattern matching.

The key points of this chapter are:

- The `match` expression is a better `switch`, without fall-through.
- If no pattern matches, a `MatchError` is thrown. Use the `case _` pattern to avoid that.
- A pattern can include an arbitrary condition, called a guard.
- You can match on the type of an expression; prefer this over `isInstanceOf/asInstanceOf`.
- You can match patterns of arrays, tuples, and case classes, and bind parts of the pattern to variables.
- In a `for` `case` expression, nonmatches are silently skipped.
- A case class is a class for which the compiler automatically produces the methods that are needed for pattern matching.
- The common superclass in a case class hierarchy should be `sealed`.
- You can declare a sealed hierarchy of case classes as a parameterized `enum`.
- A sequence of case clauses gives rise to a partial function; that is, a function that is not defined for all arguments.
- When an `unapply` method yields a pair, you can use infix notation in the `case` clause.

14.1 A Better Switch

Here is the equivalent of the C-style `switch` statement in Scala:

```
val ch: Char = ...
var sign = 0
ch match
  case '+' => sign = 1
  case '-' => sign = -1
  case _ => sign = 0
```

The equivalent of `default` is the catch-all `case _` pattern. It is a good idea to have such a catch-all pattern. If no pattern matches, a `MatchError` is thrown.

Unlike the `switch` statement, Scala pattern matching does not suffer from the “fall-through” problem. (In C and its derivatives, you must use explicit `break` statements to exit a `switch` at the end of each branch, or you will fall through to the next branch. This is annoying and error-prone.)

Note

In his entertaining book *Deep C Secrets*, Peter van der Linden reports a study of a large body of C code in which the fall-through behavior was unwanted in 97% of the cases.

Similar to `if`, `match` is an expression, not a statement. The preceding code can be simplified to

```
val sign = ch match
  case '+' => 1
  case '-' => -1
  case _ => 0
```

Use `|` to separate multiple alternatives:

```
prefix match
  case "0x" | "0X" => 16
  case "0" => 8
  case _ => 10
```

You can use the `match` statement with any types. For example:

```
color match
  case Color.RED => 0xff0000
  case Color.GREEN => 0xff00
  case Color.BLUE => 0xff
```

14.2 Guards

Suppose we want to extend our example to match all digits. In a C-style `switch` statement, you would simply add multiple `case` labels, for example `case '0': case '1': ... case '9':`. (Except that, of course, you can't use `...` but must write out all ten cases explicitly.) In Scala, you add a *guard clause* to a pattern, like this:

```
ch match
  case _ if Character.isDigit(ch) => number = 10 * number +
    Character.digit(ch, 10)
  case '+' => sign = 1
  case '-' => sign = -1
```

The guard clause can be any Boolean condition.

Patterns are always matched top-to-bottom. If the pattern with the guard clause doesn't match, the `case '+'` pattern is attempted next.

14.3 Variables in Patterns

If the `case` keyword is followed by a variable name, then the match expression is assigned to that variable. For example:

```
str(i) match
  case '+' => sign = 1
  case '-' => sign = -1
  case d => number = number * 10 + Character.digit(d, 10)
```

You can think of `case _` as a special case of this feature, where the variable name is `_`.

You can use the variable name in a guard:

```
str(i) match
  case '+' => sign = 1
  case '-' => sign = -1
  case d if Character.isDigit(d) => number = 10 * number +
    Character.digit(d, 10)
```



Caution

Unfortunately, variable patterns can conflict with constants, for example:

```
import scala.math.*
4 * a / (c * c) match
  case Pi => "a circle" // Pi equals 4 * a / (c * c)
  case q => f"not a circle, quotient ${q%f}" // q set to 4 *
    a / (c * c)
```

How does Scala know that `Pi` is a constant, not a variable? The rule is that a variable must start with a *lowercase* letter.

To match a name that starts with a lowercase letter, enclose it in backquotes:

```
import java.io.File.* // Imports
java.io.File.separatorChar
ch match
  case `separatorChar` => '\\' // ch equals
    java.io.File.separatorChar
  case _ => ch
```

14.4 Type Patterns

You can match on the type of an expression, for example:

```
obj match
  case x: Int => x
  case s: String => Integer.parseInt(s)
  case _: BigInt => Int.MaxValue
  case _ => 0
```

In Scala, this form is preferred to using the `isInstanceOf` operator.

Note the variable names in the patterns. In the first pattern, the match is bound to `x` as an `Int`, and in the second pattern, it is bound to `s` as a `String`. No `asInstanceOf` casts are needed!



Caution

When you match against a type, you must supply a variable name. Otherwise, you match the *object*:

```
obj match
  case _: BigInt => Int.MaxValue // Matches any object of
  type BigInt
  case BigInt => -1 // Matches the BigInt object of type
  Class
```

Matches are attempted in the order of the `case` clauses. When the compiler detects that a clause is unreachable, a compile-time error results:

```
ex match
  case _: RuntimeException => "RTE"
  case _: NullPointerException => "NPE" // Error: unreachable
  case
```

```
case _: IOException => "IOE"  
case _: Throwable => ""
```

Here, the second `case` clause is shadowed by the first one since `NullPointerException` is a subtype of `RuntimeException`.

Similarly, the following is an error:

```
obj match  
  case s: String => Double.parseDouble(s)  
  case null => Double.NaN
```

A `null` would match the first case.

In both examples, the remedy is to rearrange the cases and put the more specific ones first.



Caution

Matches occur at runtime, and generic types are erased in the Java virtual machine. For that reason, you cannot make a type match for a specific `Map` type.

```
case m: Map[String, Int] => ... // Don't
```

You can match a generic map:

```
case m: Map[_, _) => ... // OK
```

However, arrays are not erased. You can match an `Array[Int]`.

14.5 The `Matchable` Trait

Some types are *not matchable*. For example, the Scala library has a type `IArray` for immutable arrays:

```
val smallPrimes = IArray(2, 3, 5, 7, 11, 13, 17, 19)
```

The underlying type is a Java virtual machine array, so the following match, if allowed, would succeed at runtime, permitting mutation:

```
smallPrimes match
  case a: Array[Int] => a(1) = 4
```

This problem is solved as follows. Only values of types that extend the `Matchable` trait should participate in pattern matching. As you can see from the hierarchy diagram in [Chapter 8](#), both `AnyVal` and `AnyRef` extend `Matchable`. Types such as `IArray` do not.

Depending on your Scala version and compiler flags, you may get a warning or error if the compiler cannot prove that the matched value is `Matchable`.

In such a situation, you may be able to call `asInstanceOf[Matchable]` to keep the compiler happy. This happens when you work with variables of type `Any`, since there are types extending `Any` but not `Matchable`. A typical example is this `equals` method:

```
class Bounded(var value: Int, to: Int) :
  ...
  override def equals(other: Any) =
    other match
      case that: Bounded => value == that.value && to ==
        that.to
      case _ => false
```

With `matchable` checking enabled, the code does not compile. You need to use:

```
other.asInstanceOf[Matchable] match
```

Why the fuss? Consider these opaque types. (See [Chapter 18](#) for more information about opaque types.)

```

opaque type Minute = Bounded
def Minute(m: Int) = Bounded(m, 60)

opaque type Second = Bounded
def Second(m: Int) = Bounded(m, 60)

```

At runtime, `Minute(10)` and `Second(10)` are both `Bounded` instances with the same state, so they will compare as equal.

Of course, the `asInstanceOf` cast won't solve that problem, but at least one is now on alert and can search for alternate solutions, such as compiling with:

```
import scala.language.strictEquality
```

14.6 Matching Arrays, Lists, and Tuples

To match an array against its contents, use `Array` expressions in the patterns, like this:

```

arr match
  case Array(0) => "0"
  case Array(x, y) => s"$x $y"
  case Array(0, _) => "0 ..."
  case _ => "something else"

```

The first pattern matches the array containing `0`. The second pattern matches any array with two elements, and it binds the variables `x` and `y` to the elements. The third pattern matches any array starting with zero.

If you want to bind a `_*` match to a variable, add a `*` after the variable name:

```
case Array(0, rest*) => rest.min
```

The variable is set to a `Seq` containing the matched values.

You can match lists in the same way as you have just seen with arrays, with `List` expressions. Alternatively, you can use the `::` operator:

```
lst match
  case 0 :: Nil => "0"
  case x :: y :: Nil => s"$x $y"
  case 0 :: tail => "0 ..."
  case _ => "something else"
```

With tuples, use the tuple notation in the pattern:

```
pair match
  case (0, _) => "0 ..."
  case (y, 0) => s"$y 0"
  case _ => "neither is 0"
```

You can also match against the head and tail:

```
longerTuple match
  case h *: t => s"head is $h, tail is $t"
  case _: EmptyTuple => "empty"
```

Note the `EmptyTuple` object. You can't use `()` because that is the `Unit` literal.

Again, note how the variables are bound to parts of the list or tuple. Since these bindings give you easy access to parts of a complex structure, this operation is called *destructuring*.



Caution

The same warning applies as in [Section 14.3, “Variables in Patterns,”](#) on page 211. The variable names that you use in the pattern must start with a *lowercase* letter. In a match against `case Array(x, y)`, `x` and `y` are deemed constants, not variables.



Note

A pattern can have alternatives:

```
pair match
  case (_ , 0) | (0, _) => ... // OK, matches if one
  component is zero
```

However, you cannot use variables other than an underscore:

```
pair match
  case (x, 0) | (0, x) => ... // Error—cannot bind with
  alternatives
```

14.7 Extractors

In the preceding section, you have seen how patterns can match arrays, lists, and tuples. These capabilities are provided by *extractors*—objects with either an `unapply` or an `unapplySeq` method that extract values from the value that is being matched. The implementation of these methods is covered in [Chapter 11](#). The `unapply` method extracts a fixed number of values, while `unapplySeq` extracts a sequence whose length can vary.

For example, consider the expression

```
arr match
  case Array(x, 0) => x
  case Array(_, rest*) => rest.min
  ...
  ...
```

The `Array` companion object is an extractor—it defines an `unapplySeq` method. That method is called *with the expression that is being matched*, not with what appear to be the parameters in the pattern. The call `Array.unapplySeq(arr)` results in a sequence of values, namely the values in the array. In the first case, the match succeeds if the array has length 2 and the second element is zero. In that case, the initial array element is assigned to `x`. No call `Array(x, 0)` ever happens.

Regular expressions provide a more interesting use of extractors. When a regular expression has groups, you can match each group with an extractor

pattern. For example:

```
val pattern = "([0-9]+) ([a-z]+)".r
"99 bottles" match
  case pattern(quantity, item) => s"${ quantity: $quantity, item: $item }"
    // Sets quantity to "99", item to "bottles"
```

The call `pattern.unapplySeq("99 bottles")` yields a sequence of strings that match the groups. These are assigned to the variables `quantity` and `item`.

Note that here the extractor isn't a companion object but a regular expression object.

The general syntax is

```
value match
  case extractor(pattern1, . . . , patternn) => ...
```

To see whether the case matches, one of the following calls is made:

```
extractor.unapply(value)
```

or

```
extractor.unapplySeq(value)
```

If the call indicates failure, the match fails.

Otherwise, the call yields extracted values that are matched against the patterns. If there aren't enough values for the patterns, the match fails. If a pattern is a constant, it must match the corresponding value, or the match fails. If a pattern is a variable, the variable is assigned the corresponding value.

The rules for the allowed return types are a bit complex—see [Chapter 11](#) for details. There is some latitude for indicating success and failure, and for producing the extracted values. The details are only relevant for implementors of extractors.

14.8 Patterns in Variable Declarations

In the preceding sections, you have patterns in `match` expressions. You can also use patterns for variable declarations. For example,

```
val (a, b) = (1, 2)
```

simultaneously defines `a` as `1` and `b` as `2`. That is useful for functions that return a pair, for example:

```
val (q, r) = BigInt(10) /% 3
```

The `/%` method returns a pair containing the quotient and the remainder, which are captured in the variables `q` and `r`.

The same syntax works for extractor patterns. For example,

```
val Array(0, second, rest*) = arr
```

throws a `MatchError` if `arr` has fewer than two elements or if `arr(0)` is nonzero. Otherwise `second` becomes `arr(1)` and `rest` is set to a `Seq` of the remaining elements.



Caution

The same warning applies as in [Section 14.3, “Variables in Patterns,”](#) on page 211. A variable pattern must start with a *lowercase* letter. In a declaration

```
val Array(Pi, x, _) = arr
```

`Pi` is deemed a constant and `x` a variable. If the array has at least two elements and `arr(0) == Pi`, then `x` is set to `arr(1)`.



Note

Technically,

```
val extractor(pattern1, ..., patternn) = value
```

is equivalent to

```
value match { extractor(pattern1, ..., patternn) => () }
```

As described in the preceding section, either `extractor.unapply(value)` or `extractor.unapplySeq(value)` is called. If the result indicates failure, a `MatchError` is thrown. Otherwise the patterns are matched against the extracted values.

You don't even have to have variables on the left hand side. For example,

```
val 2 = b
```

is perfectly legal Scala code, provided `b` has been defined elsewhere. It is the same as

```
b match { case 2 => () }
```

In other words, it is equivalent to

```
if !(2 == b) then throw MatchError()
```

14.9 Patterns in `for` Expressions

You can use patterns with variables in `for` expressions. For each traversed value, the variables are bound. This makes it possible to traverse a map:

```
import scala.jdk.CollectionConverters.*  
// Converts Java Properties to a Scala map—just to get an  
interesting example
```

```
for (k, v) <- System.getProperties.asScala do
  println(s"$k -> $v")
```

For each pair in the map, `k` is bound to the key and `v` to the value.

If the left hand side of the `<-` doesn't consist of variable patterns, the match can fail. In that situation, use `for case` instead of `for`. Then any match failures are skipped. For example, the following loop yields all keys with value "UTF-8", skipping over all others:

```
for case (k, "UTF-8") <- System.getProperties.asScala
  yield k
```



Note

If you use `for` instead of `for case`, future versions of Scala will throw a `MatchError` if a match fails.

You can also use a guard. Note the placement of the `if` clause:

```
for (k, v) <- System.getProperties.asScala if v == "UTF-8"
  yield k
```

14.10 Case Classes

Case classes are a special kind of classes that are optimized for use in pattern matching. In this example, we have two case classes that extend a regular (noncase) class:

```
abstract class Amount
case class Dollar(value: Double) extends Amount
case class Currency(value: Double, unit: String) extends Amount
```

You can also have case objects for singletons:

```
case object Nothing extends Amount
```

When we have an object of type `Amount`, we can use pattern matching to match the amount type and bind the property values to variables:

```
amt match
  case Dollar(v) => s"$$${v}"
  case Currency(_, u) => s"Oh noes, I got ${u}"
  case Nothing => ""
```

 **Note**

Use `()` with case class instances, no parentheses with case objects.

 **Note**

The `Option` type has subtypes `Some` and `None` that are a case class and a case object. You can use pattern matching to analyze an `Option` value:

```
scores.get("Alice") match
  case Some(score) => println(score)
  case None => println("No score")
```

When you declare a case class, several things happen automatically.

- Each of the constructor parameters becomes a `val` unless it is explicitly declared as a `var` (which is not recommended).
- An `apply` method is provided for the companion object that lets you construct objects without `new`, such as `Dollar(29.95)` or `Currency(29.95, "EUR")`.
- An `unapply` method is provided that makes pattern matching work

- Methods `toString`, `equals`, `hashCode`, and `copy` are generated unless you provide them.

Otherwise, case classes are just like any other classes. You can add methods and fields to them, extend them, and so on.

 **Caution**

A case class cannot extend another case class. If you need multiple levels of inheritance to factor out common behavior of case classes, make only the leaves of the inheritance tree into case classes.

The `copy` method of a case class makes a new object with the same values as an existing one. For example,

```
val price = Currency(29.95, "EUR")
var discounted = price.copy()
```

By itself, that isn't very useful—after all, a `Currency` object is immutable, and one can just share the object reference. However, you can use named parameters to modify some of the properties:

```
discounted = price.copy(value = 19.95) // Currency(19.95, "EUR")
```

or

```
discounted = price.copy(unit = "CHF") // Currency(29.95, "CHF")
```

 **Note**

In type theory, a case class is called a *product type* because its value set is the “cartesian product” of the value sets of the constituent types. If the value sets are finite, the number of possible values that a case class can assume is indeed the product of the sizes of the constituent types. Consider for example:

```
enum Rank { case ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN,  
EIGHT, NINE, TEN, JACK, QUEEN, KING } // 13 values  
enum Suit { case DIAMONDS, HEARTS, SPADES, CLUBS } // 4  
values  
case class Card(r: Rank, s: Suit) // 13 × 4 = 52 values
```

Every case class extends the `Product` trait and has methods `_1`, `_2`, and so on, to access the elements:

```
val price = Currency(100, "CHF")  
price._1 // 100  
price._2 // "CHF"
```

Note

The `unapply` method of a case class is trivial—it simply returns the argument:

```
Currency.unapply(price) == price
```

To see why this works, carefully study the advanced rules for `unapply` methods in [Chapter 11](#). An `unapply` method doesn't have to return an `Option` if the extraction always succeeds, and it can return any `Product` instance. The values are extracted with the methods `_1`, `_2`, and so on.

14.11 Matching Nested Structures

Case classes are often used for nested structures. Consider, for example, items that a store sells. Sometimes, we bundle items together for a discount.

```
abstract class Item  
case class Article(description: String, price: Double) extends  
Item
```

```
case class Bundle(description: String, discount: Double, items: Item*) extends Item
```

Of course, you can specify nested objects:

```
val wish = Bundle("Father's day special", 20.0,
  Article("Scala for the Impatient", 39.95),
  Bundle("Anchor Distillery Sampler", 10.0,
    Article("Old Potrero Straight Rye Whiskey", 79.95),
    Article("Junipero Gin", 32.95)))
```

Patterns can match specific nestings, for example

```
case Bundle(_, _, Article(descr, _), _) => ...
```

matches if the first item of the bundle is an `Article` and then binds `descr` to the description.

You can bind a nested value to a variable with the `@` notation:

```
case Bundle(_, _, art @ Article(_, _), _) => ...
```

Now `art` is the first article in the bundle.

As an application, here is a function that computes the price of an item:

```
def price(it: Item): Double = it match
  case Article(_, p) => p
  case Bundle(_, disc, its*) => its.map(price _).sum - disc
```

This example can enrage OO purists. Shouldn't `price` be a method of the superclass? Shouldn't each subclass override it? Isn't polymorphism better than making a switch on each type?

In many situations, this is true. If someone comes up with another kind of `Item`, you'll need to revisit all those `match` clauses. In such a situation, case classes are not the right solution.

On the other hand, if you know that there will be a limited number of cases, case classes work well. It is often more convenient to use `match` expressions

than polymorphism. In the next section, you will see how to ensure that you know all cases.

14.12 Sealed Classes

When you use pattern matching with case classes, you would like the compiler to check that you exhausted all alternatives. To achieve this, declare the common superclass as *sealed*:

```
sealed abstract class Amount
case class Dollar(value: Double) extends Amount
case class Currency(value: Double, unit: String) extends Amount
```

All subclasses of a sealed class must be defined in the same file as the class itself. For example, if someone wants to add another class for euros,

```
case class Euro(value: Double) extends Amount
```

they must do so in the file in which `Amount` is declared.

When a class is sealed, all of its subclasses are known at compile time, enabling the compiler to check pattern clauses for completeness. It is a good idea for case classes to extend a sealed class or trait.



Note

The `Option` class is a sealed class with two subtypes: the case class `Some` and the case object `None`.

In a sealed class hierarchy, the compiler can check that a match is exhaustive:

```
scores.get("Alice") match
  case Some(score) => println(score)
  case None => println("No score")
```

If you omit the second case, the compiler warns you. You can suppress the warning with the `@unchecked` annotation:

```
(scores.get("Alice") : @unchecked) match
  case Some(score) => println(score)
```

14.13 Parameterized Enumerations

As an alternative to a sealed class hierarchy, you can use a *parameterized enumeration*:

```
enum Amount :
  case Dollar(value: Double)
  case Currency(value: Double, unit: String)
  case Nothing
```

This is similar to the enumerations that you saw in [Chapter 6](#). However, the `Dollar` and `Currency` cases have parameters. They are translated into case classes. The third case becomes a case object.

Note that the case classes are nested inside the enum type. Users access them as `Amount.Dollar`, `Amount.Currency`, and `Amount.None`.

As with all enumerations, you can add methods to the abstract `enum` class, but not to the subclasses. You can also add methods to the companion object:

```
object Amount :
  def euros(value: Double) = if value == 0 then Nothing else
    Currency(value, "EUR")
```

Enumerations can have type parameters. For example, `Option` could have been defined as an enumeration:

```
enum Option[+T] :
  case Some(value: T)
  case None
```

```
def get = this match
  case Some(v) => v
  case None => throw NoSuchElementException()
```

The type parameters are automatically applied to each case class.

Note

In the uncommon case that the type parameter variances do not apply to all case classes, a case class can declare its own:

```
enum Producer[-T] :
  case Constant[U](value: U) extends Producer[U]
  case Generator[U](next: () => U) extends Producer[U]
```

Note

In type theory, the `Amount` type is said to be the *sum* of the three alternatives `Currency`, `Dollar`, and `Nothing`. Each of those is a product type, and `Amount`, being a sum of products, is called an *algebraic data type*.

You don't generally need to know about this terminology when working with case class hierarchies. However, in [Chapter 20](#), you will see tools for analyzing sum and product types.

14.14 Partial Functions A3

A sequence of `case` clauses can be converted to a *partial function*—a function which may not be defined for all inputs. It is an instance of a class `PartialFunction[P, R]`. (`P` is the parameter type, `R` the return type.) That class has two methods: `apply`, which computes the function value from the

matching pattern, and `isDefinedAt`, which returns `true` if the input matches at least one of the patterns.

For example,

```
val f: PartialFunction[Char, Int] =  
  case '+' => 1  
  case '-' => -1  
f('-) // Calls f.apply('-), returns -1  
f.isDefinedAt('0') // false  
f('0') // Throws MatchError
```

The `collect` method of the `IterableOps` trait applies a partial function to all elements where it is defined, and returns a sequence of the results.

```
"-3+4".collect({ case '+' => 1 ; case '-' => -1 }) // Vector(-1,  
1)
```

The `case` sequence must be in a context where the compiler can infer the return type. This happens when you assign it to a typed variable or pass it as an argument.



You can also convert a `case` sequence to a `Function[P, R]`.

However, a function lacks the `isDefinedAt` method, so the user doesn't know when it is safe to call. If the `case` clauses are exhaustive, then that is not a problem. Here, a `Function[Char, Int]` is passed to the `map` method:

```
"-3+4".map({ case '+' => 1 ; case '-' => -1; case _ => 0 })  
// Vector(-1, 0, 1, 0)
```

A `Seq[A]` is a `PartialFunction[Int, A]`, and a `Map[K, V]` is a `PartialFunction[K, V]`. For example, you can pass a map to `collect`:

```
val names = Array("Alice", "Bob", "Carmen")
val scores = Map("Alice" -> 10, "Carmen" -> 7)
names.collect(scores) // Yields Array(10, 7)
```

The `lift` method turns a `PartialFunction[T, R]` into a regular function with return type `Option[R]`.

```
val f: PartialFunction[Char, Int] = { case '+' => 1 ; case '-' =>
-1 }
val g = f.lift // A function with type Char => Option[Int]
g('-') // Some(-1)
g('*') // None
```

In [Chapter 9](#), you saw that the `Regex.replaceSomeIn` method requires a function `String => Option[String]` for the replacement. If you have a map (or some other `PartialFunction`), you can use `lift` to produce such a function:

```
val varPattern = """\{([0-9]+)\}""".r
val message = "At {1}, there was {2} on {0}"
val vars = Map("{0}" -> "planet 7", "{1}" -> "12:30 pm",
"{2}" -> "a disturbance of the force")
val result = varPattern.replaceSomeIn(message, m =>
vars.lift(m.matched))
```

Conversely, you can turn a function returning `Option[R]` into a partial function by calling `Function.unlift`.



