

PROFESSIONAL SCALA

INTRODUCTION	xv
CHAPTER 1 Language Features	1
CHAPTER 2 Functional Programming	19
CHAPTER 3 Java Compatibility	37
CHAPTER 4 Simple Build Tool	45
CHAPTER 5 Maven	63
CHAPTER 6 Scala Style/Lint	79
CHAPTER 7 Testing	85
CHAPTER 8 Documenting Your Code with Scaladoc	95
CHAPTER 9 Type System	139
CHAPTER 10 Advanced Functional Programming	165
CHAPTER 11 Concurrency	179
CHAPTER 12 Scala.js	205
INDEX	215

PROFESSIONAL **Scala**

Aliaksandr Bedrytski
Janek Bogucki
Alessandro Lacava
Matthew de Detrich
Benjamin Neil



Professional Scala

Published by
John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2016 by John Wiley & Sons, Inc., Indianapolis, Indiana

Published by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-1-119-26722-5
ISBN: 978-1-119-26725-6 (ebk)
ISBN: 978-1-119-26726-3 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permissions>.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at <http://book-support.wiley.com>. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2016937234

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

ABOUT THE AUTHORS

ALIAKSANDR BEDRYTSKI is a passionate software engineer at Worldline Lyon, France. Even though it's been more than four years since he discovered Scala, he's still in love with the language and is trying to push its boundaries every day. In his spare time he's learning something new about space, physics, and bleeding edge technologies. He is currently working as a lead developer on a data-analysis project featuring Hadoop, Spark-Scala, Hive, and other Big Data tools.

JANEK BOGUCKI is a co-founder at Inferess Inc. and principal consultant (Machine Learning and Scala) at Combination One. He has a background in mathematics with an ongoing interest in computer science, data science, machine learning, graph theory, and development methodologies. He lives in Kent, UK with his wife Rebecca, son Theo, two cats, one dog, and a variable number of chickens.

ALESSANDRO LACAVA holds a degree in telecommunications engineering. He's had extensive experience with OOP before becoming very passionate about functional programming. He is currently working as a lead designer and developer on different types of applications using mainly, but not only, Scala. He's also a contributor of pretty famous open source projects, such as shapeless and cats.

MATTHEW DE DETRICH is a tech lead for Monetise Pty Ltd, where he primarily works on full stack applications with backends written in Scala. Matthew has a passion for designing, building, and integrating complex multi-domain systems from the ground up. In his spare time he is exploring new developments in the Scala space, such as Scala.js.

BENJAMIN NEIL is a full stack engineer at AppThis LLC. He is a polyglot engineer who has had the privilege of working the past eight years making websites, services, and tools for amazing companies. He is obsessed with Scala, Vim, DevOps, and making services scale smoothly.

ABOUT THE TECHNICAL EDITORS

JACOB PARK is a Scala and Typesafe enthusiast; he embraces Akka, Play, and Spark, all with Scala, to solve various problems regarding distributed systems with additional technologies such as Cassandra and Kafka. As a believer in open-source software, he contributes to various projects on GitHub related to Akka and Cassandra: <https://github.com/jparkie>. When he's not working, Jacob can be found at various Scala and Typesafe conferences and meetups, either as an attendee or a presenter.

ARIEL SCARPINELLI is a senior Java developer at VirtualMind and is a passionate developer with more than 15 years of professional experience. He currently leads four agile teams for a U.S.-based enterprise SaaS company. He has experience in a lot of languages but is currently focused on Java and JavaScript with some PHP and Python.

RADU GANCEA is a software engineer and consultant at Eloquentix with a Java background (also C/C++). He is involved in projects in the energy and advertising sectors. His current focus is on Scala and other JVM languages and frameworks.

CREDITS

PROJECT EDITOR
Charlotte Kughen

TECHNICAL EDITORS
Jacob Park
Ariel Scarpinelli
Radu Gancea

PRODUCTION EDITOR
Barath Kumar Rajasekaran

COPY EDITOR
Troy Mott

**MANAGER OF CONTENT DEVELOPMENT
AND ASSEMBLY**
Mary Beth Wakefield

PRODUCTION MANAGER
Kathleen Wisor

MARKETING MANAGER
David Mayhew

**PROFESSIONAL TECHNOLOGY & STRATEGY
DIRECTOR**
Barry Pruett

BUSINESS MANAGER
Amy Knies

EXECUTIVE EDITOR
Jim Minatel

PROJECT COORDINATOR, COVER
Brent Savage

PROOFREADER
Nancy Bell

INDEXER
Nancy Guenther

COVER DESIGNER
Wiley

COVER IMAGE
©Sergey Nivens/Shutterstock

CONTENTS

INTRODUCTION	xv
CHAPTER 1: LANGUAGE FEATURES	1
Static Types and Type Inference	2
Implicit Parameters, Conversions, and Their Resolution	3
Case Class, Tuples, and Case Object	5
Abstract Class, Traits, and Sealed	6
Pattern Matching	8
Statements Are Expressions	9
String Interpolation	9
Scala Collections, immutable and mutable	10
For Comprehension	12
Packages, Companion Objects, Package Objects, and Scoping	13
AnyVal, AnyRef, Any, and the Type Hierarchy	16
Summary	17
CHAPTER 2: FUNCTIONAL PROGRAMMING	19
Immutability	20
Pure Functions	22
Recursion	23
Higher-Order Functions	26
Core Collection Methods	27
Methods Returning a Collection	29
Methods Returning a Value	31
Currying and Partially Applied Functions	32
Null Handling (Option)	34
Strict versus Non-Strict Initialization	35
Summary	36
CHAPTER 3: JAVA COMPATIBILITY	37
Scala and Java Collections	37
Interfaces and Traits	40
Scala/Java Enumerations	42
Summary	43

CHAPTER 4: SIMPLE BUILD TOOL	45
Basic Usage	46
Project Structure	47
Single Project	47
Scopes	49
Custom Tasks	50
Dependencies	50
Resolvers	51
Advanced Usage	52
Advanced Dependencies	53
Testing in the Console	55
Release Management	56
Deploying to Sonatype	56
Packaging with SBT-Native-Packager	58
Creating a Docker Image	59
Common SBT Commands	60
Useful Plugins	61
Summary	62
CHAPTER 5: MAVEN	63
Getting Started with Maven and Scala	64
Introducing scala-maven-plugin	67
Adding Library Dependencies	70
Using the REPL	71
Getting Help	72
Running Tests	72
Joint Compilation with Java	74
Accelerating Compilation with Zinc	76
Summary	77
CHAPTER 6: SCALA STYLE/LINT	79
Scala with Style	79
Scaliform	81
Scapegoat	82
WartRemover	82
Scoverage	84
Summary	84

CHAPTER 7: TESTING	85
ScalaTest	86
Unit Tests	87
Integration Testing	87
Data-Driven Tests	88
Performance Testing	89
Acceptance Testing	90
Mocks	92
Load Testing	93
Summary	94
CHAPTER 8: DOCUMENTING YOUR CODE WITH SCALADOC	95
Why Document Your Code?	96
Revealing the Benefits	96
Bookending the Continuum	96
Choosing What to Document	96
Scaladoc Structure	97
Overall Layout	97
Index Pane	98
Content Pane	100
Invoking the Scaladoc Tool	106
Wiki Syntax	108
Formatting with Inline Wiki Syntax	108
Structuring with Block Elements	110
Linking	113
Locating Scaladoc	117
Tagging	117
Everyday Tagging	117
Tagging for Groups	123
Advanced Tagging	125
Invoking scaladoc: Additional Options	132
Integrating Scaladoc Creation with Your Project	133
Configuring Maven	133
Configuring SBT	134
Publishing Scaladoc	134
Tables and CSS	136
Summary	138

CHAPTER 9: TYPE SYSTEM	139
What Is a Type System?	140
Static versus Dynamic Typing	140
What Static Type Systems Are Good For	141
What Dynamic Type Systems Are Good For	141
Scala's Unified Type System	141
Value Classes	143
Polymorphism	145
Subtype Polymorphism	145
Parametric Polymorphism	146
Ad Hoc Polymorphism	146
Bounds	149
Context Bounds	149
Upper and Lower Bounds	150
Variance	151
Other Niceties	155
Self-Type Annotations	155
Self-Recursive Types	158
Abstract Type Members	159
Dynamic Programming	161
Structural Types	161
Dynamic Trait	162
Summary	164
CHAPTER 10: ADVANCED FUNCTIONAL PROGRAMMING	165
Higher-Kinded Types	165
Functional Design Patterns	167
Functor	167
Applicative Functor	170
Monad	172
Semigroup	173
Monoid	174
Summary	176
CHAPTER 11: CONCURRENCY	179
Synchronize/Atomic Variables	181
Future Composition	184
Parallel Collections	187
Reactive Streams	192
STM	195

Actors (Akka)	198
Spark	200
Summary	202
CHAPTER 12: SCALA.JS	205
Scala.js and Its Design	205
Getting Started: Scala.js with SBT	206
Scala.js Peculiarities	210
Webjars and Dealing with the Frontend Ecosystem	211
Summary	213
INDEX	215

INTRODUCTION

A working knowledge of Scala puts you in demand. As both the language and applications expand, so do the opportunities for experienced Scala programmers—and many positions are going unfilled. Major enterprises across industries are using Scala every day, in a number of different applications and capacities. *Professional Scala* helps you update your skills quickly to start advancing your career.

Scala bridges the gap between functional and object-oriented programming, and this book details that link with a clear discussion of both Java compatibility and the read-eval-print loop used in declarative programming. You'll learn the details of Scala testing, design patterns, concurrency, and much more as you build the in-demand skill set required to utilize Scala in a real-world production environment.

WHO THIS BOOK IS FOR

This book is for experienced programmers who already have some understanding of the Scala language and are looking for more depth. You should have a basic working knowledge of the language because this book skips over the fundamentals of programming, and the discussion launches directly into practical Scala topics.

WHAT THIS BOOK COVERS

This book explains everything professional programmers need to start using Scala quickly and effectively.

- Link functional and object-oriented programming.
- Master syntax, the SBT interactive build tool, and the REPL workflow.
- Explore functional design patterns, the type system, concurrency, and testing.
- Work effectively with Maven, Scala.js, and more.

HOW THIS BOOK IS STRUCTURED

This book was written in three parts. The first part of the book discusses language structure including syntax, practical functional programming, and Java compatibility.

The second part of the book takes the reader through tooling, including discussions of SBT, Maven, lint tools, testing, and Scaladoc. The last portion of the book continues to examine Scala through advanced topics such as polymorphism, dynamic programming, concurrency, Scala.js, and more.

WHAT YOU NEED TO USE THIS BOOK

For this book, we have used Scala version 2.11.7. The source code for the samples is available for download from the Wrox website at www.wrox.com/go/professionalscala or from GitHub at <https://github.com/backstopmedia/scalabook>.

CONVENTIONS

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

NOTE *Notes indicate notes, tips, hints, tricks, and/or asides to the current discussion.*

As for styles in the text:

- We *highlight* new terms and important words when we introduce them.
- We show code within the text like so: `persistence.properties`.
- We show all code snippets in the book using this style:

```
FileSystem fs = FileSystem.get (URI.create(uri), conf);
InputStream in = null;
try {
```

- We show URLs in text like this:

```
http://<Slave Hostname>:50075
```

SOURCE CODE

As you work through the examples in this book, you may choose either to type in all the code manually, or to use the source code files that accompany the book. All the source code used in this book is available for download at www.wrox.com. Specifically for this book, the code download is on the Download Code tab at:

```
www.wrox.com/go/professionalscala
```

You can also search for the book at www.wrox.com by ISBN (the ISBN for this book is 9781119267225 to find the code. And a complete list of code downloads for all current Wrox books is available at www.wrox.com/dynamic/books/download.aspx.

NOTE *Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 978-1-119-26722-5*

Once you download the code, just decompress it with your favorite compression tool. Alternatively, you can go to the main Wrox code download page at www.wrox.com/dynamic/books/download.aspx to see the code available for this book and all other Wrox books.

ERRATA

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, like a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata, you may save another reader hours of frustration, and at the same time, you will be helping us provide even higher quality information.

To find the errata page for this book, go to

www.wrox.com/go/professionalscala

and click the Errata link. On this page you can view all errata that has been submitted for this book and posted by Wrox editors.

If you don't spot "your" error on the Book Errata page, go to www.wrox.com/contact/techsupport.shtml and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

P2P.WROX.COM

For author and peer discussion, join the P2P forums at <http://p2p.wrox.com>. The forums are a web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At <http://p2p.wrox.com>, you will find a number of different forums that will help you, not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to <http://p2p.wrox.com> and click the Register link.
2. Read the terms of use and click Agree.

3. Complete the required information to join, as well as any optional information you wish to provide, and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

NOTE *You can read messages in the forums without joining P2P, but in order to post your own messages, you must join.*

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to This Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works, as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

1

Language Features

WHAT'S IN THIS CHAPTER?

- Outlining various control structures in Scala
- Using various tools in the standard library
- Mastering inheritance and composition in Scala

Scala borrows many of its syntax and control structures from various other languages. Methods are declared in Algol/C style with the addition of features such as optional braces, and semicolons to reduce code boilerplate.

Scala syntax also has features such as expressions for statements, local type inference, suffix/infix notation, and implicit return statements, which make typical idiomatic Scala code look like dynamic languages such as Lisp/Python/Ruby, but with the added benefit of strong static typing.

The expressive nature of Scala's syntax is useful to create custom DSL's, with the most notable example being Scala's own XML library. Scala also features several forms of inheritance (traits, classes, objects), which can be combined in various ways to structure composition in your code in a more natural and elegant way. This combined with a strong support for functional programming, syntactic sugar (for comprehension), and type inference makes it possible to create very terse domain specific code that is also type safe.

This chapter provides a broad overview of the various parts of Scala's design and syntax, as well as the features that are expected in modern generic programming languages (control of scoping, string interpolation, encapsulation, and modules).

STATIC TYPES AND TYPE INFERENCE

Scala is first and foremost a statically typed language, which is similar to other statically typed languages you can annotate types for:

- Variable declarations
- Method/function arguments
- Method/function return values
- Various types of data structures

Below is a code sample to demonstrate these types of annotations:

```
val s: String // type definition on variable declaration
def doSomething(s: String) // type definition in a parameter
def doSomething: String // type definition for return type of a function
class SomeClass(s: String) // type definition in a class constructor definition
type MyString = String // type aliasing
```

A significant reason why Scala requires mandatory type signatures is that it's very difficult to provide a practical implementation of global type inference in a statically typed language that supports subtyping. Thankfully, while Scala mandates that you provide type signatures for method parameters and type definitions, for variable declarations inside function/method bodies it features local type inference. This drastically reduces the amount of boilerplate code when it comes to function/method bodies. Here is an example of local type inference in action.

```
def doSomething(input: String) = {
  val l = List(1,3,5)
  val summed = l.sum
  val asString = summed.toString + input
  asString.toUpperCase
}
```

As you can see for variable declarations, you don't need to specify their respective types. To manually specify the types, do the following:

```
def doSomething(input: String): String = {
  val l: List[Int] = List(1,3,5)
  val summed: Int = l.sum
  val asString: String = summed.toString + input
  asString.toUpperCase()
}
```

One bonus of specifying types for function arguments is that they also act as a form of documentation, which is particularly helpful. Return types of functions are also optional; however, it is encouraged to annotate them for documentation/clarity reasons. Due to Scala's generic code functionality, it is possible to have expressions that return different values in unexpected scenarios. One good example is `Int`'s versus `Double`:

```
def f = 342342.43

def f_v2 = 342342
```

```
f == 342342 // returns true

f_v2 == 342342 // return false
```

Since `f` and `f_v2` don't have their return types annotated, we aren't aware of what types they can return (let's assume you don't know what the actual values of `f` and `f_v2` are). This means that the `f == 342342` expression returns something else compared to `f_v2 == 342342`. If, however, you know the return types of `f` and `f_v2`, you can do the following:

```
def f: Double = 342342.43

def f_v2: Int = 342342
```

As a user, you know that the return type of `f` is a `Double`, so don't compare it with an `Int` when asking for equality (since it's a `Float` by extension you should provide custom equality with a `delta`).

Manual type signatures can also be used to force the type of a declaration. This is similar to the `typecast` feature in languages like Java or C; however, it's more powerful due to type conversions available in Scala. A simple example in Scala is to assign a `Long` type to an integer literal.

```
val l: Long = 45458
```

By default, if you use a number literal in Scala it defaults to an `Int`. Note that it is also possible to use the number based suffix like so:

```
val l = 45458L
```

Arguably, the type annotation is more idiomatic, especially for custom types that may not have literal operators.

Implicit Parameters, Conversions, and Their Resolution

Scala has an incredibly powerful feature called `implicit`s. When you ask for a value implicitly, such as a method parameter, or the `implicit` keyword you are telling the Scala compiler to look for a value that has the same type in scope. This is unlike the explicit value, where you need a type to specify when it's called/used. A practical use case for `implicit`s is providing an API key for a REST web service. The API key is typically only provided once, so an `implicit` is a good use case in this instance, since you only need to define and import it once.

```
case class ApiKey(id: String)
```

Since `implicit`s are found by their type you need to create a new type, in this case `ApiKey`. Now let's define a method that makes a HTTP call.

```
import scala.concurrent.Future

def getUser(userId: Long)(implicit apiKey: ApiKey): Future[Option[User]] = ???
```

In order to call this function you must make sure that an `implicit ApiKey` is visible in the scope. Let's first make an instance of `ApiKey` and place it in an object.

```
object Config {  
    implicit val apiKey = ApiKey(System.getenv("apiKey"))  
}
```

Whenever you call `getUser`, `ApiKey` needs to be visible, and the code should look something like this:

```
import Config._ // This imports everything in Config, including implicits  
  
object Main extends App {  
    val userId = args(0).toLong  
    getUser(userId)  
}
```

It's also possible to supply an implicit argument explicitly.

```
object Main extends App {  
    val userId = args(0).toLong  
    val apiKey = ApiKey(args(1))  
    getUser(userId)(apiKey)  
}
```

Implicits can be used to make automatic conversions between types; however, such usage is considered advanced and should be done sparingly, because it's easy to abuse code that uses these features. A prominent use of implicit conversions is in DSLs. For example, if you have a trivial DSL that attempts to represent SQL queries, you might have something like this:

```
Users.filter(_.firstName is "Bob")
```

In this case, `_.firstName` may be a different type from a `String` (in this example, let's say it has type `Column[String]`). And it can only work on types of `Column[String]`:

```
val columnStringImplementation = new Column[String] {  
    def is(other: Column[String]): EqualsComparison = ???  
}
```

To compare two instances of `Column[String]` (to see if they are equal) you want an implicit conversion of `String` to `Column[String]`, so that you don't have to write:

```
Users.filter(_.firstName is ColumnString("Bob"))
```

To do this you need to define the following:

```
implicit def stringToColumnStringConverter(s: String): Column[String] = ???
```

In this case, when you import your `stringToColumnStringConverter`, it will automatically convert any instances of `String` to `Column[String]` if it's required to do so. Since the `is` method only works on `Column[String]`, the compiler will try to see if there is an implicit conversion available for the `String` "BoB," so it can satisfy the method signature for `is`.

Implicit classes are a type safe solution that Scala provides for monkey patching (extending an existing class, or in Scala's case, a type without having to modify the existing source of the type). Let's try this by adding a `replaceSpaceWithUnderScore` method to a `String`.

```
object Implicits {
  implicit class StringExtensionMethods(string: String) {
    def replaceSpaceWithUnderScore = string.replaceAll(" ", "_")
  }
}
```

Like the previous example, you need to import this class to make it available:

```
import Implicits._
"test this string".replaceSpaceWithUnderScore // returns test_this_string
```

Case Class, Tuples, and Case Object

Scala has a feature called case class, which is a fundamental structure to represent an immutable data. A case class contains various helper methods, which provide automatic accessors, and methods such as `copy` (return new instances of the case class with modified values), as well as implementations of `.hashCode` and `equals`.

```
case class User(id: Int, firstName: String, lastName: String)

User(1, "Bob", "Elvin").copy(lastName = "Jane") // returns User(1, "Bob", "Jane")
```

These combined features allow you to compare case classes directly. Case classes also provide a generic `.toString` method to pretty print the constructor values.

```
User(1, "Bob", "Elvin").toString // returns "User(1,Bob,Elvin)"
```

Scala also has tuples, which work similarly to Python’s tuples. A nice way to think of tuples is a case class that has its fields defined positionally (i.e., without its field name), and there is no restriction on a tuple requiring the same type for all of its elements. You can construct tuples by surrounding an expression with an extra set of parenthesis:

```
val userData = (5, "Maria", "Florence") //This will have type Tuple3
[Int, String, String]
```

You can then convert between tuples and case classes easily, as long as the types and positions match:

```
val userData2: Tuple3[Int, String, String] = User.unapply(User(15, "Shelly",
"Sherberty")).get
val constructedUserData: User = User.tupled.apply(userData2) // Will be
User(15, "Shelly", "Sherberty").
```

Tuples are a handy way to store grouped data of multiple types, and they can be deconstructed using the following notation:

```
val (id, firstName, lastName) = userData2
```

Note that tuples also provide an `_index` (where index is the position) and methods to access the elements of the tuple by position (in the example above, `_2` would return “Shelly”). We recommend that you use the deconstruction mentioned above, since it’s clearer what the field is meant to be.

For internal and local use, tuples can be quite convenient due to not needing to define something like a case class to store data. However, if you find yourself constantly structuring/destructuring tuples, and/or if you have methods returning tuples, it’s usually a good idea to use a case class instead.

Abstract Class, Traits, and Sealed

In the previous chapter we discussed case classes. Apart from immutability (and other benefits), one of the primary uses of case classes is for creation of ADTs (known as Algebraic Data Types). Algebraic data types allow you to structure complex data in a way that is easy to decompose, and it is also known as the visitor pattern in Java. Here is an example of how you can model a video store:

```
sealed abstract class Hemisphere
case object North extends Hemisphere
case object South extends Hemisphere

sealed abstract class Continent(name: String, hemisphere: Hemisphere)

case object NorthAmerica extends Continent("North America", North)
case object SouthAmerica extends Continent("South America", South)
case object Europe extends Continent("Europe", North)
case object Asia extends Continent("Asia", North)
case object Africa extends Continent("Africa", South)
case object Australia extends Continent("Australia", South)
```

You can then easily use Pattern Matching (explained in greater detail in the next section) to extract data, and doing so looks similar to this:

```
val continent: Continent = NorthAmerica
continent match {
  case Asia => println("Found asia")
  case _ =>
}
```

You may also notice the keyword “sealed.” Sealed in Scala means that it’s not possible for a class/case class/object outside of the file to extend what is sealed. In the example above, if `Continent` was defined in a file called `Continent.scala`, and in `Main.scala` you tried to do the following, Scala produces a compile error.

```
case object Unknown extends Continent
```

The advantage of `sealed` is that since the compiler knows all of the possible cases of a class/trait/object being extended, it’s possible to produce warnings when pattern matching against the cases of the sealed abstract class. This also applies to Traits, not just sealed abstract classes.

Traits that allow Scala to implement the mixin pattern are another one of Scala’s most powerful features. Traits, at a fundamental level, allow you to specify a body of code that other classes/traits/objects can extend. The only real limitation is that Traits can’t have a primary constructor (this is to avoid the diamond problem). You can think of them as interfaces, except that you can provide definition for certain methods that you wish (and you can extend as many traits as you want).

A good use case for a trait is in a web framework, where traits are often used as a composition model for routes. Suppose you have trait defined as the following:

```
trait LoginSupport {self: Controller =>
  lazy val database: Database
  def login(userId) = {
```

```

        //Code dealing with logging in goes here
        afterLogin()
    }
    def logout() = {
        //Code dealing with logging out goes here
    }
    def afterLogin: Unit = {
    }
}

```

Here a trait is defined that allows you to mixin login/logout functionality, but there are a few interest points here. The first is the `self: Controller =>`. This dictates that only a class of type `Controller` can extend the `LoginSupport` trait. Furthermore, there is a reference to this type through the `self` variable. This is actually an example of inversion of control, which is a basic technique needed to do simple DI (Dependency Injection) in Scala. This form of DI is statically checked, and doesn't need any dependency on external libraries, or features like macros.

The next important thing to note is the `val database: Database`. It's up to the class/trait/object extending `LoginSupport` to provide an implementation of database, which is ideally what you want (the `LoginSupport` trait shouldn't need to know how the database is being instantiated).