The `catch` clause of the `try` statement can be a partial function instead of a sequence of `case` clauses:

```
def tryCatch[T](block: => T, catcher:
PartialFunction[Throwable, T]) =
try
```

```
    block
  catch catcher
```

Then you can supply a custom catch clause like this:

```
val result = tryCatch(str.toInt, { case _:  
  NumberFormatException => -1 })
```

14.15 Infix Notation in `case` Clauses L2

When an `unapply` method yields a pair, you can use infix notation in the `case` clause. In particular, you can use infix notation with a case class that has two parameters. For example:

```
price match
  case value Currency "CHF" => ... // Same as case
  Currency(value, "CHF")
```

Of course, that is a silly example. Here is an example from the Scala API. Every `List` object is either `Nil` or an instance of a case class cunningly named `::`:

```
case class List[E]
```

As with every case class, the `::` companion object automatically has an `unapply` method. Since there are two instance variables, `unapply` yields a pair and can be used in infix position. That is why you can use `::` for deconstructing:

```
lst match
  case h :: t => ... // Same as case ::(h, t), which calls
  ::.unapply(lst)
```

If an infix extractor ends in a colon, it associates right-to-left. For example,

```
case first :: second :: rest
```

means

```
case ::(first, ::(second, rest))
```

As another example, suppose you want to extract numerator and denominator from a `Fraction` object as `case n / d`. This is easily achieved. Simply define an object, cunningly named `/`, with an `unapply` method:

```
object Fraction :  
  object / :  
    def unapply(input: Fraction) = (input.num, input.den)  
  
  import Fraction./
```

The `/` object must be put inside the `Fraction` companion object to gain access to the private `num` and `den` instance variables.

Now you can write:

```
Fraction(3, 4) * Fraction(5, 6) match  
  case n / d => n * 1.0 / d
```

Here, the first `/` is the `Fraction./` object with the `unapply` method. The second `/` is the division operator.

Note that this example does not involve case classes. For infix extraction, the symbol between the two patterns can be any object with an `unapply` method.

Exercises

1. Your Java Development Kit distribution has the source code for much of the JDK in the `src.zip` file. Unzip and search for case labels (regular expression `case [^:]++:`). Then look for comments starting with `//` and containing `[Ff]alls? thr` to catch comments such as `// Falls through or // just fall thru`. Assuming the JDK programmers follow the Java code convention, which requires such a comment, what percentage of cases falls through?

2. Using pattern matching, write a function `swap` that receives a pair of integers and returns the pair with the components swapped.
3. Using pattern matching, write a function `swap` that swaps the first two elements of an array provided its length is at least two.
4. Add a case class `Multiple` that is a subclass of the `Item` class. For example, `Multiple(10, Article("Blackwell Toaster", 29.95))` describes ten toasters. Of course, you should be able to handle any items, such as bundles or multiples, in the second argument. Extend the `price` function to handle this new case.
5. A `List[T]` is either `Empty`, or it is a `NonEmpty` with a head of type `T` and a tail that is again a `List[T]`.

Implement such a list, both as case classes and as `enum`. In both cases, add a `::` operator and `length`, `append`, and `map` methods.

6. One can use lists to model trees that store values only in the leaves. For example, the list `((3 8) 2 (5))` describes the tree

```

    / \ \
    2
   / \   |
  3 8   5
  
```

However, some of the list elements are numbers and others are lists. In Scala, you cannot have heterogeneous lists, so you have to use a `List[Any]`. Write a `leafSum` function to compute the sum of all elements in the leaves, using pattern matching to differentiate between numbers and lists.

7. A better way of modeling such trees is with case classes. Let's start with binary trees.

```

sealed abstract class BinaryTree
case class Leaf(value: Int) extends BinaryTree
case class Node(left: BinaryTree, right: BinaryTree) extends BinaryTree
  
```

Write a function to compute the sum of all elements in the leaves.

8. Extend the tree in the preceding exercise so that each node can have an arbitrary number of children, and reimplement the `leafSum` function. The tree in [Exercise 5](#) should be expressible as

```
Node(Node(Leaf(3), Leaf(8)), Leaf(2), Node(Leaf(5)))
```

9. Extend the tree in the preceding exercise so that each nonleaf node stores an operator in addition to the child nodes. Then write a function `eval` that computes the value. For example, the tree

```
+  
/ | \  
* 2 -  
/ \ |  
3 8 5
```

has value $(3 \times 8) + 2 + (-5) = 21$.

Pay attention to the unary minus.

10. Complete the `Producer` class in [Section 6.6, “Enumerations,”](#) on page 85. Add a method `get` that produces another element; either a constant value or the result from invoking the generator function. Write a program that demonstrates producers of constant and random numbers. Include an example to show that `Producer` is contravariant. What happens if you simply declare the following, and why?

```
enum Producer[-T] :  
    case Constant(value: T)  
    case Generator(next: () => T)
```

Chapter 15

Annotations

Topics in This Chapter A2

- [15.1 What Are Annotations?](#)
- [15.2 Annotation Placement](#)
- [15.3 Annotation Arguments](#)
- [15.4 Annotations for Java Features](#)
- [15.5 Annotations for Optimizations](#)
- Tail Recursion. Lazy Values.
- [15.6 Annotations for Errors and Warnings](#)
- [15.7 Annotation Declarations](#)
- [Exercises](#)

Annotations let you add information to program items. This information can be processed by the compiler or by external tools. In this chapter, you will learn how to interoperate with Java annotations and how to use the annotations that are specific to Scala.

The key points of this chapter are:

- You can annotate classes, methods, fields, local variables, parameters, expressions, type parameters, and types.

- With expressions and types, the annotation follows the annotated item.
- Annotations have the form `@Annotation`, `@Annotation(value)`, or `@Annotation(name1 = value1, ...)`.
- You can use Java annotations in Scala code. They are retained in the class files.
- `@volatile`, `@transient`, and `@native` generate the equivalent Java modifiers.
- Use `@throws` to generate Java-compatible `throws` specifications.
- Use the `@BeanProperty` annotation to generate the JavaBeans `getXxx/setXxx` methods.
- The `@tailrec` annotation lets you verify that a recursive function uses tail call optimization.
- Use the `@deprecated` annotation to mark deprecated features.

15.1 What Are Annotations?

Annotations are tags that you insert into your source code so that some tools can process them. These tools can operate at the source level, or they can process the class files into which the compiler has placed your annotations.

Annotations are widely used in Java, for example by testing tools such as JUnit and enterprise technologies such as Jakarta EE.

Annotations start with an `@`. For example:

```
case class Person @JsonbCreator (
  @JsonbProperty val name: String,
  @JsonbProperty val age: Int)
```

You can use Java annotations with Scala classes. The annotations in the preceding example are from JSON-B, a Java framework for converting between JSON and Java classes. The framework has no knowledge of Scala. We will be using JSON-B for several examples, but you don't have to be familiar with its details. If you are curious, read through the tutorial at <https://javaee.github.io/jsonbspec/users-guide.html>.

Scala provides its own annotations that are processed by the Scala compiler or a compiler plugin. (Implementing a compiler plugin is a nontrivial undertaking that is not covered in this book.)

You have seen the Scala `@main` annotation throughout the book for marking a program entry point.

Java annotations do not affect how the compiler translates source code into bytecode; they merely add data to the bytecode that can be harvested by external tools. In the example above, the constructor and its parameters are annotated in the class file.

In Scala, annotations can affect the compilation process. For example, the `@main` annotation causes the generation of a Java class with a `public static void main(String[] args)` method.

15.2 Annotation Placement

In Scala, you can place annotations before classes, methods, fields, local variables, parameters, and type parameters:

```
@deprecated class Sample : // Class
  @volatile var alive = true // Field
  @tailrec final def gcd(a: Int, b: Int): Int = if b == 0 then a
  else gcd(b, a % b) // Method
  def display(@nowarn message: String) = "" // Parameter
  case class Box[@specialized T](value: T) // Type parameter
```

You can apply multiple annotations. The order doesn't matter.

```
@BeanProperty @JsonbProperty val age: Int
```

When annotating the primary constructor, place the annotation after the class name:

```
class Person @JsonbCreator (...)
```

You can also annotate expressions. Add a colon followed by the annotation, for example:

```
(props.get(key) : @unchecked) match { ... }  
    // The expression props.get(key) is annotated
```

Annotations on a type are placed *after* the type, like this:

```
val country: String @Localized =  
    java.util.Locale.getDefault().getDisplayCountry()
```

Here, the `String` type is annotated. The method returns a localized string.

15.3 Annotation Arguments

Annotations can have named arguments, such as

```
@JsonbProperty(value="p_name", nullable=true) var name: String =  
    null
```

Most annotation arguments have defaults. For example, the `nullable` argument of the `@JsonbProperty` annotation has a default value of `false`, and the `value` attribute has a default of `""`.

When you provide only the argument named `value`, then `value=` is optional. For example:

```
@JsonbProperty("p_age") var age : Int = 0  
    // The value argument is "p_age"
```

If the annotation has no arguments, the parentheses can be omitted:

```
@JsonbTransient val nice: Boolean = true
```

Arguments of Java annotations are restricted to the following types:

- Numeric or Boolean literals
- Strings

- Class literals
- Java enumerations
- Other annotations
- Arrays of the above (but not arrays of arrays)

Arguments of Scala annotations can be of arbitrary types.

15.4 Annotations for Java Features

The Scala library provides annotations for interoperating with Java. They are presented in the following sections.

15.4.1 Bean Properties

When you declare a public field in a class, Scala provides a getter and (for a `var`) a setter method. However, the names of these methods are not what Java tools expect. The JavaBeans specification (www.oracle.com/technetwork/articles/javaee/spec-136004.html) defines a Java property as a pair of `getFoo`/`setFoo` methods (or just a `getFoo` method for a read-only property). Many Java tools rely on this naming convention.

When you annotate a Scala field with `@BeanProperty`, then such methods are automatically generated. For example,

```
import scala.beans.BeanProperty

class Person :
  @BeanProperty var name = ""
```

generates *four* methods:

1. `name: String`
2. `name_=(newValue: String): Unit`
3. `getName(): String`
4. `setName(newValue: String): Unit`

However, the `getName` and `setName` methods are *only for the benefit of Java*. The Scala compiler will refuse to invoke them. In Scala, read or write the `name` property.

The `@BooleanBeanProperty` annotation generates a getter with an `is` prefix for a Boolean method.



Note

If you define a field as a primary constructor parameter, and you want JavaBeans getters and setters, annotate the constructor parameter like this:

```
class Person(@BeanProperty var name: String)
```

15.4.2 Serialization

With serializable classes, you can use the `@SerialVersionUID` annotation to specify the serial version:

```
@SerialVersionUID(6157032470129070425L)  
class Employee extends Person, Serializable :
```

The `@transient` annotation marks a field as transient:

```
@transient var lastLogin: ZonedDateTime = null  
// Becomes a transient field in the JVM
```

A transient field is not serialized.



Note

For more information about Java concepts such as volatile fields or serialization, see C. Horstmann, *Core Java®, Twelfth Edition* (Prentice Hall, 2022).

15.4.3 Checked Exceptions

Unlike Scala, the Java compiler tracks checked exceptions. If you call a Scala method from Java code, its signature should include the checked exceptions that can be thrown. Use the `@throws` annotation to generate the correct signature. For example,

```
@throws(classOf[IOException]) def save(filename: String) = ...
```

The Java signature is

```
void save(String filename) throws IOException
```

Without the `@throws` annotation, the Java code would not be able to catch the exception.

```
try { // This is Java
    fred.save("/etc/fred.ser");
} catch (IOException ex) {
    System.out.println("Error saving: " + ex.getMessage());
}
```

The Java compiler needs to know that the `save` method can throw an `IOException`, or it will refuse to catch it.

15.4.4 Variable Arguments

The `@varargs` annotation lets you call a Scala variable-argument method from Java. By default, if you supply a method such as

```
def process(args: String*) = ...
```

the Scala compiler turns the variable argument into a sequence parameter:

```
def process(args: Seq[String])
```

That method would be very cumbersome to call in Java. If you add `@varargs`,

```
@varargs def process(args: String*) = ...
```

then a Java method

```
void process(String... args) // Java bridge method
```

is generated that wraps the `args` array into a `Seq` and calls the Scala method.

15.4.5 Java Modifiers

Scala uses annotations instead of modifier keywords for some of the less commonly used Java features.

The `@volatile` annotation marks a field as volatile:

```
@volatile var done = false // Becomes a volatile field in the JVM
```

A volatile field can be updated in multiple threads.

The `@native` annotation marks methods that are implemented in C or C++ code. It is the analog of the `native` modifier in Java.

```
@native def win32RegKeys(root: Int, path: String): Array[String]
```

15.5 Annotations for Optimizations

Several annotations in the Scala library let you control compiler optimizations. They are discussed in the following sections.

15.5.1 Tail Recursion

A recursive call can sometimes be turned into a loop, which conserves stack space. This is important in functional programming where it is common to write recursive methods for traversing collections.

Consider this method that computes the sum of a sequence of integers using recursion:

```

object Util {
    def sum(xs: Seq[Int]): BigInt =
        if (xs.isEmpty) BigInt(0) else xs.head + sum(xs.tail)
    ...
}

```

This method cannot be optimized because the last step of the computation is addition, not the recursive call. But a slight transformation can be optimized:

```

def sum2(xs: Seq[Int], partial: BigInt): BigInt =
    if (xs.isEmpty) partial else sum2(xs.tail, xs.head + partial)

```

The partial sum is passed as a parameter; call this method as `sum2(xs, 0)`. Since the *last* step of the computation is a recursive call to the same method, it can be transformed into a loop to the top of the method. The Scala compiler automatically applies the “tail recursion” optimization to the second method. If you try

```
Util.sum(1 to 1000000)
```

you will get a stack overflow error (at least with the default stack size of the JVM), but

```
Util.sum2(1 to 1000000, 0)
```

returns the sum 500000500000.

Even though the Scala compiler will try to use tail recursion optimization, it is sometimes blocked from doing so for nonobvious reasons. If you rely on the compiler to remove the recursion, you should annotate your method with `@tailrec`. Then, if the compiler cannot apply the optimization, it will report an error.

For example, suppose the method is in a class instead of an object:

```

class Util {
    @tailrec def sum3(xs: Seq[Int], partial: BigInt): BigInt =
        if (xs.isEmpty) partial else sum3(xs.tail, xs.head +
}

```

```
partial)
```

```
...
```

Now the program fails with an error message "could not optimize @tailrec annotated method sum2: it is neither private nor final so can be overridden". In this situation, you can move the method into an object, or you can declare it as `private` or `final`.



Note

A more general mechanism for recursion elimination is “trampolining”. A trampoline implementation runs a loop that keeps calling functions. Each function returns the next function to be called. Tail recursion is a special case where each function returns itself. The more general mechanism allows for mutual calls —see the example that follows.

Scala has a utility object called `TailCalls` that makes it easy to implement a trampoline. The mutually recursive functions have return type `TailRec[A]` and return either `done(result)` or `tailcall(expr)` where `expr` is the next expression to be evaluated. The expression returns a `TailRec[A]`. Here is a simple example:

```
import scala.util.control.TailCalls.*  
def evenLength(xs: Seq[Int]): TailRec[Boolean] =  
    if (xs.isEmpty) done(true) else  
        tailcall(oddLength(xs.tail))  
def oddLength(xs: Seq[Int]): TailRec[Boolean] =  
    if (xs.isEmpty) done(false) else  
        tailcall(evenLength(xs.tail))
```

To obtain the final result from the `TailRec` object, use the `result` method:

```
evenLength(1 to 1000000).result
```

15.5.2 Lazy Values

A lazy value is initialized when it is first accessed:

```
lazy val words =  
  scala.io.Source.fromFile("/usr/share/dict/words").mkString.split(  
    "\n")
```

If you never use `words`, the file is not read in at all. If you use it multiple times, the file is only read with the first use.

Since it is possible for that first use to occur concurrently in multiple threads, each access of a lazy val invokes a method that acquires a lock.

If you know that you never have such a concurrent access, you can avoid the locking with the `@threadUnsafe` annotation:

```
@threadUnsafe lazy val words =  
  
  scala.io.Source.fromFile("/usr/share/dict/words").mkString.split(  
    "\n")
```

15.6 Annotations for Errors and Warnings

If you mark a feature with the `@deprecated` annotation, the compiler generates a warning whenever the feature is used. The annotation has two optional arguments, `message` and `since`.

```
@deprecated(message = "Use factorial(n: BigInt) instead")  
def factorial(n: Int): Int = ...
```

The `@deprecatedName` is applied to a parameter, and it specifies a former name for the parameter.

```
def display(message: String, @deprecatedName("sz") size: Int,  
  font: String = "Sans") = ...
```

You can still call `draw(sz = 12)` but you will get a deprecation warning.

The `@deprecatedInheritance` and `@deprecatedOverriding` annotations generate warnings that inheriting from a class or overriding a method is now deprecated.

Some Scala features are deemed experimental. To access them, you need to enter “experimental scope” with an `@experimental` annotation:

```
@experimental @newMain def main(name: String, age: Int) =  
  println(s"Hello $name, next year you'll be ${age + 1}")
```

The `@unchecked` annotation suppresses a warning that a match is not exhaustive. For example, suppose we know that a given list is never empty:

```
(lst: @unchecked) match  
  case head :: tail => ...
```

The compiler won’t complain that there is no `Nil` case. Of course, if `lst` is `Nil`, an exception is thrown at runtime.

The `@uncheckedVariance` annotation suppresses a variance error message. For example, it would make sense for `java.util.Comparator` to be contravariant. If `Student` is a subtype of `Person`, then a `Comparator[Person]` can be used when a `Comparator[Student]` is required. However, Java generics have no variance. We can fix this with the `@uncheckedVariance` annotation:

```
trait Comparator[-T] extends  
  java.util.Comparator[T @uncheckedVariance]
```

Finally, you can selectively hide warnings with the `@nowarn` annotation. For example, the `stop` method of the Java `Thread` class is deprecated. When you call

```
myThread.stop()
```

the compiler generates a deprecation warning. You can turn it off like this:

```
myThread.stop() : @nowarn
```

or

```
myThread.stop() : @nowarn("cat=deprecation") // Silences the  
deprecation category
```

Note

The optional argument of the `@nowarn` annotation can be any valid filter for the `-Wconf` compiler flag. Run the Scala command-line compiler with the `-Wconf:help` flag to get a summary of the filter syntax.

Tip

If you use the compiler flag `-Wunused:nowarn`, the compiler checks that each `@nowarn` annotation actually suppresses a warning message.

15.7 Annotation Declarations

I don't expect that many readers of this book will feel the urge to implement their own Scala annotations. The main point of this section is to be able to decipher the declarations of the existing annotation classes.

An annotation must extend the `Annotation` trait. For example, the `unchecked` annotation is defined as follows:

```
final class unchecked extends scala.annotation.Annotation
```

An annotation extending `StaticAnnotation` persists in class files:

```
class deprecatedName(name: String, since: String) extends  
StaticAnnotation
```

A `ConstantAnnotation` can only be constructed with numbers, Boolean values, strings, enumerations, class literals, and arrays thereof. Here is an example:

```
class SerialVersionUID(value: Long) extends ConstantAnnotation
```



Caution

The annotations that Scala places in class files are in a different format than Java annotations and cannot be read by the Java virtual machine. If you want to implement a new Java annotation, you need to write the annotation class in Java.

Generally, an annotation belongs only to the expression, variable, field, method, class, or type to which it is applied. For example, the annotation

```
def display(@nowarn message: String) = ""
```

applies only to one element: the parameter variable `message`.

However, field definitions in Scala can give rise to multiple features in Java, all of which can potentially be annotated. For example, consider

```
class Person(@JsonbProperty @BeanProperty var name: String)
```

Here, there are six items that can be targets for the `@JsonbProperty` annotation:

- The constructor parameter
- The private instance field
- The accessor method `name`
- The mutator method `name_=`
- The bean accessor `getName`
- The bean mutator `setName`

By default, constructor parameter annotations are only applied to the parameter itself, and field annotations are only applied to the field. The meta-annotations `@param`, `@field`, `@getter`, `@setter`, `@beanGetter`, and `@beanSetter` cause an annotation to be attached elsewhere. For example, the `@deprecated` annotation is defined as:

```
@getter @setter @beanGetter @beanSetter  
class deprecated(message: String = "", since: String = "")  
  extends ConstantAnnotation
```

You can also apply these annotations in an ad-hoc fashion:

```
@(JsonbProperty @beanGetter @beanSetter) @BeanProperty var name:  
String = null
```

In this situation, the `@JsonbProperty` annotation is applied to the Java `getName` method.

Exercises

1. Using the Java JUnit library, write a class with test cases in Scala. Use the `@Test` annotation and a couple of other annotations of your choice.
2. Make an example class that shows every possible position of an annotation. Use `@deprecated` as your sample annotation.
3. Which annotations from the Scala library use one of the meta-annotations `@param`, `@field`, `@getter`, `@setter`, `@beanGetter`, or `@beanSetter`?
4. Write a Scala method `sum` with variable integer arguments that returns the sum of its arguments. Call it from Java.
5. Write a Scala method that returns a string containing all lines of a file. Call it from Java.
6. Write a Scala object with a volatile Boolean field. Have one thread sleep for some time, then set the field to `true`, print a message, and exit. Another thread will keep checking whether the field is `true`. If so, it

prints a message and exits. If not, it sleeps for a short time and tries again. What happens if the variable is not volatile?

7. Make a class `student` with read-write JavaBeans properties `name` (of type `String`) and `id` (of type `Long`). What methods are generated? (Use `javap` to check.) Can you call the JavaBeans getters and setters in Scala? Should you?

8. Consider these recursive functions for repeatedly applying a function to an initial value until a fix point is found (that is, a value `x` such that $f(x) == x$).

```
def fix(f: Double => Double)(x: Double): Double = val y = f(x)
  if (x == y) x
  else fix(f)(y)

def fixpath(f: Double => Double)(x: Double): List[Double] =
  val y = f(x)
  if (x == y) List()
  else y :: fixpath(f)(y)
```

The first function yields the fix point, the second the sequence of all intermediate values. For example, try out

```
fix(Math.cos)(0)
fixpath(Math.cos)(0)
```

Which is tail recursive? When not, can you provide an implementation that is?

9. Give an example to show that the tail recursion optimization is not valid when a method can be overridden.

10. Try finding an experimental feature in your Scala release and use it with the `@experimental` annotation.

11. In Scala 3.2, the `@newMain` experimental feature substantially improved on the command line parsing of its `@main` predecessor. Perhaps it is no longer experimental by the time you read this, and hopefully it has a nicer name. Use it to write a `grep`-like application that

accepts on the command line a regular expression to search for, a file name, and flags for case sensitivity and inverting the match (that is, only printing non-matching lines).

12. Experiment with the `@nowarn` annotation and the filter syntax. Write some code that produces warnings and then turn them off with `@nowarn` and appropriate filters. What happens if you add `@nowarn` to an expression that doesn't produce a warning and use the `-Wunused:nowarn` flag? Can you turn that warning off with another `@nowarn`?

Can you apply `@nowarn` to something other than expressions?

Chapter 16

Futures

Topics in This Chapter A2

- [16.1 Running Tasks in the Future](#)
- [16.2 Waiting for Results](#)
- [16.3 The Try Class](#)
- [16.4 Callbacks](#)
- [16.5 Composing Future Tasks](#)
- [16.6 Other Future Transformations](#)
- [16.7 Methods in the Future Object](#)
- [16.8 Promises](#)
- [16.9 Execution Contexts](#)
- [Exercises](#)

Writing concurrent applications that work correctly and with high performance is very challenging. The traditional approach, in which concurrent tasks have side effects that mutate shared data, is tedious and error-prone. Scala encourages you to think of a computation in a functional way. A computation yields a value, sometime in the future. As long as the computations don't have side effects, you can let them run concurrently and combine the results when they become available. In this chapter, you will see how to use the `Future` and `Promises` traits to organize such computations.

The key points of this chapter are:

- A block of code wrapped in a `Future { ... }` executes concurrently.
- A future succeeds with a result or fails with an exception.
- You can wait for a future to complete, but you don't usually want to.
- You can use callbacks to get notified when a future completes, but that gets tedious when chaining callbacks.
- Use methods such as `map/flatMap`, or the equivalent `for` expressions, to compose futures.
- A promise has a future whose value can be set once.
- Pick an execution context that is suitable for the concurrent workload of your computation.

16.1 Running Tasks in the Future

The `scala.concurrent.Future` object can execute a block of code “in the future.”

```
import java.time.*
import scala.concurrent.*
given ExecutionContext = ExecutionContext.global

Future {
    Thread.sleep(10000)
    println(s"This is the future at ${LocalTime.now}")
}
println(s"This is the present at ${LocalTime.now}")
```

When running this code, a line similar to the following is printed:

```
This is the present at 13:01:19.400
```

About ten seconds later, a second line appears:

```
This is the future at 13:01:29.140
```

When you create a `Future`, its code is run on some thread. One could of course create a new thread for each task, but thread creation is not free. It is better to

keep some pre-created threads around and use them to execute tasks as needed. A data structure that assigns tasks to threads is usually called a *thread pool*. In Java, the `Executor` interface describes such a data structure. Scala uses the `ExecutionContext` trait instead.

Each `Future` must be constructed with a reference to an `ExecutionContext`. The simplest way is to use this statement, which uses syntax from [Chapter 19](#):

```
given ExecutionContext = ExecutionContext.global
```

Then the tasks execute on a global thread pool. This is fine for demos, but in a real program, you should make another choice if your tasks block. See [Section 16.9, “Execution Contexts,”](#) on page 258 for more information.

When you construct multiple futures, they can execute concurrently. For example, try running

```
Future { for (i <- 1 to 100) { print("A"); Thread.sleep(10) } }
Future { for (i <- 1 to 100) { print("B"); Thread.sleep(10) } }
```

You will get an output that looks somewhat like

```
ABABABABABABABABABABABABABA...AABABBBABABABABABABBBBBBBBBBBB  
BBB
```

A future can—and normally will—have a result:

```
val f = Future {
  Thread.sleep(10000)
  42
}
```

When you evaluate `f` in the REPL immediately after the definition, you will get this output:

```
res12: scala.concurrent.Future[Int] = Future(<not completed>)
```

Wait ten seconds and try again:

```
res13: scala.concurrent.Future[Int] = Future(Success(42))
```

Alternatively, something bad may happen in the future:

```
val f2 = Future {  
    if LocalTime.now.getMinute != 42 then  
        throw Exception("not a good time")  
    42  
}
```

Unless the minute happens to be 42, the task terminates with an exception. In the REPL, you will see

```
res14: scala.concurrent.Future[Int] =  
Future(Failure(java.lang.Exception: not a good time))
```

Now you know what a `Future` is. It is an object that will give you a result (or failure) at some point in the future. In the next section, you will see one way of harvesting the result of a `Future`.



Note

The `java.util.concurrent` package has a `Future` interface that is much more limited than the Scala `Future` trait. A Scala future is equivalent to the `CompletionStage` interface in Java.



Tip

The Scala language imposes no restrictions on what you can do in concurrent tasks. However, you should stay away from computations with side effects. It is best if you don't increment shared counters—even atomic ones. Don't populate shared maps—even threadsafe ones. Instead, have each future compute a value. Then you can combine the computed values after all contributing futures have completed. That way, each value is only owned by one task at a time, and it is easy to reason about the correctness of the computation.

16.2 Waiting for Results

When you have a `Future`, you can use the `isCompleted` method to check whether it is completed. But of course you don't want to wait for completion in a loop.

You can make a blocking call that waits for the result.

```
import scala.concurrent.duration.*  
val f = Future { Thread.sleep(10000); 42 }  
val result = Await.result(f, 10.seconds)
```

The call to `Await.result` blocks for ten seconds and then yields the result of the future.

The second argument of the `Await.result` method is a `Duration` object. Importing `scala.concurrent.duration.*` enables conversion methods from integers to `Duration` objects, called `seconds`, `millis`, and so on.

If the task is not ready by the allotted time, the `Await.ready` method throws a `TimeoutException`.

If the task throws an exception, it is rethrown in the call to `Await.result`. To avoid exceptions, you can call `Await.ready` and then get the result.

```
val f2 = Future {  
    Thread.sleep((10000 * scala.math.random()).toLong)  
    if scala.math.random() < 0.5 then throw Exception("Not your lucky  
    day")  
    42  
}  
val result2 = Await.ready(f2, 5.seconds).value
```

The `Await.ready` method yields its first parameter, and the `value` method of the `Future` class returns an `Option[Try[T]]`. It is `None` when the future is not completed and `Some(t)` when it is. Here, `t` is an object of the `Try` class, which holds either the result or the exception that caused the task to fail. You will see how to look inside it in the next section.



In practice, you won't use the `Await.result` or `Await.ready` methods much. You run tasks concurrently when they are time-consuming and your program can do something more useful than waiting for the result. [Section 16.4, “Callbacks,”](#) on page 250 shows you how you can harvest the results without blocking.



Caution

In this section, we used the `result` and `ready` methods of the `Await` object. The `Future` trait also has `result` and `ready` methods, but you should not call them. If the execution context uses a small number of threads (which is the case for the default fork-join pool), you don't want them all to block. Unlike the `Future` methods, the `Await` methods notify the execution context so that it can adjust the pooled threads.



Note

Not all exceptions that occur during execution of the future are stored in the `result`. Virtual machine errors and the `InterruptedException` are allowed to propagate in the usual way.

16.3 The `try` Class

A `Try[T]` instance is either a `Success(v)`, where `v` is a value of type `T` or a `Failure(ex)`, where `ex` is a `Throwable`. One way of processing it is with a `match` statement.

```
t match
  case Success(v) => println(s"The answer is $v")
  case Failure(ex) => println(ex.getMessage)
```

Alternatively, you can use the `isSuccess` or `isFailure` methods to find out whether the `Try` object represents success or failure. In the case of success, you can obtain the value with the `get` method:

```
if t.isSuccess then println(s"The answer is ${t.get}")
```

To get the exception in case of failure, first apply the `failed` method which turns the failed `Try[T]` object into a `Try[Throwable]` wrapping the exception. Then call `get` to get the exception object.

```
if t.isFailure then println(t.failed.get.getMessage)
```

You can also turn a `Try` object into an `Option` with the `toOption` method if you want to pass it on to a method that expects an option. This turns `Success` into `Some` and `Failure` into `None`.

To construct a `Try` object, call `Try(block)` with some block of code. For example,

```
val t = Try(str.toInt)
```

is either a `Success` object with the parsed integer, or a `Failure` wrapping a `NumberFormatException`.

There are several methods for composing and transforming `Try` objects. However, analogous methods exist for futures, where they are more commonly used. You will see how to work with multiple futures in [Section 16.5, “Composing Future Tasks,”](#) on page 250. At the end of that section, you will see how those techniques apply to `Try` objects.

16.4 Callbacks

As already mentioned, one does not usually use a blocking wait to get the result of a future. For better performance, the future should report its result to a callback function.

This is easy to arrange with the `onComplete` method.

```
f.onComplete(t => ...)
```

When the future has completed, either successfully or with a failure, it calls the given function with a `Try` object.

You can then react to the success or failure, for example by passing a match function to the `onComplete` method.

```

val f = Future {
    Thread.sleep(1000)
    if scala.math.random() < 0.5 then throw Exception("Not your lucky
day")
    42
}
f.onComplete {
    case Success(v) => println(s"The answer is $v")
    case Failure(ex) => println(ex.getMessage)
}

```

By using a callback, we avoid blocking. Unfortunately, we now have another problem. In all likelihood, the long computation in one `Future` task will be followed by another computation, and another. It is possible to nest callbacks within callbacks, but it is profoundly unpleasant. (This technique is sometimes called “callback hell”).

A better approach is to think of futures as entities that can be composed, similar to functions. You compose two functions by calling the first one, then passing its result to the second one. In the next section, you will see how to do the same with futures.

16.5 Composing Future Tasks

Suppose we need to get some information from two web services and then combine the two. Each task is long-running and should be executed in a `Future`. It is possible to link them together with callbacks:

```

val future1 = Future { getData1() }
val future2 = Future { getData2() }

future1 onComplete {
    case Success(n1) =>
        future2 onComplete {
            case Success(n2) => {
                val n = n1 + n2
                println(s"Sum: $n")
            }
        }
}
```

```

        case Failure(ex) => ex.printStackTrace()
    }
case Failure(ex) => ex.printStackTrace()
}

```

Even though the callbacks are ordered sequentially, the tasks run concurrently. Each task starts after the `Future.apply` method executes or soon afterwards. We don't know which of `future1` and `future2` completes first, and it doesn't matter. We can't process the result until both tasks complete. Once `future1` completes, its completion handler registers a completion handler on `future2`. If `future2` has already completed, the second handler is called right away. Otherwise, it is called when `future2` finally completes.

Even though this chaining of the futures works, it looks very messy, and it will look worse with each additional level of processing.

Instead of nesting callbacks, we will use an approach that you already know from working with Scala collections. Think of a `Future` as a collection with (hopefully, eventually) one element. You know how to transform the values of a collection—with `map`:

```

val future1 = Future { getData1() }
val combined = future1.map(n1 => n1 + getData2())

```

Here `future1` is a `Future[Int]`—a collection of (hopefully, eventually) one value. We map a function `Int => Int` and get another `Future[Int]`—a collection of (hopefully, eventually) one integer.

But wait—that's not quite the same as in the callback code. The call to `getData2` is running *after* `getData1`, not concurrently. Let's fix that with a second `map`:

```

val future1 = Future { getData1() }
val future2 = Future { getData2() }
val combined = future1.map(n1 => future2.map(n2 => n1 + n2))

```

When `future1` and `future2` have delivered their results, the sum is computed.

Unfortunately, now `combined` is a `Future[Future[Int]]`, which isn't so good. That's what `flatMap` is for:

```
val combined = future1.flatMap(n1 => future2.map(n2 => n1 + n2))
```

This looks much nicer when you use a `for` expression instead of chaining `flatMap` and `map`:

```
val combined = for n1 <- future1; n2 <- future2 yield n1 + n2
```

This is exactly the same code since `for` expressions are translated to chains of `map` and `flatMap`.

You can also apply guards in the `for` expression:

```
val combined =  
  for n1 <- future1; n2 <- future2 if n1 != n2 yield n1 + n2
```

If the guard fails, the computation fails with a `NoSuchElementException`.

What if something goes wrong? The `map` and `flatMap` implementations take care of all that. As soon as one of the tasks fails, the entire pipeline fails, and the exception is captured. In contrast, when you manually combine callbacks, you have to deal with failure at every step.

So far, you have seen how to run two tasks concurrently. Sometimes, you need one task to run after another. A `Future` starts execution immediately when it is created. To delay the creation, use functions.

```
val future1 = Future { getData1() }  
def future2 = Future { getData2() } // def, not val  
val combined = for (n1 <- future1; n2 <- future2) yield n1 + n2
```

Now `future2` is only evaluated when `future1` has completed.

It doesn't matter whether you use `val` or `def` for `future1`. If you use `def`, its creation is slightly delayed to the start of the `for` expression.

This is particularly useful if the second step depends on the output of the first:

```
def future1 = Future { getData() }  
def future2(arg: Int) = Future { getMoreData(arg) }  
val combined = for (n1 <- future1; n2 <- future2(n1)) yield n1 + n2
```

 Note

Like the `Future` trait, the `Try` class from [Section 16.3, “The Try Class,”](#) on page 249 has `map` and `flatMap` methods. A `Try[T]` is a collection of, hopefully, one element. It is just like a `Future[T]`, except you don’t have to wait. You can apply `map` with a function that changes that one element, or `flatMap` if you have `Try`-valued function and want to flatten the result. And you can use `for` expressions. For example, here is how to compute the sum of two function calls that might fail:

```
def readInt(prompt: String) = Try(StdIn.readLine(s"$prompt:"))
                                         .toInt)
val combined =
  for (n1 <- readInt("n1"); n2 <- readInt("n2")) yield n1 +
  n2
```

In this way, you can compose `Try`-valued computations and you don’t need to deal with the boring part of error handling.

16.6 Other `Future` Transformations

The `map` and `flatMap` methods that you saw in the preceding section are the most fundamental transformation of `Future` objects.

[Table 16–1](#) shows several ways of applying functions to the contents of a future that differ in subtle details.

The `foreach` method works exactly like it does for collections, applying a method for its side effect. The method is applied to the single value in the future. It is convenient for harvesting the answer when it materializes.

```
val combined = for (n1 <- future1; n2 <- future2) yield n1 + n2
combined.foreach(n => println(s"Sum: $n"))
```

Table 16–1 Transformations on a `Future[T]` with success value `v` or exception `ex`

Method	Result Type	Description
<code>collect(pf: PartialFunction[T, S])</code>	<code>Future[S]</code>	Like <code>map</code> , but with a partial function. The result fails with a <code>NoSuchElementException</code> if <code>pf(v)</code> is not defined.
<code>foreach(f: T => U)</code>	<code>Unit</code>	Calls <code>f(v)</code> like <code>map</code> , but only for its side effect.
<code>andThen(pf: PartialFunction[Try[T], U])</code>	<code>Future[T]</code>	Calls <code>pf(v)</code> for its side effect and returns a future with <code>v</code> .
<code>filter(p: T => Boolean)</code>	<code>Future[T]</code>	Calls <code>p(v)</code> and returns a future with <code>v</code> or a <code>NoSuchElementException</code> .
<code>recover(pf: PartialFunction[Throwable, U])</code> <code>recoverWith(pf: PartialFunction[Throwable, Future[U]])</code>	<code>Future[U]</code> (where <code>U</code> is a supertype of <code>T</code>)	A future with value <code>v</code> or <code>pf(ex)</code> , flattened in the asynchronous case
<code>fallbackTo(f2: Future[U])</code>	<code>Future[U]</code> (where <code>U</code> is a supertype of <code>T</code>)	A future with value <code>v</code> , or if this future failed, with the value of <code>f2</code> , or if that also failed, with exception <code>ex</code> .
<code>failed</code>	<code>Future[Throwable]</code>	A future with value <code>ex</code> .
<code>transform(s: T => S, f: Throwable => Throwable)</code> <code>transform(f: Try[T] => Try[S])</code> <code>transformWith(f: Try[T] => Future[Try[S]])</code>	<code>Future[S]</code>	Transforms both the success and failure.
<code>zip(f2: Future[U])</code>	<code>Future[(T, U)]</code>	A future with a pair holding <code>v</code> and the value of <code>f2</code> , or <code>ex</code> if this future fails, or the failure of <code>f2</code>
<code>zipWith(f2: Future[U])(f: (T, U) => R)</code>	<code>Future[R]</code>	Zips both futures and applies <code>f</code>
<code>flatten</code>	<code>Future[S]</code> (where <code>T</code> is <code>Future[S]</code>)	Flattens a <code>Future[Future[S]]</code> into a <code>Future[S]</code>

The `recover` method accepts a partial function that can turn an exception into a successful result. Consider this call:

```
val f = Future { persist(data) } recover { case e: SQLEXception => 0 }
```

If a `SQLEXception` occurs, the future succeeds with result 0.

The `fallbackTo` method provides a different recovery mechanism. When you call `f.fallbackTo(f2)`, then `f2` is executed if `f` fails, and its value becomes the value of the future. However, `f2` cannot inspect the reason for the failure.

The `failed` method turns a failed `Future[T]` into a successful `Future[Throwable]`, just like the `Try.failed` method. You can retrieve the failure in a `for` expression like this:

```
val f = Future { persist(data) }
for v <- f do println(s"Succeeded with $v")
for ex <- f.failed do println(s"Failed with $ex")
```

Finally, you can zip two futures together. The call `f1.zip(f2)` yields a future whose result is a pair `(v, w)` if `v` was the result of `f1` and `w` the result of `f2`, or an exception if either `f1` or `f2` failed. (If both fail, the exception of `f1` is reported.)

The `zipWith` method is similar, but it takes a method to combine the two results instead of returning a pair. For example, here is another way of obtaining the sum of two computations:

```
val future1 = Future { getData1() }
val future2 = Future { getData2() }
val combined = future1.zipWith(future2) (_ + _)
```

16.7 Methods in the `Future` Object

The `Future` companion object contains useful methods for working on collections of futures.

Suppose that, as you are computing a result, you organize the work so that you can concurrently work on different parts. For example, each part might be a subsequence of the inputs. Make a future for each part:

```
val futures = parts.map(part => Future { process(part) })
```

Now you have a collection of futures. Often, you want to combine the results. By using the `Future.sequence` method, you can get a collection of all results for further processing:

```
val result = Future.sequence(futures);
```

Note that the call doesn't block—it gives you a future to a collection. For example, assume `futures` is a `seq[Future[T]]`. Then the result is a `Future[Seq[T]]`. When the results for all elements of `futures` are available, the `result` future will complete with a sequence of the results.

If any of the futures fail, then the resulting future fails as well with the exception of the leftmost failed future. If multiple futures fail, you don't get to see the remaining failures.

The `traverse` method combines the `map` and `sequence` steps. Instead of

```
val futures = parts.map(p => Future { process(p) })
val result = Future.sequence(futures);
```

you can call

```
val result = Future.traverse(parts)(p => Future { process(p) })
```

The function in the second curried argument is applied to each element of `parts`. You get a future to a collection of all results.

There are `reduceLeft` and `foldLeft` operations on iterables of futures. You supply an operation that combines the results of all futures as they become available. For example, here is how you can compute the sum of the results:

```
val result = Future.reduceLeft(futures)(_ + _)
// Yields a future to the sum of the results of all futures
```

So far, we have collected the results from all futures. Suppose you are willing to accept a result from any of the parts. Then call

```
val result = Future.firstCompletedOf(futures)
```

You get a future that, when it completes, has the result or failure of the first completed element of `futures`.

The `find` method is similar, but you also supply a predicate.

```
val result = Future.find(futures) (predicate)
// Yields a Future[Option[T]]
```

You get a future that, when it completes successfully, yields `Some(r)`, where `r` is the result of one of the given futures that fulfills the predicate. Failed futures are ignored. If all futures complete but none yields a result that matches the predicate, then `find` returns `None`. Note that the `predicate` parameter has type `Option[T]`.

Caution

A potential problem with `firstCompletedOf` and `find` is that the other computations keep on going even when the result has been determined. Scala futures do not have a mechanism for cancellation.

The `Future.delegate` method runs a `Future`-producing function and flattens the result:

```
def future1 = Future { getData() } // Note def, not val
val result = Future.delegate(future1) // A Future[T], not a
Future[Future[T]]
```

Finally, the `Future` object provides convenience methods for generating simple futures:

- `Future.successful(r)` is an already completed future with result `r`.
- `Future.failed(e)` is an already completed future with exception `e`.
- `Future.fromTry(t)` is an already completed future with the result or exception given in the `Try` object `t`.
- `Future.unit` is an already completed future with `Unit` result.
- `Future.never` is a future that never completes.

16.8 Promises

A `Future` object is read-only. The result of the future is set implicitly when its task has completed or failed. It cannot be set explicitly.

As a consumer of a `Future`, you would never want to set the result. The point of a `Future` is to process a result once it is ready.

However, if you produce a `Future` for others to consume, the task mechanism only works for synchronous computations. If you use an asynchronous API, you are called back when the result is available. That's the point where you want to set the result and complete the `Future`. You use a `Promise` to make that work.

Calling `success` on a promise sets the result. Alternatively, you can call `failure` with an exception to make the promise fail. As soon as one of these methods is called, the associated future is completed, and neither method can be called again. (An `IllegalStateException` is thrown otherwise.)

Here is a typical work flow:

```
def computeAnswer(arg: String) = {  
    val p = Promise[String]()  
    def onSuccess(result: String) = p.success(result)  
    def onFailure(ex: Throwable) = p.failure(ex)  
    startAsyncWork(arg, onSuccess, onFailure)  
    p.future  
}
```

Calling `future` on a promise yields the associated `Future` object. Note that the method returns the `Future` right away, immediately after starting the work that will eventually yield the result.

From the point of view of the consumer (that is, the caller of the `computeAnswer` method), there is no difference between a `Future` that was constructed with a task function and one that was produced from a `Promise`. Either way, the consumer gets the result when it is ready.

The producer, however, has more flexibility when using a `Promise`. For example, multiple tasks can work concurrently to fulfill a single promise. When one of the tasks has a result, it calls `trySuccess` on the promise. Unlike the

`success` method, that method accepts the result and returns `true` if the promise has not yet completed; otherwise it returns `false` and ignores the result.

```
val p = Promise[String]()
Future {
    val result = workHard(arg)
    p.trySuccess(result)
}
Future {
    val result = workSmart(arg)
    p.trySuccess(result)
}
```

The promise is completed by the first task that manages to produce the result. With this approach, the tasks might want to periodically call `p.isCompleted` to check whether they should continue.



Note

Scala promises are equivalent to the `CompletableFuture` class in Java 8.

16.9 Execution Contexts

By default, Scala futures are executed on the global fork-join pool. That works well for computationally intensive tasks. However, the fork-join pool only manages a small number of threads (by default, equal to the number of cores of all processors). This is a problem when tasks have to wait, for example when communicating with a remote resource. A program could exhaust all available threads, waiting for results.

You can notify the execution context that you are about to block, by placing the blocking code inside `blocking { ... }`:

```
val f = Future {
    val url = "https://horstmann.com/index.html"
    blocking {
```

```

    val contents = Source.fromURL(url).mkString
    if contents.length < 300
    then contents
    else contents.substring(0, 300) + "..."
}
}

```

The execution context may then increase the number of threads. The fork-join pool does exactly that, but it isn't designed for perform well for many blocking threads. If you do input/output or connect to databases, you are better off using a different thread pool. The `Executors` class from the Java concurrency library gives you several choices. A cached thread pool works well for I/O intensive workloads. You can pass it explicitly to the `Future.apply` method, or you can set it as the given execution context:

```

val pool = Executors.newCachedThreadPool()
given ExecutionContext = ExecutionContext.fromExecutor(pool)

```

Now this pool is used by all futures where the `given` declaration is in scope. (See [Chapter 19](#) for more information about `given` declarations.)

Exercises

1. Consider the expression

```

for
  n1 <- Future { Thread.sleep(1000) ; 2 }
  n2 <- Future { Thread.sleep(1000); 40 }
do
  println(n1 + n2)

```

How is the expression translated to `map` and `flatMap` calls? Are the two futures executed concurrently or one after the other? In which thread does the call to `println` occur?

2. Write a function `doInOrder` that, given two functions `f: T => Future[U]` and `g: U => Future[V]`, produces a function `T => Future[V]` that, for a given `t`, eventually yields `g(f(t))`.

3. Repeat the preceding exercise for any sequence of functions of type `T => Future[T]`.
4. Write a function `doTogether` that, given two functions `f: T => Future[U]` and `g: U => Future[V]`, produces a function `T => Future[(U, V)]`, running the two computations in parallel and, for a given `t`, eventually yielding `(f(t), g(t))`.
5. Write a function that receives a sequence of futures and returns a future that eventually yields a sequence of all results.
6. Write a method

```
Future[T] repeat(action: => T, until: T => Boolean)
```

that asynchronously repeats the action until it produces a value that is accepted by the `until` predicate, which should also run asynchronously. Test with a function that reads a password from the console, and a function that simulates a validity check by sleeping for a second and then checking that the password is "secret". Hint: Use recursion.

7. Write a program that counts the prime numbers between 1 and n , as reported by `BigInt.isProbablePrime`. Divide the interval into p parts, where p is the number of available processors. Count the primes in each part in concurrent futures and combine the results.
8. Write a program that asks the user for a URL, reads the web page at that URL, and displays all the hyperlinks. Use a separate `Future` for each of these three steps.
9. Write a program that asks the user for a URL, reads the web page at that URL, finds all the hyperlinks, visits each of them concurrently, and locates the `Server` HTTP header for each of them. Finally, print a table of which servers were found how often. The futures that visit each page should return the header.
10. Change the preceding exercise where the futures that visit each header update a shared Java `ConcurrentHashMap` or Scala `TrieMap`. This isn't as easy as it sounds. A threadsafe data structure is safe in the sense that you cannot corrupt its implementation, but you have to make sure that sequences of reads and updates are atomic.
11. In the preceding exercise, you updated a mutable `Map[URL, String]`. Consider what happens when two threads concurrently query for the same

key whose result is not yet present. Then both threads expend effort computing the same value. Avoid this problem by using a `Map[URL, Future[String]]` instead.