`def login` and `def logout` provide implementations for login and logout respectively. You then have an implementation of `afterLogin()`, which is called after you `login()` a user. Now let's assume you have a controller, or something like:

```

class ApplicationController(implicit val database: Database) extends LoginSupport
{self:
Controller =>
}

```

Inside this class, you now have access to the `login`, `logout`, and `afterLogin` methods. Doing just the above however, would generate a compile error, since you haven't defined the database. Using an assumption that the database is being passed into the `ApplicationController` (let's say implicitly), you can do this:

```

class ApplicationController(implicit val database: Database) extends LoginSupport
{
}

```

You can provide an instance of the database as a constructor for `ApplicationController`, and it will be used by the `LoginSupport`. Alternatively, you can do this:

```

class ApplicationController2 extends LoginSupport {self: Controller =>
lazy val database: Database = Config.getDb
}

```

Similar to extending in Java classes, you can actually override one of these methods. In this case you can execute something after the user logins:

```

class ApplicationController3(implicit val database: Database) extends
LoginSupport{self:
Controller =>
override def afterLogin() = Logger.info("You have successfully logged in!")
}

```

As you can see, traits are incredibly flexible. They allow you to cleanly split responsibility (i.e., what is implemented by the trait versus what is needed, but not implemented).

PATTERN MATCHING

Pattern matching is one of the most used features in the Scala language. It provides a Swiss Army knife of capabilities that are the most common ways of inspecting and dealing with object data.

Pattern matching has the ability to deal with the following:

- **Equality comparison.** Similar to the `switch` statement, pattern matching allows you to branch out different executions, which depend on the value of a variable. Scala's pattern matching, however, has far better capabilities than just equality checking. It can be combined with `if` statements to allow for more fine-grained branching, and it's also the primary way to deconstruct ADTs (Abstract Data Types).
- **Typesafe forced casting.** Pattern matching can be used to do safe type casting. In Java, manual checks with `instanceOf` need to be made to ensure that safe cast is being applied. Pattern matching allows you to do this pattern in a safe way.
- **Destructive assignment of case classes and types that provide Unapply.** Destructive assignment, loosely speaking, is the ability to inspect the contents of a type and act on them as they are being deconstructed. This is also used to deal with `Option` in a type safe way and can aid construction of immutable data structures such as a `List` (reference link).

The beautiful thing about pattern matching is that all of the above capabilities can be combined to deconstruct complex business logic, as this example shows:

```
sealed abstract class SomeRepresentation
case class NumberRepr(number: Double) extends SomeRepresentation
case class StringRepr(s: String) extends SomeRepresentation
val s: SomeRepresentation = NumberRepr(998534)
s match {
  case NumberRepr(n) if n > 10 => println("number is greater than 10, number is $n")
  case NumberRepr(n) => println("number is not greater than 10, number is $n")
  case StringRepr(s) => println("is a string, value is $s")
}
```

This example demonstrates switching on an ADT, destructuring the values inside the ADT, and also matching on those destructured values. Pattern matching has the wildcard operator (`_`) which is the fallback if all other options are exhausted (in Java this is `default`). A good example of this is an actor:

```
case class StringMessage(s: String)
case class IntMessage(i: Int)
class SomeActor extends Actor {
  def receive = {
    case StringMessage(s) => println("message received, it's a string of value $s")
    case IntMessage(i) => println("message received, its an int of value $i")
    case _ => println("unknown message")
  }
}
```

Another use of pattern match is in the construction of partial functions, which is particularly powerful when combined with methods like `collect`:

```
val l: List[Any] // List of values/references of any possible type
l.collect{
  case i: Int if i % 2 == 0 => i
  case s: String => s.length
}
```

In the case above, `collect` will grab all items from the list only if they happen to be an even number of type `Int` OR if the item is of type `String`, then the length of the `String` is returned. Pattern matching also provides a typesafe way of dealing with `Option` values.

```
val param: Option[String]
param match {
  case Some(s) => "parameter found, its value is $s"
  case None => "no parameter is found"
}
```

Statements Are Expressions

In Scala, any statement is also an expression. This is particularly powerful, since it represents a consistent way of working with branching of expressions (especially when combined with pattern matching as described before).

```
import scala.concurrent.Future
val parameter: Option[String] = ???
val httpCall: Future[String] = parameter match {
  case Some(s) => Http.get("/api/doSomething/$s")
  case None => Future("No parameter supplied")
}
```

The important thing to note here is that you are making sure that the return result of the statement is of type `Future` (so you can reuse this as a proper value later on).

NOTE *Scala uses the least upper bound in determining what the return type of a statement is. This means that it's possible to get the returning type of an expression to be `Any`, `AnyVal` or `AnyRef`. For example, if you have an `if` expression that has an evaluation of one branch to be a `String`, and another `Int`, the resulting type will be `Any`.*

String Interpolation

String interpolation provides a performant, concise, and typesafe way to print out values in the middle of string literals. The basic way to use the `$` is to print the value of a variable:

```
s"the value of this variable is $s"
```


It's also possible to use block syntax so you can interpolate expressions, and not just values:

```
log.info(s"The id of the current user is ${user.id}")
```

Scala Collections, immutable and mutable

In the previous example, you can see the usage of methods like `.map` and `.flatMap`. These are actually fundamental patterns in functional programming, and they are used frequently in Scala as part of its collection framework. Scala collections are one of the few languages that provide a generic, strongly/statically typed framework for collections that also deal with type transformations. Let's start off with one of the most fundamental types in Scala, the `List`.

A `List` in Scala represents an immutable `List`, whereas a `Seq` represents an arbitrary sequence (can be mutable or immutable). Below is an example of how to use `list`:

```
val l = List(3,6,2342,8)
```

The type of this expression is `List[Int]`. One thing to note is that you can use the `Seq` constructor like so:

```
val s = Seq(3,6,2342,15)
```

`Seq`'s default constructor is an immutable `List`, so in fact these expressions are equal. However, as stated before, a `Seq` can represent any sequence whereas a `List` is much more strict.

```
def listLength(l: List[_]) = l.length
def sequenceLength(s: Seq[_]) = s.length
```

It is perfectly legitimate to pass any sequence into `sequenceLength` (this can mean a `Vector`, a mutable list, or even a `String`), however, if you try to pass these types in `listLength`, you get a compile error. This is very powerful, as it allows you to specify the required granularity that is needed for collections, but it can also be dangerous. Consuming a list, while doing the map that is mutable versus immutable, can have unintended consequences.

`.map` allows you to apply an anonymous function to every element in a collection, with the item being replaced with the returning value of that anonymous function.

```
val l = List(3,6,2342,8)
l.map(int => int.toString) // Result is List("3","6","2342","8")
```

The example above converts every number to a `String`, which shows you how Scala on a simple level deals with converting types from within a collection (in this case `Int -> String`). One of the great things about the Scala collection library is that it provides implicit conversions that allow you to do complex transformations.

```
val m = Map(
  "3" -> "Bob",
  "6" -> "Alice",
  "10" -> "Fred",
  "15" -> "Yuki"
)
```

This Map will have the type `Map[String, String]`. However, let's say that you want to convert the keys to a `String`, while at the same time splitting out the names into its characters (i.e. `Array[Char]`).

A first naive solution to the problem can be:

```
val keys = m.keys.map{key => key.toInt}.toList
val values = m.values.map{name => name.toCharArray}
(keys zip values).toMap
```

As you can see, this attempt is quite wordy, and it's also inefficient. You have to manually get the keys/values out of the map and transform them. Then you have to manually zip the keys with the values and convert them to a map. A more idiomatic solution is:

```
m.map{case (key,name) => key.toInt -> name.toCharArray }
```

This version is definitely more readable and succinct than the former, but you may be wondering how this works behind the scenes. Below is the definition of `map` from the Scala collections library.

```
def map[B, That](f: A => B)(implicit bf: CanBuildFrom[Repr, B, That]): That
```

While this looks quite complex, the thing to note here is the `implicit bf: CanBuildFrom[Repr, B, That]`. This brings an implicit `CanBuildFrom`, which allows the Scala collection library to transform between different types to produce the desired results.

In this case, when you call the `.map` on the `Map[String]`, the argument to the anonymous function is actually a tuple (which you deconstruct with the `case` statement).

```
m.map{x => ...} // x is of type Tuple2[String,String], with the first String being
                // a key, and the second value
m.map{case (key,value) => ... } // Deconstruct the Tuple2[String,String]
                                // immediately. key is of type String, as is value
```

If you return a tuple in the anonymous function that you place in `map`, `CanBuildFrom` will handle the conversion of `Tuple2` as a key-value pair for the `Map` you originally operated off. The `->` syntax you noticed earlier is actually an alias to construct a tuple:

```
implicit final class ArrowAssoc[A](private val self: A) extends AnyVal {
  @inline def -> [B](y: B): Tuple2[A, B] = Tuple2(self, y)
  def →[B](y: B): Tuple2[A, B] = ->(y)
}
```

In other words, `key.toInt -> name.toCharArray` is the same as `(key.toInt, name.toCharArray)`. This means the return type of `{case (key,name) => key.toInt -> name.toCharArray }` is `Tuple2[Int, Char[Array]]`, so all that happens is you go from `Map[String, String]` to `Map[Int, Array[Char]]` in the final expression.

You may ask at this point, what would happen if you return a completely different type, such as if you return an `Int` instead of a `Tuple2[A, B]` (where `A` and `B` are arbitrary types). Well, it so happens that the map has its `(key,value)` entry replaced with just a value, so you end up converting from a `Map` to an `Iterable`.

This means that `m.map{case (_,value) => value }` is the same as `m.values`, and `m.map{case (key,_) => key }` is the same as `m.keys`. A lot of these powerful transformations are possible due to `CanBuildFrom`, so you can implement equivalent functionality in other strongly, statically typed languages. This needs to be implemented into the actual compiler (in Scala, the collection is a standard Scala library that is included by default in the Scala distribution).

For Comprehension

You may have noticed earlier that when you composed `map`'s and `flatMap`'s, you end up with undesirable nesting (i.e., a horizontal pyramid). `for` comprehension is a very powerful feature to deal with such issues.

```
import scala.util._

def firstTry: Try[String] = Success { "first response" }
def secondTry(string: String): Try[Int] = Failure { new
  IllegalArgumentException(s"Invalid length ${string.length}")
}
def finalTry(int: Int): Try[String] = Success { int.toString }

firstTry.map{result =>
  s"value from firstResult: $result"
}.flatMap{anotherResult =>
  secondTry(anotherResult).map{finalResult =>
    finalTry(finalResult).map(_._toUpperCase)
  }
}
```

The equivalent `for` comprehension for this statement would be:

```
for {
  result <- firstTry
  anotherResult = s"value from firstResult: $result"
  secondResult <- secondTry(anotherResult)
  finalResult <- finalTry(secondResult)
} yield finalResult.uppercase
```

The `for` comprehension has completely flattened out the above comprehension, making the intention very clear. `for` comprehension also provides syntactic sugar over `filter`/`withFilter`. A good example of this is shown using the `Range` class:

```
for (i <- 1 to 10000 if i % 2 == 0 ) yield i
```

This provides you with all of the even numbers up to 10,000. The equivalent without the `for` comprehension is:

```
(1 to 10000).withFilter(i => i % 2 == 0)
```

PACKAGES, COMPANION OBJECTS, PACKAGE OBJECTS, AND SCOPING

Another fantastic feature of Scala is how it provides access controls and encapsulation in every vertical of the language. This goes from controlling scoping on the package level to controlling scoping down to a module level.

At the highest level, Scala provides packages that work the same way they do in Java. In essence, packages are a purely static construct, which the compiler can use to group a collection of `.scala` source files. Packages can't be referenced in runtime apart from providing them as an import. So, packages are often used to collect source files on a very high level. As an example, the Scala Futures are contained within `scala.concurrent`, with `scala.concurrent` being the package.

The benefit of packages is that they are purely static, and packages can be combined with completely separate dependencies as long as there are no conflicts in naming. As an example, if you want to make your own version of `Future` (let's call it `ImprovedFuture`), you can do the following:

```
package scala.concurrent

trait ImprovedFuture {
  // Implementation goes here
}
```

If you package this as a dependency, and include it in one of your projects, you can import both the standard `Future`, our `ImprovedFuture`, and everything else under the namespace `scala.concurrent`.

The other namespacing utility that Scala has is called objects. In contrast to packages, objects have an actual runtime type representation. Another common name for objects is singletons. A singleton is a class that can only ever have one global instance, which is also automatically instantiated.

```
object MyObject {
  // Implementation goes here
}
```

`MyObject` has a type called `MyObject$` at runtime. In actual Scala code, this type can also be retrieved by calling the `.type` method (i.e. `MyObject.type`). This pattern essentially provides “dynamic” modules, or in other words, allows you to deal with packages within your own code. For example, you can make a function that accepts `MyObject` and does something with it:

```
def doSomething(myObject: MyObject)
```

A more realistic example is to create a module that a function can work with. As an example, let's create a trait that defines logging:

```
trait LoggerImplementation {
  def publishLog(level:String, message: String)
}
```

Now since this is just a trait, you need some way to instantiate it, so let's create a `MyLogger` object:

```
object ConsoleLogger extends LoggerImplementation {
  def publishLog(level: String, message: String) = {
    println(s"level: $level, message: $message")
  }

  object Implicits {
    implicit lazy val consoleLogger = ConsoleLogger
  }
}
```

This provides an interface, and you have also provided an implementation as an object. Since it is an object, you don't have to worry about instantiating `ConsoleLogger`, you can just import it. Now let's define a logger trait:

```
trait Logger {
  def log(level: String, message: String)(implicit loggerImplementation:
    LoggerImplementation) = {
    loggerImplementation.publishLog(level, message)
  }
}
```

Now let's create a basic `Main` function to test out logger:

```
import ConsoleLogger.Implicits._

object Main extends App with Logger {
  log("info", "This is a log statement")
}
```

In this case, `ConsoleLogger` represents a module (i.e., an implementation of a `Logger`). If you remove the `ConsoleLogger.Implicits._` import, you get a compile error saying something similar to the following:

```
error: could not find implicit value for parameter loggerImplementation:
  this.LoggerImplementation
```

There is also a notion of companion objects, which is an object that also references a class of the same name and path. In this sense, companion objects work the exact same way that normal objects do, but the difference is that a class can access the private members of a companion object directly.

```
object MyClass {
  private val statement = "This is a statement"
}

case class MyClass (additionalStatement: String) {
  def printStatement = println(s"$additionalStatement ${MyClass.statement}")
}
```

As with sealed traits, companion objects have to be defined in the same file that their companion class is defined in. Factory and instantiation related code is commonly placed within companion objects.

NOTE *When using the Scala REPL, you need to define the companion and its object within another enclosing object. The REPL is unable to properly determine a companion object if it's evaluated separately and without an enclosing object.*

Scala also offers another feature called package objects. Package objects are similar to packages described earlier; however, they don't have limitations regarding structures, which normally can't be top level (such as implicit classes, implicit conversions, or type aliases). Normally, you would have to place these constructs inside an object and then manually import them. As an example, you may have some type alias like:

```
package mypackage
object InternalTypeAliases {
  type Number = BigDecimal
  type Price = BigDecimal
}
```

To use these types, you then have to explicitly import the `InternalTypeAliases`. An alternative is to use a package object, and you can define it like so.

```
package mypackage
package object internalTypeAliases {
  type Number = BigDecimal
  type Price = BigDecimal
}
```

Now, anything inside of `mypackage` automatically has reference to the content defined inside of `internalTypeAliases`. This feature also means that package objects are commonly used inside a package, such as `util` functions, implicit conversions, and factory methods without having to deal with massive lists of import statements.

Scala, also similar to Java, provides various access control modifiers that allow designers to restrict how code is accessed (the only real exception is that in Scala, everything is public unless specified otherwise). You may have noticed earlier the keyword `private`, which essentially means that the value can only be accessed within the object itself (and a companion object if it's defined). The other access modifier is `protected`, which means the value can only be accessed within the class and any of its subclasses.

```
object Internals {
  private val privateInt = 1 //Can only be accessed within Internals, or
    a companion case class Internals if it's defined
  protected val protectedInt = 5 //Can only be accessed within Internals, or
    a companion case class Internals, or any subclass of Internals
}
object MoreInternals extends Internals {
  val publicInt = protectedInt + 10 // Is public
  val publicInt2 = privateInt + 5 // Does not compile
```

```
}  
  
object Main {  
  print(Internals.privateInt) // Does not compile  
  print(MoreInternals.publicInt)  
}
```

The usage of access modifiers is important in library and API design, and they are an important principle in encapsulation. Combined with Scala's already powerful features regarding modules, it's possible to provide both extensibility and restriction as desired.

AnyVal, AnyRef, Any, and the Type Hierarchy

Unlike languages like Java and C, Scala makes a specific distinction between value's (also known as primitive types) and references in the type system generically. This allows you to specify your own custom values (in a limited fashion).

NOTE *Scala also has the `Object` type, which is a type alias for `AnyRef` that is carried over from Java for compatibility reasons.*

An `AnyVal` in Scala represents a value that is not boxed, i.e., they are represented as actual values. Common types that inherit `AnyVal` include number types (`Int`, `Long`, `Double`, and `Float`) as well as other types like `Boolean`. Since the memory representation of these types is often very small, it's much more efficient to store just the actual value in the host system, rather than a reference to the value.

Since Scala 2.10, you can define your own `AnyVal` types by extending the `AnyVal` class. Previously in the implicit section of this chapter we defined an `ApiKey` class, and the definition is repeated below.

```
case class ApiKey(id: String)
```

If you want to turn it into a value type, simply make it extend `AnyVal`:

```
case class ApiKey(id: String) extends AnyVal
```

Now whenever `ApiKey` is instantiated, you won't get a performance penalty due to boxing, yet you still have the benefits of treating `ApiKey` as a different type. You can write functions that take `ApiKey`, instead of having to deal with `String`.

NOTE *The usage for `ApiKey` previously forced you to create a new type due to the usage of implicits.*

There are limitations when it comes down to using `AnyVal` (<http://docs.scala-lang.org/overviews/core/value-classes.html#limitations>), and these limitations are mainly due to the underlying host (in this case, the JVM).

As opposed to `AnyVal`, Scala also has the concept of `AnyRef`, which represents a reference to an object. Essentially any instantiated variable that isn't an `AnyVal` has to be an `AnyRef` (either directly, or indirectly by the type hierarchy). One notable difference with `AnyRefs` is how to treat both equality and identity. Since `AnyRef` stores a reference to either a value or another object, rather than the actual value itself, there are different ways to treat equality. Typically, languages such as C and Java use reference equality to deal with the comparison of non-primitive types, which often means methods have to be separately defined to deal with equality by its contents (also known as deep or structural equality).

Similarly, methods often need to be defined to allow an efficient representation of the reference as a value (this is known as `hashCode` in Java).

In Scala, as in other functional languages such as ML and Haskell, deep equality is used by default for comparison with structures such as case classes, rather than just comparing whether the two objects have the same reference. The same also applies for `hashCode`.

```
case class Example(s: String)

Example("test") == Example("test") // returns true

val a = Example("test")
val b = Example("test")

a == b // Also returns true

class Example2(s:String)

val c = new Example2("test")
val d = new Example2("test")

c == d // Returns false
```

Finally, you have the `Any` type, which basically denotes that the type can be either a reference or a value. `Any` is the supertype of every other type in Scala (that is, everything can be of type `Any`). This is in stark contrast to Java, which although it has an `Object` type, it doesn't have a type to represent values.

SUMMARY

As you can see in this introductory chapter, Scala is a language that has quite a few orthogonal features, which when combined together, provide a highly extensible language that is able to provide expressive and rich libraries, while also being largely correct.

The advanced type system, combined with implicit parameters and subtyping, allows you to apply type safety to very complex business logic, giving the ability for the Scala compiler to detect errors before they get pushed into production. The type system also provides a powerful form of documentation, allowing you to get an initial overview of a library, as well as the powerful and accurate type completion in IDEs, that types provides. Advanced usage of types will be looked at in greater detail in Chapter 9.

Scala also provides the necessary tools to improve performance without too much sacrifice in expressiveness and abstractions. `AnyVal` vs `AnyRef` is an example of such a feature. Case classes, case objects, and sealed traits set up the basis required to model GADTs, an elegant solution to model data structures and ASTs. Pattern matching, essentially a souped-up switch statement whose power is unmatched in many modern languages, gives you a unified solution to many problems, such as deconstructing data structures as well as safe runtime type casts.

A comprehensive collection library provides a vast array of both mutable and immutable data structures with a common interface to maximize reusability, as well as consistency of methods and functions used when calling typical collection methods. It also provides a transformation between different data structures. A functional design underpins the collection methods, which pave the way for the basis of functional programming, which is explored in Chapter 2, and much more advanced material in Chapter 10.

Finally, the explicit control that Scala gives you over both runtime and static modularization of code provides a principal way to approach many issues that are applicable in modern and large scale systems, including, but not limited to, dependency injection and loosely coupled modules. The SBT build tool (explained in greater detail in Chapters 4 and 12) allows you to structure, segment and control how your code is loaded and injected. The SBT tool also excels at supporting the creation of artifacts and the deploying of binaries.

The combination of modularity and functional concepts form the base design of Scala as a language, that is, “*Unifying functional and object-oriented programming*,” which is a direct quote from Martin Odersky, the creator of the Scala programming language.

While this chapter has gone over many of these features to give a general overview, Scala itself has an almost boundless ability to express code in the most desirable fashion. Due to the huge breadth that is available in the Scala language, the later chapters in this book go into greater depth for a smaller range of essential topics, to help pave the way for you to enhance your programming experience, plus a more fundamental basis for Scala knowledge.

2

Functional Programming

WHAT'S IN THIS CHAPTER?

- Understanding the advantages of functional programming compared to the traditional imperative code
- Providing ways to improve readability of your code by using declarative programming
- Handling null-pointer exceptions in a functional way
- Improving the application through refactoring to the functional style

Object-oriented programming has been a standard in large-scale applications for many years, and it isn't going to change any time soon. The advantage of Scala is that it allows you to choose functional programming, without abandoning the good parts of object-oriented architecture. It is possible to program in Scala in the same way that you program in Java. Sure, there would be some different keywords here and there, but the overall structure would be the same. You may start by writing your application in a boring imperative style, and then transforming it to the immutable-functional one.

When a “functional feature” comes to an imperative object-oriented language (see streams in Java 8), you may want to reject it at first. But then, after trying it for some time, you find that you can't develop without it. This chapter provides you with hints about the advantages of functional programming, but it's up to you to decide whether or not functional programming is your cup of tea.

The previous chapter covered Scala's syntax, and in this chapter we discuss the classical pillars of functional programming language, and how they can be implemented using Scala. Sometimes there will be object-oriented/imperative counterparts, to show you the difference between the two styles of coding.

So, what are the main characteristics of a functional programming language? It should be transparent, it should have higher order functions and tools to better work with recursion (to use it without being afraid of stack overflows), its functions should be side-effect free, and there should be a possibility of curing and non-strict (also known as lazy) evaluation. This is not a final definition of functional programming, because it varies from language to language (with Haskell being the most known standard), but it covers the basics. You will see how those techniques can increase your productivity, as well as the readability and the safety of your code.

There are two kinds of people coming to Scala: Java developers who want a more powerful and expressive language that has better tools for working with concurrent systems, and Haskell developers looking for Haskell on a JVM. Due to the limitations of JVM that can't be avoided, the latter group will be disappointed with its capabilities. Those limitations are, however, outside of the scope of this chapter, because they are nonexistent for someone beginning this adventure in functional programming.

IMMUTABILITY

Programming languages have a different approach to immutability: some of them embrace it and don't allow any mutable state, and others allow it only through constants that, arguably, are semantically different from a simple immutable value. Scala allows both approaches through `val` and `var` keywords. Let's look at an abstract example of swapping values in variables that makes mutable state more obvious:

```
var a = 1
var b = 2
// ... other code goes here, but something happens, we need to swap
var c = a // a = 1 b = 2
a = b // a = 2 b = 2
b = c // a = 2 b = 1
```

If you have not seen this pattern before, this code may seem cryptic to you. Also, there is a state of uncertainty at the line with `a = b`, where a thread accessing the values of `a` and `b` will see `a = 2` and `b = 2`. This is something you wouldn't expect to see (it should be either `a = 1 b = 2` at the beginning or `a = 2 b = 1` after the swapping).

So, how can you better handle this situation? It may seem obvious, but wouldn't it be preferable to use additional values:

```
// todo: find real world example with meaningful names
val a = 1
val b = 2
// something happens, we need to swap
val c = a
val d = b
```

Instead of reusing old variables, you can create new values with a new semantic meaning (from new value's name) and immutability. As these are now constant values, you will remain assured that all the way through the method, the values of `a` and `b` will stay the same, no matter how you use them. But what about a more real life example:

```
var users = dao.findAll()
// ... here goes some code that uses users variable
// so that the next line cannot be directly chained ...
users = users.filter(_.age >= 18)
// ... reusing the same users variable through the rest of the method ...
```

In this code adult users are selected. A small schema of what this operation looks like is shown in Figure 2-1.