12. Using futures, run four tasks that each sleep for ten seconds and then print the current time. If you have a reasonably modern computer, it is very likely that it reports four available processors to the JVM, and the futures should all complete at around the same time. Now repeat with forty tasks. What happens? Why? Replace the execution context with a cached thread pool. What happens now? (Be careful to define the futures *after* declaring the given execution context.)
13. Using Swing or JavaFX, implement a function that returns a future for a button click. Use a promise to set the value to the button label when the button is clicked. Fail the promise when a timeout has expired.
14. Write a method that, given a URL, locates all hyperlinks, makes a promise for each of them, starts a task in which it will eventually fulfill all promises, and returns a sequence of futures for the promises. Why would it not be a good idea to return a sequence of promises?
15. Use a promise for implementing cancellation. Given a range of big integers, split the range into subranges that you concurrently search for palindromic primes. When such a prime is found, set it as the value of the future. All tasks should periodically check whether the promise is completed, in which case they should terminate.

Chapter 17

Type Parameters

Topics in This Chapter L2

- 17.1 Generic Classes
- 17.2 Generic Functions
- 17.3 Bounds for Type Variables
- 17.4 Context Bounds
- 17.5 The `ClassTag` Context Bound
- 17.6 Multiple Bounds
- 17.7 Type Constraints L3
- 17.8 Variance
- 17.9 Co- and Contravariant Positions
- 17.10 Objects Can't Be Generic
- 17.11 Wildcards
- 17.12 Polymorphic Functions
- Exercises

In Scala, you can use type parameters to implement classes and functions that work with multiple types. For example, an `Array[T]` stores elements of an arbitrary type `T`. The basic idea is very simple, but the details can get

tricky. Sometimes, you need to place restrictions on the type. For example, to sort elements, `T` must provide an ordering. Furthermore, if the parameter type varies, what should happen with the parameterized type? For example, can you pass an `Array[String]` to a function that expects an `Array[Any]`? In Scala, you specify how your types should vary depending on their parameters.

The key points of this chapter are:

- Classes, traits, methods, and functions can have type parameters.
- Place the type parameters after the name, enclosed in square brackets.
- Type bounds have the form `T <: UpperBound, T >: LowerBound, T : ContextBound`.
- You can restrict a method with a type constraint such as `(given ev: T <:< UpperBound)`.
- Use `+T` (covariance) to indicate that a generic type's subtype relationship is in the same direction as the parameter `T`, or `-T` (contravariance) to indicate the reverse direction.
- Covariance is appropriate for parameters that denote outputs, such as elements in an immutable collection.
- Contravariance is appropriate for parameters that denote inputs, such as function arguments.

17.1 Generic Classes

As in Java or C++, classes and traits can have type parameters. In Scala, you use square brackets for type parameters, for example: `class Pair[T, S](val first: T, val second: S)`

This defines a class with two type parameters `T` and `S`. You use the type parameters in the class definition to define the types of variables, method parameters, and return values.

A class with one or more type parameters is *generic*. If you substitute actual types for the type parameters, you get an ordinary class, such as `Pair[Int, String]`.

Pleasantly, Scala attempts to infer the actual types from the construction parameters:

```
val p = Pair(42, "String") // It's a Pair[Int, String]
```

You can also specify the types yourself:

```
val p2 = Pair[Any, Any](42, "String")
```



Note

Of course, Scala has a pair type `(T, S)`, so there is no actual need for a `Pair[T, S]` class. However, this class and its variations provide simple and convenient examples for discussing the finer points of type parameters.



Note

You can write any generic type with two type parameters in infix notation, such as `Double Map String` instead of `Map[Double, String]`.

If you have such a type, you can annotate it as `@showAsInfix` to modify how it is displayed in compiler and REPL messages. For example, if you define

```
@showAsInfix class x[T, U](val first: T, val second: U)
```

then the type `x[String, Int]` is displayed as `String x Int`.

17.2 Generic Functions

Functions and methods can also have type parameters. Here is a simple example:

```
def getMiddle[T](a: Array[T]) = a(a.length / 2)
```

As with generic classes, you place the type parameter after the name.
Scala infers the actual types from the arguments in the call.

```
getMiddle(Array("Mary", "had", "a", "little", "lamb")) // Calls  
getMiddle[String]
```

If you need to, you can specify the type:

```
val f = getMiddle[String] // The function, saved in f
```

17.3 Bounds for Type Variables

Sometimes, you need to place restrictions on type variables. Consider a generic `Pair` where both components have the same type, like this:

```
class Pair[T](val first: T, val second: T)
```

Now we want to add a method that produces the smaller value:

```
class Pair[T](val first: T, val second: T) :  
    def smaller = if first.compareTo(second) < 0 then first else  
        second // Error
```

That's wrong—we don't know if `first` has a `compareTo` method. To solve this, we can add an *upper bound* `T <: Comparable[T]`.

```
class Pair[T <: Comparable[T]](val first: T, val second: T) :  
    def smaller = if first.compareTo(second) < 0 then first else  
        second
```

This means that `T` must be a subtype of `Comparable[T]`.

Now we can instantiate `Pair[java.lang.String]` but not `Pair[java.net.URL]`, since `String` is a subtype of `Comparable[String]` but `URL` does not implement `Comparable[URL]`. For example:

```
val p = Pair("Fred", "Brooks")
p.smaller // "Brooks"
```

Note

In this chapter, I use the `Comparable` type from Java for several examples. In Scala, it is more common to use the `Ordering` trait. See the next section for details.

Caution

If you construct a `Pair(4, 2)`, you will get a `Pair[java.lang.Integer]` since the Scala `Int` type does not extend the `java.util.Comparable` trait.

You can also specify a lower bound for a type. For example, suppose we want to define a method that replaces the first component of a pair with another value. Our pairs are immutable, so we need to return a new pair. Here is a first attempt:

```
class Pair[T] (val first: T, val second: T) :
  def replaceFirst(newFirst: T) = Pair[T](newFirst, second)
```

But we can do better than that. Suppose we have a `Pair[Student]`. It should be possible to replace the first component with a `Person`. Of course, then the result must be a `Pair[Person]`. In general, the replacement type must be a supertype of the pair's component type. Use the `>:` symbol for the supertype relationship: `def replaceFirst[R >: T] (newFirst: R) = Pair[R](newFirst, second)`

Caution

The supertype relationship symbol is `>:`, even though `:>` would be more symmetrical. This is analogous to the `<=` and `>=` operators.

The example uses the type parameter in the returned pair for greater clarity. You can also write

```
def replaceFirst[R >: T](newFirst: R) = Pair(newFirst, second)
```

Then the return type is correctly inferred as `Pair[R]`.



Caution

If you omit the lower bound,

```
def replaceFirst[R](newFirst: R) = Pair(newFirst, second)
```

the method will compile, but it will return a `Pair[Any]`.

17.4 Context Bounds

A *context bound* has the form `T : M`, where `M` is another generic type. It requires that there is a “given value” of type `M[T]`. We discuss given values in detail in [Chapter 19](#).

For example,

```
class Pair[T : Ordering]
```

requires that there is a given value of type `Ordering[T]`. That given value can then be used in the methods of the class. When you declare a method that uses the given value, you have to add a “using parameter.” Here is an example:

```
class Pair[T : Ordering](val first: T, val second: T) :  
  def smaller(using ord: Ordering[T]) =
```

```
if ord.compare(first, second) < 0 then first else second
```

A type parameter can have both context bounds and subtype bounds. In that case, subtype bounds come first:

```
class Pair[T <: Serializable : Ordering]
```

17.5 The `ClassTag` Context Bound

To instantiate a generic `Array[T]`, one needs a `ClassTag[T]` object. This is required for primitive type arrays to work correctly. For example, if `T` is `Int`, you want an `int[]` array in the virtual machine. If you write a generic function that constructs a generic array, you need to help it out and pass that class tag object. Use a context bound, like this:

```
import scala.reflect._

def makePair[T : ClassTag](first: T, second: T) =
  Array[T](first, second)
```

If you call `makePair(4, 9)`, the compiler locates the given `ClassTag[Int]` instance and actually calls `makePair(4, 9)(classTag)`. Then the `Array` constructor is translated to a call `classTag newArray`, which in the case of a `ClassTag[Int]` constructs a primitive array of type `int[]`.

Why all this complexity? In the virtual machine, generic types are erased. There is only a single `makePair` method that needs to work for *all* types `T`.

17.6 Multiple Bounds

A type variable can have both an upper and a lower bound. The syntax is this:

```
T >: Lower <: Upper
```

You can't have multiple upper or lower bounds. However, you can still require that a type implements multiple traits, like this:

```
T <: Comparable[T] with Serializable with Cloneable
```

You can have more than one context bound:

```
T : Ordering : ClassTag
```

17.7 Type Constraints L3

Type constraints give you another way of restricting types. There are two relationships that you can use:

```
T <:< U  
T =:= U
```

These constraints test whether `T` is a subtype of `U` or whether `T` and `U` are the same type. To use such a constraint, you add a “using parameter” like this:

```
class Pair[T](val first: T, val second: T)(using ev: T <:<  
Comparable[T]) :  
  def smaller = if first.compareTo(second) < 0 then first else  
    second
```



See [Chapter 19](#) for an explanation of the syntax, and for an analysis of the inner workings of the type constraints.

In the example above, there is no advantage to using a type constraint over a type bound `class Pair[T <: Comparable[T]]`. However, type constraints give you more control in specialized circumstances.

In particular, type constraints let you supply a method in a generic class that only makes sense for some types. Here is an example:

```

class Pair[T](val first: T, val second: T) :
  def smaller(using ev: T <:< Comparable[T]) =
    if first.compareTo(second) < 0 then first else second

```

You can form a `Pair[URL]`, even though `URL` is not a subtype of `Comparable[URL]`. You will get an error only if you invoke the `smaller` method.

The `orNull` method in the `Option` class also uses a type constraint. Here is how to use the method:

```

val friends = Map("Fred" -> "Barney", ...)
val friendOpt = friends.get("Wilma") // An Option[String]
val friendOrNull = friendOpt.orNull // A String or null

```

The `orNull` method can be useful when working with Java code where it is common to encode missing values as `null`.

But now consider this code:

```

val numberOpt = if scala.math.random() < 0.5 then Some(42) else
  None
val number: Int = numberOpt.orNull // Error

```

The second line should not be legal. After all, the `Int` type doesn't have `null` as a valid value. For that reason, `orNull` is implemented using a constraint `Null <:< A`. You can instantiate `Option[Int]`, as long as you stay away from `orNull` for those instances.

See on page for another example with the `=:=` constraint.

17.8 Variance

Suppose we have a function that does something with a `Pair[Person]`:

```
def makeFriends(p: Pair[Person]) = ...
```

If `Student` is a subclass of `Person`, can I call `makeFriend` with a `Pair[Student]`? By default, this is an error. Even though `Student` is a subtype of `Person`, there is *no* relationship between `Pair[Student]` and `Pair[Person]`.

If you want such a relationship, you have to indicate it when you define the `Pair` class:

```
class Pair[+T] (val first: T, val second: T)
```

The `+` means that the type is *covariant* in `T`—that is, it varies in the same direction. Since `Student` is a subtype of `Person`, a `Pair[Student]` is now a subtype of `Pair[Person]`.

It is also possible to have variance in the other direction. Consider a generic type `Friend[T]`, which denotes someone who is willing to befriend anyone of type `T`.

```
trait Friend[T] :  
    def befriend(someone: T): String
```

Now suppose you have a function

```
def makeFriendWith(s: Student, f: Friend[Student]) =  
    f.befriend(s)
```

Can you call it with a `Friend[Person]`? That is, if you have

```
class Person(name: String) extends Friend[Person] :  
    override def toString = s"Person $name"  
    override def befriend(someone: Person) = s"$this and $someone  
are now friends"  
class Student(name: String, major: String) extends Person(name) :  
    override def toString = s"Student $name majoring in $major"  
val susan = Student("Susan", "CS")  
val fred = Person("Fred")
```

will the call `makeFriendWith(susan, fred)` succeed? It seems like it should. If Fred is willing to befriend any person, he'll surely like to be friends with

Susan.

Note that the type varies in the opposite direction of the subtype relationship. `Student` is a subtype of `Person`, but `Friend[Student]` needs to be a supertype of `Friend[Person]`. In that case, you declare the type parameter to be *contravariant*:

```
trait Friend[-T] :  
  def befriend(someone: T): String
```

You can have both variance types in a single generic type. For example, single-argument functions have the type `Function1[-A, +R]`. To see why these are the appropriate variances, consider a function

```
def friendsOfStudents(students: Seq[Student],  
                     findFriend: Function1[Student, Person]) =  
  // You can write the second parameter as findFriend: Student  
  => Person  
  for s <- students yield findFriend(s)
```

Suppose you have a function

```
def findStudentFriendsOfPerson(p: Person) : Student = ...
```

Can you call `friendsOfStudents` with that function? Of course you can. It's willing to take any person, so surely it will take a `Student`. That's contravariance in the first type parameter of `Function1`. It yields `Student` results, which can be put into a collection of `Person` objects. The second type parameter is covariant.

17.9 Co- and Contravariant Positions

In the preceding section, you saw that functions are contravariant in their arguments and covariant in their results. Generally, it makes sense to use contravariance for the values an object consumes, and covariance for the values it produces. (Aide-mémoire: **contravariance consumes.**)

If an object does both, then the type should be left *invariant*. This is generally the case for mutable data structures. For example, in Scala, arrays are invariant. You can't convert an `Array[Student]` to an `Array[Person]` or the other way around. This would not be safe. Consider the following:

```
val students = Array[Student](length)
val people: Array[Person] = students // Not legal, but suppose it
was ...
people(0) = Person("Fred") // Oh no! Now students(0) isn't a
Student
```

Conversely,

```
val people = Array[Person](length)
val students: Array[Student] = people // Not legal, but suppose it
was ...
people(0) = Person("Fred") // Oh no! Now students(0) isn't a
Student
```



Note

In Java, it is possible to convert a `Student[]` array to a `Person[]` array, but if you try to add a nonstudent into such an array, an `ArrayStoreException` is thrown. In Scala, the compiler rejects programs that could cause type errors.



The `IArray[+T]` class is a wrapper for immutable JVM arrays. Note that the element type is covariant.

Suppose we tried to declare a covariant *mutable* pair. This wouldn't work. It would be like an array with two elements, and one could produce the same

kind of error that you just saw.

Indeed, if you try

```
class Pair[+T] (var first: T, var second: T) // Error
```

you get an error complaining that the covariant type `T` occurs in a *contravariant position* in the setter

```
first_= (value: T)
```

Parameters are contravariant positions, and return types are covariant.

However, inside a function parameter, the variance flips—its parameters are covariant. For example, look at the `foldLeft` method of `Iterable[+A]`:

```
foldLeft[B] (z: B) (op: (B, A) => B): B  
- + + - +
```

Note that `A` is now in a covariant position.

These position rules are simple and safe, but they sometimes get in the way of doing something that would be risk-free. Consider the `replaceFirst` method from [Section 17.3, “Bounds for Type Variables,”](#) on page 265 in an immutable pair:

```
class Pair[+T] (val first: T, val second: T) :  
  def replaceFirst(newFirst: T) = Pair[T](newFirst, second) //  
Error
```

The compiler rejects this, because the parameter type `T` is in a contravariant position. Yet this method cannot damage the pair—it returns a new pair.

The remedy is to come up with a second type parameter for the method, like this:

```
def replaceFirst[R >: T](newFirst: R) = Pair[R](newFirst, second)
```

Now the method is a generic method with another type parameter `R`. But `R` is *invariant*, so it doesn’t matter that it appears in a contravariant position.

17.10 Objects Can't Be Generic

It is not possible to add type parameters to objects. Consider, for example, immutable lists. A list with element type `T` is either empty, or it has a head of type `T` and a tail of type `List[T]`:

```
abstract sealed class List[+T] :  
    def isEmpty: Boolean  
    def head: T  
    def tail: List[T]  
    def toString = if isEmpty then "" else s"$head $tail"  
  
case class NonEmpty[T] (head: T, tail: List[T]) extends List[T] :  
    def isEmpty = false  
  
case class Empty[T] extends List[T] :  
    def isEmpty = true  
    def head = throw UnsupportedOperationException()  
    def tail = throw UnsupportedOperationException()
```



Note

Here I use `NonEmpty` and `Empty` for clarity. They correspond to `::` and `Nil` in Scala lists.

It seems silly to define `Empty` as a class. It has no state. But you can't simply turn it into an object:

```
case object Empty[T] extends List[T] : // Error
```

You can't add a type parameter to an object. In this case, a remedy is to inherit `List[Nothing]`:

```
case object Empty extends List[Nothing] :
```

Recall from [Chapter 8](#) that the `Nothing` type is a subtype of all types. Thus, when we make a one-element list

```
val lst = NonEmpty(42, Empty)
```

type checking is successful. Due to covariance, a `List[Nothing]` is convertible into a `List[Int]`, and the `NonEmpty[Int]` constructor can be invoked.

17.11 Wildcards

In Java, all generic types are invariant. However, you can vary the types where you use them, using wildcards. For example, a method

```
void makeFriends(List<? extends Person> people) // This is Java
```

can be called with a `List<Student>`.

You can use wildcards in Scala too. They look like this:

```
def makeFriends(people: java.util.List[? <: Person]) = ... //  
This is Scala
```

In Scala, you don't need the wildcard for a covariant `Pair` class. But suppose `Pair` is invariant:

```
class Pair[T] (var first: T, var second: T)
```

Then you can define

```
def makeFriends(p: Pair[? <: Person]) = ... // OK to call with a  
Pair[Student]
```

You can also use wildcards for contravariance:

```
import java.util.Comparator  
def min[T] (p: Pair[T], comp: Comparator[? >: T]) =
```

```
if comp.compare(p.first, p.second) < 0 then p.first else  
p.second
```

If necessary, you can have both variances. Here `T` must be a subtype of `Comparable[U]`, where `U` (the parameter of the `compare` method) must be a supertype of `T`, so that it can accept a value of type `T`:

```
def min[T <: Comparable[? >: T]](p: Pair[T]) =  
  if p.first.compareTo(p.second) < 0 then p.first else p.second
```

17.12 Polymorphic Functions

Consider this top-level function declaration:

```
def firstLast[T](a: Array[T]) = (a(0), a(a.length - 1))
```

In [Chapter 12](#), you saw how to rewrite a `def` into a `val` and a lambda expression:

```
val firstLast = (a: Array[T]) => ...
```

But that can't be quite right. You need to put the type parameter somewhere. You can't put it with `firstLast`—there are no generic `val`:

```
val firstLast[T] = (a: Array[T]) => (a(0), a(a.length - 1)) //  
Error
```

Instead, you should think of `firstLast` to have two curried parameters: the type `T` and the array `a`. Here is the correct syntax:

```
val firstLast = [T] => (a: Array[T]) => (a(0), a(a.length - 1))
```

The type of `firstLast` is

```
[T] => Array[T] => (T, T)
```

Such a type is called a *polymorphic function type*.

Note that `firstLast[String]` is an ordinary function with type `Array[String] => (String, String)`. The type parameter has been “curried”.

Polymorphic functions are useful when you need a lambda that must work with different types. Suppose you want to wrap all elements of a tuple in `Some`, transforming `(1, 3.14, "Fred")` into `(Some(1), Some(3.14), Some("Fred"))`.

There is a `Tuple.map` method for just this purpose. The function to be mapped requires a type parameter since each element can have a different type. Here is how to do it:

```
val tuple = (1, 3.14, "Fred")
tuple.map([T] => (x: T) => Some(x))
```

The type of that lambda expression is `[T] => T => Some[T]`.



Caution

In [Chapter 20](#), you will encounter *type lambdas* such as `[X] =>> Some[X]`. They look similar but are completely different. A type lambda is a type-level *function*. It consumes a type and produces another. In contrast, a polymorphic function type is a type.

Exercises

1. Define an immutable class `Pair[T, S]` with a method `swap` that returns a new pair with the components swapped.
2. Define a mutable class `Pair[T]` with a method `swap` that swaps the components of the pair.
3. Given a class `Pair[T, S]`, write a generic method `swap` that takes a pair as its argument and returns a new pair with the components swapped.

4. Why don't we need a lower bound for the `replaceFirst` method in [Section 17.3, “Bounds for Type Variables,”](#) on page 265 if we want to replace the first component of a `Pair[Person]` with a `Student`?
5. Why does `RichInt` implement `Comparable[Int]` and not `Comparable[RichInt]`?
6. Write a generic method `middle` that returns the middle element from any `Iterable[T]`. For example, `middle("World")` is 'r'.
7. Add a method `zip` to the `Pair[T]` class that zips a pair of pairs:

```
val p = Pair(Pair(1, 2), Pair(3, 4))
p.zip // Pair(Pair(1, 3), Pair(2, 4))
```

Of course, this method can only be applied if `T` is itself a `Pair` type. Use a type constraint.

8. Write a class `Pair[T, S]` that implements a *mutable* pair. Using the `=:=` type constraint, provide a `swap` method that swaps the components in place, provided that `S` and `T` are the same type. Demonstrate that you can invoke `swap` on `Pair("Fred", "Brooks")` and that you can construct `Pair("Fred", 42)` but you cannot invoke `swap` on it.
9. Look through the methods of the `Iterable[+A]` trait. Which methods use the type parameter `A`? Why is it in a covariant position in these methods?
10. In [Section 17.9, “Co- and Contravariant Positions,”](#) on page 270, the `replaceFirst` method has a type bound. Why can't you define an equivalent method on a mutable `Pair[T]`?

```
def replaceFirst[R >: T](newFirst: R) =
  first = newFirst // Error
```

11. It may seem strange to restrict method parameters in an immutable class `Pair[+T]`. However, suppose you could define

```
def replaceFirst(newFirst: T)
```

in a `Pair[+T]`. The problem is that this method can be overridden in an unsound way. Construct an example of the problem. Define a subclass `NastyDoublePair` of `Pair[Double]` that overrides `replaceFirst` so that it makes a pair with the square root of `newFirst`. Then construct the call `replaceFirst("Hello")` on a `Pair[Any]` that is actually a `NastyDoublePair`.

Chapter 18

Advanced Types

Topics in This Chapter L2

- [18.1 Union Types](#)
- [18.2 Intersection Types](#)
- [18.3 Type Aliases](#)
- [18.4 Structural Types](#)
- [18.5 Literal Types](#)
- [18.6 The Singleton Type Operator](#)
- [18.7 Abstract Types](#)
- [18.8 Dependent Types](#)
- [18.9 Abstract Type Bounds](#)
- [Exercises](#)

In this chapter, you will see all the types that Scala has to offer, including some of the more technical ones.

The key points of this chapter are:

- A value belongs to a union or intersection type if it belongs to any or all of the constituent types.

- Structural types are similar to “duck typing” but are checked at compile-time
- Singleton types are useful for method chaining and methods with object parameters.
- A type alias gives a short name for a type.
- An opaque type alias hides a representation type from the public.
- An abstract type must be made concrete in a subclass.
- A dependent type is a type that depends on a value.

18.1 Union Types

If T_1 and T_2 are types, then the *union type* $T_1 \mid T_2$ is the type whose instances belong either to T_1 or T_2 . Here is a concrete example. Response choices could be indicated by a string, separated by white space such as "Yes Maybe No" or an array `Array("yes", "maybe", "no")`. This function accepts either:

```
def isValid(choice: String, choices: String | Array[String]) =
  choices match
    case str: String => str.split("\\s+").contains(choice)
    case Array(elems*) => elems.contains(choice)
```

In general, to process a union type, you use a type match.

Union types can be suitable for “ad hoc” alternatives. Of course, you can follow a more object-oriented approach with a trait `Choices` and subclasses `StringChoices` and `ArrayChoices`:

```
enum Choices :
  case StringChoices(str: String)
  case ArrayChoices(elems: String*)
```

This is sometimes called a “discriminated union”. It is equivalent to a union type, but the types have names. Generally, we prefer named types in Scala, and union types are uncommon.



Caution

When you produce a value that can be one type or another, and you want the result to be considered a union type, you need to specify the union type explicitly. Consider this example:

```
def inverse(x: Double):  
  Double | String = if x == 0 then "Divide by zero" else 1  
  / x
```

Without the `Double | String` return type, the compiler infers the type `Matchable`, which is not very useful.

Note that type union is commutative: The types $T_1 + T_2$ and $T_2 + T_1$ are the same type.

18.2 Intersection Types

An intersection type has the form

$$T_1 \And T_2 \And T_3 \dots$$

where T_1 , T_2 , T_3 , and so on are types. In order to belong to the intersection type, a value must belong to all of the individual types.

You use an intersection type to describe values that must provide multiple traits. For example,

```
val img = ArrayBuffer[java.awt.Shape & java.io.Serializable]
```

You can draw the `img` object as `for (s <- img) graphics.draw(s)`. You can serialize the `image` object because you know that all elements are serializable.

Of course, you can only add elements that are both shapes and serializable objects:

```
val rect = Rectangle(5, 10, 20, 30)
img += rect // OK—java.awt.Rectangle is a Shape and
Serializable
```

But this does not work:

```
img += Area(rect) // Error—java.awt.Area is a Shape but
not Serializable
```



Note

When you have a declaration

```
trait ImageShape extends Shape, Serializable
```

this means that `ImageShape` extends the intersection type `Shape & Serializable`.

In Scala, you are more likely to use such traits than raw intersection types. But you will sometimes see the intersection types of traits in compiler messages.

Like type union, type intersection is commutative: $T_1 \& T_2$ and $T_2 \& T_1$ are the same type.

???????:

!!!! It may seem surprising that type intersection is commutative since order matters when using multiple traits. For example, in [Chapter 10](#), you saw this example:

```
val acct1 = new LoggedAccount() with TimestampLogger with
ShortLogger
val acct2 = new LoggedAccount() with ShortLogger with
TimestampLogger
```

The `log` methods of the two objects act differently. Nevertheless, both objects have type

```
LoggedAccount & TimestampLogger & ShortLogger
```

which is the same as

```
LoggedAccount & ShortLogger & TimestampLogger
```

The type describes objects that have a `log` method, but it doesn't say how it is implemented.

18.3 Type Aliases

You can create a simple *alias* for a complicated type with the `type` keyword, like this:

```
object Book :  
    type Index = scala.collection.mutable.HashMap[String, (Int,  
    Int)]  
    ...
```

Then you can refer to `Book.Index` instead of the cumbersome name `scala.collection.mutable.HashMap[String, (Int, Int)]`.



Caution

To instantiate `Book.Index`, you need `new`:

```
val idx = new Book.Index()
```

The expression `Book.Index()` does not work since `Index` is a type, not a value with an `apply` method. If you really want to avoid `new`, add a value for the companion object:

```
val Index = scala.collection.mutable.HashMap
```

A type alias is particularly useful for a type that doesn't have a name by itself:

```
type Pair[T] = (T, T)
type Choices = String | Array[String]
```

A type alias can be *opaque*, indicating that the actual type is only known in the scope of the declaration. For example:

```
object Doc :
    opaque type HTML = String
    def format(text: String): HTML = ...
    class Chapter :
        private val paragraphs = ArrayBuffer[String]()
        def paragraph(index: Int): HTML = paragraphs(index)
        def append(paragraph: HTML) = paragraphs += paragraph
```

Inside the scope of the `Doc` object, it is known that `HTML` is an alias for `String`. But elsewhere, calling the `append` method with a `String` is an error. One can only pass instances of the type `Doc.HTML`. The features of the `Doc` object are responsible for the integrity of the HTML formatting.

Note that the opacity is not achieved by “boxing” the value inside another object. A value of type `Doc.HTML` is simply a `String` reference, but you cannot invoke `String` methods on it.

You can reveal some methods of an opaque type by giving a type bound:

```
opaque type HTML <: CharSequence = String
```

Now you can invoke any methods of the `CharSequence` interface.



Note

By using extension methods (see [Chapter 19](#)), you can create opaque types with methods of your choice. This allows you to define types that are as efficient as value classes (see [Chapter 5](#)).



Caution

Note that in the example, the opaque type was defined in an object. It is legal to define an opaque type in a class, but then each instance has its own type. Had the declaration of the opaque `HTML` type been in the `Chapter` class, then `ch1.HTML` and `ch2.HTML` would be different types, and you could not copy paragraphs between chapters.



Note

The `type` keyword is also used for *abstract types* that are made concrete in a subclass, for example:

```
abstract class Reader :  
    type Contents // an abstract type  
    def read(filename: String): Contents
```

We will discuss abstract types in [Section 18.7, “Abstract Types,”](#) on page 286.

18.4 Structural Types

A “structural type” is a specification of abstract methods, fields, and types that a conforming type should possess. Here we declare a structural type and assign it to a type alias:

```
type Appendable = { def append(str: String): Any }
```

A structural type is more flexible than defining a `Appendable` trait, because you might not always be able to add that trait to the classes you are using. For example, there is a `java.lang.Appendable` interface with an `append`

method. The `javax.swing.JTextArea` class also has an `append` method, but it doesn't implement the `java.lang.Appendable` interface. Yet instances of any `java.lang.Appendable` (which includes `StringBuilder` and `Writer`), as well as `JTextArea`, belong to the `Appendable` type that was just declared.

What can you do with this type? Here is a function that almost works:

```
def appendLines(target: Appendable, lines: Iterable[String]) =  
    for l <- lines do  
        target.append(l); // Error  
        target.append("\n") // Error
```

However, the compiler rejects the invocations of the `append` method, even though it knows that the method exists.

The problem is that compiler doesn't know *how* to invoke the method.

That sounds a bit strange. Why would the compiler not know how to invoke a method? But consider what happens in the situation where `Appendable` was a trait. In that case, the compiler generates a virtual machine instruction for dynamic invocation. The virtual machine has an efficient mechanism for invoking methods, using a run-time data structure called a virtual method table. That only works when the virtual machine knows the class or interface in which the method is defined.

There is another way of invoking a method: through the reflection API. In our example, that is a reasonable approach. Still, reflection is expensive, and the Scala compiler wants us to confirm that's what we want. You achieve this with a special import:

```
def appendLines(target: Appendable, lines: Iterable[String]) =  
    import reflect.Selectable.reflectiveSelectable  
    for l <- lines do  
        target.append(l); // OK  
        target.append("\n") // OK
```

Note that this approach is typesafe. You can call the `appendLines` method with an instance of *any* class that has a suitable `append` method. The

compiler reports an error if you call the `appendLines` function with an instance of a class without such an `append` method.

 **Note**

In [Chapter 11](#), you saw how to use the `Selectable` interface for typesafe selection and method application. Importing

```
reflect.Selectable.reflectiveSelectable introduces a  
conversion from any type  
to a Selectable instance that uses reflection .
```

You can also create instances of a structural type on the fly, like this:

```
val appender: Appendable = new :  
  def append(str: String): Any = println(str)
```

 **Note**

Instead of the type intersection `T & { def append(str: String): Any }`, you can write `T { def append(str: String): Any }`. In the latter case, the expression in braces is called a *type refinement*.

 **Note**

Calling structural types with reflection is similar to “duck typing” in dynamically typed programming languages such as JavaScript or Ruby. In those languages, variables have no type. When you write `obj.quack()`, the *runtime* figures out whether the particular object to which `obj` refers at this point has a `quack` method. In other words, you don’t have to declare `obj` as a `Duck` as long as it walks

and quacks like one. In contrast, Scala checks at compile-time that the call will succeed.

18.5 Literal Types

You can define a type with exactly one value, which must be a literal constant: a number, string, character, or Boolean literal (but not `null`).

```
type Zero = 0
```

A variable of this type can hold a single value:

```
val z: Zero = 0
```

That doesn't sound useful, but you can define meaningful union types:

```
type Bit = 0 | 1
var b: Bit = 0
b = 1 // OK
```

The following is an error:

```
b = -1 // Error
```

Here is a more interesting example. We use language tags to ensure at compile-time that only messages of the same language are combined.

```
class Message[L <: String](val text: String)
val m1 = Message["de"]("Achtung") // m1 has type Message["de"]

class Messages[L <: String] :
  private val sb = StringBuilder()
  def append(message: Message[L]) =
    sb.append(message.text).append("\n")
  def text = sb.toString
```

```
val germanNews = Messages["de"]()
germanNews.append(m1) // OK to add message with the same language
```

But the following does not compile:

```
val m2 = Message["en"]("Hello")
germanNews.append(m2) // Error
```



Tip

By default, literal types are widened in type inference. For example, consider this function for constructing a string in a given language

```
def message[L <: String](language: L, text: String) =
  Message[L](text)
```

The type of `message("en", "hello")` is `Message[String]`, not `Message["en"]`. To avoid this widening, add the upper bound `Singleton`:

```
def message[L <: String & Singleton](language: L,
  text: String) =
  Message[L](text)
```



Caution

Every literal type is a subtype of `Singleton`, but not every subtype of `Singleton` is a singleton. By general principles, since `"en" <: Singleton` and `"fr" <: Singleton`, it also holds that `"en" | "fr" <: Singleton`.

In [Chapter 20](#), you will see other uses of literal types. For example, we will define a parameterized vector type. The dimension is a type parameter,

which prevents you from adding a `Vec[3]` and a `Vec[4]`. Note the `3` and `4` inside the square brackets. They are literal types, not values!

18.6 The Singleton Type Operator

Given a value `v`, you can form the singleton type `v.type` which contains the value `v`. This sounds like the literal types from the preceding section, but there are a couple of differences.

Literal types are defined with *literals*. Here, the value is given as a *variable or path*, for example, `e.name.type`. (Of course, the value must be immutable. If a variable `v` is declared as `var`, you cannot form a singleton type `v.type` since it might have more than one value.)

The type is tied to the path, not its value. If `v` and `w` have the same value, then `v.type` and `w.type` are nevertheless different:

```
val v = 1
val w = 1
val x: v.type = v // OK

val y: v.type = w // Error
val z: v.type = 1 // Error
```



Caution

If `v` is a reference type, then `v.type` is not actually a singleton type. It has two values—`v` and `null`:

```
val v = "Hello"
val x: v.type = null // OK
```

The same is true for literal string types:

```
val v: "Hello" = null // OK
```

This is perhaps unfortunate but unavoidable. The `Null` type (with the single value `null`) is a subtype of every reference type.

At first glance, singleton types don't appear to be very useful. But they have several practical applications.

Consider a method that returns `this` so you can chain method calls:

```
class Document :  
    def setTitle(title: String) = { ...; this }  
    def setAuthor(author: String) = { ...; this }  
    ...
```

You can then call

```
article.setTitle("Whatever Floats Your Boat").setAuthor("Cay  
Horstmann")
```

However, if you have a subclass, there is a problem. Consider this subclass:

```
class Book extends Document :  
    def addChapter(chapter: String) = { ...; this }  
    ...
```

The following call fails:

```
val book = Book()  
book.setTitle("Scala for the Impatient").addChapter("Advanced  
Types") // Error
```

Since the `setTitle` method returns `this`, Scala infers the return type as `Document`. But `Document` doesn't have an `addChapter` method.

The remedy is to declare the return type of `setTitle` as `this.type`:

```
def setTitle(title: String): this.type = { ...; this }
```

Now the return type of `book.setTitle("...")` is `book.type`, and since `book` has an `addChapter` method, the chaining works.

Tip

Sometimes, you need to convince the compiler that an object extends a particular trait. You could cast it, for example:

```
v.asInstanceOf[Matchable]
```

but that may not be good enough. The result is `Matchable`, but it has lost whatever type `v` had before. The remedy is to add the type of `v`:

```
v.asInstanceOf[v.type & Matchable]
```

Another use of singleton types are methods that take an `object` instance as parameter. You may wonder why you would ever want to do that. After all, if there is just one instance, the method could simply use it instead of making the caller pass it.

However, some people like to construct “fluent interfaces” that read like English, for example:

```
book set Title to "Scala for the Impatient"
```

This is parsed as

```
book.set>Title).to("Scala for the Impatient")
```

For this to work, `set` is a method whose argument is the singleton `Title`:

```
object Title // This object is used as an argument for a fluent
interface
```

```
class Document :
    private var title = ""
```

```

private var useNextArgAs: Any = null
def set(obj: Title.type): this.type = { useNextArgAs = obj;
this }
def to(arg: String) = if (useNextArgAs == Title) title = arg
// else ...
...

```

Note the `Title.type` parameter. You can't use

```
def set(obj: Title) ... // Error
```

since `Title` denotes the singleton *object*, not a type.

18.7 Abstract Types

A class or trait can define an *abstract type* that is made concrete in a subclass. For example:

```

trait Reader :
  type Contents
  def read(url: String): Contents

```

Here, the type `Contents` is abstract. A concrete subclass needs to specify the type:

```

class StringReader extends Reader :
  type Contents = String
  def read(url: String) =
    String(URL(url).openStream.readAllBytes)

class ImageReader extends Reader :
  type Contents = BufferedImage
  def read(url: String) = ImageIO.read(URL(url))

```

The same effect could be achieved with a type parameter:

```

trait Reader[C] :
    def read(url: String): C

class StringReader extends Reader[String] :
    def read(url: String) =
        String(URL(url).openStream.readAllBytes)

class ImageReader extends Reader[BufferedImage] :
    def read(url: String) = ImageIO.read(URL(url))

```

Which is better? In Scala, the rule of thumb is:

- Use type parameters when the parameter types should be supplied with each instance of the type. For example, when declaring a map as a `Map[String, Int]`, you specify the types at the declaration site.
- Use abstract types when the types are supplied in subclasses. That is the case in our `Reader` example. You can't simply instantiate a `Reader[JSONObject]`. You need form a subclass and declare the `read` method to produce JSON.

18.8 Dependent Types

Let's put the readers from the preceding section to work for making user interface components:

```

val ir = ImageReader()
val imageLabel =
    makeComponent(ir, "https://horstmann.com/cay-tiny.gif", b =>
    JLabel(ImageIcon(b)))

```

The `makeComponent` function takes a reader, a URL, and a function that transforms the reader's contents into a component. What should be the parameter type of that function? It can't just be `Contents`. That's an abstract type. Instead, you need the `Contents` type of the specific reader:

```
def makeComponent(r: Reader, url: String, transform: r.Contents  
=> JComponent) =  
  val contents = r.read(url)  
  transform(contents)
```

The type `r.Contents` is called a *dependent type* since it depends on the value `r`. For example, `ir.Contents` is the type `BufferedImage`.



Caution

You cannot make a dependent type with a mutable variable. If you declare `var r: Reader`, then the compiler cannot tell what `r.Contents` is. It would depend on the current value of `r`.

In Scala, dependent types belong to instances, not classes. For example, there is no type `ImageReader.Contents`.

In general, if you have a type `p.q.r.T`, then the components `p`, `q`, `r` must be package components, object names, or immutable variables (which includes `this` and `super`). Such a construct is called a *path*—you may come across that term in compiler messages.

If you need to refer to the `Contents` of all `ImageReader` instances, use a *type projection* `ImageReader#Contents`—see [Chapter 5](#).



Note

Internally, the compiler translates type expressions `p.q.r.T` to type projections `p.q.r.type#T`. For example, `r.Contents` becomes `r.type#Contents`—any `Contents` inside the singleton `r.type`. That is not something you generally need to worry about. However, sometimes you may see an error message with a type of the form `p.q.r.type#T`. Just translate it back to `p.q.r.T`.

You have just seen that the return type of the `read` method is a dependent type. Now let's turn that into a function:

```
val readContents = (r: Reader, url: String) => r.read(url)
```

What is the type of the function? It can't be `(Reader, String) => Reader.Contents` because there is no type `Reader.Contents`.

In order to refer to the proper type `r.Contents`, you need to add variable names, like this:

```
(r: Reader, url: String) => r.Contents
```

It doesn't matter what names you choose. You have to supply them for *all* parameters, even if you don't need them (like `url` above). Such a type is called a *dependent function type*.

18.9 Abstract Type Bounds

Abstract types can have type bounds, just like type parameters. For example:

```
class Event
class EventSource :
    type E <: Event
    ...

```

A subclass must provide a compatible type, for example:

```
class Button(val text: String) extends EventSource :
    class ButtonEvent(val source: Button) extends Event
    type E = ButtonEvent // OK, it's a subtype of Event
    ...

```

Note that this example could not have been done with type parameters since the bound is an inner class.

Let us continue this example to demonstrate how abstract types can describe subtle interdependencies between types.

Generally, it is a challenge to model families of types that vary together, share common code, and preserve type safety. In our event handling scenario, we want different event sources to fire events of different types, and manage listeners that receive notifications with the appropriate types.

Let us add functionality for managing listeners and firing events to the `EventSource` class:

```
type L <: Listener
trait Listener :
    def occurred(e: E): Unit

    private val listeners = ArrayBuffer[L]()
    def add(l: L) = listeners += l
    def remove(l: L) = listeners -= l
    def fire(e: E) =
        for (l <- listeners) l.occurred(e)
```

Note the type variables `E` and `L` for the event and listener types. The `occurred` listener method requires a parameter of the specific type `E`, not just any event. You can only add listeners of the specific type `L`.

In the `Button` class, we already set `E` to `ButtonEvent`. We still need to specify the `L` type. The `click` method simulates a button click:

```
trait ButtonListener extends Listener
type L = ButtonListener
def click() = fire(ButtonEvent(this))
```

What is the point? Now you can add a listener that receives button events:

```
val b = Button("Click me!")
val listener = new b.ButtonListener :
    override def occurred(e: b.ButtonEvent) =
        println(s"Clicked the ${e.source.text} button")
```

```
b.add(listener)  
b.click()
```

But trying to add any other kind of listener will fail (see [Exercise 11](#) on page 291). This type safety is achieved even though the code for managing listeners is entirely contained in the `EventSource` superclass.

Note

In the examples for abstract types, I used single-letter names for the abstract types, to show the analogy with the version that uses type parameters. It is common in Scala to use more descriptive type names, which leads to more self-documenting code:

```
class EventSource :  
    type EventType <: Event  
    type ListenerType <: Listener  
    ...  
  
class Button extends EventSource :  
    type EventType = ButtonEvent  
    type ListenerType = ButtonListener  
    ...
```

Exercises

1. Instead of using the `Try[T]` type, you could use a union type `T | Throwable`. Give advantages and disadvantages of that approach.
2. This type models a list of strings:

```
enum List :  
    case Empty  
    case NonEmpty(head: String, tail: List)
```

Can you use a union type instead? If so, how? If not, why is it not possible?

3. Show that the types $(T \mid U) \Rightarrow R$ and $(T \Rightarrow R) \& (U \Rightarrow R)$ are equivalent.

4. Write a function `printValues` with three parameters `f`, `from`, `to` that prints all values of `f` with inputs from the given range. Here, `f` should be any object with an `apply` method that consumes and yields an `Int`. For example,

```
printValues(Array(1, 1, 2, 3, 5, 8, 13, 21, 34, 55), 3, 6) //  
Prints 3 5 8 13  
printValues((x: Int) => x * x, 3, 6) // Prints 9 16 25 36
```

Hint: In the Scala 3.2 implementation, function literals such as `(x: Int) => x * x` do *not* have a method `apply(n: Int): Int` that can be called by reflection. Use a union type to accept either `Function1[Int, Int]` instances or objects with a `apply(n: Int): Int` method.

5. Using a structural type instance, Scala lets you define objects as flexibly as JavaScript:

```
val fred = new :  
    val id = 1729  
    var name = "Fred"
```

Unfortunately, you can't access the fields. Describe two distinct modifications of this declaration that allow field access (without defining a `Person` class).

6. Implement a method that receives an object of any class that has a method

```
def close(): Unit
```

together with a function that processes that object. Call the function and invoke the `close` method upon completion, or when any exception occurs.

7. Implement a `Bug` class modeling a bug that moves along a horizontal line. The `move` method moves in the current direction, the `turn` method makes the bug turn around, and the `show` method prints the current position. Make these methods chainable. For example,

```
bugsy.move(4).show().move(6).show().turn().move(5).show()
```

should display 4 10 5.

8. Provide a fluent interface for the `Bug` class of the preceding exercise, so that one can write

```
bugsy move 4 and show and then move 6 and show turn around  
move 5 and show
```

9. Complete the fluent interface in [Section 18.6, “The Singleton Type Operator,”](#) on page 284 so that one can call

```
book set Title to "Scala for the Impatient" set Author to "Cay  
Horstmann"
```

10. The `cast v.asInstanceOf[v.type & Matchable]` looks a little unsightly. Define a generic method `as` so that you can add an arbitrary trait to a value by calling `as[Matchable](v)`.

11. Try defining a `ButtonListener` whose `occurred` method accepts an event other than a `ButtonEvent`. Try adding a listener other than a `ButtonListener` to a `Button`. What happens in each case?

12. Rewrite the `EventSource` class using parameterized types instead of abstract types. Is it still guaranteed that a `ButtonListener` only accepts `ButtonEvent` values, and that you can only add a `ButtonListener` to a `Button`?

13. Given the following trait

```
trait UnitFactory :  
    type U <: Unit  
    def create(x: Double): U  
    trait Unit :
```

```
val value: Double
def name: String
override def toString = s"$value $name"
def + (that: U): U =
  of(this.value + that.value)
```

create objects

```
object MeterFactory extends UnitFactory
```

and

```
object LiterFactory extends UnitFactory
```

whose `create` methods yield instances of classes `Meter` and `Liter`. Demonstrate that you can only add units of the same type.

14. Improve the code from the preceding exercises so that you also support units such as kilometers and milliliters, where the compiler checks that you only add units of like types.
15. Can you implement [Exercise 13](#) on page 291 with parameterized types instead of abstract types? If not, why not?
16. Try writing a dependent function type for the `makeComponent` function in [Section 18.8, “Dependent Types,”](#) on page 287. What happens? Fix it by currying the function. What is the type now?

Chapter 19

Contextual Abstractions

Topics in This Chapter L3

- 19.1 Context Parameters
- 19.2 More about Context Parameters
- 19.3 Declaring Given Instances
- 19.4 Givens in `for` and `match` expressions
- 19.5 Importing Givens
- 19.6 Extension Methods
- 19.7 Where Extension Methods Are Found
- 19.8 Implicit Conversions
- 19.9 Rules for Implicit Conversions
- 19.10 Importing Implicit Conversions
- 19.11 Context Functions
- 19.12 Evidence
- 19.13 The `@implicitNotFound` Annotation
- Exercises

The Scala language provides a number of power tools for carrying out work that depends on a *context*—settings that are configured depending on

current needs. In this chapter, you will learn how to build capabilities that can be added in an ad-hoc fashion to class hierarchies, how to enrich existing classes, and to carry out automatic conversions. With contextual abstractions, you can provide elegant mechanisms that hide tedious details from users of your code.

The key points of this chapter are:

- An `an` context parameter requests a `given` object of a specific type. You can define suitable `given` objects for different contexts.
- The `summon[T]` function summons the current `given` object for type `T`.
- They can be obtained from implicit objects that are in scope, or from the companion object of the desired type.
- There is special syntax for declaring `given` objects that extend a trait and define methods.
- You can build complex `given` objects, using type parameters and context bounds
- If an implicit parameter is a single-argument function, it is also used as an implicit conversion.
- Typically, you use import statements to supply the `given` instances that you need.
- Using extension methods, you can add methods to existing classes.
- Implicit conversions are used to convert between types.
- You control which extensions and implicit conversions are available at a particular point.
- A context function type describes a function with context parameters.

19.1 Context Parameters

Typically, systems have certain settings that are needed by many functions, such as a database connection, or a logger, or a locale. Passing those settings as arguments is impractical, because every function would need to pass them to the functions that it calls. You can use globally visible objects,

but that is inflexible. Scala has a powerful mechanism to make values available to functions without explicitly passing them, through *context parameters*. Let's first understand how a function receives those parameters.

A function or method can have one or more parameter lists marked with the `using` keyword. For those parameters, the compiler will look for appropriate values to supply with the function call, using a mechanism that we will study later. Here is a simple example:

```
case class QuoteDelimiters(left: String, right: String)

def quote(what: String) (using delims: QuoteDelimiters) =
    delims.left + what + delims.right
```

You can call the `quote` method with an explicit `QuoteDelimiters` object, like this:

```
quote("Bonjour le monde") (using QuoteDelimiters("«", "»"))
// Returns «Bonjour le monde»
```

Note that there are two argument lists. This function is “curried”—see [Chapter 12](#).

However, the point of context parameters is that you don't want to provide an explicit `using` parameter. You simply want to call:

```
quote("Bonjour le monde")
```

Now the compiler will look for a suitable value of type `QuoteDelimiters`. This must be a value that is declared as given:

```
given englishQuoteDelims : QuoteDelimiters = QuoteDelimiters("""",
""")
```

Now the delimiters are supplied implicitly to the `quote` function.