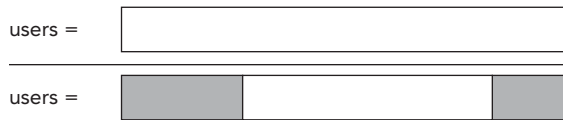


FIGURE 2-1

The darker part of the rectangle is the part that was filtered out of the `users` variable. As you can see (see Figure 2-2), the developer didn't take time to create another variable with a more appropriate name, because it wasn't needed, but if the immutable values were used, it would have been a requirement.

```
val users = dao.findAll()
// ... here goes some code that uses users variable
// so that the next line cannot be directly chained ...
val adults = users.filter(_.age >= 18)
// reusing adults that stays the same during the rest of the method
```

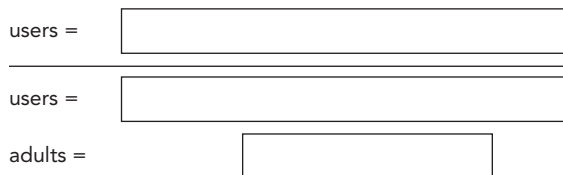


FIGURE 2-2

Here you may see that even after the transformation the user variable has the same value and meaning as before, and the result is stored in a new variable, with a name that is semantically correct.

As a rule of thumb, whenever you have a computation that may be given a name, assign it to a new value with that name. It's far better than a commentary, because it makes your code more readable. This is especially useful when breaking chained methods into meaningful pieces.

But `val` alone does not guarantee immutability. In the following code only `val` is used, but the value is not the same during the execution:

```
val string = new StringBuilder()
string.append("ab")
println(string) // "ab"
string.append("cd")
println(string) // "abcd"
```

When you are using mutable structures, you lose all of the benefits of `val` for some small improvements in performance that will be negated by the time spent debugging your code. You can read the “mutable vs immutable collections” explanation in Chapter 1.

When everything is stateless (no variables or mutable structures) there is also another benefit: your code becomes thread-safe, and it may be called from different threads without any synchronization needed.

Although avoiding `var` at all cost may be great (you can use tools like <http://www.scalastyle.org/> to enforce this rule), there may be some cases where `var` may seem like a necessity. But before going with `var`, look to see if there is no alternative implementation in the part of code you are working on: the forced use of a `var` (outside of Scala library internals) is usually a sign of a “code smell.”

PURE FUNCTIONS

Pure functions, also known as “side-effect free,” are functions that take some value as an input and return a value without using any outside scope for the computations. Such functions are also called “referentially transparent.” That means a call to a side-effect free function may be directly replaced with its returned value. How can you find out if a function is not pure? If it accesses an outside scope or database, mutates input values, prints anything, or returns `Unit`, it has side effects. As an example of such functions, consider this:

```
def helloWorld(): Int = {  
    println("Hello")  
    println("World")  
    42  
}
```

The function is not side-effect free, because it can’t be replaced with its return value, a number 42, as there are still calls to a `println()` that do something, such as showing a string in the terminal. This is why it is not referentially transparent. This function, however, is pure:

```
def add(a: Int, b: Int): Int = a + b
```

It doesn’t use any outside scope and it can be directly replaced with its return value, `add(1, 2)`, which can be replaced with 3 without any consequences. Otherwise, if the `add()` function stores something in the database, it can’t be possible to replace the call with the results so easily.

The benefits of pure functions are:

- **Improved testability:** Your code doesn’t need any mocks or dependency injections. A simple call to the function with predefined input values will always return the same result no matter what happens outside of it.
- **Improved readability:** Methods are loosely coupled because there is no shared scope between them. You will not break the functionality by creating another pure function. Also method signatures are more explicit in side-effect free functions.
- **Improved performance:** If your pure function is computation heavy, you may add caching to prevent it from recomputing the same value twice.

- **Improved concurrency:** Side-effect free functions are thread safe, because they don't use any shared scope. This is especially important in concurrent code, as you will see in Chapter 11. But for instance, imagine calling the `helloWorld()` function above by many threads at the same time. It would be impossible to predict in what order the words “Hello” and “World” would be printed!

Though pure functions are great, it's often impossible to create an application without using functions with side effects. For instance, every access to a database is a side effect. The solution to this problem is to extract side-effect free code into separate functions with meaningful names and keep them as small as possible. For example, instead of:

```
def extractAdultUserNames(): List[String] = {
    val users = dao.findAll()
    users.filter(_.age >= 18).map(_.name)
}
```

You can do:

```
def extractAdultUserNames(users: List[User]): List[String] =
    users.filter(_.age >= 18).map(_.name)

val users = dao.findAll()
extractAdultUserNames(users)
```

As you may see, in this oversimplified example, we refactored the `extractAdultUserNames` method so that it is now side-effect free. As a benefit, this method is now far less painful to test (no need to mock `dao` dependency injection); it may be used in concurrent structures (in a `map()` method of a parallel collection for example). It also has a more accurate and meaningful signature (just judging by the input type and the output type we can guess how the method works).

Because of the abundance of the mutable state in imperative languages, very few talk about side-effect free methods outside of functional programming. Even if the concept of pure function is hard to grasp at first, it may greatly improve your code once applied.

RECURSION

Recursion is rarely used in imperative languages, except for some canonical cases like tree traversal or for writing a Fibonacci function during your job interview. It seems that such functions are often feared because of the complexity they bring, and the potential problems with stack overflow. Surprisingly enough in functional languages, recursion is one of the main ways of doing unbounded loops, because these languages do not have a `while` construct. Scala doesn't remove `while` from your toolbox; instead it gives you tools to handle recursion and use it to your own advantage.

Let's see why `while` is considered a bad practice, with this minimalistic example:

```
var flag = true

while (flag){
    println(flag)
    flag = false
}
```

Easy, right? Also very familiar. So, what's wrong with this code? First of all, it's mutable: there is a `var` that cannot be directly replaced with a `val`. Secondly, the block inside the `while` is not side-effect free, it should somehow modify the scope outside of it; otherwise the condition will always be true and the `while` loop will never end. These two issues were discussed previously in this chapter. Lastly, it is difficult to extract the block inside `while` into a separate method due to the closure (the link to the external flag variable). Otherwise, you need to mutate the variable that is passed in parameters, which is generally considered a poor practice.

So, doing `while` is bad, but is recursion better? Let's take for a fact that any `while` can be written in a recursive way, so you can prove it by creating `whileFunc()`, which does exactly the same thing as `while`, but recursively:

```
def whileFunc[T](block: => T, condition: => Boolean): Unit = {
  if (condition) {
    block
    whileFunc(block, condition)
  }
}

// and an example of usage:
var i = 0

whileFunc({
  println(i)
  i = i + 1
}, i < 10) // the boolean statement will be re-evaluated as it is a call-by-name
```

This code will print numbers from 0 to 9. It is a challenge to convince anyone that this code is better than the 4 lines in the `while` loop. Not only this, but the code will still get stack overflow if you increase the condition number. So, how can you deal with this problem? You can use a tail recursion: a recursive function that has a call to itself as the last call. Here is a classical example of a recursive function representing a factorial:

```
def fact(n: Int): Int = {
  if (n < 1) 1
  else n * fact(n - 1)
}
```

But this is not tail recursive: as you can see, the last call to that function isn't a call to `fact()`; instead it's a call to `*`. To make it a tail recursive you need to modify its signature:

```
@tailrec
def fact(n: Int, acc: Int = 1): Int = {
  if (n < 1) acc
  else fact(n - 1, acc * n)
}

fact(6) // returns 720
```

This last call is to the `fact()` function, and the `@tailrec` annotation verifies that the function is tail-recursive during compilation. If it is, the compiler can transform it into its `while` version, so that

the “stack overflow” exception is no longer an issue. To better see the benefits of a recursive function, consider this code:

```
val ids = List(0, 3, 4, 7, 9) // generally those Ids are stored in the database
var generatedId = 0 // or null, doesn't matter

do {
  generatedId = scala.util.Random.nextInt(10)
} while (ids.contains(generatedId))

println(generatedId)
```

The code is not hard, but everything is here: mutable state, side effects, and a variable declaration outside of the while block. Let’s refactor it in a functional way:

```
val ids = List(0, 3, 4, 7, 9) // generally those Ids are stored in the database

@tailrec
def generateId(currentIds: List[Int]): Int = {
  val id = scala.util.Random.nextInt(10)

  if (currentIds.contains(id)) generateId(currentIds)
  else id
}

println(generateId(ids))
```

Even if you only use recursion once in a blue moon, this code shouldn’t be a problem. There is no modification of outside scope, which is an elegant execution. Every time you enter the body of `generateId()` function, you have a clean state without needing to look outside of the scope.

But sadly, not all recursive functions are easily transformed into tail recursive ones. This is a Fibonacci function:

```
def fibo( n : Int): Int = {
  if (n < 2) n
  else fibo(n-1) + fibo(n-2)
}
```

Its definition is a sum of the previous two values in the Fibonacci sequence. Here is its tail-recursive version:

```
@tailrec
def fibo(n: Int, a: Int = 0, b: Int = 1): Int = {
  if (n < 1) a
  else fibo(n-1, b, a+b)
}
```

It takes some time to understand how it works. But even if it still compiles to a `while` loop internally, it is still better than the actual Fibonacci’s `while` version:

```
def fiboWhile( n : Int ) : Int = {
  var first = 0
```



```
var second = 1
var i = 0

while( i < n ) {
  val result = first + second
  first = second
  second = result
  i = i + 1
}
first
}
```

So, as a rule of thumb, when dealing with `while` in your code, try to see if there is an alternative using Collection API, which is discussed later in this chapter. Otherwise, find a tail-recursive function that will do the job, and don't forget about the `@tailrec` annotation, because it will do all of the complicated checking for you. It will be hard at first, but soon recursion won't be a mystery anymore.

HIGHER-ORDER FUNCTIONS

Higher-order functions accept another function as a parameter and/or return a new one. Purely functional programming languages don't have any objects; instead they have functions as first-class citizens. This means that functions are not different from a normal value like a `String` or an `Int`. And applications are made by combining them into one top-level function. As for a hybrid language like Scala, higher order functions are most useful for removing code repetitions. Consider this simplified example of a stateful backend:

```
val authenticatedUsers = List("Alex", "Sam")

// example url /hello/{userName}
def hello(userName: String): String = {
  if (authenticatedUsers.contains(userName)) s"world $userName"
  else "Unauthorized access"
}

// example url /foo/{userName}
def foo(userName: String): String = {
  if (authenticatedUsers.contains(userName)) s"bar $userName"
  else "Unauthorized access"
}

println(s"request to /hello/Alex: ${hello("Alex")}")
println(s"request to /hello/David: ${hello("David")}")
println(s"request to /foo/Alex: ${foo("Alex")}")
```

The server-side has a list of authenticated users, so when the client-side requests `hello()` action, the server checks first if the user is authenticated and then returns the string generated by the business logic. So, `hello()` and `foo()` methods are nearly identical, it's just the domain logic that is a bit different. Let's extract a higher order function that will handle user authorization in one place:

```
def userAwareAction(userName: String,
                    authUsers: List[String],
```

```

        f: String => String): String = {
    if (authUsers.contains(userName)) f(userName)
    else "Unauthorized access"
}

```

and the refactored methods `hello()` and `foo()`:

```

val authenticatedUsers = List("Alex", "Sam")

def hello(userName: String): String = userAwareAction(userName, authenticatedUsers,
    userName => s"world $userName")

def foo(userName: String): String = userAwareAction(userName, authenticatedUsers,
    userName => s"bar $userName")

```

In addition to the removed repetitions, you can create the method `userAwareAction()` without side-effects, so it has all of the advantages we've discussed, such as the testability without needing to mock anything. Aside from that, this method may also be moved to the parent controller so that any child controller can benefit from it. In this case, if you need to change the way a user is authenticated, such as fetch this information from a database instead of a list, or modify the reply in case authentication failed with a "403" page instead of just a `String`, you can do it in one place from now on.

To summarize, if you are dealing with an application that has a lot of code repetitions, look for a higher order function that will contain the common code. The next part of the chapter covers higher order functions that help you with writing more meaningful and declarative code applied to collections.

As for the part where a method returns a function, that is covered in the section about currying.

CORE COLLECTION METHODS

Collections, including sequences, sets, and maps, may be considered one of the most used data structure in programming languages. In fact, some languages are entirely constructed from lists and primitives; just look at Lisp! Functional programming languages contain quite a number of collection methods, so it is important to know them. In imperative languages you are used to dealing with collections using loops. For example, to transform a collection of `User` objects into a collection of users' names, you can do the following:

```

case class User(name: String, age: Int)

val users = List(User("Alex", 26), User("Sam", 24))

var names = List[String]()

for (user <- users) {
    names = names :+ user.name
}

println(users)

```

A common pattern is to take an element from one array, transform it, and insert it into another. Let's put aside the version with `while`, because we already know why it is considered bad. But look at the previous code: doesn't it have the same problems? It has a mutable state coupled with side effects, and poor method extraction. The fact that this is called micro management is another reason and looks like what is shown in Figure 2-3.

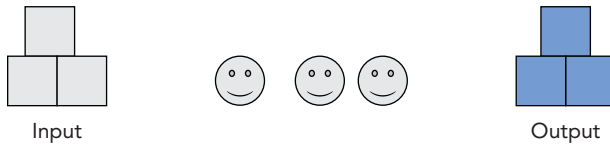


FIGURE 2-3

You need to transform the pile in Figure 2-3 on the left into the pile on the right. When doing `for` loops, you are telling each one of the workers to go to the first pile, take the box, change its color, go to the second pile, and put it there. This is for every loop and iteration. This is quite tiresome for real life management. Why not just tell them to transform this pile of white boxes into the dark ones and leave them to do their job? Luckily, in functional programming there is a function that does such a thing. It is called `map()`, and here is how you can refactor the previous iterative code:

```
val users = List(User("Alex", 26), User("Sam", 24))

var names = users.map(user => user.name)

Or its shortened and, arguably, more concise version:

var names = users.map(_.name)
```

The function that is passed to the method is called `lambda expression`. If at first the `map()` doesn't sound like a "transform" to you, don't worry, because after time in functional programming, you will find it normal because the `map()` method is everywhere. But there is more on this in Chapter 10 on monadic types.

The second most common operation on a collection is ridding it of unwanted elements:

```
val users = List(User("Alex", 12), User("Sam", 22))

var adults = List[User]()

for (user <- users) {
  if (user.age >= 18) {
    adults = adults :+ user
  }
}

println(users)
```

Here you are like club security checking everyone personally, instead of just filtering the flow:

```
val adults = users.filter(_.age >= 18)
```

By now the benefit of declarative programming should be obvious, and the iterative version of the method is no longer needed to prove it. You may say that this is less debuggable, but in a modern IDE, you already have a possibility to directly debug lambda expressions.

The Collection API is divided into two categories: methods that return other collections (including maps) and methods that return values. When doing an operation on an iterable that sounds like it may be already implemented in the core library, you should have a reflex of going to the corresponding API section (like this one: <http://www.scala-lang.org/api/current/index.html#scala.collection.Seq> for Seq) and to look up if there is already an implementation.

The following is a list of selected methods returning a collection. They are not sorted in any particular order, but it is useful to know that they exist as they are implemented in many functional programming languages. More often than not, the purpose of the method may be guessed from its name, but if it's not obvious, the description and the example will better describe the function's goal.

Methods Returning a Collection

`map()` and `filter()` methods have been covered, but there is another function that is useful called `flatMap()`. As an example, let's transform a text file into a list of words from it:

```
val shakespeare = List(
  "Words are easy",
  "like the wind",
  "faithful friends",
  "are hard to find"
)

val words: List[String] = shakespeare.flatMap(line => line.split(" "))
```

The function that is passed as a parameter to the `flatMap()` should return the same type of collection as the `shakespeare`'s value type. In this case, it returns an `Array` of words (that is implicitly transformable into a `List`) for each line. As a result, you have a list of words that is then “flattened” into a single concatenated list of words. If you already have a collection of collections as a value, you may flatten it without any `flatMap`. This is often useful when there is no way to change a `map()` method into a `flatMap()`:

```
val wordsNotFlattened: List[Array[String]] = shakespeare.map(_.split(" "))
val words: List[String] = wordsNotFlattened.flatten
```

The `distinct` method takes no parameters and returns a collection of the same type without repetitions. It may be often replaced with a cast to a `Set`, because it can be semantically better:

```
val listWithRepetitions = List("Alex", "Alex", "Sam")

println(listWithRepetitions.distinct) // prints List(Alex, Sam)
println(listWithRepetitions.toSet) // prints Set(Alex, Sam)
```

The `groupBy()` method is handy when working with a collection of structures that have a unique identifier each. This is particularly useful while analyzing the contents of a database table:

```
case class User(identifier: String, likes: String)

val users = List(User("Alex", "Kiwi"),
                  User("Sam", "Banana"),
                  User("Alex", "Apple"))
val likesByUser = users.groupBy(_.identifier)
/* likesByUser contains:
Map(
  Alex -> List(User2(Alex,Kiwi), User2(Alex,Apple)),
  Sam  -> List(User2(Sam,Banana))
)*/
```

The `partition()` method is not used very often, but it is nice to know that it exists. When applied to a collection, it accepts a function-predicate returning a Boolean and outputs a tuple with two values: the first one is a collection of elements for which the predicate is true, and the second one is with the elements that are left. So, technically, it is the equivalent of doing two `filter()` operations, but only in one traversal:

```
val (moreThanTwo, lessOrEqualThanTwo) = List(1, 2, 3, 4).partition(_ > 2)
```

If you need to sort a collection that contains only standard Scala’s “primitives” (String, Int, etc.), you may apply a sorted method to it. However, more often than not, you are dealing with more complex elements that may not be sorted in that manner easily as they are not “primitives.” Luckily there are two methods that may help you: `sortBy()` and `sortWith()`. The former accepts a function that takes an element from the collection and returns a “primitive” that may be ordered (the age in our case as it is an integer). The latter accepts a function with two parameters that decides if the first element passed to it is less than the second one:

```
List(3, 4, 1, 2).sorted

case class User(name: String, age: Int)

val users = List(User("Alex", 26), User("Sam", 24), User("David", 25))

users.sortBy(_.age)
users.sortWith((a, b) => a.age < b.age) // same as sortBy example,
    but more flexible
```

If your collection is already sorted, but you need it in a different order, you may do it with the reverse method:

```
List(3, 4, 1, 2).sorted.reverse // returns List(4, 3, 2, 1)
```

Sometimes when traversing over a collection, you need to know the index of the element you are working with. The only way to do it in a functional way is to transform your collection using `zipWithIndex()` into a list of tuples where each element is made of a corresponding index and a value. After that you may iterate over it as you would do with any list containing tuples.

Methods Returning a Value

One of the most known methods returning a value is, of course, `foreach()`. It returns a `Unit` value that is a sure sign of side effects. That's why it would be better to avoid it if a better alternative is available. Here is an example of how you can print the contents of a list one element at a time:

```
List(1, 2, 3).foreach(println)
```

There are a few mathematical methods that are just nice to be aware of so that you won't reimplement them again. The three of them that come immediately to mind are `sum()`, `max()`, and `min()`. They do exactly what their names stand for and don't work with certain types, like how `sum()` won't work with a collection of `String` out of the box.

Useful methods returning `Boolean` are `contains()`, along with its variations: `exists()` and `forall()`. The former returns `true` if the element passed in parameters is included into the collection, yet the latter returns `true` only if the predicate (a function that accepts an element and returns a `Boolean`) is true for all elements in the collection. The `exists()` method is similar, but verifies if the predicate is true for at least one element:

```
List(1, 2, 3).contains(1) // returns true
List(1, 3, 5).exists(_ % 2 == 0) // returns false
List(2, 4, 6).forall(_ % 2 == 0) // returns true
```

The second example tries to find at least one even number, and the third example verifies if all the numbers in the collection are even.

The `find()` method, as its name suggests, returns an `Option` containing the first element that the predicate (passed as a parameter) holds true. We will talk more about `Option` (`Some` and `None`) later in this chapter. Here is an example:

```
List(1, 2, 4, 6).find(_ % 2 == 0) // returns Some(2)
```

Two other handful methods are `count()` and `mkString()`. The first one accepts a predicate and counts the elements for which the predicate is true. The second one transforms a collection into a `String` with a parameter as a delimiter between the elements:

```
List(1, 2, 4, 6).count(_ % 2 == 0) // returns 3
List(1, 2, 4, 6).mkString("|") // returns "1|2|4|6"
```

Finally, there are some methods that are somehow complicated for developers coming from imperative programming languages, but they are still useful to know to be able to read functional code. The most noteworthy are the `fold()` and `reduce()` methods. `fold()`'s signature is quite complex, but it boils down to `fold(initialAccumulatorValue)((accumulator, element) => newAccumulator)`. In the currying part of the chapter we will discuss what that parentheses madness is all about, but for now treat it as if `fold()` accepted two parameters: the initial value of the

accumulator and the lambda expression that transforms current accumulator and an element of the collection into a new accumulator. Here is how you can rewrite the `sum()` method with `fold()`:

```
fold(0)((acc, value) => acc + value)
```

Here each element is added to the accumulator with the initial value as zero (as the sum of an empty list is zero). The `reduce()` method is often referred to in the “map-reduce” model, that’s why it may be a little more recognizable. It’s essentially the same thing as `fold()`, but without an initial value, because it is deduced from the operation on the first two values. Here is the `sum()` equivalent:

```
reduce((acc, value) => acc + value)
```

If you apply `reduce` on an empty list, it will throw an error. So, is there any need to use `reduce()` instead of `sum()`? Not in the slightest, but imagine that you would like to divide the elements in the collection between them. The best way is to use the `reduce()` method, or `fold()` if the collection may be empty:

```
List(1, 2, 3).reduce(_ / _)
List(1, 2, 3).fold(1)(_ / _)
```

To conclude, you can see how the `collection` API helps to remain immutable and side-effect free. Its wide variety of methods helps you write readable and elegant code. That said, you should prefer readability above everything else. If you see that a version with `for` will be easier to read, use it without hesitation:

```
val listX = List(1, 2, 3)
val listY = List(1, 2, 3)

for {
  x <- listX
  y <- listY
} yield (x, y)

listX.flatMap(x => listY.map(y => (x, y)))
```

The version with `for` is clearly more readable.

CURRYING AND PARTIALLY APPLIED FUNCTIONS

Partially applied functions or currying (also known as Schönfinkelization, thanks to the work of mathematician Moses Schönfinkel) are, one of the most difficult concepts to put into practice among those mentioned in this chapter. It is often used for code factorization, local dependency injection, or simply to improve code readability.

As an example, let’s take this `add` method:

```
def add(a: Int, b: Int) = a + b
```

It may seem to be quite useless, but if you have `add5` and `add7` in your code, you can simplify their definition with a more generalized one using partially applied functions:

```
def add5 = add(5, _: Int)
def add7 = add(7, _: Int)

add5(3) // returns 8
add7(2) // returns 9
```

Here you have the adding logic in one place: inside the `add()` function. If someday it changes, you can take care of it by changing the `add()` method, similarly to the technique you already saw in the section on higher order functions. We can also write the `add()` function using currying:

```
def add(a: Int)(b: Int) = a + b

def add5 = add(5) _
def add7 = add(7) _
```

So, what's the difference? These two are essentially the same, but the curry version is used more because it is visually better when the second parameter is a function, because you can use curly braces instead of parentheses:

```
def using[T](fileName: String)(f: List[String] => T): T = {
    val file = scala.io.Source.fromFile("file.txt").getLines().toList
    f(file)
}

val words = using("Hello.txt") { lines =>
    lines.flatMap(_.split(" "))
}
```

Let's see another example where currying is useful. Imagine you have a huge list of users and you need to modify each user into some other value. This transformation is handled by a `transform()` method that takes an instance of `Config` and a user to output the same user with a slightly changed name:

```
def transform(config: Config, user: User): User = {
    user.copy(name = s"${user.name} ${config.userNameSuffix}")
}

val transformedUsers = users.map(user => transform(config, user))
```

And here is a version with currying:

```
def transform(config: Config)(user: User): User = {
    user.copy(name = user.name + config.userNameSuffix)
}

val transformedUsers = users.map(transform(config))
```


As you can see, other than the added parentheses, there were no changes to the function. This is cleaner, but a bit harder to read if you don't know about the syntax.

It is possible that you are already using partially applied functions, when dealing with Maps. Using pattern matching, you can use this syntax:

```
val usersWithId = Map(1 -> User("Alex", 27), 2 -> User("Sam", 23))

val users = usersWithId.map{ tup => s"${tup._1}: ${tup._2.name}" }
// extracting user's name prepended with her id
```

But this is ugly. You can use pattern matching instead of accessing tuple's cryptic properties (`_1`, `_2` and so on):

```
val usersNames = usersWithId.map{ user => user match {
  case (id, value) => s"$id ${value.name}" // using "user" for the value would
  be bad because we already have a "user" as a parameter
}
```

And here is a version with a partially applied function (with Scala compiler's help):

```
val usersNames = usersWithId.map {
  case (id, user) => user.name
}
```

To conclude, use currying when you need to “prepare” functions with some dependencies. Also use it when it makes a more readable code thanks to the curly braces for functional parameters. Partially applied functions may be useful in conjunction with pattern matching to avoid useless repetitions.

NULL HANDLING (OPTION)

Scala has a very neat way to avoid nulls. In fact, if there was no need to keep the compatibility with Java, it's certain that Scala would remove `null` from the language altogether. Pure functional programming languages don't have `null` at all, but instead they have a `Maybe` type that is represented in Scala by `Option`. Consider this method signature:

```
def findUserById(id: Int): User
```

Let's say you don't have access to its internals, and here is an example of this usage:

```
val user = findUserById(10)
```

Would you check the resulting user value for being `null`? If not, have you considered what happens if the user doesn't exist and you try to access the name with `user.name`, which throws the famous `NullPointerException`. Instead let's define the method the other way around:

```
def findUserById(id: Int): Option[User]
```

Just by reading the signature you may guess what this method would do if it does not find the user in the database: it would return `None`; otherwise it would be `Some(user)`. So, every time there is a

possibility of an absence of the result, use `Option`. There are different ways to handle this type, and you may do it immediately:

```
findUserId(10) match {
  case Some(user) => user.name
  case None => "anonymous"
}

// or
val user = findUserId(10).getOrElse(User("anonymous", 0))
```

Or you may just modify the value inside the `Option` and leave the handling of `None` to the layer above.

```
val name: Option[String] = findUserId(10).map(_.name)
```

The `map()` here will transform the result from `Option[User]` to `Option[String]` containing the user's name. If there is no user, the final result is still `None`. It is easier to understand if you imagine `Option` as a sort of list that is either empty or containing one element.

If you are dealing with something that may return `null`, just wrap it into an `Option`, and it will transform the returned `null` into `None`. Also, never use the `Option`'s `get()` method, because it will throw an exception if the element is `None`, effectively replacing `NullPointerException` problem with a similar one.

STRICT VERSUS NON-STRICT INITIALIZATION

It's worth writing a few words on a non-strict initialization. There are two kinds of it: lazy values and call-by-name parameters. There is a "code smell" when you know you need the former:

```
class foo{
  var database = _

  def bar() = {
    database = initDb()
    // ... the rest of the code that uses database here ...
  }
}
```

Here we didn't initialize the configuration directly because it would consume a connection from a connection pool, or it would simply throw an exception (cannot be initialized in the constructor). You may also notice a `var` and a `null` that are bad, and were discussed previously in this chapter. All of these problems can be solved using `lazy`:

```
class foo{
  lazy val database = initDb()

  def bar() = {
    // ... the rest of the code that uses configuration here ...
  }
}
```

Other than a benefit, which is an immutable code without null instances, the initialization of the database value becomes thread-safe (it uses double-checked locking internally), so different workers accessing the value won't see an uninitialized database.

Call-by-name parameters are useful when there is a need to wrap your code with some initializations. For instance, if you want to measure code execution with `System.nanoTime`, the solution is to do the following:

```
def measure[T] (code: => T): String = {  
  val start = System.nanoTime  
  code  
  val end = System.nanoTime  
  s"It took ${end - start} nanoseconds to execute the code"  
}  
  
measure(1 + 1)
```

In the `measure()` function, the code passed in parameters will be executed only when called in the body of the function. This provides better flexibility of what code to test. As a note, in Chapter 9 you will see that there is a better tool for micro-benchmarking than `nanotime`.

SUMMARY

This chapter covered the basic building blocks of functional programming. Functional code was compared to its imperative counterpart, and new techniques were detailed that help you write more readable and maintainable code.

Immutability makes it possible to create a thread-safe, stateless code that insists on writing more intermediate values with meaningful names, making the code more readable. Free functions were also covered that don't change the external scope, providing referential transparency that makes unit testing a breeze. Details about why `while` is bad, and how the recursion may be used for something more than just traversing recursive structures, were also covered.

Higher order functions help you write more concise, declarative code, as you saw in the section about the `collection` API. Higher order functions also provide you with tools to factorize your code more effectively.

We examined two useful features of functional programming: the `Option` type and lazy execution. The former helps you avoid `NullPointerExceptions`, and provides a way to describe a function that may have an absent value just by its return type, while the latter is a nice way to bypass value initialization until later when it's needed.

3

Java Compatibility

WHAT'S IN THIS CHAPTER?

- Converting back and forth between Java and Scala collections
- Understanding the relations between Java interfaces and Scala traits
- Using Scala with enumerations

Right from the start, the creators of Scala took the Java compatibility issue very seriously. This makes a lot of sense, given all of the already existent Java libraries that are out there.

This chapter starts by showing you how to convert Java collections to Scala and vice versa. It covers how Scala traits relate to Java interfaces, and details how both things can cooperate. Java enums are then *mapped* into the Scala world. You'll see that, in Scala, you have more than one alternative.

SCALA AND JAVA COLLECTIONS

Collections are probably one of the more used APIs, both in Scala and Java. When interoperating with a Java library it's important to know how you can go from a Java collection to a Scala one and back again.

In the `scala.collection` package, there are two objects that can be used for this purpose, namely `JavaConversions` and `JavaConverters`.

The former provides a bunch of implicit conversions supporting interoperability between Scala and Java collections. For example, Java's `List` does not have a `map` method, but you can still call `map` on it thanks to the implicit conversion into `ArrayBuffer`:

```
import java.util.{ArrayList => JArrayList, List => JList}

import scala.collection.JavaConversions._
import scala.collection.mutable
```

```
val javaList: JList[Int] = new JArrayList()
javaList.add(42)
javaList.add(1)

val asScala = javaList.map(_ + 1)
println(s"Java List to Scala: $asScala")
```

The output is as follows:

```
Java List to Scala: ArrayBuffer(43, 2)
```

JavaConversions provide two-way implicit conversions among the following types:

```
scala.collection.Iterable <=> java.lang.Iterable
scala.collection.Iterable <=> java.util.Collection
scala.collection.Iterator <=> java.util.{ Iterator, Enumeration }
scala.collection.mutable.Buffer <=> java.util.List
scala.collection.mutable.Set <=> java.util.Set
scala.collection.mutable.Map <=> java.util.{ Map, Dictionary }
scala.collection.concurrent.Map <=> java.util.concurrent.ConcurrentMap
```

There are also the following one-way implicit conversions:

```
scala.collection.Seq => java.util.List
scala.collection.mutable.Seq => java.util.List
scala.collection.Set => java.util.Set
scala.collection.Map => java.util.Map
java.util.Properties => scala.collection.mutable.Map[String, String]
```

JavaConverters, on the other hand, extends the Java and Scala collections by adding to them the asScala and asJava methods, respectively. It is newer than JavaConversions and makes the conversions explicit.

As a rule of thumb, always remember that explicit is better than implicit. For this reason prefer JavaConverters to JavaConversions, unless you have a very good reason not to.

Table 3-1 shows the available conversions from Scala to Java collections, while Table 3-2 shows the other way around. Note that in both tables *s.c* stands for *scala.collection*, while *j.u* stands for *java.util*.

TABLE 3-1: From Scala to Java

TYPE	METHOD	RETURNED TYPE
<i>s.c</i> .Iterator	asJava	<i>j.u</i> .Iterator
<i>s.c</i> .Iterator	asJavaEnumeration	<i>j.u</i> .Enumeration
<i>s.c</i> .Iterable	asJava	<i>java.lang</i> .Iterable
<i>s.c</i> .Iterable	asJavaCollection	<i>j.u</i> .Collection
<i>s.c</i> .mutable.Buffer	asJava	<i>j.u</i> .List

TYPE	METHOD	RETURNED TYPE
s.c.mutable.Seq	asJava	j.u.List
s.c.Seq	asJava	j.u.List
s.c.mutable.Set	asJava	j.u.Set
s.c.Set	asJava	j.u.Set
s.c.mutable.Map	asJava	j.u.Map
s.c.Map	asJava	j.u.Map
s.c.mutable.Map	asJavaDictionary	j.u.Dictionary
s.c.mutable.ConcurrentMap	asJavaConcurrentMap	j.u.concurrent.ConcurrentMap

TABLE 3-2: From Java to Scala

TYPE	METHOD	RETURNED TYPE
j.u.Iterator	asScala	s.c.Iterator
j.u.Enumeration	asScala	s.c.Iterator
java.lang.Iterable	asScala	s.c.Iterable
j.u.Collection	asScala	s.c.Iterable
j.u.List	asScala	s.c.mutable.Buffer
j.u.Set	asScala	s.c.mutable.Set
j.u.Map	asScala	s.c.mutable.Map
j.u.concurrent.ConcurrentMap	asScala	s.c.mutable.ConcurrentMap
j.u.Dictionary	asScala	s.c.mutable.Map
j.u.Properties	asScala	s.c.mutable.Map[String, String]

You can easily rewrite the previous example using `JavaConverters`:

```
import scala.collection.JavaConverters._
import java.util.{ArrayList => JArrayList, List => JList}

val javaList: JList[Int] = new JArrayList()
javaList.add(42)
javaList.add(1)

val asScala = javaList.asScala.toList.map(_ + 1)
println(s"Java List to Scala: $asScala")
```

Here you import `JavaConverters` instead of `JavaConversions`, and explicitly convert the `Java List` to `Scala` using the `asScala` method.

Note also that, as shown in Table 3-2, calling `asScala` on a `Java List` you get back a mutable `Buffer`. Since we love living in an immutable world when it comes to functional programming, we converted the `Buffer` into an immutable `Scala List` by calling `toList` on it.

Finally, this is the output:

```
Java List to Scala: List(43, 2)
```

INTERFACES AND TRAITS

Interoperability problems in collections are what makes `Java` libraries hard to use in `Scala` code. But what about general things like classes, interfaces, and traits? Given that `Scala`'s collection of language features is a superset of `Java`'s ones, the use of `Java` libraries in your `Scala` projects is easy and straightforward. Classes and abstract classes are already present in `Scala`, and an interface is just a trait without any method implementations. Here is how it works:

```
// Java interface
public interface CompatibilityInterface {
    public void print();
}

// Scala implementation
class ScalaClass extends CompatibilityInterface{
    override def print() = println("lorem ipsum")
}

(new ScalaClass).print()

// Java class
public class CompatibilityClass {
    public void print(){
        System.out.println("lorem ipsum");
    }
}

// Scala implementation
class ScalaClass extends CompatibilityClass

(new ScalaClass).print()
```

If you need to use the `Java` library in your project, you shouldn't have any problems at all. But if you need to export your `Scala` code so that it can be used in `Java`, there may be some complications. For instance, the following example won't compile:

```
// Scala trait
trait ScalaTrait {
    def print() = println("lorem ipsum")
}
```

```
// Java class
public class JavaUseOfTrait implements ScalaTrait {}

// Java application
public class JavaApplication {
    public static void main(String[] args) {
        JavaUseOfTrait j = new JavaUseOfTrait();
        j.print(); // This will not work!
    }
}
```

To be more precise, it will throw the following exception: `Error: com.professionalscala.ch3.java.JavaUseOfTrait is not abstract and does not override abstract method print() in com.professionalscala.ch3.ScalaTrait`. Java can't find the implementation that is provided in the trait. To understand why it is the case, you must open the folder with the generated byte-code (for example “project/target/scala-2.11/classes/com/professionalscala/ch3”) and see how Java sees the compiled code:

```
ch3 $: javap ScalaTrait
public interface com.professionalscala.ch3.ScalaTrait {
    public abstract void print();
}
```

As you may note, for Java `ScalaTrait` is only an interface without any implementation. This is the case because Scala's byte-code should be compatible with Java 6, but this version does not have any method implementations in the interface (by contrast to Java 8). So these methods should be stored somewhere else. You may have already noticed the compiled `.class` file named `ScalaTrait$class`. `.class`, let's see what it contains:

```
ch3 $: javap ScalaTrait\$.class
public abstract class com.professionalscala.ch3.ScalaTrait$.class {
    public static void print(com.professionalscala.ch3.ScalaTrait);
}
```

Here's where the implementation is located, but notice that the method is static and it accepts an instance of `ScalaTrait`. How can you use it in your Java code? Here is where it becomes somewhat complicated:

```
public class JavaUseOfTrait implements ScalaTrait {
    @Override
    public void print(){
        ScalaTrait$.class.print(this);
    }
}
```

You should implement the generated interface and use Trait's implementation in the overridden method. This is how a Scala class that extends `ScalaTrait` looks like when decompiled by `javap`.

Now let's talk about static methods. Scala's object methods are plainly transformed into static ones:

```
class ScalaObject {
    def say() = println("Hello")
}
```



```
object ScalaObject{  
  def print() = println("World!")  
}
```

Decompilation yields the following result:

```
ch3 $: javap ScalaObject  
public class com.professionalscala.ch3.ScalaObject {  
  public static void print();  
  public void say();  
  public com.professionalscala.ch3.ScalaObject();  
}
```

As you see, the resulting class contains methods from both Scala's class and its companion object.

To wrap up, any Java library should work in Scala out of the box. Other than type casting, as in the case of collections, there is nothing else that is problematic. The usage of Scala's traits in Java, however, needs some knowledge about how the trait is compiled to the byte-code. If you need your library to be usable on both platforms, you should consider this specific case and create some adapters for Java developers.

SCALA/JAVA ENUMERATIONS

Unfortunately, there is not a direct translation from Java to Scala in terms of Enumeration. You can split your project to include enumeration using Java enumerations by having Java files just for using Java enumeration. However, if you can handle a slight paradigm shift in how to utilize enumerations, you can enact one of two Scala-based alternatives. The first is an example of using an object to an encapsulated number of objects that extend a sealed trait.

```
sealed trait Color { val hex }  
  
object Color {  
  case object Red extends Color { val hex = "#FF0000" }  
  
  case object Yellow extends Color{ val hex = "#ffff00" }  
  
  case object Green extends Color{ val hex = "#00ff00" }  
}
```

While this implementation doesn't give you the ability to iterate over the inner objects (although you can add that either through a macro or list addendum), you get the immediate benefit of being able to use case matches. Another positive is the ability to create even more custom field values for the enumerated type, which can be quite handy depending on the enumeration you are attempting.

To try out the other method of creating enumerations in Scala, you only need to create an object and extend the `scala Enumeration` abstract class.

```
object Colors extends Enumeration {  
  val Red, Yellow, Green = Value  
}
```

Note how we left off the hex implementation, because that would require the creation of an abstract class to accommodate that lost functionality. Even if you had added that additional functionality, you would still lose the ability to do a non-exhaustive search, which makes pattern matching less ideal. That said, you inherently maintain the ability to iterate over those values. This can be helpful, and you'll note that the actual implementation is much simpler to understand.

SUMMARY

This chapter examined collections, which are used in APIs in both Scala and Java. You also examined the available conversions between Scala and Java collections. Given that Scala's collection of features is a superset of Java's features, it helps make it more straightforward when using Java libraries in your Scala projects. Unfortunately, there is not a direct translation from Java for Scala in terms of enumerations, but there are Scala-based alternatives that were covered. These are all examples that illustrate how the creators of Scala took the Java compatibility issue very seriously.

4

Simple Build Tool

WHAT'S IN THIS CHAPTER?

- Choosing a scala version
- Adding Library dependencies
- Using the REPL
- Running a program
- Running tests

The Simple Build Tool, or simply SBT, is a robust tool that uses the Scala language with an easy interface to create simple or complex behaviors within your project. While using SBT you can create a project with little to no configuration, and you can package/publish JARs and be presented with a very powerful shell that can be used to test your code from the outside.

With so many features, it's not hard to see why SBT is known as the de facto build tool for compiling, testing, and packaging Scala services and applications. Therefore, learning how to use and extend SBT for your own projects results in a very maintainable and extensible project.

To give a comparison, most developers have experiences in using build tools such as npm, gradle, and bundler. SBT allows you to pull in different dependencies, set up custom tasks, interact with external repositories, and even create your own testing frameworks through a slick DSL. All that's required is getting a basic understanding of the concepts to which SBT subscribes. Then build toward an understanding of how these techniques can support very small to very large scale Scala code bases. This is of course the purpose of this chapter.

While going over these techniques, keep in mind that the tool presented here is vast, and that configurations can become extremely customized toward your own goals. In this chapter we'll focus on the maintainable benefits of a highly structured SBT environment, and provide examples that support good SBT usage.

First, make sure that you have downloaded and installed SBT from the main website, currently <http://www.scala-sbt.org/download.html>, and follow the instructions for your particular platform. At the time of writing the current version number is SBT 13.9, but classically SBT has been very good about supporting backward comparability—so if you are on a later version, you should be able to follow along.

SBT Configs

SBT has a lot of flexibility around configuring where you want to keep your global settings and plugins for your projects. By default this content is stored in the following locations:

```
~/.sbtconfig:
```

This file is deprecated, since it was used for global settings that should be moved into `/usr/local/etc/sbtopts`.

```
~/.sbt/0.13/plugins/:
```

Here you can install a plugin for use in all projects.

```
~/.sbt/0.13/global.sbt:
```

Here you define global settings for plugins.

```
/usr/local/etc/sbtopts:
```

You'll need to modify to adjust global settings applied to SBT on startup, for example, to change your debug port or max memory.

BASIC USAGE

SBT doesn't require a build definition file for simple programs, nor does it require that the file be placed in a standard directory structure. So, for quick scripting it can be easy to touch a file, add the main method, then start coding away.

For example, let's create a Scala file that does a simple factorial computation, since that's the rave in interviews on Scala:

```
1 object Factorial {  
2   def main(args: Array[String]): Unit = {  
3     println(factorial(5))  
4   }  
5  
6   def factorial(n: Int): Int = {  
7     if(n == 0){ 1 }  
8     else{ n * factorial(n-1) }  
9   }  
10 }
```

- It's important to pay attention to the main signature, since if you return anything but a unit, SBT will be unable to find the main method.
- By default SBT will compile and build the project with the same version of Scala that SBT was run with.

The next step is to run the SBT command in the same folder where the file exists and to use the `run` command. You should see a result of 120. This is just a simple example of running a Scala program through SBT, and more advanced usage of the SBT console will be provided as we go along.

If you decide to create multiple Scala files with main methods in that same folder and use the `run` command, SBT will automatically give you a choice of which file to run. This can be great when you have a test for your application and want to have a few choices on which example app to run.

Please take a look at <https://github.com/milessabin/shapeless>. Switch to the examples project and then execute the `run` command. It gives you a good breakdown of all the code examples that correlate to the features of the project. This same pattern is used in the code base used for this book, to create more interesting coding samples.

Project Structure

While the setup of a simple code snippet using SBT doesn't require any special ceremony other than having SBT run in the same directory as a Scala file with a main method, you'll quickly find that having some order to your upcoming project will make a good deal of sense. As such, the folder layout for any basic single Scala project should resemble the following:

```
src/
  main/
    resources/
      <files to include in main jar here>
    scala/
      <main Scala sources>
    java/
      <main Java sources>
  test/
    resources
      <files to include in test jar here>
    scala/
      <test Scala sources>
    java/
      <test Java sources>
```

In the event you need to add more directories to this model, such as the `main` or `test` folders, SBT will by default or ignore those custom directories.

Single Project

When first starting out with using SBT it can be helpful to implement the setup of a bare bones build file, and work toward a more complex example. SBT 13.9 has some nice improvements on another earlier version (mostly in a more opinionated multi-project setup). Using the previous example, you can create a `build.sbt` file in the same directory as the `Factorial.scala` file.

```
1 lazy val root =
2   (project in file("."))
3     .settings(
4       name := "Factorial",
5       version := "1.0.0",
6       scalaVersion := "2.11.7"
7     )
```

You can now start up SBT again in interactive mode using the `run` command, which will still result in a 120. So, what changed? During the startup of SBT, an immutable map describing the build persisted within SBT that was fed from the build definition that was just provided.

For example, within the SBT prompt you can type out any of the values that were just provided:

```
> name
[info] Factorial
> version
[info] 1.0.0
> scalaVersion
[info] 2.11.7
```

What has occurred is that a build file defines a sequence of `Setting[T]`, in this case `Setting[String]`, as `name`, `version`, etc. This feeds into SBT's immutable map of settings for each key/value on startup. You can also use those settings to expand out the `build.sbt` file programmatically and break up the settings easily into their own variable by abstracting the file out a bit further:

```
1 lazy val buildSettings = Seq(
2   organization := "com.professionalscala",
3   version := "1.0.0",
4   scalaVersion := "2.11.7"
5 )
6
7 lazy val root =
8   (project in file("."))
9     .settings(buildSettings)
10    .settings(
11      name := "Factorial",
12      description := s"${name.value} using ${scalaVersion.value}"
13    )
```

- You may notice that the code provided is using `lazy val`, which is purely to avoid order problems during the SBT startup process. You can use `val`, `lazy val`, or `defs` as well.

By doing this breakup of `buildSettings` in another variable, the technique allows reuse across other project definitions. Not only that, but since `SettingKey`'s are computed, once on startup you can setup some interesting behaviors. If, for example, you create a build on Jenkins, you can have Jenkins increment an environment variable for a minor build version of the project. This is useful after deploying out to your cluster.

```
3   version := {
4     val minorV =
5       Option(System.getenv.get("currentMinorRelease"))
6         .getOrElse("0").toInt
7     s"1.0.${minorV}"
8   },
```

The above would result in the version being 1.0.0 unless Jenkins incremented the environment, in which case the output would be “1.0.1.” It’s useful to remember that `settingsKeys` can be modified for any number of unique situations within your project. So far we have only used Setting Keys in the project definition, but it’s important to note that there are three kinds of keys that can be used:

- **SettingKey[T]**: A key for a value computed once (the value is computed when loading the project, and kept around)
- **TaskKey[T]**: A key for a value, called a task, that has to be recomputed each time, potentially with side effects
- **InputKey[T]**: A key for a task that has command line arguments as input

Scopes

When working with SBT, one of the concepts to understand is “scope.” Each key type can have an associated value in more than one context, which is described as a “scope.”

A scope can be overwritten in each sub-project through the use of the multi-project build.sbt files found in those multi-project directories. This enables different build settings, different packaging, and can allow for different values to be held within.

Scope axis is the phrase SBT uses to distinguish how particular settings or behaviors work together. The Scope axes are:

- Projects
- Configurations
- Tasks

In the most basic terms, scope allows different values to apply to various events within SBT. For example, if you want to disable scaladoc generation in a play application, you can add the following: <http://stackoverflow.com/a/21491331>.

```
// Disable scaladoc generation
publishArtifact in (Compile, packageDoc) := false,
publishArtifact in packageDoc := false,
sources in (Compile, doc) := Seq.empty,
```

The first line above shows that the key `publishArtifact`, when doing a `compile` or `packageDoc` task, is instructed to NOT generate the projects documentation.

You typically will only need to muck with a scopes values if the plugin, custom code, or framework is doing something undesired, while you are in the life-cycle of development. Normally, you won’t need to touch these values.

Custom Tasks

Custom tasks or `TaskKey`'s can be useful in developing out build automation, and later creating custom plugins for SBT. Without going too heavily into creating a plugin for SBT, let's go over a very basic example inside the SBT interactive shell.

```
> set TaskKey[java.util.UUID]("newId") := { java.util.UUID.randomUUID }
```

Execute that task by using the `show` command:

```
> show newId
[info] b43726fa-0c73-4e6c-ba7d-6c386f03a969
> show newId
[info] 91256835-8d62-446e-ae61-690304472c50
```

Now you have an easy generator for new `uuid`'s if you don't already have `uuidgen` available on your local environment.

Say, for example that you need a task that requires some information from the box it was currently running on, and that information couldn't be obtained through other Java libraries. You can build a custom task to accomplish this as well:

```
> set TaskKey[String]("getUname", "get the uname") := { Process("uname -a")!! }
> show getUname
[info] Linux bucket 3.13.0-75-generic ...
```

Moving those custom tasks into your build only involves a slight change to the syntax within the `build.sbt` file.

```
val getUname = TaskKey[String]("getUname", "gets the local machines uname")
lazy val root =
  (project in file("."))
    .settings(buildSettings)
    .settings(
      name := "Factorial",
      getUname := {
        Process("uname -a")!!
      },
      description := "a simple factorial"
    )
```

After reloading SBT, you can then use `show genUname` to see the same result as the previous console execution results.

Dependencies

When working with the build file, you want to bring in third-party dependencies. This task can be accomplished with little effort by adding the JAR directly to the `lib/` directory (in the base folder) as unmanaged dependencies, or by adding the dependency directly to the `build.sbt` file as a managed dependency.

```
val http4sVersion = "0.11.2"
lazy val commonSettings = Seq(
  resolvers ++= Seq(
```



```

    Resolver.sonatypeRepo("releases"),
    Resolver.sonatypeRepo("snapshots")
  ),
  libraryDependencies += Seq(
    "org.http4s" %% "http4s-blaze-server" % http4sVersion,
    "org.http4s" %% "http4s-dsl"          % http4sVersion,
    "org.http4s" %% "http4s-circe"        % http4sVersion
  )
)

```

Note the `libraryDependencies` section: since each of those entries directly maps to a formula similar to patterns, you may see a maven xml file.

```
groupId %% artifactId % version % scope
```

This pattern is used to look up the POM and JAR files stored on either the defined or default ivy2/maven repositories. Using the `repo1.maven.org` repository as an example, you can follow each stage of the formula being applied and then see how those endpoints line up when they are being pulled:

- `GroupId` = `https://repo1.maven.org/maven2/org/http4s/`
- `ArtifactId` = `https://repo1.maven.org/maven2/org/http4s/http4s-blaze-server_2.11/`
- `Version` = `https://repo1.maven.org/maven2/org/http4s/http4s-blaze-server_2.11/0.11.2/`

The double percentage sign is used to specify a lookup for a binary compatible version of the `http4s-blaze-server`, and was also used in the `ArtifactId` to silently append 2.11 onto that string. You can also specify a single percentage sign, which would be an exact string match, rather than depending on SBT to append the version of Scala to the lookup.

```
"org.http4s" % "http4s-blaze-server_2.11" % http4sVersion
```

The above is an equivalent to using the double percentage signs. It's also worth noting that you may find some libraries using triple percentage signs, which is indicative of a scalajs dependency. More about that can be found in the Webjars section of the Scalajs chapter later in this book.