Note

As you will see soon, there are more compact ways of declaring given values. Right now, I am using a form that is easy to understand.

However, you typically don't want to declare global given values. In order to switch between contexts, you can declare objects such as the following:

```
object French :  
    given quoteDelims: QuoteDelimiters = QuoteDelimiters("«", "»")  
    ...  
  
object German :  
    given quoteDelims: QuoteDelimiters = QuoteDelimiters("„", "“")  
    ...
```

Then import the given values from one such object, using the following syntax:

```
import German.given
```



Note

For each type, there can only be one given value. Thus, it is not a good idea to have using parameters of common types. For example,

```
def quote(what: String) (using left: String, right: String)  
// No!
```

would not work—there could not be two different given values of type String.

You have now seen the essence of context parameters. In the next three sections, you will learn about the finer points of using clauses, the syntax of given declarations, and the ways to import them.

19.2 More about Context Parameters

In the preceding section, you saw a function with a `using` parameter:

```
def quote(what: String) (using delims: QuoteDelimiters) = ...
```

If the function body doesn't refer to the `using` parameter, you don't have to give it a name:

```
def attributedQuote(who: String, what: String) (using
  QuoteDelimiters) =
  s"$who said: ${quote(what)}"
```

The `using` parameter still gets passed to the functions that need it—in this case, the `quote` function.

You can always obtain a given value by calling `summon`:

```
def quote(what: String) (using QuoteDelimiters) =
  summon[QuoteDelimiters].left + what +
  summon[QuoteDelimiters].right
```

By the way, here is the definition of `summon`:

```
def summon[T] (using x: T): x.type = x
```



Just as Shakespeare's “vasty deep” is filled with spirits that any man can summon, the Scala nether world is populated with given objects, at most one for each type. Any programmer can summon them. Will they come? That depends on whether the compiler can locate a unique given object of the summoned type.

A function can have more than one `using` parameter. For example, we may want to localize messages in addition to quotes. Each language maps keys

to `MessageFormat` templates:

```
case class Messages(templates: Map[String, String])

object French :

  ...

given msgs: Messages =
  Messages(Map("greeting" -> "Bonjour {0}!",
  "attributedQuote" -> "{0} a dit {1}{2}{3}"))
```

This method receives the quote delimiters and message map through `using` parameters:

```
def attributedQuote(who: String, what: String) (
  using delims: QuoteDelimiters, msgs: Messages) =
  MessageFormat.format(msgs.templates("attributedQuote"),
  who, delims.left, what, delims.right)
```

You can curry the `using` parameters:

```
def attributedQuote(who: String, what: String) (
  using delims: QuoteDelimiters)(using msgs: Messages) = ...
```

With this example, there is no need to use currying. But there can be situations where currying helps with type inference.

A primary constructor can have context parameters:

```
class Quoter(using delims: QuoteDelimiters) :
  def quote(what: String) = ...
  def attributedQuote(who: String, what: String) = ...
  ...
```

The `delims` variable is initialized from the context parameter when the object is constructed. Afterwards, there is nothing special about it. Its value is accessible in all methods.

A `using` parameter can be declared by-name (`using ev: => T`), to delay its production until it is needed. This can be useful if the given value is

expensive to produce, or to break cycles. This is not common.

19.3 Declaring Given Instances

In this section, we go over the various ways of declaring `given` instances. You have already seen the explicit form with a name, a type, and a value.

```
given englishQuoteDelims : QuoteDelimiters = QuoteDelimiters(""", """)
```

The right-hand side doesn't have to be a constructor—it can be any expression of type `QuoteDelimiters`.



Caution

There is no type inference for `given` instances. The following is an error:

```
given englishQuoteDelims = QuoteDelimiters(""", """)  
// Error—no type inference
```

If you use a constructor, you can omit the type and the `=` operator:

```
given englishQuoteDelims : QuoteDelimiters(""", """)
```

You can also omit the name, since you don't normally need it:

```
given QuoteDelimiters(""", """)
```

Inside an abstract class or trait, you can define an *abstract given*, with a name and type but with no value. The value must be supplied in a subclass.

```
abstract class Service :  
  given logger: Logger
```

It is common to declare `given` instances that override abstract methods. For example, when declaring a `given` instance of type `Comparator[Int]`, you must provide an implementation of the `compare` method:

```
given intComp: Comparator[Int] =
  new Comparator[Int]() :
    def compare(x: Int, y: Int) = Integer.compare(x, y)
```

There is a convenient shortcut for this common case:

```
given intComp: Comparator[Int] with
  def compare(x: Int, y: Int) = Integer.compare(x, y)
```

Presumably, the `with` keyword was chosen to avoid having two colons in a row.

The name is optional:

```
given Comparator[Int] with
  def compare(x: Int, y: Int) = Integer.compare(x, y)
```

You can create parameterized `given` instances. For example:

```
given comparableComp[T <: Comparable[T]] : Comparator[T] with
  def compare(x: T, y: T) = x.compareTo(y)
```

The instance name is optional, but the colon before the instance type is required.

Let us use this example to introduce one more concept: `given` instances with `using` parameters. Consider the comparison of two `List[T]`. If neither are empty, we compare the heads.

How do we do that? We need to know that values of type `T` can be compared. That is, we need a `given` of type `Comparator[T]`. As always, this need can be expressed with a `using` clause:

```
given listComp[T] (using tcomp: Comparator[T]) :
  Comparator[List[T]] =
    new Comparator[List[T]]() :
```

```

def compare(xs: List[T], ys: List[T]) =
  if xs.isEmpty && ys.isEmpty then 0
  else if xs.isEmpty then -1
  else if ys.isEmpty then 1
  else
    val diff = tcomp.compare(xs.head, ys.head)
    if diff != 0 then diff
    else compare(xs.tail, ys.tail)

```

As before, you can omit the name and take advantage of the `with` syntax:

```

given [T] (using tcomp: Comparator[T]) : Comparator[List[T]] with
  def compare(xs: List[T], ys: List[T]) =

```



Note

Instead of an explicit `using` parameter, you can use a *context bound* for the type parameter:

```
given [T : Comparator] : Comparator[List[T]] with ...
```

As you saw in [Chapter 17](#), there must be a given `Comparator[T]` instance. That is equivalent to a `using` clause. However, there is one difference: you don't have a parameter for the given instance. Instead, you summon it from the “nether world”:

```

val ev = summon[Comparator[T]]
val diff = ev.compare(xs.head, ys.head)

```

It is common to use the variable name `ev` for such summoned objects. It is shorthand for “evidence”. The fact that a given value can be summoned is evidence for their existence.

Let us reflect on the strategy that underlies the example. We declare ad-hoc given instances for `Comparator[Int]`, `Comparator[Double]`, and so on. Using a parameterized rule, we obtain `Comparator[T]` for all types `T` that extend `Comparable[T]`. This includes `String` and a large number of Java types such as `LocalDate` or `Path`. Next, we have `Comparator[List[T]]` instances lists of any type `T` that itself has a `Comparator[T]` instance. That process can be carried out for other collections, and, with more technical effort, for tuples and case classes. Note that this happens without cooperation from individual classes. For example, we did not have to modify the `List` class in any way.

What is the benefit of this effort? It allows us to write generic functions that require an ordering. They will work with all types `T` with a given `Comparator[T]` instance. Here is a typical example:

```
def max[T] (a: Array[T]) (using comp: Comparator[T]) =  
  if a.length == 0 then None  
  else  
    var m = a(0)  
    for i <- 1 until a.length do  
      if comp.compare(a(i), m) > 0 then m = a(i)  
    Some(m)
```

The Scala library does just that, except with the `Ordering` trait that extends `Comparator`.

19.4 Givens in `for` and `match` expressions

You can declare a `given` in a `for` loop:

```
val delims = List(QuoteDelimiters("""", """"), QuoteDelimiters("«",  
"»"),  
  QuoteDelimiters("", ""))  
for given QuoteDelimiters <- delims yield  
  quote(text)
```

In each iteration of the loop, the given `QuoteDelimiters` instance changes, and the `quote` function uses different delimiters.

In general, a `for` loop `for (x <- a) yield f(x)` is equivalent to `a.map(f)`, in which the values of `a` become arguments of `f`. In the given form `for (given T <- a) yield f()`, the values of `a` become using arguments of `f`.

A pattern `a match` expression can be declared as given:

```
val p = (text, QuoteDelimiters("«", "»"))
p match
  case (t, given QuoteDelimiters) => quote(t)
```

The given value is introduced in the right hand side of the matched case.

Note that in both situations, you specify the type of the `given` value, not the name of a variable. If you need a name, use the `@` syntax:

```
for d @ given QuoteDelimiters <- delims yield
  quote(d.left + d.right)

p match
  case (t, d @ given QuoteDelimiters) => quote(d.left + d.right)
```

19.5 Importing Givens

Usually, you define `given` values in an object:

```
object French :
  given quoteDelims: QuoteDelimiters = QuoteDelimiters("«", "»")
  given NumberFormat =
    NumberFormat.getNumberInstance(Locale.FRANCE)
```

You import them as needed, either by name or by type:

```
import French.quoteDelims // imported by name
import French.given NumberFormat // imported by type
```

You can import multiple given values. Imports by name come before imports by type.

```
import French.{quoteDelims, given NumberFormat}
```

If the type is parameterized, you can import all given instances by using a wildcard. For example,

```
import Comparators.given Comparator[?]
```

imports all given Comparator instances from

```
object Comparators :  
    def ≤[T : Comparator](x: T, y: T) =  
        summon[Comparator[T]].compare(x, y) <= 0  
    given Comparator[Int] with  
        ...  
    given [T <: Comparable[T]] : Comparator[T] with  
        ...  
    given [T : Comparator] : Comparator[List[T]] with  
        ...
```

To import all given values, use

```
import Comparators.given
```

Note that the wildcard import

```
import Comparators.*
```

does *not* import any given values. To import everything, you need to use:

```
import Comparators.{*, given}
```

You can also export given values:

```
object CanadianFrench :  
    given NumberFormat =
```

```
NumberFormat.getNumberInstance(Locale.CANADA)
    export French.given QuoteDelimiters
```

19.6 Extension Methods

Did you ever wish that a class had a method its creator failed to provide? For example, wouldn't it be nice if the `java.io.File` class had a `read` method for reading a file:

```
val contents = File("/usr/share/dict/words").read
```

As a Java programmer, your only recourse is to petition Oracle Corporation to add that method. Good luck!

In Scala, you can define an *extension method* that provides what you want:

```
extension (f: File)
  def read = Files.readString(f.toPath)
```

Now it is possible to call `read` on a `File` object.

An extension method can be an operator:

```
extension (s: String)
  def -(t: String) = s.replace(t, "")
```

Now you can subtract strings:

```
"units" - "u"
```

Extensions can be parameterized:

```
extension [T <: Comparable[? >: T]](x: T)
  def <(y: T) = x.compareTo(y) < 0
```

Note that this extension is *selective*: It adds a `<` method only to types extending `Comparable`.

You can add multiple extension methods to the same type:

```
extension (f: File)
  def read = Files.readString(f.toPath)
  def write(contents: String) = Files.writeString(f.toPath,
contents)
```

An `export` clause can add multiple methods:

```
extension (f: File)
  def path = f.toPath
  export path.*
```

Now all `Path` methods are applicable to `File` objects:

```
val home = File("")
home.toAbsolutePath() // On my computer, that is Path("/home/cay")
```

19.7 Where Extension Methods Are Found

Consider a method call

```
obj.m(args)
```

where `m` is not defined for `obj`. Then the Scala compiler needs to look for an extension method named `m` that can be applied to `obj`.

There are four places where the compiler looks for extension methods:

1. In the scope of the call—that is, in enclosing blocks and types, in supertypes, and in imports
2. In the companion objects of all types that make up the type `T` of `obj`. For historical reasons, this is called the “implicit scope” of the type `T`. For example, the implicit scope of `Pair[Person]` consists of the members of the `Pair` and `Person` objects.
3. In all given instances that are available at the point of the call
4. In all given instances in the implicit scope of the type of `obj`

Let us look at examples of each of these situations.

You can place extension methods in an object or package, and then import it:

```
object FileOps : // or package
  extension(f: File)
    def read = Files.readString(f.toPath)

  import FileOps.* // or import FileOps.read
  File("/usr/share/dict/words").read
```

You can put extension methods in a trait or class and extend it

```
trait FileOps :
  extension(f: File)
    def read = Files.readString(f.toPath)

  class Task extends FileOps :
    def call() =
      File("/usr/share/dict/words").read
```

Here is an example of an extension method in a companion object:

```
case class Pair[T](first: T, second: T)

object Pair :
  extension [T](pp: Pair[Pair[T]])
    def zip = Pair(Pair(pp.first.first, pp.second.first),
                  Pair(pp.first.second, pp.second.second))

  val obj = Pair(Pair(1, 2), Pair(3, 4))
  obj.zip // Pair(Pair(1, 3), Pair(2, 4))
```

The `zip` method can't be a method of the `Pair` class. It doesn't work for arbitrary pairs, but only for pairs of pairs. Since the `zip` method is invoked on an instance of `Pair[Pair[Int]]`, both the `Pair` and `Int` companion objects are searched for an extension method.

Now let us look at extension methods in given values. Consider a `stringify` method that formats numbers, strings, arrays, and objects using JSON. It has to be an extension method since there is not currently such a method for numbers, strings, pairs, or arrays.

Moreover, we need a way of expressing the requirement that the elements of pairs or arrays can be formatted. As with the `Comparator` example in [Section 19.3, “Declaring Given Instances,”](#) on page 299, we will use a context bound.

The following `JSON` trait requires the existence of `stringify` as an extension method. A companion object declares given values `JSON[Double]` and `JSON[String]`:

```
trait JSON[T] :  
    extension (t: T) def stringify: String  
  
object JSON :  
    given JSON[Double] with  
        extension (x: Double)  
            def stringify = x.toString  
    def escape(s: String) = s.flatMap(  
        Map('\\' -> "\\\\", '\"' -> "\\\"", '\n' -> "\\n", '\r' ->  
        "\\r").withDefault(  
            _.toString))  
    given JSON[String] with  
        extension (s: String)  
            def stringify = s""""${escape(s)}"""
```

Now let’s define a suitable extension method for `Pair[T]`, provided that `T` can be formatted:

```
object Pair :  
    given [T : JSON] : JSON[Pair[T]] with  
        extension (p: Pair[T])  
            def stringify: String =  
                s"""{"first": ${p.first.stringify}, "second":  
                    ${p.second.stringify}}"""
```

Have a close look at the expression `p.first.stringify`. How does the compiler know that `p.first` has a `stringify` method? The type of `p.first` is `T`. Because of the context bound, there is a given object of type `JSON[T]`. That given object defines an extension method for the type `T` with name `stringify`. You have just seen an example of rule #3 for locating extension methods.

Finally, for rule #4, consider the call `Pair(1.7, 2.9).stringify`. The extension method is not declared in the `Pair` companion object but in a given value that is declared in `Pair`.

19.8 Implicit Conversions

An *implicit conversion* is a function with a single parameter that is a given instance of `Conversion[S, T]` (which extends the type `S => T`). As the name suggests, such a function is automatically applied to convert values from the source type `S` to the target type `T`.

Consider the `Fraction` class from [Chapter 11](#). We want to convert integers `n` to fractions `n / 1`.

```
given int2Fraction: Conversion[Int, Fraction] = n => Fraction(n,  
1)
```

Now we can evaluate

```
val result = 3 * Fraction(4, 5) // Calls int2Fraction(3)
```

The implicit conversion turns the integer `3` into a `Fraction` object. That object is then multiplied by `Fraction(4, 5)`.

You can give any name, or no name at all, to the conversion function. Since you don't call it explicitly, you may be tempted to drop the name. But, as you will see in [Section 19.10, “Importing Implicit Conversions,”](#) on page 308, sometimes it is useful to import a conversion function. I suggest that you stick with the `source2target` convention.

Scala is not the first language that allows the programmer to provide automatic conversions. However, Scala gives programmers a great deal of

control over when to apply these conversions. In the following sections, we will discuss exactly when the conversions happen, and how you can fine-tune the process.

Note

Even though Scala gives you tools to fine-tune implicit conversions, the language designers realize that implicit conversions are potentially problematic. To avoid a warning when using implicit functions, add the statement `import scala.language.implicitConversions` or the compiler option `-language:implicitConversions`.

Note

In C++, you specify implicit conversions as one-argument constructors or member functions with the name `operator Type()`. However, in C++, you cannot selectively allow or disallow these functions, and it is common to run into unwanted conversions.

19.9 Rules for Implicit Conversions

Implicit conversions are considered in two distinct situations:

1. If the type of an argument differs from the expected type:

```
Fraction(3, 4) * 5 // Calls int2Fraction(5)
```

The `*` method of `Fraction` doesn't accept an `Int` but it accepts a `Fraction`.

2. If an object accesses a nonexistent member:

```
3 * Fraction(4, 5) // Calls int2Fraction(3)
```

The `Int` class doesn't have an `*(Fraction)` member but the `Fraction` class does.

On the other hand, there are some situations when an implicit conversion is *not* attempted:

1. No implicit conversion is used if the code compiles without it. For example, if `a * b` compiles, the compiler won't try `a * convert(b)` or `convert(a) * b`.
 2. The compiler will never attempt multiple conversions, such as `convert1(convert2(a)) * b`.
 3. Ambiguous conversions are an error. For example, if both `convert1(a) * b` and `convert2(a) * b` are valid, the compiler will report an error.
-



Caution

Suppose we also have a conversion

```
given fraction2Double: Conversion[Fraction, Double] = f =>
f.num * 1.0 / f.den
```

It is *not* an ambiguity that

```
3 * Fraction(4, 5)
```

could be either

```
3 * fraction2Double(Fraction(4, 5))
```

or

```
int2Fraction(3) * Fraction(4, 5)
```

The first conversion wins over the second, since it does not require modification of the object to which the `*` method is applied.



Tip

If you want to find out which using parameters, extension methods, and implicit conversions the compiler uses, compile your program as

```
scalac -Xprint:typer MyProg.scala
```

You will see the source after these contextual mechanisms have been added.

19.10 Importing Implicit Conversions

Scala will consider the following implicit conversion functions:

1. Implicit functions or classes in the companion object of the source or target type
2. Implicit functions or classes that are in scope

For example, consider the `int2Fraction` and `fraction2Double` conversions. We can place them into the `Fraction` companion object, and they will be available for conversions involving fractions.

Alternatively, let's suppose we put the conversions inside a `FractionConversions` object, which we define in the `com.horstmann.impatient` package. If you want to use the conversions, import the `FractionConversions` object, like this:

```
import com.horstmann.impatient.FractionConversions.given
```

You can localize the import to minimize unintended conversions. For example,

```
@main def demo =  
    import com.horstmann.impatient.FractionConversions.given  
    val result = 3 * Fraction(4, 5) // Uses imported conversion  
    fraction2Double  
    println(result)
```

You can even select the specific conversions that you want. If you prefer `int2Fraction` over `fraction2Double`, you can import it specifically:

```
import com.horstmann.impatient.FractionConversions.int2Fraction  
val result = 3 * Fraction(4, 5) // result is Fraction(12, 5)
```

You can also exclude a specific conversion if it causes you trouble:

```
import com.horstmann.impatient.FractionConversions.  
{fraction2Double as _, given}  
// Imports everything but fraction2Double
```



If you want to find out why the compiler *doesn't* use an implicit conversion that you think it should use, try adding it explicitly, for example by calling `fraction2Double(3) * Fraction(4, 5)`. You may get an error message that shows the problem.

19.11 Context Functions

Consider our first example of a function with a context parameter:

```
def quote(what: String) (using delims: QuoteDelimiters) =  
    delims.left + what + delims.right
```

What is the type of `quote`? It can't be

```
String => String
```

because there is that second parameter list. But clearly it's not

```
String => QuoteDelimiters => String
```

either, because then we would call it as

```
quote(text) (englishQuoteDelims)
```

and not

```
quote(text) (using englishQuoteDelims)
```

There has to be some way of writing that type, and this is what the Scala designers came up with:

```
String => QuoteDelimiters ?=> String
```



Perhaps you are surprised that the `?` is with the second arrow.

Recall that a curried function type `T => S => R` is really `T => (S => R)`. When you fix the first parameter, you get a function `S => R`.

In our case, fixing the `text` to a particular value, we have a function with one `using` parameter, whose type is denoted as

```
QuoteDelimiters ?=> String.
```



If a method takes only `using` parameters, such as

```
def randomQuote(using delims: QuoteDelimiters) =
```

```
delims.left  
+ scala.io.Source.fromURL(  
"https://horstmann.com/random/quote").mkString.split(" - ")  
(0)  
+ delims.right
```

then its type has the form `QuoteDelims ?=> String`

Now that you have seen this curious syntax, let us put it to practical use. Suppose you develop a library where you work with the `Future` trait. In [Chapter 16](#), we used the global `ExecutionContext`, but in practice, you will need to tag all your methods with `(using ExecutionContext)`:

```
def GET(String url) (using ExecutionContext): Future[String] =  
  Future {  
    blocking {  
      Source.fromURL(url).mkString  
    }  
  }
```

You can do slightly better than that by defining a type alias

```
type ContextualFuture[T] = ExecutionContext ?=> Future[T]
```

Then the method becomes

```
def GET(url: String): ContextualFuture[String] = ...
```

Here is a more interesting technique for implementing “magic identifiers”. I want to design an API for web requests where I can say:

```
GET("https://horstmann.com") { println(res) }
```

The first parameter of the `GET` method is a URL. The second is a handler—a block of code that is executed when the response is available. Note the

magic identifier `res` that denotes the response string. Here is how the magic will unfold:

- The `GET` method makes the HTTP request and obtains the response
- The `GET` method calls the handler, with the (wrapped) response as a `using` parameter
- `res` is a method that summons the response from the “vasty depth” of given objects

We put everything into a `Requests` object and use a wrapper class for the given response, so that we don’t pollute the givens with an API class.

```
object Requests :  
    class ResponseWrapper(val response: String)  
    def res(using wrapper: ResponseWrapper): String =  
        wrapper.response
```

The `GET` method has two curried parameters: the URL and the handler. The handler is a context function—note the `?=>` arrow:

```
def GET(url: String)(handler: ResponseWrapper ?=> Unit) =  
    Future {  
        blocking {  
            val response = Source.fromURL(url).mkString  
            handler(using ResponseWrapper(response))  
        }  
    }
```

Note that the response is wrapped and passed to the handler as a `using` parameter. Then the handler code executes, and the `res` function retrieves and unwraps the response object.

This is not meant to be an exemplary design for an HTTP request library. It would be better to work with the `Future` API. The point of the example is that `res` is used as a “magic variable” that is available in the scope of the handler function.

See [Exercise 12](#) on page 313 for another example of using context functions.

19.12 Evidence

In [Chapter 17](#), you saw the type constraints

```
T =:= U
T <:< U
```

The constraints test whether `T` equals `U` or is a subtype of `U`. To use such a type constraint, you supply a `using` parameter, such as

```
def firstLast[A, C](it: C)(using ev: C <:< Iterable[A]) : (A, A)
=
  (it.head, it.last)

firstLast(0 until 10) // the pair (0, 9)
```

The `=:=` and `<:<` are classes that produce given values, defined in the `Predef` object. For example, `<:<` is essentially

```
abstract class <:<[-From, +To] extends Conversion[From, To]

object `<:<` :
  given conforms[A]: (A <:< A) with
    def apply(x: A) = x
```

Suppose the compiler processes a constraint using `ev: Range <:< Iterable[Int]`. It looks in the companion object for a given object of type `Range <:< Iterable[Int]`. Note that `<:<` is contravariant in `From` and covariant in `To`. Therefore the object

```
<:<.conforms[Range]
```

is usable as a `Range <:< Iterable[Int]` instance. (The `<:<.conforms[Iterable[Int]]` object is also usable, but it is less specific and therefore not considered.)

We call `ev` an “evidence object”—its existence is evidence of the fact that, in this case, `Range` is a subtype of `Iterable[Int]`.

Here, the evidence object is the identity function. To see why the identity function is required, have a closer look at

```
def firstLast[A, C](it: C)(using ev: C <:< Iterable[A]) : (A, A)  
=  
(it.head, it.last)
```

The compiler doesn't actually know that `c` is an `Iterable[A]`—recall that `<:<` is not a feature of the language, but just a class. So, the calls `it.head` and `it.last` are not valid. But `ev` is a `Conversion[C, Iterable[A]]`, and therefore the compiler applies it, computing `ev(it).head` and `ev(it).last`.