Resolvers

In the same way the above examples exposed third party dependencies, it can be necessary to specify third party repositories. This can be helpful if you are working with a third party library and need snapshots rather than stable releases. SBT comes with a few predefined repositories that can be found at <http://www.scala-sbt.org/0.13/docs/Resolvers.html>—but in most cases simply following the above example will be enough to clear up any missing dependencies.

In the event you are unable to find the repository that has the dependencies, you are attempting to add to your local ivy cache. You can do a manual search for them using the maven central repository (<http://search.maven.org/>) and looking up the dependency by its `GroupId` or `ArtifactId`. Once you've found the correct dependency, the information page will provide you with resolver and matching `libraryDependency` code for including that dependency in your `build.sbt` file.

ADVANCED USAGE

Often when building services in Scala, the problem of needing a multi services or a shared library of common code between said services will arise. As an option you can make this shared code into a completely separate project, and then include that as a dependency within the other sub projects. However, when a core library is being actively developed, that technique can create confusion and result in breakage within your code base.

As an alternative, SBT provides the means to set up a multi project, which gives you the flexibility to create a shared library. This can depend on the dependencies apart from the other sub projects and enables other core programmers to expand the shared library with more confidence that those changes won't break other sub projects depending on that core.

Unlike the single project setup, folders in the base directory are going to take on another form, where you can move the `src` folder under a common name for the service that it reflects. Such a folder structure may look similar to the following:

```
common/  
  src/  
    main/  
    test/  
services-a/  
  src/  
    main/  
    test/  
    resources/  
services-b/  
  src/  
    main/  
    test/  
    resources/
```

Figuring out the locations of the `build.sbt` files is a crux, since nothing stops you from having all of your `build.sbt` files stay in the root folder. We advise against this, however, and place `build.sbt` files into each of those sub-projects. During your compiling/building of the root project, all of the sub-project SBT files will merge together with the build definition for the entire project.

This is especially helpful if you prefer to run things under one large umbrella application, or run projects from their sub project view in the interactive console. It is helpful to keep `build.sbt` files in each sub-project, since when you attempt to package your application you can be specific about the dependencies and relationships of those sub projects—keeping them isolated for deployments.

After moving the common code into that sub directory you'll need to make a slight change to the root `build.sbt` file to specify the other sub-projects as projects.

```
lazy val buildSettings = Seq(  
  organization := "com.example",  
  version := "0.1.0",  
  scalaVersion := "2.11.7"  
)  
  
lazy val core = (project in file("core"))
```

```

    .settings(buildSettings)
    .settings(
      // other settings
    )

    lazy val service = (project in file("services-a"))
      .settings(buildSettings)
      .settings(
        // other settings
      )

    lazy val services-b = (project in file("services-b"))
      .settings(buildSettings)
      .settings(
        // other settings
      )

```

Notice that the build settings are included in each of the services now defined. This will ensure that the basic information of organization and Scala version are defined for all projects, meaning they will be easier to maintain.

Modifications made in the sub-projects build.sbt files will be reflected in the SBT console as well. So, now you need to learn about how to depend on different sub-projects for using the core library in other sub-projects, which can be accomplished by using the `dependsOn` method call:

```

lazy val services-a = (project in file("services-a"))
  .dependsOn(core)
  .settings(buildSettings)
  settings(
    // other settings
  )

```

Code in the core library will be accessible by `services-a`. This will allow you to run a task on the top level project that will force the other aggregated projects to also run the same task. You can add the `aggregate` method onto the service project in the same way as the `dependsOn` method.

Advanced Dependencies

A quick note on keeping things sane while doing large multi project builds: It's possible to have each build.sbt in each sub-project define its individual dependencies and also use the core projects dependencies as a `dependsOn` project that brings in the remainder. In doing this, however, duplication will arise between all of these files. In that case, creating a dependencies Scala file within the project folder `project/Dependencies.scala` can create a one stop shop when you want to add and update dependencies.

```

//Dependencies.scala
import sbt._

object Dependencies {
  // Versions
  lazy val akkaVersion = "2.3.8"
  lazy val http4sVersion = "0.11.2"

```

```
// Libraries
val akkaActor    = "com.typesafe.akka" %% "akka-actor" % akkaVersion
val akkaCluster = "com.typesafe.akka" %% "akka-cluster" % akkaVersion
val specs2core  = "org.specs2"      %% "specs2-core" % "2.4.14"
val blazeServer = "org.http4s"      %% "http4s-blaze-server" % http4sVersion
val http4sDSL   = "org.http4s"      %% "http4s-dsl" % http4sVersion
val http4sCirce = "org.http4s"      %% "http4s-circe" % http4sVersion
val psql        = "postgresql"      % "postgresql" % "9.1-901.jdbc4"
val quill        = "io.getquill"    %% "quill-async" % "0.2.1"

// Projects
val http4s = Seq(
  blazeServer, http4sDSL, http4sCirce
)

val db = Seq(
  psql, quill
)
val backendDeps =
  Seq(akkaActor, specs2core % Test) ++ http4s ++ db
}
```

To use them within your sub-projects, you only need to specify them under that group's individual settings.

```
//root build.sbt
import Dependencies._

scalaVersion in Global := "2.11.7"

lazy val buildSettings = Seq(
  organization := "com.example",
  description := "the root description.",
  version := "1.0.0",
  scalaVersion := "2.11.7"
)

resolvers += Seq(
  Resolver.sonatypeRepo("releases"),
  Resolver.sonatypeRepo("snapshots")
)

lazy val core =
  (project in file("core"))
    .settings(buildSettings)

lazy val servicesa =
  (project in file("services-a"))
    .settings(buildSettings)
    .dependsOn(core)
    .aggregate(core)

lazy val servicesb =
  (project in file("services-b"))
```

```
.settings(buildSettings)
.settings(libraryDependencies ++= backendDeps)
.dependsOn(core)
.aggregate(core)
```

Using this technique ensures that only the dependencies that you want for a sub-project are included per project.

Testing in the Console

By placing tests in the corresponding test directories in either a single or multi-project, you can use SBT to run through those tests with ease. First, you need to get a testing library and add it as a managed dependency. For this example you're going to use `specs2`, since it comes standard in a few popular Scala frameworks, and also has many similarities to `scalatest`. Start by adding the dependency to the `Dependencies.scala` file under the root project folder.

```
libraryDependencies ++= Seq("org.specs2" %% "specs2-core" % "3.7" % "test")
```

With that dependency defined, you'll need to either reload or restart SBT so that the dependency can be loaded up. Now, assuming you have tests in the standard paths of your project, you can begin by using `test`, `test-only`, and `test-quick`.

The `test` task takes no arguments, but once executed it will traverse all projects and find any tests that are in the correct locations, and then it runs all tests. The `test-only` task is useful when you have a particular test you want to run.

test-only com.example.FactorialSpec

The `test-quick` task allows similar behavior to `test-only`, but only under the following circumstances:

- The tests that failed on the previous run
- The tests that were not run before
- The tests that have one or more transitive dependencies, maybe in a different project, recompiled

Remember, you can always prepend the test task execution with a tilda sig, which will re-test the specified tests after a change in your codebase.

Another nice fix that can enable a cleaner style within the testing side of your codebase is to filter on tests that adhere to a filename convention and only run those tests. For example, add the following to your build SBT:

```
testOptions in Test := Seq(Tests.Filter(s => s.endsWith("Spec")))
```

Generating Documentation

If you have been documenting your code using valid `scaladoc` syntax, you can use the SBT `doc` command to start generating Scala documentation in the target folder of your application. Running this

from the root project will also pull in the sub-project documentation as well. There are a few settings that can be helpful before generating the Scala documentation, which is discussed at length in Chapter 8.

RELEASE MANAGEMENT

Before packaging your application, you'll need to decide if you are going to use the project as a dependency in another project, or if you are creating an application that will be stand alone. Assuming the former, you can use the command `SBT package`, which will create a simple JAR that contains the main artifact of your package. Then add that new JAR file to the `lib` folder of your other project as an unmanaged dependency.

That approach is the best, since you're going to end up having to manually migrate that JAR file every time you want to create updates. A better solution is to use the command `sbt publish-local`. That will store the new JAR inside your ivy2 cache (if you want to publish to your local maven repo, you can use the command `publish-m2`) and can then be referenced by your other project, by adding the dependency entry into your SBT or dependencies Scala file.

If you want to build an executable JAR with all of the included dependencies of your project, you'll need to use the `sbt-assembly` plugin and then invoke the command `sbt assembly`. This is considered a fat/uber JAR, because it's an all in one solution for deployment. You can then run it by using the command :

```
java -jar your.jar com.example.MainMethod
```

Your new Scala project should now be running properly. Thanks JVM.

Deploying to Sonatype

The Sonatype Nexus is a repository manager. It gives you a place to store all of the packaged JARs you create, and gives you a single place to then share those JARs with other developers. You can also set up your own local sonatype repo for your company by following the steps here: <https://books.sonatype.com/nexus-book/reference/install.html>, but the following documentation is for deploying to the central sonatype nexus.

If you have already read over the documentation and terms of service provided at <http://central.sonatype.org/pages/ossrh-guide.html>, and you have received email notice from sonatype about provisioning your repo, you can start setting up SBT to deploy to your repo.

First, you need to PGP sign your artifacts for the sonatype repository, using the `sbt-gpg` plugin. To do this, add the following to your `~/.sbt/0.13/plugins/gpg.sbt` file:

```
addSbtPlugin("com.jsuereth" % "sbt-gpg" % "1.0.0")
```

This document assumes that you have already created a PGP key, and that you have sent the key to the keyserver pool.

Next, to publish to the maven repo, you'll need to add the settings:

```
publishMavenStyle := true
```

Add this to either the sub project or the root project. Then you'll need to add the repositories for pushing to sonatype:

```
publishTo := {
  val nexus = "https://oss.sonatype.org/"
  if (isSnapshot.value)
    Some("snapshots" at nexus + "content/repositories/snapshots")
  else
    Some("releases" at nexus + "service/local/staging/deploy/maven2")
}
```

The next step is getting the POM metadata that isn't generated by SBT into the build. This can be accomplished by using `pomExtra`:

```
pomExtra := (
  <url>http://sample.org</url>
  <licenses>
    <license>
      <name>MIT-style</name>
      <url>http://opensource.org/licenses/MIT</url>
      <distribution>repo</distribution>
    </license>
  </licenses>
  <scm>
    <url>git@github.com:sample/sample.git</url>
    <connection>scm:git:git@github.com:sample/sample.git</connection>
  </scm>
  <developers>
    <developer>
      <id>sample</id>
      <name>John Doe</name>
      <url>http://github.com/sample</url>
    </developer>
  </developers>
)
```

Finally, you need to add your credentials. This is normally handled from within the `~/sbt/0.13/sonatype.sbt`.

```
credentials += Credentials("Sonatype Nexus Repository Manager", "oss.sonatype.org",
  "<your username>", "<your password>")
```

Then, in SBT, you can run the `publishSigned` task that will package and deploy your application right from the interactive console. Assuming all goes well, you have just deployed your first artifact to Nexus.

Packaging with SBT-Native-Packager

The `sbt-native-packager` plugin can be useful, if not essential, in the packaging of your shiny new micro service, allowing all you need to package and deploy your services or applications to almost any environment. As of writing this you can deploy out to:

- Universal zip,tar.gz, xz archives
- deb and rpm packages for Debian/RHEL based systems
- dmg for OSX
- msi for Windows
- Docker images

Choosing which environment to deploy to will depend on your business needs. That stated, this chapter focuses on deployment to Debian packages and Docker instances, since both are fairly standard.

Creating a Debian Package

First, append to your `project/plugins.sbt` file the following:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-native-packager" % "x.y.z")
```

Then, from within your root `build.sbt` file, append:

```
enablePlugins(JavaServerAppPackaging)
```

It's worth mentioning that if you are currently using the Play framework, `sbt-native-packager` is already installed, but it will require that you add the following to create Debian packages:

```
enablePlugins(DebianPlugin)
```

Also, you will need to have installed the following system packages to provide native package creation:

- dpkg-deb
- dpkg-sig
- dpkg-genchanges
- lintian
- fakeroot

Once completed, you'll also have to make sure that the `build.sbt` for the sub project contains at the minimum the following lines:

```
name := "Services-A"  
version := "1.0"  
maintainer := "John Doe <jdoe@example.com>"  
packageSummary := "The greatest test service ever"  
packageDescription := "Lorem ipsum of ipsum Lorem"
```


After that final step you only need to restart SBT and give packaging your application a try. In the SBT console navigate to one of the sub project created earlier, such as `project services-a`, and use the command `debian:packageBin`. This will by default create the native Debian package, which can be installed using `dpkg`, located in the target directory of the subproject.

The above steps are great for getting a native build out, but in terms of customization, there are a few steps you can take to make the build a bit more informative. You can specify dependencies that your build will require by adding the following to your `build.sbt`:

```
debianPackageDependencies in Debian += Seq("java8-runtime", "bash (>= 2.05a-11)")
```

You can also recommend packages by adding the following:

```
debianPackageRecommends in Debian += "cowsay"
```

Finally you can hook into the debian package life-cycle by adding the `preinst`, `postinst`, `prerm`, or `postrm` hooks into your build SBT file:

```
import DebianConstants._
maintainerScripts in Debian := maintainerScriptsAppend(
  (maintainerScripts in Debian).value)(
  Preinst -> "echo 'Thanks for installing Service-A'",
  Postinst -> s"echo 'installed ${ (packageName in Debian).value }'"
)
```

Or you can add those hooks into your project's file structure:

```
src/debian/DEBIAN
```

You may also need to add specific extra definitions to your build. This can be accomplished by modifying your `build.sbt` and appending `bashScriptExtraDefines`:

```
bashScriptExtraDefines += """addJava "-Djava.net.preferIPv4Stack=true"""
```

Now that you've created a Debian package, you can follow similar instructions and create a Docker image.

Creating a Docker Image

Docker, as described in the documentation, “is an open platform for developing, shipping and running applications” through containers, which are a sandbox for images. To get a more general introduction to Docker you can visit the documentation: <https://docs.docker.com/engine/understanding-docker/> or visit <https://www.youtube.com/watch?v=aLipr7tTuA4>, which has a really easy introduction to the technology.

Similar to the setup for the debian packager, you'll need to modify the `build.sbt` for the sub-project to include:

```
enablePlugins(DockerPlugin)
```

Once completed, you'll need to set some default configurations in `build.sbt`:

```
packageName in Docker := "service-b"
version in Docker := "latest"
```

There are some other customizations that you can use, such as specifying the base image, the Docker repository, or the specific image customizations. You can view those settings at <http://www.scala-sbt.org/sbt-native-packager/formats/docker.html>, but here is a basic setup:

```
.settings(
  packageName in Docker := "service-b",
  version in Docker := "latest",
  NativePackagerKeys.dockerBaseImage := "dockerfile/java:oracle-java8",
  NativePackagerKeys.dockerExposedPorts := Seq(9000, 9443),
  NativePackagerKeys.dockerExposedVolumes := Seq("/opt/docker/logs"),
)
```

One issue that's common, especially when working with multi project setups and creating Docker containers, is that you may end up wanting to create a Docker image for the root project. By default the creation of the Docker image for the root project will aggregate and generate Docker images for all of the sub projects.

One way to get around this is to disable the aggregation in Docker from within the root project definition:

```
lazy val root = (project in file("."))
  .enablePlugins(DockerPlugin)
  //...(any other plugins or settings)
  .settings( //see above
    packageName in Docker := "root-project",
    version in Docker := "latest",
    NativePackagerKeys.dockerBaseImage := "dockerfile/java:oracle-java8",
    NativePackagerKeys.dockerExposedPorts := Seq(4000, 4443),
    NativePackagerKeys.dockerExposedVolumes := Seq("/opt/docker/logs"),
  )
  .dependsOn(servicesa).aggregate(servicesa)
  .dependsOn(servicesb).aggregate(servicesb)
  .dependsOn(core).aggregate(core)
  .settings( // the below will disable the subproject docker generation
    aggregate in Docker := false
  )
```

Publishing the Docker image can be broken down in one of two ways. `docker:publishLocal` will build the image using the local Docker server or `docker:publish`, which will do the same as `publishLocal`, and will then push the image to the configured remote repository.

Common SBT Commands

When you first start SBT, you can run standard tasks that come built into SBT. It's worth noting that you can use the `~` symbol to watch for changes within your code base.

Common Commands

- `clean`—Deletes files produced by the build
- `compile`—Compiles sources
- `console`—Starts the Scala interpreter

- `run`—Runs a main class, along with any command line arguments
- `test`—Executes all tests
- `testOnly`—Executes the tests provided as arguments
- `testQuick`—Executes the tests that failed before
- `update`—Resolves and optionally retrieves dependencies, producing a report
- `Reload`—Reloads the project in the current directory

REPL Commands

- `consoleProject`—Starts the Scala interpreter with the sbt build definition on the classpath and useful imports
- `consoleQuick`—Starts the Scala interpreter with the project dependencies on the classpath

Package Commands

- `package`—Produces the main artifact, such as a binary JAR, which is typically an alias for the task that actually does the packaging
- `packageBin`—Produces a main artifact, such as a binary JAR
- `packageDoc`—Produces a documentation artifact, such as a JAR containing API documentation
- `packageSrc`—Produces a source artifact, such as a JAR containing sources and resources

Documentation-Related Commands

- `doc`—Generates API documentation

Publish Commands

- `publish`—Publishes artifacts to a repository
- `publishLocal`—Publishes artifacts to the local Ivy repository
- `publishM2`—Publishes artifacts to the local Maven repository

Useful Plugins

- `sbt-dependency-graph` <https://github.com/jrudolph/sbt-dependency-graph>:
Creates a nice graph to visualize a project's dependencies. Really interesting for seeing where all of the cyclic references are in a project and getting an idea of why a project is taking a long time to compile.
- `sbt-revolver` <https://github.com/spray/sbt-revolver>:
When invoked, it creates a fork of the application in the background, which is great for fast background starting/stopping of applications and re-triggered starts. This is usually the first plugin added in this chapter to a new project, since it helps development move along quickly

and doesn't require an alternative like `dcevm`. It's worth noting that `Jrebel` is no longer supported by `sbt-revolver`.

- `tut` <https://github.com/tpolecat/tut>:

A documentation tool for Scala that reads markdown files and results in being able to write documentation that is typechecked and run as part of your build. The plugin really shines when creating tutorials that need to be typechecked by default.

- `sbt-updates` <https://github.com/rtimush/sbt-updates>:

Used to find updates for all of your project dependencies. This is great for Monday morning, when you want to check out all of the updates that are now available.

- `sbt-git` <https://github.com/sbt/sbt-git>:

Installing this plugin means never having to leave the interactive console to check in a file. More importantly, having `git` at your command from within SBT can allow you to write custom tasks from within SBT that interact with `git`.

- `sbt-musical` <https://github.com/tototoshi/sbt-musical>:

One of the just-for-fun plugins, and assuming you are on a Mac with iTunes open, you can use this plugin to play music whenever you want for any task. Simply add the `♪` prefix before any command.

- `ammonite-repl` <http://lihaoyi.github.io/Ammonite/>:

Another favorite plugin, this is truly a great way to experience the console in SBT or in standalone mode. Ammonite boasts the ability to dynamically load ivy dependencies/artifacts, multiline edits, pretty printed output, and much more. You should place this in your `global.sbt` file:

```
~/.sbt/0.13/plugins/build.sbt
```

- `sbt-release` <https://github.com/sbt/sbt-release>:

This is a customizable release process management. It is an all-in-one solution for setting up steps to distribute or manage releases. It can include release notes and check tests and it can do a number of remarkable things, all in an easy to digest DSL.

SUMMARY

You should now have a good understanding of how to use the Simple Build Tool with the Scala language to simplify complex behaviors within your project. SBT helps you package/publish JARs and also provides a powerful shell that can be used to test your code. You can see why SBT is the de facto build tool for compiling, testing, and packaging Scala services and applications. Take advantage of this popular tool and you will better enjoy your Scala experience.

5

Maven

WHAT'S IN THIS CHAPTER?

- Compiling and testing Scala with Maven
- Exploring the REPL
- Mixing Java and Scala at compile time
- Reducing development cycle times

While Scala has established itself as the de facto language of choice in certain areas, for example streaming analytics, many organizations still carry a technological or political legacy that prevents wholesale tooling and language transitions. In this situation incremental change is your ally. In concrete terms, you might not be able to use SBT and Scala, but if you are already working in a Maven based environment you can add Scala and learn how to coexist with Java source code in the same project with little disruption. Using Scala with Maven requires no additional installation activities, thereby preserving any current Jenkins builds you might have configured.

In this chapter, you will learn how to work with Maven to manage your Scala-based project. Along the way, you can see how Maven is extended to support Scala to a similar level as Java, how you can use Scala at an interactive prompt, and how you can slash compile times via configuration.

This chapter recognizes that no build lifecycle is complete without a dash of testing, so you will find out how to integrate ScalaTest with the assistance of Maven. You will learn how to leverage the power of compilation as a service, and see how Java and Scala sources can be combined in the same project.

GETTING STARTED WITH MAVEN AND SCALA

This chapter assumes you have prior experience working with Maven and you are familiar with the key notions of goals, phases, lifecycles, dependencies, plugin configuration, and the use of Maven from the command line. You should also be familiar with using Maven from your IDE.

If you need a Maven refresher, head off to <https://maven.apache.org> for a couple of hours and resume this chapter once you are up to speed. As a further preparation, you should have Maven installed, and be able to execute the command shown here.

```
$ mvn --version
Apache Maven 3.3.9
  (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-10T16:41:47+00:00)
Maven home: /usr/share/maven3
Java version: 1.8.0_60, vendor: Oracle Corporation
Java home: /usr/lib/jvm/java-8-oracle/jre
Default locale: en_GB, platform encoding: UTF-8
OS name: "linux", version: "3.13.0-63-generic", arch: "amd64", family: "unix"
```

Any version of Maven from 3.0.5 onward should be fine.

With these prerequisites met you are ready to work through the examples below to create a Scala aware Maven project.

Step 1. Start by adding a minimal POM at the root level of your project folder as shown below. Name the file `pom.xml`, which stands for Project Object Model.

```
<?xml version="1.0" encoding="UTF-8"?>
<project
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd
>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.scalacraft</groupId>
  <artifactId>professional-scala</artifactId>
  <version>0.1.0-SNAPSHOT</version>

  <properties>
    <scala.version>2.11.7</scala.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.scala-lang</groupId>
      <artifactId>scala-library</artifactId>
      <version>${scala.version}</version>
    </dependency>
    <dependency>
      <groupId>org.scala-lang</groupId>
```

```

        <artifactId>scala-compiler</artifactId>
        <version>${scala.version}</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>net.alchim31.maven</groupId>
            <artifactId>scala-maven-plugin</artifactId>
            <version>3.2.2</version>
            <executions>
                <execution>
                    <id>compile</id>
                    <goals>
                        <goal>compile</goal>
                        <goal>testCompile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
</project>

```

You should take a moment to run this. Only the final part of the output is shown next.

```
$ mvn clean install
```

```

[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.739s
[INFO] Finished at: Tue Jan 26 22:18:52 GMT 2016
[INFO] Final Memory: 11M/245M
[INFO] -----

```

```
Process finished with exit code 0
```

Provide some Scala source code to put this POM to work.

Step 2. Create a source directory layout as shown here.

```

$ mkdir -p src/main/scala/com/scalacraft/professionalscala/chapter5
$ mkdir -p src/test/scala/com/scalacraft/professionalscala/chapter5

```

NOTE Create this folder hierarchy using the tools of your choice, but if you are on Windows then an easy and non-disruptive route to gaining access to a compact Unix environment is offered by Git BASH, which is available for download from <https://git-for-windows.github.io/>. Equally you can use Powershell, CMD, or your IDE.

The first directory will contain your main application code, or the second test code. Now, enter the Scala class `Probe` as shown below, into the main source code hierarchy.

```
package com.scalacraft.professionalscala.chapter5

class Probe {
  def touchdown: Unit = println("Hello, New World!")
}
```

Step 3. Build the project and confirm some compilation took place.