Tip

To test whether a given evidence object exists, you can call the `summon` function in the REPL. For example, type `summon[Range <:< Iterable[Int]]` in the REPL, and you get a result (which you now know to be a function). But `summon[Iterable[Int] <:< Range]` fails with an error message.



Note

If you need to check whether a given value does *not* exist, use the `scala.util.NotGiven` type:

```
given ts[T](using T <:< Comparable[? >: T]):  
java.util.Set[T] =  
    java.util.TreeSet[T]()  
given hs[T](using NotGiven[T <:< Comparable[? >: T]]):  
java.util.Set[T] =  
    java.util.HashSet[T]()
```

19.13 The `@implicitNotFound` Annotation

The `@implicitNotFound` annotation raises an error message when the compiler cannot construct a given value of the annotated type. The intent is to give a useful error message to the programmer. For example, the `<:<` class is annotated as

```
@implicitNotFound(msg = "Cannot prove that ${From} <:< ${To}.")  
abstract class <:<[-From, +To] extends Function1[From, To]
```

For example, if you call

```
firstLast[String, List[Int]](List(1, 2, 3))
```

then the error message is

```
Cannot prove that List[Int] <:< Iterable[String]
```

That is more likely to give the programmer a hint than the default

```
Could not find implicit value for parameter ev: <:  
<[List[Int],Iterable[String]]
```

Note that `${From}` and `${To}` in the error message are replaced with the type parameters `From` and `To` of the annotated class.

Exercises

1. Call `summon` in the REPL to obtain the given values described in [Section 19.3, “Declaring Given Instances,”](#) on page 299.
2. What is the difference between the following primary constructor context parameters?

```
class Quoter(using delims: QuoteDelimiters) : ...  
class Quoter(val using delims: QuoteDelimiters) : ...  
class Quoter(var using delims: QuoteDelimiters) : ...
```

3. In [Section 19.3, “Declaring Given Instances,”](#) on page 299, declare `max` with a context bound instead of a `using` parameter.
4. Show how a given instance of a `Logger` class can be used in a body of code, without having to pass it along in method calls. Compare with the usual practice of retrieving logger instances with calls such as `Logger.getLogger(id)`.
5. Provide a given instance of `comparator` that compares objects of the class `java.awt.Point` by lexicographic comparison.
6. Continue the previous exercise, comparing two points according to their distance to the origin. How can you switch between the two orderings?
7. Produce a given `Ordering` for `Array[T]` instances, provided that there is a given `Ordering[T]`.
8. Produce a given `JSON` for `Iterable[T]` instances, provided that there is a given `JSON[T]`. Stringify as a JSON array.
9. Improve the `GET` method in [Section 19.11, “Context Functions,”](#) on page 309 so that it can retrieve bodies of type other than `String`. Use the Java `HttpClient` instead of `scala.io.Source`. Use a context bound that recognizes the (few) types for which there are body handlers.
10. Did you ever wonder how `"Hello" -> 42` and `42 -> "Hello"` can be pairs `("Hello", 42)` and `(42, "Hello")`? Define your own operator that achieves this.
11. Define a `!` operator that computes the factorial of an integer. For example, `5.!` is `120`.
12. Consider these function invocations for making an HTML table:

```
table {  
    tr {  
        td("Hello")  
        td("Bongiorno")  
    }  
    tr {  
        td("Goodbye")  
        td("Arrivederci")  
    }  
}
```

```
    }  
}
```

A `Table` class stores rows:

```
class Table:  
  val rows = new ArrayBuffer[Row]  
  def add(r: Row) = rows += r  
  override def toString = rows.mkString("<table>\n", "\n", "  
</table>")
```

Similarly, a `Row` stores data cells.

How do the `Row` instances get added to the correct `Table`? The trick is for the `table` method to introduce a `given` instance of type `Table`, and for the `tr` method to have a `using` parameter of type `Table`. Pass along a `Row` in the same way from the `tr` to the `td` method.

There is one technical complexity. The `table` and `tr` functions consume functions with no parameter, except that they need to pass along the `using` parameter. Their types are `Table ?=> Any` and `Row ?=> Any` (see [Section 19.11, “Context Functions,” on page 309](#)).

Complete the `Table` and `Row` classes and the `table`, `tr`, and `td` functions.

13. Improve the functions of the preceding exercise so that you can't nest a `table` inside another `table` or a `tr` inside another `tr`. Hint: `NotGiven`.

14. Look up the `=:=` object in `Predef.scala`. Explain how it works.

Chapter 20

Type Level Programming

Topics in This Chapter L3

- [20.1 Match Types](#)
- [20.2 Heterogeneous Lists](#)
- [20.3 Literal Type Arithmetic](#)
- [20.4 Inline Code](#)
- [20.5 Type Classes](#)
- [20.6 Mirrors](#)
- [20.7 Type Class Derivation](#)
- [20.8 Higher-Kinded Types](#)
- [20.9 Type Lambdas](#)
- [20.10 A Brief Introduction into Macros](#)
- [Exercises](#)

Why use Scala instead of Java on the JVM? Sure, the Scala syntax is more concise and more regular than Java, bu in the end, it is all about the types. Types help us to express what we expect our programs to do. Type mismatches help detect programming errors early, when they are cheap to fix. You have already seen many instances where the Scala type system is richer than Java. This chapter introduces you to advanced techniques of

“programming with types”—analyzing and generating types at compile time. These type manipulations are the foundation of some of the most powerful Scala libraries. The details are undeniably complex, and only of interest for advanced library creators. This overview should enable you to understand how those libraries work, and what is involved in their implementation.

The key points of this chapter are:

- Match types express types that vary with another type.
- A heterogeneous list has elements of different types.
- You can make compile-time computations with literal integer types.
- With inline functions, you can conditionally inject code at compile time.
- A type class defines behavior that classes can declare to support in an adhoc fashion.
- The mirror API provides support for analyzing types at compile time. An important application is the automatic derivation of type class instances.
- A higher-kinded type has a type parameter that is itself a parameterized type.
- A type lambda denotes a transformation of a type into another type.
- Macros programmatically generate and transform code at compile time.

20.1 Match Types

At times, it can be useful to perform pattern matching on the type level. Scala has *match types* for this purpose. Here is the canonical introductory example:

```
type Elem[C] = C match
  case String => Char
  case Iterable[t] => t
  case Any => C
```

The type `Elem[String]` is `Char`, but `Elem[List[Int]]` is `Int`. Note that the type variables, like any variables in a match expression, are *lowercase*.

Given this type, we can now define a function yielding the initial element of different types:

```
def initial[C <: Matchable](c: C): Elem[C] =  
  c match  
    case s: String => s(0)  
    case i: Iterable[t] => i.head  
    case _: Any => c
```



Caution

The `match` clause must have the exact same shape as the match type. The clauses must come in the same order. You cannot add other clauses. In our example, it might be tempting to add a clause such as `case "" => '\u0000'`, but that is a compile-time error.

Match types can be recursive. Given a list of lists, we may want the initial element of the initial list:

```
type FlatElem[C] = C match  
  case String => Char  
  case Iterable[t] => FlatElem[t] // Recursion  
  case Any => C
```

Now `FlatElem[List[List[Int]]]` is `Int`.

There is a wrinkle when working with this match type. The compiler cannot tell whether the element type `t` is `Matchable`. You finesse that with a cast:

```
def flatInitial[C](c: C): FlatElem[C] =  
  c.asInstanceOf[c.type & Matchable] match  
    case s: String => s(0)
```

```
case i: Iterable[t] => flatInitial(i.head)
case x: Any => x
```

The Scala API has a slightly prettier mechanism for the same purpose. With the statement

```
import compiletime.asMatchable
```

you can call `asMatchable` on any object to add the `Matchable` trait:

```
def flatInitial[C](c: C): FlatElem[C] =
  c.asMatchable match
    case s: String => s(0)
    case i: Iterable[t] => flatInitial(i.head)
    case x: Any => x
```

Now you can call

```
flatInitial(List(List(1, 7), List(2, 9))) // 1
```

20.2 Heterogeneous Lists

A `List[T]` collects elements of type `T`. What if we want to collect elements of different types? Of course, we can use a `List[Any]`, but then we lose all type information. In this section, we will build a *heterogeneous* list class that holds elements of different types.

A list is either empty, or it has a head of type `H` and a tail that is again a `HList`:

```
abstract sealed class HList :
  def ::[H, T >: this.type <: HList](head: H): HNonEmpty[H, T] =
    HNonEmpty(head, this)

  case object HEmpty extends HList
  case class HNonEmpty[H, T <: HList](head: H, tail: T) extends
    HList
```

With the `::` operator, you can build up heterogeneous lists:

```
val lst = "Fred" :: 42 :: HEmpty // Type is HNonEmpty[String,  
HNonEmpty[Int, HEmpty.type]]
```



Caution

There is a subtlety in the type parameters of the `::` method. If you define

```
def ::[H] (head: H): HNonEmpty[H, this.type] =  
  HNonEmpty(head, this)
```

then the type of `lst` has an unwanted wildcard: `HNonEmpty[String, ? <: HNonEmpty[Int, HEmpty.type]]`.

The problem is that `this.type` is *not* the type of the right hand side (i.e. `not HNonEmpty[Int, HEmpty.type]`). Instead, it is the singleton type consisting only of the right hand side object. You need an additional type variable to obtain the list type.

The Scala compiler can determine the type of each list element. Try evaluating the following expressions:

```
lst.head // Type is String  
lst.tail.head // Type is Int  
lst.tail.tail // Type is HEmpty
```

Consider the following function that concatenates two heterogeneous lists:

```
def append(a: HList, b: HList): HList =  
  a match  
    case HEmpty => b  
    case ne: HNonEmpty[?, ?] => HNonEmpty(ne.head,  
      append(ne.tail, b))
```

Unfortunately, the function is not very useful because it doesn't have a precise return type:

```
val lst2 = append(lst, lst) // Type is HList
```

You can't even ask for `lst2.head` because a `HList` doesn't have a `head` method. Only `HNonEmpty` does, but the compiler has no way of knowing that `lst2` is nonempty.



Note

If you try these instructions in the REPL, you will find that `lst2` has the value

```
HNonEmpty(Fred, HNonEmpty(42, HNonEmpty(Fred, HNonEmpty(42, HEmpty))))
```

However, that is only known at runtime.

To accurately determine the return type at compile time, use a match type:

```
type Append[A <: HList, B <: HList] <: HList = A match
  case HEmpty.type => B
  case HNonEmpty[h, t] => HNonEmpty[h, Append[t, B]]
```

This match type recursively computes the type of the concatenation. Let's look at the example of the computation `append(lst, lst)`. Both `A` and `B` are `HNonEmpty[String, HNonEmpty[Int, HEmpty]]`. Then you can work out `Append[A, B]`. After a couple of recursive steps, it is

```
HNonEmpty[String, HNonEmpty[Int, B]]
```

which is

```
HNonEmpty[String, HNonEmpty[Int, HNonEmpty[String, HNonEmpty[Int, HEmpty]]]]
```

That is the type that we need to use for the return type of `append`. Here is the correct function:

```
def append[A <: HList, B <: HList](a: A, b: B): Append[A, B] =  
  a match  
    case _: HEmpty.type => b  
    case ne: HNonEmpty[?, ?] => HNonEmpty(ne.head,  
      append(ne.tail, b))
```

Since the types of `a` and `b` can vary, the `append` function needs type parameters `A` and `B`, both subtypes of `HList`. The return type is accurately computed as `Append[A, B]`.

Now the compiler can determine the types of expressions such as

```
lst2.head // Type is String  
lst2.tail.tail.tail.tail // Type is HEmpty
```



Caution

The `match` clause must *exactly* mirror the match type. You must write

```
case _: HEmpty.type => b
```

in the function since the type has

```
case HEmpty.type => B
```

It would be an error to write

```
case HEmpty => b
```



Note

The `HList` example shows you how to carry out type-level programming. If you want to use heterogeneous lists in your Scala code, you don't need to implement your own `HList` class. Just use tuples. They are implemented with the same techniques that this chapter covers. For example, the Scala compiler knows that the type of `("Fred", 42) ++ ("Fred", 42)` is `(String, Int, String, Int)`.

20.3 Literal Type Arithmetic

[Chapter 18](#) introduced literal types, such as the type `3` that holds a single value, namely the integer 3.

Why have values at the type level? It allows for checks and computations at compile time. For example, we can define a `Vec[N]` as a vector with `N` components, where `N` is a compile-time constant integer. Then we can check at compile time that a `Vec[3]` can be added to another `Vec[3]` but not a `Vec[4]`:

```
class Vec[N <: Int] :  
  private val xs: Array[Double] = ...  
  def +(other: Vec[N]) = ... // Must have the same size
```

Note that the type parameters must be compile-time constants. You can declare a `Vec[3]` but not a `Vec[n]`, where `n` is a variable of type `n`.

The bound `N <: Int` is not enough to guarantee that `N` is a singleton type. To fix that, add a `using` parameter of type `ValueOf[N]`, like this:

```
class Vec[N <: Int] (using n: ValueOf[N]) :  
  private val xs: Array[Double] = Array.ofDim[Double](n.value)  
  def +(other: Vec[N]) = // Must have the same size  
    val result = Vec[N]()  
    for i <- 0 until xs.length do  
      result.xs(i) = xs(i) + other.xs(i)
```

```
result  
...
```

A given instance of `valueOf[S]` exists for all singleton types whose single occupant is known to the compiler. It has a method `value` that produces the single value, forming a bridge from the world of types to the world of values. In the code snippet above, the `value` is used to construct the array of vector coordinates.



Caution

The `using` parameter is required. It is not enough to constrain the type of `N` to `Int & Singleton`. The following does not work:

```
class Vec[N <: Int & Singleton] :  
  private val xs: Array[Double] = Array.ofDim[Double]  
(summon[ValueOf[N]].value)
```

Here, the compiler has insufficient information to infer the single value of `N`.

Now consider the operation of concatenating a `Vec[N]` with a `Vec[M]`. The result is a vector with `N + M` components. Of course, `N` and `M` are types. Can you add two types?

If you import `scala.compiletime.ops.int.*`, you can. That package defines types with names `+`, `-`, and so on, which carry out arithmetic at the type level. If `N` and `M` are singleton integer types, the type `N + M` is the type whose sole inhabitant is the sum of the inhabitants of `N` and `M`.

Here is how to define the concatenation:

```
def ++[M <: Int](other: Vec[M])(using ValueOf[N + M]) =  
  val result = Vec[N + M]()  
  for i <- 0 until xs.length do  
    result.xs(i) = xs(i)  
  for i <- 0 until other.xs.length do
```

```
result.xs(i + xs.length) = other.xs(i)  
result
```



Caution

In Scala 3.2, the singleton type inference is rather fragile. If the `using` parameter has type `valueOf[M]`, the Scala compiler does not infer that both `N` and `M` are singletons that can be combined with `+`.

As a final use of type-level arithmetic, let us provide a `middle` method that yields the middle component of a vector, provided the number of components is odd.

The `scala.compiletime.ops.any` package defines `==` and `!=` type operators. The type `N % 2 != 0` is the literal type `true` when `N` is inhabited by an odd integer.

```
def middle(using N % 2 != 0 =:= true) = xs(n.value / 2)
```

A call `v.middle` yields a compile-time error if `v` has an even number of components. In that case, `N % 2 != 0` is `false`, and it is impossible to summon a given `false =:= true` instance.

20.4 Inline Code

In the preceding section, you saw how to carry out computations at the type level, to ensure that two integer type parameters match, or to form their sum, or to ensure that one is odd. These computations happen when the program is compiled.

For more complex compile-time computations, you can use the Scala facilities for generating *inline code*. In this section, you will see basic examples that don't involve type-level programming. [Section 20.6, “Mirrors,”](#) on page 328 puts the features to work for automatically generating code for case classes.

Even more complex compile-time transformations are possible with macros—see [Section 20.10, “A Brief Introduction into Macros,”](#) on page 335. However, programming with macros is quite a bit more complex. The inline code generation was designed to avoid that complexity in common use cases.

This example nicely shows off the mechanics of inline code generation:

```
inline def run[T] (inline task: => T, profile: Boolean) =  
  var start: Long = 0  
  inline if profile then start = System.nanoTime()  
  val result = task  
  inline if profile then  
    val end = System.nanoTime()  
    println(f"Duration of ${codeOf(task)}: ${(end - start) * 1E-  
9}%.3f")  
  result
```

The function runs some task, and if the `profile` flag is `true`, the running time is printed. Read the code, skipping over the `inline` keywords. You see that there is a by-name parameter `task` that is executed. The result, of type `T`, is returned. A typical call would be:

```
println(run(Thread.sleep(1000), true))
```

The output is something like

```
Duration of Thread.sleep(1000L): 1.001 ()
```

Note the message that is generated by the `println` statement inside the `run` function, indicating the time that elapsed. The `()` that follows is from the `println` around the call to `run`, printing the `Unit` return value.

The call

```
println(run(Thread.sleep(1000), false))
```

simply prints

()

That is not remarkable by itself. To see what is special about the inline code generation, look at the generated bytecodes—see [Exercise 4](#) on page 339. There is no separate `run` function. Instead, the code of `run` is placed inside the function that calls `run`. The first copy of the code, when `profile` is `true`, contains the calls to `System.nanoTime` and the string formatter, in addition to the `Thread.sleep` call. The second copy of the code only calls `Thread.sleep`.

Note the three uses of the `inline` keyword:

- The `inline` function causes the function code to be placed into the location of the function call.
 - The `inline` parameter causes the call expression to be placed inside the function code.
 - `inline if` discards the code in non-matching branches.
-



Caution

Each use of an `inline` parameter is expanded. The implementation

```
inline def run[T](inline task: => T, profile: Boolean) =  
  var start: Long = 0  
  if profile then  
    start = System.nanoTime()  
    val result = task  
    val end = System.nanoTime()  
    println(f"Duration of ${codeOf(task)}: ${(end -  
      start) * 1E-9}%.3f")  
    result  
  else  
    task
```

would inline the code for `task` twice.

Finally, note the nifty `codeOf(task)` that yields a string representing the inline expression. The string comes from the parse tree for the expression, not the source code. Carefully look at the output `Thread.sleep(1000L)`, whereas the source code is `Thread.sleep(1000)`, without the `L`. The `codeOf` function is in the `scala.compiletime` package.

An `inline match`, similar to an `inline if`, generates code only for the matching branch:

```
inline def sizeof[T <: Matchable] =
  inline erasedValue[T] match
    case _: Boolean => 1
    case _: Byte => 1
    case _: Char => 2
    case _: Short => 2
    case _: Int => 4
    case _: Float => 4
    case _: Long => 8
    case _: Double => 8
    case _ => error("Not a primitive type")
```

For example, `sizeof[Int]` is replaced with 4.

The `scala.compiletime.erasedValue` method is needed because `T` is a type, but the selector of a `match` expression needs to be a value. It pretends to produce a value of the given type, that then is matched against one of the `case` branches. Actually, the compiler just matches the types, but this way, no new syntax for type matching had to be invented.

The `scala.compiletime.error` method reports a compile-time error with the given message.

When each branch of an `inline match` returns a different type, you may want to declare the function as `transparent inline`. Then the compiler doesn't use the common supertype as the return type. Here is an example:

```
transparent inline def defaultValue[T <: Matchable] =
  inline erasedValue[T] match
    case _: Boolean => false
```

```

case _: Byte => 0.asInstanceOf[Byte]
case _: Char => '\u0000'
case _: Short => 0.asInstanceOf[Short]
case _: Int => 0
case _: Float => 0.0f
case _: Long => 0L
case _: Double => 0.0
case _ => error("Not a primitive type")

```

Now the type of `defaultValue[Int]` is `Int`. Without the `transparent` keyword, it would be `AnyVal`.

There are no inline loops, but inline functions can be recursive. This function concatenates the elements of a tuple:

```

inline def mkString[T <: Tuple](inline args: T, separator:
String): String =
  inline args match
    case EmptyTuple => ""
    case t1: (? *: EmptyTuple) => t1.head.toString
    case t2: NonEmptyTuple => t2.head.toString + separator +
      mkString(t2.tail, separator)

```

For example, `mkString(("Fred", 42), ", ")` is first reduced to `"Fred".toString + ", " + mkString((42), ", ")` and then to `"Fred".toString + ", " + 42.toString`.

20.5 Type Classes

If a class wants to compare its elements in a standard fashion, it can extend the `Comparable` trait. Dozens of Java classes do just that. Inheritance is the object-oriented way of acquiring a capability.

But that mechanism is inflexible. Consider a `List[T]` type. It can't extend `Comparable` because the element type might not be comparable. That's a shame because there is a natural comparison for lists with comparable element types.

In [Chapter 19](#), you saw a different approach. We provided given `Comparator` instances for numbers, strings, and any class extending `Comparable`. Then we synthesized `Comparator` instances for those lists whose element types have a given `Comparator`. Algorithms such as sorting make use of the comparison with a `using` parameter.

This selective synthesis is much more flexible than the object-oriented approach. In order to compare instances, it is not necessary to modify the class at all. Instead, one provides a given instance.

A trait such as `Comparator` is called a *type class*. A type class defines some behavior, and a type can join the class by providing that behavior. The term comes from Haskell, and “class” is not used in the same way as in object-oriented programming. Think of “class” as in a “class action”—types banding together for a common purpose.

To see how a type joins a type class, let us look at a simple example. We want to compute averages, $(x_1 + \dots + xn) / n$. To do that, we need to be able to add two values and divide a value by an integer. There is a type class `Numeric` in the Scala library, which requires that values can be added, multiplied, and compared. But it doesn’t have any requirement that values can be divided by an integer. Therefore, let’s define our own:

```
trait Averageable[T] :  
    def add(x: T, y: T): T  
    def divideBy(x: T, n: Int): T
```

Next, to make sure that the type class is useful out of the gate, let’s include some common types. That’s easy to do by providing given objects in the companion object:

```
object Averageable :  
    given Averageable[Double] with  
        def add(x: Double, y: Double) = x + y  
        def divideBy(x: Double, n: Int) = x / n  
  
    given Averageable[BigDecimal] with
```

```
def add(x: BigDecimal, y: BigDecimal) = x + y
def divideBy(x: BigDecimal, n: Int) = x / n
```

Now we are ready to put the type class to use. In the `average` method, we need an instance of the type class so that we can call `add` and `divideBy`. (Note that these are methods of the type class, not the argument type.)

Here, we'll just compute the average of two values. The general case is left as an exercise. There are two ways in which we can supply the type class instance: as a using parameter, or with a context bound. Let's do the latter:

```
def average[T : Averageable](x: T, y: T) =
  val ev = summon[Averageable[T]]
  ev.divideBy(ev.add(x, y), 2)
```

Finally, let's see what a type needs to do to join the `Averageable` type class. It must provide a given object, just like the given instances for `Double` and `BigDecimal` that we provided out of the gate. Here is how `Point` can join:

```
case class Point(x: Double, y: Double)

object Point :
  given Averageable[Point] with
    def add(p: Point, q: Point) = Point(p.x + q.x, p.y + q.y)
    def divideBy(p: Point, n: Int) = Point(p.x / n, p.y / n)
```

Here we added the given object to the companion object of `Point`. If you can't modify the `Point` class, you can put the given object elsewhere and import it as needed.

The standard Scala library provides useful type classes, such as `Equiv`, `Ordering`, `Numeric`, `Fractional`, `Hashing`, `IsIterableOnce`, `IsIterable`, `IsSeq`, `IsMap`, `FromString`. As you have seen, it is easy to provide your own.

The important point about type classes is that they provide an “ad-hoc” way of providing polymorphism that is less rigid than inheritance.

20.6 Mirrors

In this section, we will use a type class that provides a JSON representation for any object:

```
trait JSON[A] :  
    def stringify(a: A): String
```

Here are given instances for common types:

```
object JSON :  
    given jsonDouble: JSON[Double] with  
        def stringify(x: Double) = x.toString  
  
    given jsonString: JSON[String] with  
        def escape(s: String) = s.flatMap(  
            Map('\\' -> "\\\\", '\"' -> "\\\"", '\n' -> "\\\n", '\r' ->  
            "\\\r").withDefault(  
                _.toString))  
        def stringify(s: String): String = s"\\"${escape(s)}\""
```

Given any case class, we can define a given instance that stringifies the fields:

```
case class Person(name: String, age: Int)  
  
given jsonPerson: JSON[Person] with  
    def stringify(p: Person) =  
        val name = summon[JSON[String]].stringify(p.name)  
        s"""{"name": $name, "age": ${p.age}}"""
```

But that is clearly tedious. In Java, one could use reflection to automatically stringify arbitrary instances at runtime. In Scala, we can do better, and generate the converter code at compile time.