```
$ mvn clean install

[INFO] BUILD SUCCESS

$ find target/ -name *.class
target/classes/com/scalacraft/professionalscala/chapter5/Probe.class
```

EXTERNAL MAVEN INSTALLATIONS

IDE bundled versions of Maven are used by default. Convenient though this is, to ensure the same experience on the command line as in the IDE, configure your IDE to use the external installation.

IDEA—File ⇄ Settings ⇄ Maven

Eclipse—Window ⇄ Preferences ⇄ Maven ⇄ Installations

Before you learn about the details of the Maven plugin that enables the compilation, be sure to jump onto the command line and spark up the Scala REPL via the plugin's console goal. Create an instance of `Probe` and send a message to it.

```
$ mvn scala:console
Type in expressions to have them evaluated.
Type :help for more information.

scala> import com.scalacraft.professionalscala.chapter5._
import com.scalacraft.professionalscala.chapter5._

scala> new Probe touchdown
Hello, New World!

scala>
```

EPFL we have a touchdown!

LOCATING THE SOURCE CODE

How does the plugin know where to find the source code? The plugin applies documented defaults for many configuration items, including the `mainSourceDir` property. The default values for all configuration items are documented at <http://davidb.github.io/scala-maven-plugin/plugin-info.html>. In the case of `mainSourceDir` the default is `${project.build.sourceDirectory}/../scala`. The `project.build.sourceDirectory` is the name of a property inherited implicitly from the Maven Super POM.

INTRODUCING SCALA-MAVEN-PLUGIN

By the end of this section you will have a working overview of the essentials of the `scala-maven-plugin`. Through running some examples, you will gain a compact, but practical overview.

The `scala-maven-plugin` is the de facto Maven plugin for working with Scala under Maven. Along one dimension it can be thought of as being an adapter sitting between Maven and the Scala compiler. However, that is not the sum of it. In later sections you will see how the plugin goes beyond a simple compiler adapter. First, let's examine an illustration of the adapter aspect. Follow these steps to start unpeeling the plugin behavior beginning with the link from POM configuration through to the Scala compiler.

Step 1. Add some configuration to the plugin execution using the `<configuration>` element as shown here.

```
<plugin>
  <groupId>net.alchim31.maven</groupId>
  <artifactId>scala-maven-plugin</artifactId>
  <version>3.2.2</version>
  <executions>
    <execution>
      <id>compile</id>
      <goals>
        <goal>compile</goal>
        <goal>testCompile</goal>
      </goals>
      <configuration>
        <displayCmd>true</displayCmd>
      </configuration>
    </execution>
  </executions>
</plugin>
```

When `displayCmd` is true, the command line used to invoke the Scala compiler is dumped to the Maven log.

Step 2. Build the project.

```
$ mvn clean install
```

Scrutinize the logging output until you locate the dumped command line. This is formatted below for clarity. The final line is of the most relevance for this discussion.

```
[INFO] cmd:
C:\Program Files\Java\jdk1.8.0_66\jre\bin\java
-Xbootclasspath/a:
C:\Users\jdb\.m2\repository\org\scala-lang\scala-library\2.11.7\
  scala-library-2.11.7.jar;
C:\Users\jdb\.m2\repository\org\scala-lang\scala-compiler\2.11.7\
  scala-compiler-2.11.7.jar;
C:\Users\jdb\.m2\repository\org\scala-lang\scala-reflect\2.11.7\
  scala-reflect-2.11.7.jar;
C:\Users\jdb\.m2\repository\org\scala-lang\scala-library\2.11.6\
  scala-library-2.11.6.jar;
C:\Users\jdb\.m2\repository\org\scala-lang\modules\scala-parser-combinators_2.11\
  1.0.4\scala-parser-combinators_2.11-1.0.4.jar;
C:\Users\jdb\.m2\repository\org\scala-lang\scala-library\2.11.4\
  scala-library-2.11.4.jar;
C:\Users\jdb\.m2\repository\org\scala-lang\modules\scala-xml_2.11\
  1.0.4\scala-xml_2.11-1.0.4.jar
-classpath
C:\Users\jdb\.m2\repository\net\alchim31\maven\scala-maven-plugin\3.2.2\
  scala-maven-plugin-3.2.2.jar
scala_maven_executions.MainWithArgsInFile scala.tools.nsc.Main
C:\Users\jdb\AppData\Local\Temp\scala-maven-1801724026178648094.args
```

You may have spotted the presence of multiple versions of the same libraries in the bootstrap JARs. This is not known to cause any problems. If you see this and are curious you can join the `scala-maven-plugin` community at <https://groups.google.com/forum/#!forum/maven-and-scala> and find out more.

The main class is `MainWithArgsInFile`, which calls `scala.tools.nsc.Main`, supplying the contents of an `args` file as the arguments to the compiler class in `scala.tools.nsc.Main`. This is shown below where you can see the plugin is handling the details of mapping the POM configuration to equivalent compiler options.

```
-classpath
C:\Users\jdb\.m2\repository\org\scala-lang\scala-library\2.11.7\
  scala-library-2.11.7.jar;
C:\Users\jdb\.m2\repository\org\scala-lang\scala-compiler\2.11.7\
  scala-compiler-2.11.7.jar;
C:\Users\jdb\.m2\repository\org\scala-lang\scala-reflect\2.11.7\
  scala-reflect-2.11.7.jar;
C:\Users\jdb\.m2\repository\org\scala-lang\modules\scala-xml_2.11\
  1.0.4\scala-xml_2.11-1.0.4.jar;
```

```
C:\Users\jdb\.m2\repository\org\scala-lang\modules\scala-parser-combinators_2.11\
  1.0.4\scala-parser-combinators_2.11-1.0.4.jar

-d
C:\Users\jdb\workspaces\main\professional-scala\target\classes
C:\Users\jdb\workspaces\main\professional-scala\src\main
  \scala\com\scalacraft\professionalscala\chapter5\Probe.scala
```

NOTE *The sole purpose of `MainWithArgsInFile` is to work around Windows command line size limitations.*

Exercise the link from POM to compiler once more.

Step 3. Add the compiler options shown here to the plugin configuration section.

```
<configuration>
  <args>
    <arg>-verbose</arg>
    <arg>-Xgenerate-phase-graph</arg>
    <arg>phase-graph</arg>
  </args>
  <displayCmd>true</displayCmd>
</configuration>
```

Step 4. Build the project.

```
$ mvn clean install
```

The logging output is excerpted below showing the effect of adding `-verbose`.

```
[INFO] [loaded package loader lang in 14ms]
[INFO] [loaded package loader annotation in 1ms]
[INFO] [promote the dependency of lazyvals: erasure => posterasure]
[INFO] [promote the dependency of explicitouter: tailcalls => specialize]
[INFO] Phase graph of 25 components output to phase-graph*.dot.
```

Step 5. Check the temporary `args` file to see the POM configuration passed through to the compiler command. The head of the `args` files is shown:

```
-verbose
-Xgenerate-phase-graph
phase-graph
-classpath
```

To see a compiler generated phase diagram, point your browser to <https://goo.gl/O7kGiv>.

This concludes the nickel tour of the `scala-maven-plugin` from a compiler adapter perspective. You now know how to dive through the layers from the POM to the compiler should the need ever arise. To see the full range of options applicable to the compiler visit <http://davidb.github.io/scala-maven-plugin/compile-mojo.html>. You will see further compiler options in use later.

NOTE *The `nsc` in `scala.tools.nsc.Main` stands for “New Scala Compiler.” What naming-related lessons can be drawn from history? In the world of Java there was a time when NIO stood for “New IO” but then it became “Non-blocking IO.” You have been warned about the ephemerality of “New” prefixes!*

ADDING LIBRARY DEPENDENCIES

Now that you have Maven compiling your code, pull in an additional Scala library by adding a new dependency that you will need for the REPL section. Follow these steps to add `atto`, an incremental text-parsing library, into your project as a dependency.

Step 1. Isolate the library version into the `<properties>` section as shown below.

```
<properties>
  <scala.version>2.11.7</scala.version>
  <atto.version>0.4.2</atto.version>
</properties>
```

Factoring out library versions into properties is an advisable practice from a maintenance point of view.

Step 2. Navigate to the `<dependencies>` section and add the new dependency shown here.

```
<dependency>
  <groupId>org.tpolecat</groupId>
  <artifactId>atto-core</artifactId>
  <version>${atto.version}</version>
</dependency>
```

Step 3. Build the project.

```
$ mvn clean install
```

You will see the POM and JAR for `atto-core` being downloaded and then the build will complete as before. All of this may appear somewhat pedestrian, and that is in fact the point. Using Scala libraries in Maven is exactly the same as using Java libraries.

NOTE *There are a number of good practices worth weaving into your Maven life. Entry-level best practices include using modules to organize projects, defining properties to factor out artifact versions, and depending on release versions of third-party libraries instead of snapshots. Search for “Maven: The Complete Reference” to get extensive documentation from the creators and maintainers of Maven.*

At times you may hear it said that the key best practice for Maven is to switch to SBT. You won’t see that option pursued further in this chapter for obvious reasons.

USING THE REPL

How else can the plugin power up your everyday development activities? One way is by providing frictionless access to the Scala REPL with a classpath matching your project. Build up your acquaintance with this powerful Scala feature by following these steps. Note that the Scala console is not a Maven feature. What you see in the following is how Maven understands your project dependencies when running the console via a plugin goal.

Step 1. In the root level of your project folder run Maven specifying the console goal, which is provided by the `scala-maven-plugin`. The console output has been edited for brevity.

```
$ mvn scala:console
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building professional-scala 0.1.0-SNAPSHOT
[INFO] -----
[INFO]
[INFO] --- scala-maven-plugin:3.2.2:console (default-cli) @ professional-scala ---
Welcome to Scala version 2.11.6 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_60).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Step 2. Type in these commands. Start with imports that use `atto-core`:

```
scala> import atto._
import atto._
scala> import Atto._
import Atto._
```

Now define a `val`. This is a parser combinator that parses a dot.

```
scala> val dot = char('.')
dot: atto.Parser[Char] = '.'
```

Next, build up a version number parser using predefined parsers from the library and your own `dot` parser.

```
scala> val versionParser = many(digit) ~ dot ~ many(digit)
versionParser: atto.Parser[((List[Char], Char), List[Char])] =
  ((many(digit)) ~ '.') ~ many(digit)
```

To cap it all off throw a string at the parser and review the output.

```
scala> versionParser parseOnly "2.11"
res0: atto.ParseResult[((List[Char], Char), List[Char])] =

  Done(((List(2),.),List(1, 1)))
```

The Scala console is a powerful ally and with Maven integration simplifying access, you will be able to explore new libraries interactively with no classpath management beyond the POM.

GETTING HELP

Help is available for the `scala-maven-plugin` through the standard Maven help plugin. Execute this command to see a summary of the plugin goals. Append `-Ddetail` to see an expanded version of the help. The results are summarized in Table 5-1.

```
$ mvn help:describe -Dplugin=net.alchim31.maven:scala-maven-plugin
```

TABLE 5-1: Plugin Goals

GOAL	DESCRIPTION
<code>add-source</code>	Add more source directories to the POM.
<code>cc</code>	Continuously compile the main and test sources. For use on the CLI only.
<code>cctest</code>	Continuously compile the main and test sources then run unit test cases. For use on the CLI only.
<code>compile</code>	Compile the main Scala sources.
<code>console</code>	Run the Scala console with all project classes and dependencies available.
<code>doc</code>	Produce Scala API documentation.
<code>doc-jar</code>	Create a jar of non-aggregated Scaladoc for distribution.
<code>help</code>	Display the Scala compiler help.
<code>run</code>	Run a Scala class using the Scala runtime.
<code>script</code>	Run a Scala script.
<code>testCompile</code>	Compile the test Scala sources.

RUNNING TESTS

You need to test your space probe before deploying it. Do this now via the `ScalaTest` Maven plugin. Follow the steps below to add the plugin, create, and run the smoke test.

Step 1. Add a dependency on `ScalaTest` to your POM. The details are shown below. Drop these changes into the appropriate POM locations.

```
<properties>
  <scala.version>2.11.7</scala.version>
  <atto.version>0.4.2</atto.version>
  <scalatest.version>2.2.6</scalatest.version>
</properties>

<dependency>
  <groupId>org.scalatest</groupId>
```

```

    <artifactId>scalatest_2.11</artifactId>
    <version>${scalatest.version}</version>
  </dependency>

```

Step 2. Configure scalatest-maven-plugin by adding the plugin configuration shown below. Position this `<plugin>` element after the existing `scala-maven-plugin`.

```

<plugin>
  <groupId>org.scalatest</groupId>
  <artifactId>scalatest-maven-plugin</artifactId>
  <version>1.0</version>
  <configuration>
    <reportsDirectory>
      ${project.build.directory}/surefire-reports
    </reportsDirectory>
    <junitxml>.</junitxml>
    <filereports>${project.artifactId}.txt</filereports>
    <!-- W: Suppress ANSI color codes -->
    <!-- T: Failed test reminders with short stack traces -->
    <stdout>WT</stdout>
  </configuration>
  <executions>
    <execution>
      <id>test</id>
      <goals>
        <goal>test</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

Step 3. Enter the unit test shown here into the test source code hierarchy.

```

package com.scalacraft.professionalscala.chapter5

import org.scalatest.{FlatSpec, Matchers}

class ProbeSpec extends FlatSpec with Matchers {

  behavior of "A Probe"

  it should "touchdown without puffing out smoke" in {
    new Probe touchdown
  }
}

```

Step 4. Build the project

```
$ mvn clean install
```

The test report will complete. Your report will be similar to this one.

```

Run starting. Expected test count is: 1
ProbeSpec:
A Probe

```

```
Hello, New World!
- should touchdown without puffing out smoke
Run completed in 285 milliseconds.
```

With your probe now trundling toward the launch pad, let's move onto other matters.

NOTE *Before you can have certainty in finding failure with tests, you must test for failure with certainty.*

Exactly what does that mean? Make sure the tests you think are running actually are running. In this case a simple way to achieve this is to temporarily throw a spanner into your probe by modifying the touchdown method to throw an exception.

```
def touchdown: Unit = ???
```

Run the test, and confirm that your unit test failed, then revert the method. This protects against going forward with a misconfigured POM that is not running tests. Under a test driven development methodology this step is superfluous.

JOINT COMPILATION WITH JAVA

Scala is interoperable with Java and you may find you have a requirement to mix Java and Scala source code in the same Maven project. In this case you will be well served by the joint compilation capabilities of the Scala compiler, which is configurable using `scala-maven-plugin`. Follow these steps to get a feel for the joint compilation set up. Along the way you will encounter the following players:

- Transmitter: Java interface
- Probe: Scala class extending Java interface
- CoreProbe: Java class extending Scala class

Step 1. Create a source directory for your Java code.

```
$ mkdir -p src/main/java/com/scalacraft/professionalscala/chapter5/
```

Step 2. Add a Java interface in this directory as shown here.

```
package com.scalacraft.professionalscala.chapter5;

public interface Transmitter {
    byte[] transmit();
}
```

Step 3. Implement the Java interface in the Scala Probe class following this example here.

```
class Probe extends Transmitter {
    def touchdown: Unit = println("Hello, New World!")
}
```



```

    override def transmit(): Array[Byte] = (0xda7a * 0xfeed).toString getBytes
  }

```

Step 4. Build the project and confirm all classes were compiled to bytecode.

```

$ find -name *.class
./target/classes/com/scalacraft/professionalscala/chapter5/Probe.class
./target/classes/com/scalacraft/professionalscala/chapter5/Transmitter.class
./target/test-classes/com/scalacraft/professionalscala/chapter5/
  ProbeSpec$$anonfun$1.class
./target/test-classes/com/scalacraft/professionalscala/chapter5/ProbeSpec.class

```

At this point you have a Scala class implementing a Java interface. What happens if you then have a requirement to extend a Scala class with a Java class? Try it.

Step 5. Add `CoreProbe` in the Java source directory as shown below.

```

package com.scalacraft.professionalscala.chapter5;

public class CoreProbe extends Probe {}

```

Building the project at this point ends in a compilation error.

```

[ERROR] COMPILATION ERROR :
[ERROR] \professional-scala\src\main\java\com\scalacraft\professionalscala\
chapter5\CoreProbe.java:[3,31] error: cannot find symbol

```

This error occurs because by default `maven-compiler-plugin` is executed before `scala-maven-plugin`. To unlock the benefits of joint compilation, you need to instruct Maven to execute the Scala compiler before the Java compiler. Follow the remaining steps to achieve the utopia of joint compilation.

Step 1. Modify the `<executions>` element of the `scala-maven-plugin` to stipulate the lifecycle phase to which the two Scala compilation goals are tied. The exact code to achieve this is shown below.

```

<plugin>
  <groupId>net.alchim31.maven</groupId>
  <artifactId>scala-maven-plugin</artifactId>
  <version>3.2.2</version>
  <executions>
    <execution>
      <id>compile</id>
      <phase>process-resources</phase>
      <goals>
        <goal>compile</goal>
      </goals>
    </execution>
    <execution>
      <id>test-compile</id>
      <phase>process-test-resources</phase>
      <goals>
        <goal>testCompile</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

The criteria for selecting the process-resources phase are twofold:

1. Must be before the compile phase to ensure precedence over the Java compile goal.
2. Must be as close to the compile phase as possible to preserve the semantics of the build life-cycle as much as possible.

Step 2. Confirm the project builds and check for the expected class files.

```
$ find -name *.class
./target/classes/com/scalacraft/professionalscala/chapter5/CoreProbe.class
./target/classes/com/scalacraft/professionalscala/chapter5/Probe.class
./target/classes/com/scalacraft/professionalscala/chapter5/Transmitter.class
./target/test-classes/com/scalacraft/professionalscala/chapter5/
  ProbeSpec$$anonfun$1.class
./target/test-classes/com/scalacraft/professionalscala/chapter5/ProbeSpec.class
```

You now have the option to mix Java and Scala freely within the same project. Pause to reflect on the options this brings. You can now test Java code with `ScalaTest`.

ACCELERATING COMPILATION WITH ZINC

Zinc is a Lightbend project that provides access to compilation as a service. This can reduce your project compilation time significantly. Zinc is based on the incremental compiler that is part of SBT and `scala-maven-plugin` supports easy integration with the zinc service. In this section, you will work through the steps required to install and start the zinc server, and then configure your project to use the service. At the end are sample compile times charted to show the difference between compiling with and without zinc.

Step 1. Download and install zinc to `~/Apps` or some other directory of your choice. Zinc can be downloaded from <https://github.com/typesafehub/zinc>.

Step 2. Start zinc as shown here:

```
$ ~/Apps/zinc-0.3.9/bin/zinc -nailed -scala-home ~/lib/scala-2.11.7 -start
```

The `-nailed` option runs the service as a daemon. You should specify the location of a Scala distribution using the `-scala-home` option. Starting zinc only needs to be done once.

Step 3. Update the `scala-maven-plugin` compiler configuration to delegate compilation to the zinc server. Follow the configuration below. The pertinent options are `recompileMode` and `useZincServer`. Both options must be present for the zinc server to be used.

```
<configuration>
  <recompileMode>incremental</recompileMode>
  <useZincServer>true</useZincServer>
</configuration>
```

Step 4. Build your project and look at the compilation logging. It will be similar to the logging output below.

```
[INFO] Using zinc server for incremental compilation
[warn] Pruning sources from previous analysis, due to incompatible CompileSetup.
```

```
[info] Compiling 30 Scala sources to /home/jdb/workspaces/main/professional-scala
/target/classes...
[warn] there were 5 deprecation warnings; re-run with -deprecation for details
[warn] there was one feature warning; re-run with -feature for details
[warn] two warnings found
[info] Compile success at 31-Jan-2016 21:02:19 [1.602s]
```

So is it faster? Your mileage may vary, but for your convenience Figure 5-1 charts the time in seconds to compile the project from Chapter 8 with and without zinc. The quicker times are zinc times.

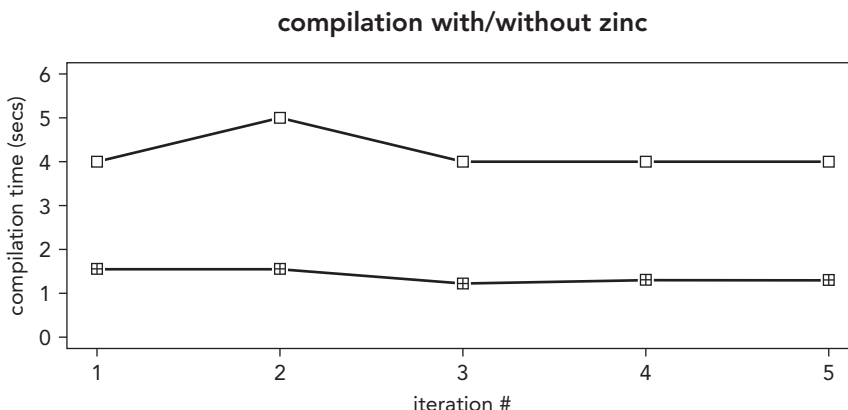


FIGURE 5-1

SUMMARY

Maven supports building mixed Scala and Java projects courtesy of the Scala Maven plugin. This plugin has eleven goals that provide support for compilation, documentation, testing, project configuration, scripting, and a project aware REPL.

A minimum viable POM that gets your Scala project building can be defined with a handful of XML elements. This configuration can be augmented to perform a range of tasks spanning documentation generation, unit test integration and more.

The Scala Maven plugin delegates to the Scala compiler in a way that you can pick apart should the need arise. The plugin provides convenient defaults with many configuration options allowing you to tailor the build to your scenarios. Some configuration relates to Maven while other configuration is specific to the compiler or scaladoc.

With a couple of config additions you can start to win back compilation time by throwing your sources onto the Zinc compiler. Other build tools for Scala include SBT, Gradle, and Pants Build System. These tools are in many ways improvements over Maven. But Maven enjoys the advantage of wide currency within the existing Java development world—a key advantage in a change adverse culture.

6

Scala Style/Lint

WHAT'S IN THIS CHAPTER?

- Understanding Scala style tools
- Using Lint tooling
- Mastering tools of the trade

Code smell is often defined “as certain structures in code that indicate violation of fundamental design principles and negatively impact design quality.” Code smells are not bugs, but instead are indications that the structure of code hasn’t been fully fleshed out, or was rushed to accommodate features at the cost of code quality. These are indications of a weakness in design and must be handled with a good eye, equally good coding habits, and help from strong style/lint tooling.

This translates to testing your Scala application, meeting its functional and business requirements, and also working to ensure that your code has no structural problems. Luckily, the open source world has created some extremely well programmed and maintained styling/lint tooling for Scala. You can use them to increase the reliability and efficiency of your applications/services, in effect allowing your code over its life cycle to become easier to maintain, understand, and extend.

This chapter takes you through the setting up of the most popular tools for styling and linting. It will also show you how to automate linting, and give your code the best possible chance to stay within current coding conventions.

SCALA WITH STYLE

The first tool to integrate into this book’s sample project is a little plugin called scalastyle (<http://www.scalastyle.org/>). Scalastyle “examines your Scala code and indicates potential problems with it.” This can become very beneficial, since new code and more sections

are added. To start working with scalastyle, your best bet is to install the SBT plugin by adding the following to your `project/plugins.sbt`:

```
addSbtPlugin("org.scalastyle" %% "scalastyle-sbt-plugin" % "0.8.0")
```

You'll need to then run the following command to generate the `scalastyle-config.xml` into the root directory:

```
sbt scalastyleGenerateConfig
```

After that file has been generated you can take a look into it and note that any of the rules can be modified to fit your individual needs. For a full breakdown of those rules, you can check out the Scalastyle: Implemented Rules page located here: <http://www.scalastyle.org/rules-0.8.0.html>. You can now run `sbt scalastyle`, which will generate a `target/scalastyle-result.xml` that also gives back feedback in the CLI for any style errors that arise during the check.

One of the nicer features of scalastyle is the ability to share a single scalastyle configuration file across multiple projects by supplying a `scalastyleConfigUrl` in your build SBT. Typically, you can use the `s3cmd` to connect to a private s3 bucket, which can be set up similar to the example from the scalastyle documentation:

```
lazy val updateScalaStyle = taskKey[Unit]("updateScalaStyle")

updateScalaStyle := {
  val secretKey = sys.env.get("S3_SECRET_KEY")
  val accessKey = sys.env.get("S3_ACCESS_KEY")
  val configUrl = "s3://bucket_name/configs/scalastyle-config.xml"
  (secretKey, accessKey) match {
    case (Some(sk), Some(ak)) =>
      val result: Int = file("target/scalastyle-config.xml") #< s"s3cmd
        --secret-key=$sk --access-key=$ak get $configUrl" !

    case _ =>
      println(s"Was unable to retrieve secretKey: $secretKey or accessKey:
        $accessKey from system variables")
  }
}
```

This will attempt to grab the Scala style configuration file from s3 when you execute the `updateScalaStyle` task. You can also automate the task during a compile or test run by adding the following:

```
(scalastyle in Compile) <=< (scalastyle in Compile) dependsOn updateScalaStyle

(scalastyle in Test) <=< (scalastyle in Test) dependsOn updateScalaStyle
```

Another customization feature that is available to you is the ability to create your own rules and add them inside the `scalastyle-config.xml`. To accomplish this it can be helpful to create a utility class within the project using scalastyle.

SCALIFORM

Scaliform is a code formatting utility, which can be used to keep all of your code in a single style. With a wide range of configuration options, any Scala project can immediately benefit by adding this dependency. The library can be used as a standalone CLI tool available here: <https://github.com/scala-ide/scalariform>. Or it can be integrated into SBT through a plugin by adding the following to your projects build SBT file:

```
resolvers += "Sonatype OSS Releases" at "https://oss.sonatype.org/service/
  local/staging/deploy/maven2"

addSbtPlugin("org.scalariform" % "sbt-scalariform" % "1.6.0")
```

After this, you can start up SBT and run the following:

```
sbt scalariformFormat
```

This will go through the code in your project and format according to the default settings of scaliform. Usually it's worth some discussion within your team before applying styling and coming to a consensus for which advanced options you may want to include in the code base for a meaningful discussion.

The below advanced configuration has resulted in the least amount of bikeshedding, since the styling is conservative enough not to raise many issues.

```
//build.sbt
import scalariform.formatter.preferences._
import com.typesafe.sbt.SbtScalariform

SbtScalariform.scalariformSettings

ScalariformKeys.preferences := ScalariformKeys.preferences.value
  .setPreference(AlignSingleLineCaseStatements, true)
  .setPreference(DoubleIndentClassDeclaration, true)
  .setPreference(placeScaladocAsterisksBeneathSecondAsterisk, true)
  .setPreference(IndentLocalDefs, true)
  .setPreference(IndentPackageBlocks, true)
  .setPreference(IndentSpaces, 2)
  .setPreference(MultilineScaladocCommentsStartOnFirstLine, false)
```

The phrase `SbtScalariform.scalariformSettings` will by default start formatting your project when the compile task or `test:compile` is run. If you only want to use Scalariform when invoked, make sure you use `defaultScalariformSettings` instead. This can be helpful if you don't want to have your code restructuring itself after every compile. That said, when using editors like vim or sublime, it can be a reassuring sight to see your code automatically reformatting itself after you save while having a watch on the compile task.

SCAPEGOAT

Another popular lint tool for your arsenal is Scapegoat, a younger project than scalastyle, but with some great linting operations (and a huge list of 107 inspections) that can benefit any code base. Scapegoat will find issues with code and report them to the console and generate html/xml reports that can give you some good feedback in your browser.

To install, you only need to add the autoplugin to your project/plugin.sbt:

```
addSbtPlugin("com.sksamuel.scapegoat" %% "sbt-scapegoat" % "1.0.4")
```

Now specify the version in your build via:

```
scapegoatVersion := "1.1.0"
```

Now you only need to invoke the scapegoat task in SBT to have it generate the above reports, which you'll find in target/scala-2.11/scapegoat-report. The scapegoat task by default will regenerate those reports after each invocation, but you can customize the report generation to only account for changes between runs by changing the setting `scapegoatRunAlways` to false.

One of the nice features of Scapegoat is the ability to suppress a warning by using the `java.lang.SuppressWarning` annotation around any method that you want the linter to skip over. This can be beneficial when you're aware that the code you're writing is breaking the rules, but that it is for the greater good.

```
Class Temp {  
  @SuppressWarnings(Array("BigDecimalDoubleConstructor"))  
  def test() = {  
    val x: BigDecimal = BigDecimal(1.2)  
    x  
  }  
}
```

The reason to like scapegoat is because it does one thing really well. It checks your code for structural errors and announces them loudly when you run the scapegoat task. It gives you immediate feedback, which is good to have on any project that doesn't already have any structural analysis tooling.

WARTREMOVER

Probably the easiest to customize out of the lint tools reviewed so far, WartRemover is a stable inclusion into your linting stack. First install the SBT plugin:

```
addSbtPlugin("org.brianmkenna" % "sbt-wartremover" % "0.14")
```

When running WartRemover for the first time, all of the checks and patterns are turned off by default. This allows you to slowly start leveling up the heat on what "warts" you want to start verifying and fixing. You can do this by adding a small configuration to the build.sbt.

```
wartremoverErrors += Warts.unsafe
```

Then it's only a matter of running `run` or `compile` and getting feedback from the compiler. If you want to ensure that things were configured properly, run the task `wartremoverErrors`, which shows you a list of warts that have been loaded into SBT. You can also add a quick wart to your code to test that the auto-plugin has hooked in properly:

```
var i = 100
```

Now compile and watch WartRemover do its magic.

Aside from the quick setup of WartRemover, you can take a quick tour of the current list of warts by visiting <https://github.com/puffnfresh/wartremover>, or start creating your own. Following the example provided by the WartRemovers github page, let's break down what it's doing to create your own:

```
//full example
import org.brianmckenna.wartremover.{WartTraverser, WartUniverse}

object Unimplemented extends WartTraverser {
  def apply(u: WartUniverse): u.Traverser = {
    import u.universe._
    import scala.reflect.NameTransformer

    val notImplementedName: TermName = NameTransformer.encode("???)
    val notImplemented: Symbol = typeOf[Predef.type].member(notImplementedName)
    require(notImplemented != NoSymbol)
    new Traverser {
      override def traverse(tree: Tree) {
        tree match {
          case rt: RefTree if rt.symbol == notImplemented =>
            u.error(tree.pos, "There was something left unimplemented")
          case _ =>
        }
        super.traverse(tree)
      }
    }
  }
}
```

During execution of the above code, WartTraverser is going to check out any code that has a method that is implementing `???` and throws an error with the line stating “There was something left unimplemented.” If you are not familiar with the Scala reflection libraries, you can get some background by checking out the Scala documentation here: <http://docs.scala-lang.org/overviews/reflection/symbols-trees-types.html>. This will give you some insight into the `require` statement from the above code snippet:

```
require(notImplemented != NoSymbol)
```

`NoSymbol` is “is commonly used in the API to denote an empty or default value.”

```
val notImplementedName: TermName = NameTransformer.encode("???)
val notImplemented: Symbol = typeOf[Predef.type].member(notImplementedName)
require(notImplemented != NoSymbol)
```


This is working as a guard to detect that the method contains that ??? symbol. While this can be a little confusing when you first start to implement your own warts or anti-pattern checks in your code base, it can become highly beneficial.

SCOVERAGE

Statement coverage is a white box testing technique, which involves the execution of all the statements at least once in the source code. It is a metric, which is used to calculate and measure the number of statements in the source code that have been executed. Why would you want that in Scala? Often in your Scala statements you may end up creating a truly magnificent one liner. It is one of those one-liners that may impress your friends, like so (adapted from http://rosettacode.org/wiki/FizzBuzz#One-liner_geek):

```
def fizzBuzz(): Seq[String] = for (i <- 1 to 100) yield(Seq(15 -> "FizzBuzz",
  3 -> "Fizz", 5 -> "Buzz").find(i % _. _1 == 0).map(_. _2).getOrElse(i.toString))
```

Since so much is going on in this line, getting back information about if this line is covered through a test is nearly impossible. You should split the functionality up and create a unit test that is able to test each of the sections. However, thanks to Scoverage you won't have too.

Basically, all you have to do is write regular unit tests! Then, Scoverage (a tool that offers aforesaid statement and branch coverage) can be used to give you a report of how much of your code base has been covered.

The setup is straightforward. First, you add:

```
addSbtPlugin("org.scoverage" % "sbt-scoverage" % "1.3.5")
```

Then you go into the SBT console and type **coverage** and then type **test**, which will generate a report of your code base with percentage of complete coverage for each file. The green represents coverage and the red represents missing coverage. Depending on how many unit tests you've used while developing your project, this will reflect that commitment in the report.

One of the nice things with Scoverage is that it has integrations with Sonarqube (<http://www.sonarqube.org/#>), which is a “central place to manage code quality.” If you haven't used it before, it can be seen in the demo here: <https://nemo.sonarqube.org/>. Sonarqube is a really nice way to start getting visibility on your code coverage and a great way to show off just how much technical debt you've been able to crunch through.

SUMMARY

This chapter has covered many of the most popular styling/lint tooling to be used with Scala. These tools come from the open source world, and they of course increase the efficiency of your Scala applications and services. Be sure to try these tools, whether it is Scaliform, Scapegoat, WartRemover, or Scoverage, since they will make your Scala code easier to maintain.

7

Testing

WHAT'S IN THIS CHAPTER?

- Introducing Scala testing frameworks
- Using property based testing
- Tapping into unit testing
- Applying integration testing
- Mastering mocking services

Learning to write tests is one of the foundations that leads to a reliable and maintainable code base. With frameworks like `ScalaTest`, `Spec2` and other mature frameworks, this chapter examines some of the most popular ways to create tests that can provide value to the longevity of your code.

Let's quickly review the common terminology for the categories of tests that are covered:

- **Unit Tests:** Single pieces of code that you want to test, which are usually best for identifying failures in functions and algorithms.
- **Integration Tests:** Shows the major system components working together properly. There are no mock objects, and everything is real as it would be in the live system.
- **Acceptance Tests:** Basically these are “your feature stories,” asking yourself if you built the right thing based on the business requirements.
- **Property Tests:** Tests the behavior of a function using generated inputs to “stress test” the validity of a piece of code.
- **Test-Driven Development (TDD):** writing tests ahead of the feature (that's it!), which is handy when you want to describe the actions of a feature before you start implementing. You then rely on refactoring to create a suite of tests at a low level of regression.

- **BDD (Business-Driven Development):** writing tests in a more natural semantic way. Really it's the same as TDD, with the test descriptions being as human readable as possible, as well as a stronger focus on the “feature” rather than the function.
- **DDD (Domain-Driven Development):** Is a software development approach that attempts to bridge a project's core domain and logic with collaboration between technical and domain experts.
- **ATDD (Acceptance Test-Driven Development):** A methodology similar to TDD, but different.

These glossary terms are now out of the way. The world of testing in Scala is fairly large and it takes a good deal of time to understand what frameworks your code base may benefit from. Even if you don't end up using all of the techniques, it's always beneficial to give them a try. Each test you write for yourself and your team will help you learn the functionality and limitations of your code base. At the end of the day this will help you and your team grow. So don't get frustrated!

SCALATEST

ScalaTest (<http://www.scalatest.org/>) is a flexible and popular testing framework for both Java and Scala. It takes the best parts of Cucumber, while integrating with a rich feature set of tools to extend SBT, Maven, scalaCheck, IntelliJ, and the list goes on to accommodate almost any testing requirement. It even provides extensions to write in all of the popular testing methodologies for you or your team, no matter what the previous background. Are you used to writing tests from Ruby's Rspec tool? You can use FunSpec and experience the same advantages. Coming from testing in Play using specs2? You can use WordSpec to simulate the same advantageous testing style.

The biggest bonus of using ScalaTest is that you have freedom to work out what standard you want to start writing your tests in without having to download a ton of different libraries and testing each individually. With ScalaTest it's as easy as plugging in the dependency and getting to work writing tests to verify your code base. To get started, add the following dependencies to your build.sbt file or Dependencies.scala file:

```
libraryDependencies += Seq("org.scalactic" %% "scalactic" % "2.2.6",
  "org.scalatest" %% "scalatest" % "2.2.6" % "test")
```

It's recommended to include the “SuperSafe Community Edition” Scala compiler plugin, which “will flag errors in your ScalaTest (and Scalactic) code at compile time.” It's worth noting that scala 2.11.7 does have some of that functionality included, which can be added to the file ~/.sbt/0.13/global.sbt:

```
resolvers += "Artima Maven Repository" at "http://repo.artima.com/releases"
```

Finally, add the autoplugin to your project's plugins.sbt file:

```
addSbtPlugin("com.artima.supersafe" % "sbtplugin" % "1.1.0-RC6")
```

That's it for the setup. Be sure to read from their website about the subject: http://www.scalatest.org/user_guide/selecting_a_style.

UNIT TESTS

Unit tests are the zerg of the testing world. They are to be used as much as possible for testing individual chunks of code with little variance between input and output. As such, if bugs are found from quality assurance, it's usually a good idea to write down the bug as a unit test and solve it so that in the future you can be sure that any new features don't end up breaking a previously found issue. For example, you can setup a simple unit test using the `funSpec` like the following:

```
class ExampleSpec extends FunSpec {
  describe("Adding 1 to 1") {
    it("should equals 2"){
      assert(1+1 == 2)
    }
  }
}
```

This asserts that 1 plus 1 equals 2, but with matchers you can use `should be` instead of `assert`:

```
class ExampleSpec extends FunSpec with Matchers{
  describe("Adding 1 to 1") {
    it("should equals 2"){
      1+1 should be(2)
    }
  }
}
```

Or you can test for an error being thrown:

```
"A number error" should "throw a divide by zero" in {
  a[java.lang.ArithmeticException] should be thrownBy {
    1 / 0
  }
}
```

You can also test for type inheritance:

```
sealed trait Animal
sealed trait Mineral
object Cat extends Animal
"An object" should "be inherit properly" in {
  val cat = Cat
  cat shouldBe a [Animal]
  cat should not be a [Mineral]
}
```

There are a plethora of other small tests you can perform, but the main thing to get out of unit tests is that they are a fantastic way to get one-to-one matches for expected outputs of a function.

INTEGRATION TESTING

Integration testing is an umbrella term that can encompass data-driven tests, performance tests (benchmarking like `ScalaMeter`: <https://scalameter.github.io/>), and acceptance tests that use

Selenium DSL. Each one gives you a better understanding of your system at sub production levels, and can be insightful given certain criteria.

Data-Driven Tests

As stated earlier, these are tests that hook into data to throw them against a system in your code-base. Consider the object `HorseOfCourse`:

```
object HorseOfCourse{

  def isHorseOrMrEd: Animal => String = {
    case PaintHorse => "horse"
    case MrEd => "mr ed"
    case _ => "not a horse or mr ed"
  }

  def apply(x: Animal) = isHorseOrMrEd(x)
}
```

You can set up a few tests that ensure the proper test validation for horses and Mr. Ed:

```
val allHorses = Array(PaintHorse, PaintHorse, MrEd)

"A Horse" should "always be a horse" in {
  for(horse <- allHorses.filter(_ == PaintHorse)){
    horse should be (PaintHorse)
  }
}

it should "unless its the famous Mr. Ed" in {
  val mrEd = allHorses contains MrEd
  mrEd should be (true)
}

it should "understand the difference between horses" in {
  for(horse <- allHorses){
    val isItAHorse = HorseOfCourse(horse)
    horse match {
      case PaintHorse => isItAHorse should be("horse")
      case MrEd => isItAHorse should be("mr ed")
      case _ => isItAHorse should be("not a horse or mr ed")
    }
  }
}
```

These sorts of tests can be bit cumbersome, since you build up more input from end users. However, that's part of the benefit, since you're using all that user input to then drive the tests in your suite. Along those same lines, property testing has huge value when creating data-driven tests in lieu of real data from logs or QA, since property tests can generate the inputs, leaving you only having to worry about the tests and underlying code (and possibly tweaking the generator). Let's take a quick look over property testing in `ScalaTest`. Given a simple class that generates the sum of a series:

```
import scala.language.postfixOps

object MathFun {
  def sumOfSeries(range: Int): Double = {
    1 to range map (x => 1.0 / (x * x)) sum
  }
}
```

Using this simple sum, you can then create a property test to determine if there are any holes in the logic.

```
import org.scalatest._
import org.scalatest.prop.GeneratorDrivenPropertyChecks

class MathFunSpec extends WordSpec with GeneratorDrivenPropertyChecks
  with Matchers {

  implicit override val generatorDrivenConfig = PropertyCheckConfig
    (minSuccessful = 3)

  "A series of numbers" should {
    forAll { (n: Int) =>
      whenever (n > 1) { MathFun.sumOfSeries(n) should be > 0.0 }
    }
  }
}
```

You'll note that the `minSuccessful` is set to a small value for a successful threshold, but this test should quickly fail. The sum of ranges has no protection for an extremely large number or a zero, neither of which we prepared for, which is the beauty of using property-based checking (this can also give your CPU a run for its money).

```
def sumOfSeries(range: Int): Double = {
  if(range <= 0) 0.0
  else if (range >= 32767) 0.0
  else {
    1 to range map (x => 1.0 / (x * x)) sum
  }
}
```

While these are not the best defaults, they will accommodate the edge cases that property checking has found.

Performance Testing

Usually, when it comes to micro benchmarking you shouldn't try to over optimize your code until the system/application has been completed. When it comes to all of the different libraries and approaches for getting something done in Scala, there should be no fundamental issue getting an idea of what the performance of each approach produces, even if those results vary to some degree based on hardware. To those ends, a micro benchmarking framework such as `ScalaMeter` (<https://scalameter.github.io/>) can be incredibly useful in finding inefficiencies in code, and help you start taking a granular approach to the logic you build in your systems.

To get started, add the following to your `build.sbt`:

```
resolvers += "Sonatype OSS Snapshots" at
  "https://oss.sonatype.org/content/repositories/snapshots"

libraryDependencies += "com.storm-enroute" %% "scalameter" % "0.8-SNAPSHOT"

testFrameworks += new TestFramework("org.scalameter.ScalaMeterFramework")

logBuffered := false
```

You can use the 0.7 ScalaMeter version if you prefer a stable version, but always check Maven, because these versions may have changed. You will also want to disable parallel execution in tests, which we usually disable in the SBT console before running benchmarking tests. You can also have the `build.sbt` file set to disable this by default using the following setting:

```
parallelExecution in Test := false
```

After that setup you can begin writing some inline benchmarking. A good feature that has been introduced in 0.7 is to wrap any function or method with the `measure` method and instantly get feedback on the performance of that code. This feature can be a great asset while writing code that can be executed. For a quick example, let's borrow for the ScalaMeter examples:

```
import org.scalatest.FunSuite
import org.scalameter._

class InlineBenchmarkTest extends FunSuite {
  test("Should correctly execute an inline benchmark") {
    val time = measure {
      for (i <- 0 until 100000) yield i
    }
    println(s"Total time: $time")
  }
}
```

Hopefully this gives you a small look into how to integrate ScalaMeter into your project, because that integration can give you some fantastic looks into bottlenecks and other performance-based issues early on in your coding.

Acceptance Testing

Acceptance testing is a methodology that also has concrete support in `ScalaTest`. Usually, the best way to think about acceptance testing is the same way you would think about black box testing, where you need to test an external API, a web page, or really anywhere you want to simulate actions in your system as if it were in production. Using a tool like Selenium is one way you can create a phantom user to interact with your application or external API—collect results and then provide feedback in terms of a real user running through the acceptance criteria of your application.

To start, you may want to take a look at the Selenium documentation from `ScalaTest` that is available (http://www.scalatest.org/user_guide/using_selenium). To get started using Selenium we'll need to modify the library dependencies in the `build.sbt` file.

```
libraryDependencies += "org.seleniumhq.selenium" % "selenium-java"
  % "2.35.0" % "test"
```

You can then reload SBT and compile it to fetch that dependency. Next, you can simulate an API to then test with Selenium. Let's use `http4s` with the blaze server to set up a quick API.

```
import org.http4s.HttpService
import org.http4s.dsl._
import org.http4s.server.blaze.BlazeBuilder

object Api extends App{
  val service = HttpService {
    case GET -> Root / "testRoute" =>
      Ok("This is a test route")
  }
  BlazeBuilder.bindHttp(8080)
    .mountService(service, "/")
    .run
    .awaitShutdown()
}
```

Go ahead and start that up in SBT and check it out in your browser by going to `http://localhost:8080/testRoute`, where you should see the phrase “This is a test route.” You can automate this test by creating the Selenium Test as follows:

```
import org.scalatest.{ShouldMatchers, FlatSpec}
import org.scalatest.selenium.HtmlUnit

class ApiSpec extends FlatSpec with ShouldMatchers with HtmlUnit {

  val host = "http://localhost:8080/"

  "This is a test route" should "should have the test content" in {
    go to (host + "testRoute")
    pageSource should be ("This is a test route")
  }
}
```

The DSL for Selenium makes it incredibly easy to read the logic within this test. Basically “go to” a URL, then check that the source is the proper phrase. While this is somewhat simplistic, you can do some fairly impressive testing. In the next example, you will do a search on Google for the rotten tomatoes score for *Groundhog Day*, use the `cssSelector` to find the rating field from the search results, and make sure it's about a 90% rating.

```
"This is a google search for Groundhog Day" should "should be greater than 90%"
in {
  go to "http://www.google.com"
  click on "q"
  textField("q").value = "Groundhog Day Rotten Tomatoes"
  submit()
  eventually {
    val rawRating = find(cssSelector("div.f.slp")).get.underlying.getText
    val rating = rawRating.replaceAll("\n", "").replaceAll(" Rating: ", "")
    rating should be > 90
  }
}
```


You'll need to add the `Eventually` trait, since the search takes a moment for `HtmlUnit` to complete.

```
import org.scalatest.concurrent.Eventually
```

You should now have an idea of just how easy it is to get some feedback for the internal and external APIs within your system, and you can start using Selenium to make sure the results you see are within normal working parameters.

Mocks

Given how common they are in testing, let's examine stubs and mocks. Stubs are a simulation of an object's behavior, and mocks are an expectation of what a stub will produce. With `ScalaTest` and a `mockito` dependency, you can achieve simple testing of traits, classes, and objects by "training" those asserts to see what expectations you can get out of them.

In terms of how stub/mock testing works in Scala, let's use the `mockito` library and implement some basic tests for a bowling league service.

First, add the following library dependency to the `build.sbt`:

```
"org.mockito" % "mockito-all" % "1.10.19"
```

Then, let's add the trait for the bowling league service:

```
trait BowlingLeague {  
  val leagueName: String  
  val leagueRules: LeagueRules  
  
  def addTeamToLeague(team: Team): Future[Boolean]  
  def removeTeamFromLeague(team: Team): Future[Boolean]  
  
  def teamStandings() : Future[Standings]  
}
```

You can start "mocking" out the adding and removing of leagues from the service:

```
class BowlingLeagueSpec  
  extends FunSuite  
    with ShouldMatchers  
    with MockitoSugar  
    with ScalaFutures {  
  
  val players: Seq[Player] = Seq(  
    Player("Charlie", "Brown", 100),  
    Player("Linus", "Pelt", 105),  
    Player("Lucy", "Pelt", 125),  
    Player("Sally", "Brown", 85)  
  )  
  
  val testTeam = Team(  
    "Pea Shooters",  
    players  
  )  
}
```

```

test("Adding a team") {
  val bowlingLeague = mock[BowlingLeague]

  when(bowlingLeague.addTeamToLeague(testTeam)).thenReturn(Future(true))

  val addTeam = bowlingLeague.addTeamToLeague(testTeam)

  addTeam.futureValue should equal(true)
}

test("Removing a team") {
  val bowlingLeague = mock[BowlingLeague]

  when(bowlingLeague.addTeamToLeague(testTeam)).thenReturn(Future(true))
  when(bowlingLeague.removeTeamFromLeague(testTeam)).thenReturn(Future(true))

  val remTeam = bowlingLeague.removeTeamFromLeague(testTeam)

  remTeam.futureValue should equal(true)
}
}

```

By creating a mock of the bowling service, you can create an expectation of what adding and removing a team from league methods does. In the above code, you add an expectation that adding a team will return a future of true. This is a great way to skip having a hermetic server that is actually connecting to a database for a bowling league service, and just ensures that your interface is working properly.

LOAD TESTING

Using load testing within your application can find memory leaks and slow code paths, and can help you find issues in your application before launching in production. Performance testing provides insight into how your application will perform under a controlled amount of stress. For load testing in Scala, let's examine the fantastic Gaitling project.

Gaitling is a load testing tool that allows you to write scenarios in Scala to stress test your application. While this product is typically used for testing the speed and efficiency of internal APIs, you can customize Gaitling for other protocols as well. To get started you'll need to add the Gaitling dependency to your SBT build:

```

"io.gatling.highcharts" % "gatling-charts-highcharts" % "2.1.7" % "test",
"io.gatling"             % "gatling-test-framework"   % "2.1.7" % "test"

```

Also, add the SBT plugin, by placing the following in project/plugins.sbt:

```
addSbtPlugin("io.gatling" % "gatling-sbt" % "2.1.5")
```

Then, much as we coded up the tests for Selenium, begin by creating the scenario for the load test:

```

import io.gatling.core.scenario.Simulation
import io.gatling.core.Predef._

```

```
import io.gatling.http.Predef._

import scala.concurrent.duration._

class ApiSim extends Simulation {
  val numUsers = 100
  val host = "http://127.0.0.1:8080"

  val httpConf = http.baseUrl(host)

  val scn = {
    scenario(s"testing the testResource ( $host )")
      .exec(
        http("testRouteSim")
          .get("/testRoute")
          .check(status.is(200))
      )
  }

  setUp(scn.inject(rampUsers(numUsers) over 10.seconds)).protocols(httpConf)
}
```

The above code is attempting to check the `/testRoute` for a 200 status code 100 times, and then reporting back the request time metrics along with other helpful information. This includes response time distribution, response time percentiles, the number of requests per second, and the path to the report that displays at the end of the simulations run. While expanding your own simulations, this provides a result similar to when we used `ScalaMeter`, which provides a benchmark for how much traffic your application can handle at a small level.