To analyze types at compile time, we need to use *mirrors*. Every case class `T` gives rise to a type `Mirror.Product[T]`. (Recall from [Chapter 14.10](#) that

case classes extend the `Product` trait.) The mirror describes the following features of the class:

- `MirroredElemTypes`, a tuple of the field types
- `MirroredElemLabels`, a tuple of the field names
- `MirroredLabel`, the name of the class

Note that all of these are *types*. For example, if `m` is the given `Mirror.Product[Person]`, then `m.MirroredElemTypes` is the tuple type `(String, Int)`, `m.MirroredElemLabels` is a tuple type of two singleton types `("name", "age")`, and `m.mirroredLabel` is the singleton type `Person`.

In the JSON representation, we need to use the field names. But we need them as string objects, not singleton types.

The `scala.compiletime` package has three methods for turning singleton types into values:

- `constValue[T]` yields the sole value of the singleton type `T`
- `constValueOpt[T]` yields an `Option` with the sole value if `T` is a singleton type, `None` otherwise
- `constValueTuple[T]` yields a tuple of singleton values, where `T` is a tuple type of singleton types.

You can move from types to objects as follows:

```
val fieldNames = constValueTuple[m.MirroredElemLabels] // A tuple  
of strings
```

For JSON formatting, we don't need the class name, but if you need it, it is

```
val className = constValue[m.MirroredLabel]
```

We need to obtain the given JSON instances for each of the field types. There are three methods for summoning given instances in an inline function:

- `summonInline[T]` summons the given instance of type `T`
- `summonAll[T]`, where `T` is a tuple type, summons a tuple of all given instances of the element types of `T`

- `summonFrom` attempts to summon instances given in a `case` clause; for example

```
summonFrom { case given Ordering[T] => TreeSet[T](); case _ =>
  HashSet[T]() }
```

Here is one way to summon the given JSON instances:

```
inline def summonGivenJSON[T <: Tuple]: Tuple =
  inline erasedValue[T] match
    case _: EmptyTuple => EmptyTuple
    case _: (t *: ts) => summonInline[JSON[t]] *:
      summonGivenJSON[ts]
```

The result is a tuple of JSON instances, such as `JSON[String]` and `JSON[Int]` if `T` is `Person`. Note that again we go from a tuple of types to a tuple of values.

You will see more elegant ways of collecting such type class instances in [Section 20.8, “Higher-Kinded Types,”](#) on page 332 and [Section 20.9, “Type Lambdas,”](#) on page 334.

At this point, we have all the tools to produce a JSON instance for an arbitrary `Product`. Every case class is a subclass of `Product`.

```
inline given jsonProduct[T] (using m: Mirror.ProductOf[T]): JSON[T] =
  new JSON[T] :
    def stringify(t: T): String =
      val fieldNames = constValueTuple[m.MirroredElemLabels]
      val jsons = summonGivenJSON[m.MirroredElemTypes]
      "{ " +
        fieldNames.productIterator.zip(jsons.productIterator)
          .zip(t.asInstanceOf[Product].productIterator).map {
            case ((f, j), v) => s""""$f": ${j.asInstanceOf[JSON[Any]].stringify(v)}"""
          }.mkString(", ") + " }"
```

The `cast j.asInstanceOf[JSON[Any]]` is necessary since `jsons` is only known to be a Tuple.



Caution

With Scala 3.2, you cannot use the `with` syntax with `inline given`.

Now any case class instance can be formatted as JSON:

```
case class Item(description: String, price: Double)

summon[JSON[Item]].stringify(Item("Blackwell Toaster", 19.95))
// { "description": "Blackwell Toaster", "price": 19.95 }
```

20.7 Type Class Derivation

In the preceding section, you saw how to automatically obtain a `JSON` instance for any product type. You can do the same for sum types. Recall that a sum type is given by a sealed class or an `enum` class, such as

```
sealed abstract class Amount
case class Dollar(value: Double) extends Amount
case class Currency(value: Double, unit: String) extends Amount
```

or

```
enum BinaryTree[+A] :
  case Node(value: A, left: BinaryTree[A], right: BinaryTree[A])
  case Empty
```

Analogous to the process of the preceding section, here is how to produce a `JSON` instance for a sum type:

```
inline given jsonSum[T] (using m: Mirror.SumOf[T]): JSON[T] =
  new JSON[T] :
```

```

def stringify(t: T): String =
  val index = m.ordinal(t)
  val jsons = summonGivenJSON[m.MirroredElemTypes]

  jsons.productElement(index).asInstanceOf[JSON[Any]].stringify(t)

```

With a `Mirror.SumOf`, `m.MirroredElemTypes` is a tuple of all subtypes, and the call `m.ordinal(t)` yields the index of the type `t` in that tuple.

Summoning a `JSON[Amount]` will summon `JSON` instances of all subtypes, namely `JSON[Dollar]` and `JSON[Currency]`. These are case classes, and the instances are generated using `jsonProduct`.

However, if you try `summon[JSON[BinaryTree[String]]]`, you run into a problem. The subclass `BinaryTree[String].Node` yields an error since the compiler doesn't yet know how to deal with the fields of type `BinaryTree`.

This can be finessed through *type class derivation*. `BinaryTree` declares that it is an instance of the `JSON` type class, with the syntax

```

enum BinaryTree[+A] derives JSON :
  case Node(value: A, left: BinaryTree[A], right: BinaryTree[A])
  case Empty

```

The `JSON` companion object must provide a method or value called `derived` that yields the type class instance. Here is a suitable method:

```

inline def derived[T] (using m: Mirror.Of[T]): JSON[T] =
  inline m match
    case s: Mirror.SumOf[T] => jsonSum(using s)
    case p: Mirror.ProductOf[T] => jsonProduct(using p)

```

Keep in mind that few programmers will implement their own type classes. Libraries for interacting with JSON, SQL, and so on, will define appropriate type classes. Users of such libraries indicate with the `derives` syntax that a particular class is an instance of such a type class.

20.8 Higher-Kinded Types

The generic type `List` has a type parameter `T` and produces a type. For example, given the type `Int`, you get the type `List[Int]`. For that reason, a generic type such as `List` is sometimes called a *type constructor*, denoted as `List[_]`. In Scala, you can go up another level and have type parameters that are type constructors.

To see why this can be useful, consider the `summonGivenJSON` method from [Section 20.6, “Mirrors,”](#) on page 328. If we want to collect elements of another type class, we have to write the method all over again, just with the other type class instead of `JSON`. It would be nice if we could pass the type class as a parameter. That is a typical use for a higher-kinded type:

```
inline def summonGiven[T <: Tuple, C[_]]: Tuple =  
  inline erasedValue[T] match  
    case _: EmptyTuple => EmptyTuple  
    case _: (t *: ts) => summonInline[C[t]] *: summonGiven[ts,  
C]
```

Now the call `summonGiven[m.MirroredElemTypes, JSON]` gathers up all given `JSON[t]` instances.

We say that `summonGiven` is *higher-kinded* because it has a type parameter that itself has a type parameter. (There is a theory of *kinds* that describes the complexity of type parameters. The details are not important for most programmers because it is very uncommon to have situations that are more complex than the one you are seeing here.)

This example is concise, but rather specialized. For another example, let’s start with the following simplified `Iterable` trait:

```
trait Iterable[E] :  
  def iterator(): Iterator[E]  
  def map[F](f: E => F): Iterable[F]
```

Now consider a class implementing this trait:

```

class Buffer[E] extends Iterable[E] :
    def iterator(): Iterator[E] = ...
    def map[F](f: E => F): Buffer[F] = ...

```

For a buffer, we expect that `map` returns a `Buffer`, not a mere `Iterable`. That means we cannot implement `map` in the `Iterable` trait itself, which we would like to do. A remedy is to parameterize the `Iterable` with a type constructor, like this:

```

trait Iterable[E, C[_]] :
    def iterator(): Iterator[E]
    def build[F](): C[F]
    def map[F](f : E => F) : C[F]

```

Now an `Iterable` depends on a type constructor for the result, denoted as `C[_]`. This makes `Iterable` a higher-kinded type.

The type returned by `map` may or may not be the same as the type of the `Iterable` on which `map` was invoked. For example, if you invoke `map` on a `Range`, the result is not generally a range, so `map` must construct a different type such as a `Buffer[F]`. Such a `Range` type is declared as

```
class Range extends Iterable[Int, Buffer]
```

Note that the second parameter is the type constructor `Buffer`.

To implement `map` in `Iterable`, we need a bit more support. An `Iterable` needs to be able to produce a container that holds values of any type `F`. Let's define a `Container` trait—it is something to which you can add values:

```

trait Container[E : ClassTag] :
    def +=(e: E): Unit

```

We require the `ClassTag` context bound since some containers need to be able to construct a generic `Array[E]`.

The `build` method is required to yield such an object:

```

trait Iterable[E, C[X] <: Container[X]] :
  def build[F : ClassTag](): C[F]
  ...

```

The type constructor `C` has now been constrained to be a `Container`, so we know that we can add items to the object that `build` returns. We can no longer use a wildcard for the parameter of `C` since we need to indicate that `C[X]` is a container for the same `X`.

The `map` method can be implemented in the `Iterable` trait:

```

def map[F : ClassTag](f : E => F) : C[F] =
  val res = build[F]()
  val iter = iterator()
  while iter.hasNext do
    res += f(iter.next())
  res

```

`Iterable` classes no longer need to supply their own `map`. Here is the `Range` class:

```

class Range(val low: Int, val high: Int) extends Iterable[Int, Buffer] :
  def iterator() = new Iterator[Int] :
    private var i = low
    def hasNext = i <= high
    def next() = { i += 1; i - 1 }

  def build[F : ClassTag]() = Buffer[F]

```

Note that a `Range` is an `Iterable`: You can iterate over its contents. But it is not a `Container`: You can't add values to it.

A `Buffer`, on the other hand, is both:

```

class Buffer[E : ClassTag] extends Iterable[E, Buffer] with
  Container[E] :
  private var capacity = 10

```

```

private var length = 0
private var elems = Array[E](capacity) // See note

def iterator() = new Iterator[E] :
  private var i = 0
  def hasNext = i < length
  def next() = { i += 1; elems(i - 1) }

def build[F : ClassTag]() = Buffer[F]

def +=(e: E) =
  if length == capacity then
    capacity = 2 * capacity
    val nelems = Array[E](capacity) // See note
    for (i <- 0 until length) nelems(i) = elems(i)
    elems = nelems
  elems(length) = e
  length += 1

```

This example showed a typical use of higher-kinded types. An `Iterable` depends on `Container`, but `Container` isn't a type—it is a mechanism for making types.

20.9 Type Lambdas

We can think of a type constructor such as `List[_]` as a type-level function, mapping a type `x` to the type `List[x]`. This can be expressed with a special syntax, called a *type lambda*:

`[X] =>> List[X]`

In this simple case, the type lambda syntax is not necessary, but it is useful for expressing more complex type-level mappings.



Caution

The `[]` are required around the type arguments. For example, `x =>> List[x]` is not a valid type lambda.

Here is another example. Recall the integer type arithmetic from [Section 20.3, “Literal Type Arithmetic,”](#) on page 322. Now we want a general mechanism to check for conditions such as “a compile-time constant is even”. Here it is:

```
def validate[V <: Int, P[_ <: Int] <: Boolean] (
    using ev: P[V] =:= true, v: ValueOf[V]) = v.value
```

`v` is a singleton integer type and `P` a type constructor turning integers to Booleans; that is, a condition. If the condition is fulfilled at the type level, `P[V] =:= true` can be summoned, and the value is retrieved.

You can specify `P` as a type lambda. Here is how to check that a value is even:

```
validate[10, [V <: Int] =>> V % 2 == 0]
```

Note the type lambda. Without it, you would need to define a named type

```
type Even[V <: Int] = V % 2 == 0
```

and call

```
validate[10, Even]
```

Finally, let us use the concept of type lambdas to make one more simplification of the `summonGivenJSON` function of [Section 20.6, “Mirrors,”](#) on page 328. We handcrafted a transformation from a tuple of types to a tuple of given instances. There is a function `summonAll` in the `scala.compiletime` package that will summon the given instances from a tuple of types. But we couldn’t use it because we didn’t have a tuple of the right types. We had a tuple `(String, Int)` but need to summon the

instances of `(JSON[String], JSON[Int])`. If we could just map `[X] =>> JSON[X]`, we would be all set.

That is exactly what the `Tuple.Map` type does. It applies a type-level transformation to a tuple of types. The type

```
type ElemTypesJSON = Tuple.Map[m.MirroredElemTypes, [X] =>>
JSON[X]]
```

is the tuple of `JSON` type classes of the field types.

You don't actually need the type lambda here. Simply summon the instances as:

```
val jsons = summonAll[Tuple.Map[m.MirroredElemTypes, JSON]]
```

20.10 A Brief Introduction into Macros

In this chapter, you have seen several techniques for carrying out compile-time computations. Macros provide a far more general mechanism for arbitrary code transformation at compile time. However, writing macros is complex and requires significant insight into the compilation process. Macro programming is a very specialized skill that is only needed to implement advanced libraries with seemingly magical powers.

This section covers a couple of use cases for macros and briefly shows you the implementation. Hopefully, that will give you a flavor of what is possible with macros, so that you have a basic understanding how the implementors of those advanced libraries achieve their magical results.

Let us start with a simple example. When debugging, it is common to write print statements such as

```
println("someVariable = " + someVariable)
```

It is annoying that one has to write the variable name twice: once to print the name, and again to get the value. That problem can actually be solved with an inline function:

```
inline def debug1(inline expr: Any): Unit = println(codeOf(expr)
+ " = " + expr)
```

You already saw the `codeOf` function in [Section 20.4, “Inline Code,”](#) on page 323. It prints the code of an inline value. Here is an example:

```
val x = 6
val y = 7
debug1(x * y) // Prints x.*(y) = 42
```

But suppose we also want to see the type of the expression. There is no library function for that. Instead, we need to write a macro.

Macros permit manipulation of syntax trees. If `e` is a Scala expression of type `T`, then `'{e}` is the syntax tree of type `Expr[T]`. There are methods for analyzing and generating syntax trees.

Conversely, if you have a value `s` of type `Expr[T]`, you can use the expression that it represents inside a Scala expression, using the syntax `$(s)`.

The `'` and `$` operations are called *quote* and *splice*.

In our `debug1` example, we will produce a syntax tree for a call to `println`:

```
'{println(${...} + ": " + ${...} + " = " + ${...})}
```

In general, the idea is to do as much as possible in Scala syntax, and use the syntax tree API only when necessary. Here, we use Scala syntax for the concatenation and the invocation of `println`.

Here is the complete implementation:

```
inline def debug1[T](inline arg: T): Unit = ${debug1Impl('arg)}

private def debug1Impl[T : Type](arg: Expr[T])(using Quotes):
Expr[Unit] =
  '{println(${Expr(arg.show)} + ": " + ${Expr(Type.show[T])} + "
= " + $arg)}
```

For technical reasons, a macro must be called from an inline function; here, `debug1`, using a top-level splice and quoting its arguments. It is customary to use an `Impl` suffix for the macro function.

Note that `'arg` is the same as `'{arg}`. No braces are required with a variable name.

The macro function receives arguments of type `Expr`, and it needs to use a given `Quotes` instance.

The call `arg.show` yields a string representation of the syntax tree of `arg`. To splice it in, we need to turn it to an `Expr`.

Similarly, `Type.show[T]`, which yields a string representation of the type parameter, is turned into an `Expr`.

Finally, the `arg` variable, which is itself a syntax tree, is spliced in. Again, since it is a variable, no braces are required.

The `debug1` function prints a single expression. It would be nice to print more than one:

```
debugN("Test description", x, y, x * y)
```

Now there is no easy way of printing the types, but instead, let us not print the expressions for literal values. It would look silly to print `Test description = Test description`.

The implementation is a bit more complex:

```
inline def debugN(inline args: Any*): Unit = ${debugNImpl('args) }

private def debugNImpl(args: Expr[Seq[Any]]) (using q: Quotes):
  Expr[Unit] =
    import q.reflect._

    val exprs: Seq[Expr[String]] = args match
      case Varargs(es) =>
        es.map { e =>
          e.asTerm match
            case Literal(c: Constant) => Expr(c.value.toString)
```

```

    case _ => '{$Expr(e.show)} + " = " + $e}
}

'{println(${exprs.reduce((e1, e2) => '{$e1 + ", " + $e2}))})

```

As before, the `debugN` function calls the `debugNImpl` macro, passing the quoted arguments, now as an `Expr[Seq[Any]]`. And as before, we need to use a given `Quotes` instance.

Now some more knowledge of Scala internals is required. The `args` parameter is actually an instance of `Varargs`, which we destructure to get its elements. Each element is either a literal, or an expression for which both the description and value are printed.

In the first case, we produce an `Expr` with the string representation of the literal. In the second case, we concatenate the expression's string representation `e.show` and the string `" = "` at compile time, and produce an expression to concatenate with the expression's value at runtime.

The call to `reduce` concatenates the values of the expressions. Note the nested quote and splice to compute the concatenation of `e1` and `e2`. It is a good idea to look at the generated byte codes to understand the intricacies of the macro expansion (see [Exercise 27](#) on page 341).

As a final example, let us expand on the type checking that you saw in [Section 20.3, “Literal Type Arithmetic,”](#) on page 322. We were able to check at compile time that two vectors have the same length, or that the length is odd. Suppose we want to check strings instead. For example, we might want to verify at compile time that a string literal contains only digits before parsing it as an integer.

Such a check requires invoking a method such as `String.matches` at compile time, which is not possible with an `inline` function. Again, a macro is required.

Specifically, we will define a type `StringMatching[regex]` for literal strings that match a regular expression, also specified as a literal string. For example, the declaration

```
val decimal: StringMatching["[0-9]+"] = "1729"
```

will succeed, but

```
val octal: StringMatching["[0-7]+"] = "1729"
```

will not compile.

Here is the `StringMatching` type:

```
opaque type StringMatching[R <: String & Singleton] = String
```

In the companion object, we define an implicit conversion:

```
inline given string2StringMatching[S <: String & Singleton, R <:  
String & Singleton]:  
    Conversion[S, StringMatching[R]] = str =>  
        inline val regex = constValue[R]  
        inline if matches(str, regex)  
            then str.asInstanceOf[StringMatching[R]]  
            else error(str + " does not match " + regex)
```

The `matches` call is a macro:

```
inline def matches(inline str: String, inline regex: String):  
Boolean =  
    ${matchesImpl('str, 'regex)}  
  
def matchesImpl(str: Expr[String], regex: Expr[String])(using  
Quotes): Expr[Boolean] =  
    Expr(str.valueOrAbort.matches(regex.valueOrAbort))
```

It would have been nicer to use quotes and splices '``${str.matches($regex)}`', but that does not inline. Therefore, the `matchesImpl` method manually converts the expression trees to values, invokes `String.matches`, and produces an `Expr` of the Boolean result.

This chapter has given you an overview of advanced techniques that make some of the most advanced Scala libraries possible. You have now reached the end of this book. I hope that you found it useful as a rapid introduction

into all aspects of the Scala language, and that you will find it equally effective as a reference in the future.

Exercises

1. Using a recursive match type and union types, define a type `Interval[MIN <: Int, MAX <: Int] <: Int` that describes an interval of integers, with membership checked at compile time. For example, a variable of type `Interval[1, 10]` can be set to any constant integer between 1 and 10, and not to any other value.
2. Define an opaque type `IntMod[N]` for integers modulo `N`, with operators `+, -, *` that only combine integers with the same modulus.
3. Define a `zip` function that takes two `HList` of the same length and yields an `HList` of corresponding pairs. You will also need a recursive match type `Zip`.
4. Look at the bytecodes that are generated when you call the `run` function of [Section 20.4, “Inline Code,”](#) on page 323. Compile the `Section5` class from the companion code with the Scala compiler, then run `javap -c -private` on the class file that is called from the `@main` method. How can you tell that the code of the `run` function was inlined?
5. Verify that the code in the Caution note of [Section 20.4, “Inline Code,”](#) on page 323 inlines the code for the `task` parameter twice.
6. How does the `codeOf(task)` string appear in the bytecodes that are generated when you call the `run` function of [Section 20.4, “Inline Code,”](#) on page 323? Try invoking `run` with two different tasks.
7. Explain why `Ordering` is a type class and why `Ordered` is not.
8. Generalize the `average` method in [Section 20.5, “Type Classes,”](#) on page 326 to a `Seq[T]`.
9. Make `String` a member of the `Averageable` type class in [Section 20.5, “Type Classes,”](#) on page 326. The `divBy` method should retain every `n`th letter, so that `average("Hello", "World")` becomes `"Hlool"`.
10. Why can’t you implement the `debugN` function from [Section 20.10, “A Brief Introduction into Macros,”](#) on page 335 as an `inline` function?

Could you do it if you pass the arguments as a tuple, for example `debugN((x, y + z))`?

11. Write a recursive inline function that computes the sum of the elements of a `Vec` from [Section 20.3, “Literal Type Arithmetic,”](#) on page 322.
12. Write a recursive inline function that yields the element with index `n` of a `HList` with return type `Any`.
13. Improve upon the preceding exercise with an accurate return type. For example, `elem("Fred", 42), 1)` should have type `Int`. Hint: Use a recursive match type `Elem[T, N]`. You may still need a cast in the implementation. If you are stuck, look into the source code of the Scala `Tuple` class.
14. Create a fluent API for reading integers, floating-point numbers, and strings from the console. For example: Read in a `String` askingFor "Your name" and an `Int` askingFor "Your age" and a `Double` askingFor "Your weight". Return a `Tuple` of the appropriate type, such as `(String, Int, Double)` in this example.
15. Rewrite the `average` function in [Section 20.5, “Type Classes,”](#) on page 326 with the `using` syntax.
16. Rewrite the `average` function in [Section 20.5, “Type Classes,”](#) on page 326 to compute the average of any number of arguments.
17. Define a `Fraction` class and make it an instance of the standard `Fractional` type class.
18. Define a type class `Randomized` for producing a random instance of a type. Provide given instances for `Int`, `Double`, `BigDecimal`, and `String`.
19. Implement an inline function that makes a binary or a linear search through an array, depending on whether the component type admits an ordering.
20. Change the `summonGivenJSON` function in [Section 20.6, “Mirrors,”](#) on page 328 so that it returns a `List[JSON[Any]]` instead of a `Tuple`. Can you then avoid invoking `asInstanceOf` in `jsonProduct`?

21. The automatic derivation in [Section 20.6, “Mirrors,”](#) on page 328 does not work for the `Person` class. How can you fix that?
22. Extend the `JSON` type class in [Section 20.6, “Mirrors,”](#) on page 328 so that it can handle `Boolean` values and arrays.
23. Add a `parse` method to the `JSON` type class in [Section 20.6, “Mirrors,”](#) on page 328 and implement parsing for case classes.
24. Confirm that the cast `asInstanceOf[Product]` can be avoided in the `jsonProduct` function of [Section 20.6, “Mirrors,”](#) on page 328 if one adds the very sensible bound `<: Product` to the type parameter. What problem does that cause in [Section 20.7, “Type Class Derivation,”](#) on page 330?
25. A (potentially infinite) set of elements of type `T` can be modelled as a function `T => Boolean` that is `true` whenever the argument is an element of the set. Declare a type `Set` in two ways, with a type parameter and with a type lambda.
26. The result of `"abc".map(_.toUpper)` is a `String`, but the result of `"abc".map(_.toInt)` is a `Vector`. Find out how that is implemented in the Scala library.
27. Write a program that uses the `debugN` macro of [Section 20.10, “A Brief Introduction into Macros,”](#) on page 335. (You need to put the macro and the code using it into separate source files.) Use `javap -c -private` to analyze the class file. Can you correlate the string concatenations with the quote and splice levels? What happened to the calls to `map` and `reduce`?