Load testing also helps you manage production levels of traffic in a simulated stage environment where you can set up “clusters” of gatling boxes to test the infrastructure. The only downside is that currently there is no way to correlate that information other than SCP’n the results to a central box or to your laptop, and then running the report against all of the logs generated by gatling.

SUMMARY

You now have a better understanding of the tools necessary to test your Scala code. This chapter covered the many benefits of using `ScalaTest`, and also examined the benefits of using unit testing, integration testing, data-driven testing, and performance and acceptance testing. We also examined stubs, which are a simulation of an object’s behavior; and mocks, which are an expectation of what a stub will produce. Load testing allows you to hunt down memory leaks and slow code paths prior to launching your app in production, and the Gaitling project is used in load testing. As with all other languages, it is important to test out all of your Scala code, and the tools described in this chapter will help.

8

Documenting Your Code with Scaladoc

WHAT'S IN THIS CHAPTER?

- Understanding the structure of Scaladoc documentation
- Applying wiki syntax and tags
- Generating API documentation from commented source code
- Publishing your API documentation on the web
- Learning advanced documentation techniques

Code documentation can take many forms, ranging from manually authored web pages, README.md files, and generated API docs to no documentation at all. The nature of the code and the intended audience have a role to play in choosing what is right for your project. Suppose you have decided to document your code using Scaladoc. What's next? In this chapter you will learn about how to use Scaladoc syntax and the associated tooling to create documentation from Scaladoc comments embedded in your source code.

To get started you need to know what can be documented and what the formatting and linking options are. You also need to be aware of the tooling options you can apply to generate the API docs. Once you have your documentation generated, you need to know what to do next to expose it to a wider audience.

You will also learn about the structure of Scaladoc, both at the source level and in the rendered form. What options exist for generating the documentation from the command line, or from Maven or SBT? What are the limits of wiki syntax? Examples in this chapter will illustrate these general rules.

WHY DOCUMENT YOUR CODE?

“Lack of documentation is becoming a problem for acceptance.”

—WIETSE VENEMA, CREATOR OF POSTFIX

Creating high quality useful documentation is a time consuming business. You might reasonably ask the question: Why bother? You have to decide where on the documentation continuum you want to land. Should you provide no documentation at all because the unit tests are adequate documentation? Or go with comprehensive documentation that has embedded REPL examples? Factors that feed into your decision include the type of project you are developing, the anticipated project lifecycle, any coding guidelines in force, the expected audience, and the cost of creating the documentation and taking it forward through project iterations. This section will help you make these decisions.

Revealing the Benefits

Scaladoc plays a critical role making your code accessible to others, particularly when the luxury of face-to-face contact is not an option. With a few concise words and pointed examples, you can orient the user and get them on a productive track. Scaladoc is an especially good choice for documenting library code. As an example of effective Scaladoc, take a look at the documentation for the Scala Standard Library Regex class (<http://www.scala-lang.org/api/current/#scala.util.matching.Regex>). With a handful of code examples linked with explanatory text, the class becomes immediately accessible.

Scaladoc is far from the only option for documentation. Some libraries flourish while eschewing Scaladoc almost entirely. Scaladoc sports the key advantage of being embedded in your code and supporting a simple wiki syntax. Nearly all documentation authoring can be accomplished without switching out of your IDE.

Bookending the Continuum

As a Scala developer you already have a good idea of what makes for useful documentation. This helps when you start authoring your own documentation.

Some projects, such as ScalaTest, are ahead of the documentation curve (<http://doc.scalatest.org/2.2.6/index.html#org.scalatest.FlatSpec>). But how is this layout achieved? You will find out later in this chapter. Let's first examine Scala documentation in general.

Choosing What to Document

Let's explore what type of documentation is useful. If you have coding guidelines that stipulate project documentation requirements then you'll want to use the remainder of this chapter to learn how to unleash the power of Scaladoc. If you have no guidelines, then consider creating a set of documentation objectives.

If, on the other hand, you are developing a library for others to use, you will need to determine if Scaladoc can form part of the adoption path for your end users. If you decide to take this route, later sections in this chapter cover the inclusion of examples in Scaladoc.

One final question to explore is whether you have the opportunity to add Scaladoc after development activities are complete. If the answer is no, then document the code incrementally as you develop it. An advantage of interlacing documentation and development is that there is less context switching. Even when your planned activities include a later documentation phase, you should evaluate bringing this forward and document as you code. The results are better and the net effort will be lower. Let's dive into Scaladoc and discover the many advantages of implementing it.

SCALADOC STRUCTURE

Scaladoc refers to three different things: the command line tool `scaladoc`, source code comments with wiki syntax and tags, and the generated documentation viewed in a browser. The meaning will normally be clear from the context. In this section you will see a breakdown of the structural elements of the generated documentation, and you will learn how to use the Scaladoc UI.

NOTE *Throughout this section you will be shown examples taken from the Scala Standard Library 2.11.7 API docs available at <http://www.scala-lang.org/api/>.*

Overall Layout

The generated Scaladoc is a collection of HTML pages that can be viewed with a browser. The files can reside locally or they can be published on the web. Options for web publication are covered later in this chapter.

These are the files and directory you see after generating the documentation using the `scaladoc` command:

```
$ ls -l scaladocs/  
com/  
deprecated-list.html  
index/  
index.html  
index.js  
lib/  
package.html
```

Navigate to <http://www.scala-lang.org/api/> and take a look at the overall layout of the page. You will see in Figure 8-1 how it is divided into two sections: the index pane and the content pane.



FIGURE 8-1

The index pane is where you search the documentation. The content pane displays documentation grouped by class, trait, object or package. The UI is unsurprising to use, but investing a few minutes studying each element will pay off. Time to dive in.

Index Pane

Your entry point into the documentation is the index pane shown in Figure 8-2. This pane is partitioned into the search field, the full index, the kind filter, and finally the entities grouped under the package they belong to.

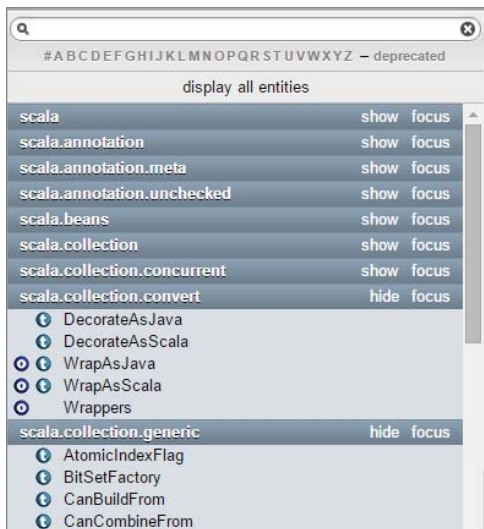


FIGURE 8-2

Searching for Entities

When you type text into the search box, the list of entities is restricted. The filtering behavior differs based on the presence or not of capital letters in the search text (see Figure 8-3). When the query term is all lower case, the filter is case insensitive and contains a match on the fully qualified name of each entity.

This example shows packages, classes, and traits.

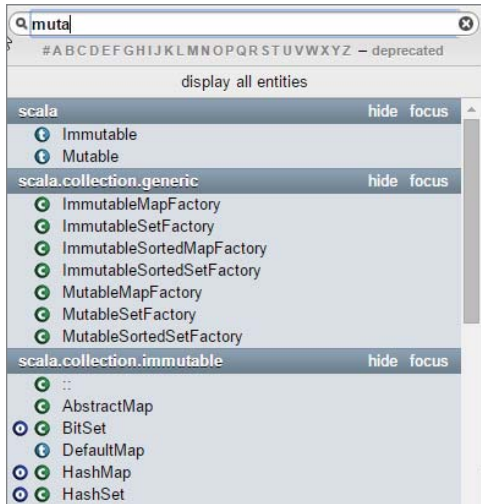


FIGURE 8-3

Including one or more capital letters in the query term applies a camel case match (see Figure 8-4).



FIGURE 8-4

The query box has a hidden feature in Scaladoc generated from Scala 2.11.7. Append a dollar to your query term and this works in the same way as if it were part of a regular expression. For example, Map\$ will match AbstractMap not ImmutableMapFactory.

NOTE Use *Alt-/* to jump to the query box. This also works for the query box on the content pane.

Indexing It All

Underneath the query box you will find a horizontal list of clickable letters bookended by a hash and the word deprecated (see Figure 8-5). Clicking any of these elements will show a listing of entities, methods, functions, and public and protected vals and vars.



FIGURE 8-5

When you click **R** you will see what is shown in Figure 8-6.

RESET
AnsiColor
REVERSED
AnsiColor
Random
util
Range
scala immutable
RangedProxy
runtime

FIGURE 8-6

Under each entry are links to the entities that contain the entry. You should be aware that when the default Scaladoc generation options are used, nested types will not be found from the query box. An example of this can be found in the **Z** section where **Zero** is shown contained in **Duration**, but querying for **Z** does not show **Zero**.

NOTE *Although protected variables are listed in the index, they are not visible on the content pane. An example of this is the `log val` from the `scala.collection.mutable.History` class.*

Occupying the next slot down you will encounter the kind filter. This toggles between hiding packages and not hiding them. This is useful when you are seeking an overview of the packages. Once you have arrived at the package of interest, click **show** to reveal the contained entities.

When you have restricted the display to packages, entering query text will automatically turn off the package restriction.

Figure 8-7 shows the index pane restricted by query text with some packages collapsed. This figure also serves to illustrate the entities type indicator: **O** for object, **C** for class, and **T** for trait. Two columns suffice for this as a class, and a trait cannot share the same name in the same package. The **O**, **C**, and **P** letters are clickable and navigate the content pane directly to the entity type. Clicking the entity name navigates to the class or trait unless there is only an object for the name.

Content Pane

The content pane is on the right (see Figure 8-8). After you have clicked on a package, class, trait, or object, information about the selected entity is displayed here. The pane is divided into three

sections: top level information, filtering and ordering options, and entity member details. Each section is described in the following text. Before you step through the process of adding Scaladoc to the example project, a quick tour of the notable features of the content pane is in order. `ParMap` from the standard library has this documentation.

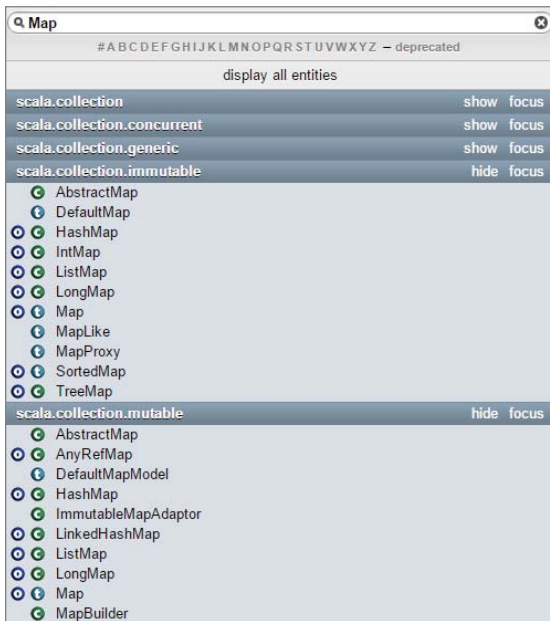


FIGURE 8-7

Top-Level Information

Depending on what you select, you will be presented with information regarding a class, a trait, an object, or a package. The types of information that are shown depend on both the type of entity and the entity itself. Objects, for example, will not include a Known Subclasses section because objects do not have subclasses.

Starting from the top and working down you will see a large letter: **O**, **C**, **T**, or **P** denoting Object, Class, Trait, or Package respectively. Next to this is the entity name. The large letter is clickable. If you start at a class `Foo`, which has a companion object, clicking the **C** navigates to the object `Foo`, and the letter switches to **O**. One way to think about this is that clicking takes you to the next entity related by name if such an entity exists. There are a limited set of such relationships that can exist:

- Class and Object
- Trait and Object

Over to the right (see Figure 8-8) you find the label `Related Docs`: introducing links to the related entity if it exists and a link to the package the entity is in. Hover the mouse over the top right to reveal a hidden Scaladoc feature. A link icon appears. Right click and copy to obtain a link to use to navigate directly to the entity page you are currently on. Permalinks are also available for each entity member. See Figure 8-9 for an example.

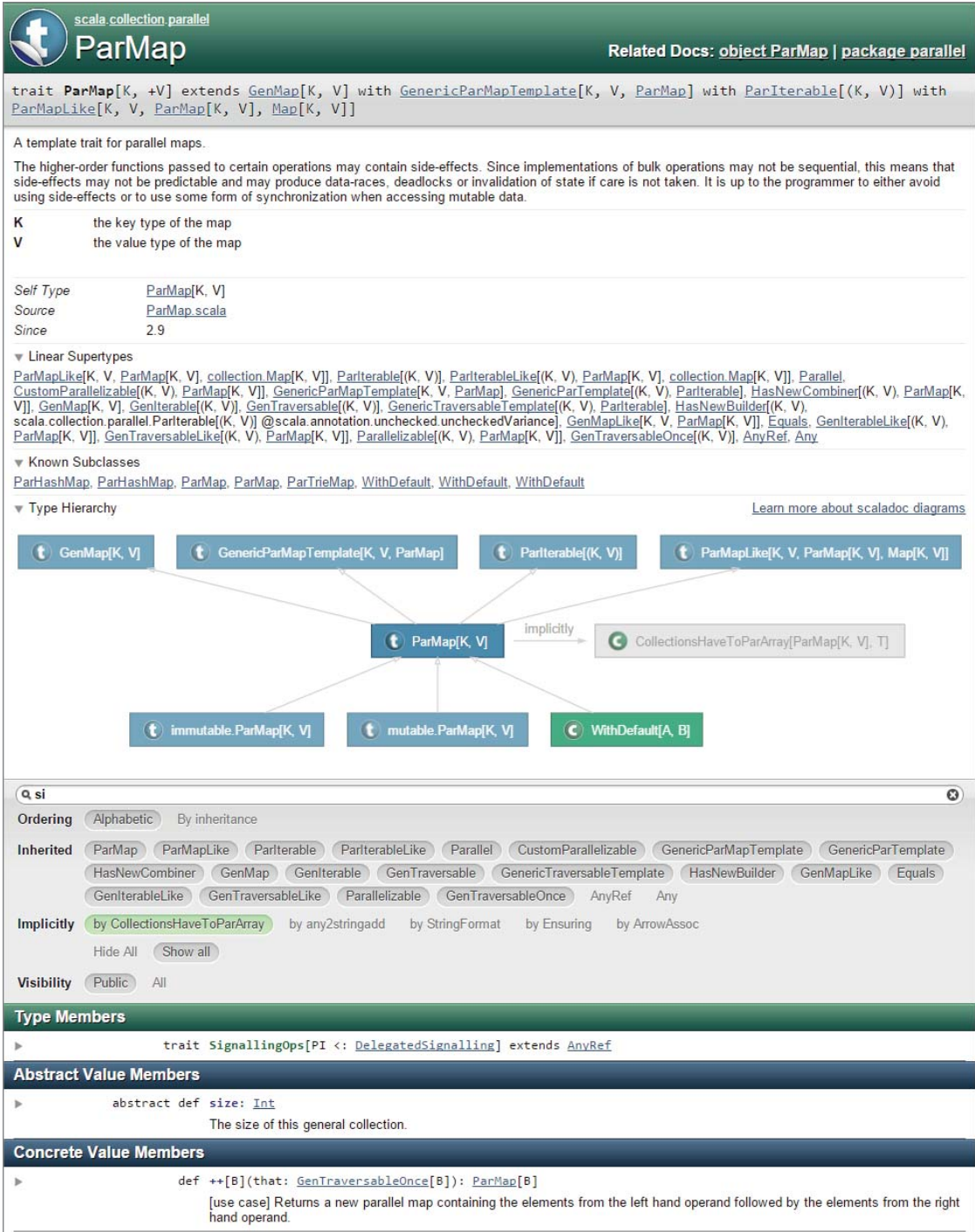


FIGURE 8-8

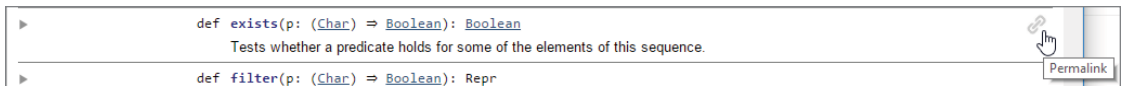


FIGURE 8-9

Following the entity name is the entity signature. This corresponds to the code you find in a source file without the template body. Class parameters, variance annotations, optional superclass, and mixins will be seen here. Hovering over a name will show the fully qualified version of the name.

Residing under the entity signature is the class, trait, object, or package level documentation. The text here is derived from the Scaladoc comments the author added to the entity before the entity header. Beneath the main comment are the type parameter comments, then the attribute block containing, in this case: Self Type, Source, and Since. Later you will see the full list of tags and wiki syntax you can use to directly control the content of the sections described in this paragraph.

Moving down from the attribute block you can see Linear Supertypes that show the supertypes in linearization order. Known Subclasses is self-explanatory. Type Hierarchy shows a view of the type hierarchy centered around the current entity. It shows the directly declared supertype and mixins, implicit views to other types, and some subclasses.

The boxes are clickable and will switch the content pane to the new entity. If you select a package, the Content Hierarchy will be shown. There is also a type hierarchy diagram that shows the entities contained directly in the package. Click the diagram to have it pop up in a larger window.

Filtering and Ordering Options

After the diagrams you are presented with controls for filtering and ordering the member documentation that occupies the remainder of the content pane. The first element is the query box.

NOTE Use *Alt-/* to jump to query box.

Try this out:

- Navigate to `http://www.scala-lang.org/api/current/#package`.
- Use the index pane filter to filter by LL.
- Pick LinkedList.
- Enter head into the content pane query box. You will see method with head in the method name or description.
- Add tail to the content page query box. The box now contains “head tail” and the listed members swells to include tail and a few other methods.
- Replace the query box content with `withFilter`.

When you added `tail` into the query box, you saw a demonstration of the additive nature of the content pane query box. The query for `withFilter` included `WithFilter` in turn, demonstrating the case insensitivity of the content pane query. Remember, you saw that the index page search was case sensitive.

Underneath the query box are buttons controlling the ordering. This will be touched on momentarily. First a word about the Inherited buttons. Deselecting an inherited entity removes its member from display, but that's not the sum of it. There is valuable interaction with the query box to remember. For a member to show, it must satisfy the query box filter and be in an entity that is selected. Try this now:

- Remain on `LinkedList`.
- Enter `++` in the query box.
- Note the presence of the `++` method.
- Deselect `TraversableLike`.

`++` is removed from view when `TraversableLike` is deselected.

The ordering options commonly seen are: Alphabetic, By Inheritance. There is another option that can be seen by navigating to the package documentation for `scala.langauge`, which is shown in Figure 8-10.

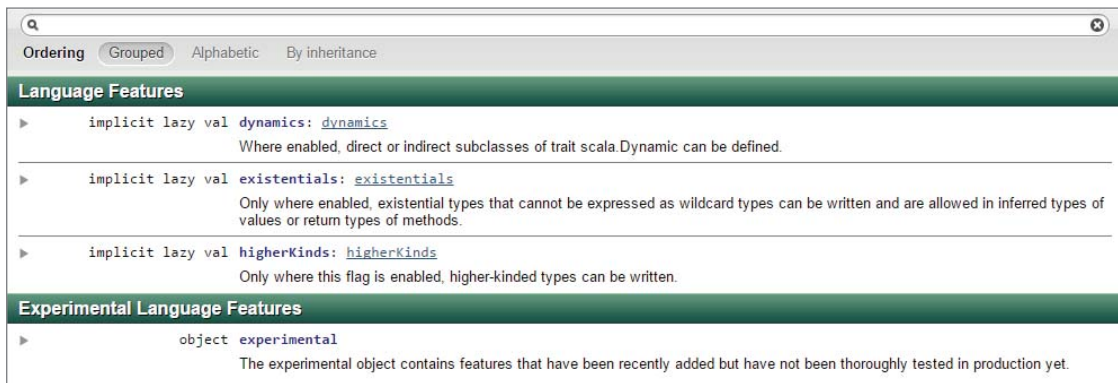


FIGURE 8-10

You can see two groups in use: Language Features and Experimental Language Features. These groups are defined by Scaladoc tags. You will be finding out how to define your own groups later.

THE CLASS OF A THOUSAND METHODS BEGINS WITH A SINGLE DEF

Filtering and ordering of entity members is a welcome feature when working with Scala where there is a tendency toward rich APIs with numerous methods.

Pop quiz: Which class in the Standard Library has the most methods? Would you have guessed Byte with over 600 methods? Here's the proof:

```
$ for f in $(find -name '*.html'); do echo $(grep -c 'def</span>' $f) $f
; done|
sort -rn|head -3
667 ./library/scala/Byte.html
581 ./library/scala/Char.html
558 ./library/scala/Short.html
```

Entity Member Details

A member can be a constructor, a method, a type, a type alias, or a value. The documentation provided for each member will be covered in depth when you learn how to add Scaladoc yourself. Here, an important aspect of the alphabetic member ordering is described. Knowing the logic behind the default ordering will allow you to benefit more deeply from any API documentation you study.

Take a look at the documentation for Set from the Standard Library. In Alphabetic mode the members are grouped under Instance Constructors, Type Members, Abstract Value Members, Concrete Value Members, and Shadowed Implicit Value Members. Within each group the members are ordered alphabetically. If you were looking at an entity with a member for each category you would see the groups from Table 8-1.

TABLE 8-1: Entities by Category

GROUP	DESCRIPTION
Instance Constructors	Primary and auxiliary constructors. Can included deprecated constructors.
Type Members	Type aliases, traits and classes. Can included deprecated members.
Abstract Value Members	Abstract methods and values. Can included deprecated members.

continues

TABLE 8-1 (continued)

GROUP	DESCRIPTION
Value Members/Concrete Value Members	Concrete methods and values. Excludes shadowed implicit members and deprecated members. When the Abstract Value Members group is not empty this group is called Concrete Value Members; otherwise it is Values Members.
Shadowed Implicit Value Members	Shadowed or ambiguous implicits. Excludes any deprecated members.
Deprecated Value Members	Deprecated concrete value members.

The trait `StringLike` from the Standard Library uses five of the six alphabetic groups. See Figure 8-11.

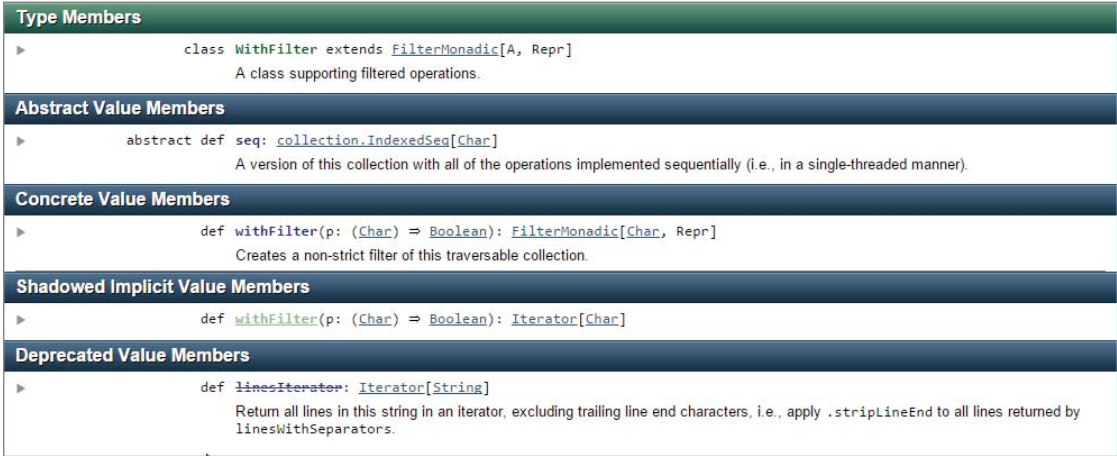


FIGURE 8-11

Entity member information is included in this order: Instance Constructors, Type Members, Abstract Value Members, Value Members/Concrete Value Members, depending on presence of abstract value members, Shadowed Implicit Value Members, and Deprecated Value Members.

INVOKING THE SCALADOC TOOL

Scaladoc generation is realized at the command line by the scaladoc tool. In the following exercise you will create Scaladoc for `Universe.scala`, which is shown here:

```
package com.scalacraft.professionalscala.chapter8.cosmos

object Universe {
  type Cluster = String
  def getClusters: Seq[Cluster] = Nil
}
```

Create the Scaladoc for Universe.scala by following these steps:

Step 1. Open a command prompt.

Step 2. Change directory to professional-scala/src/main/scala/com/scalacraft/professionalscala/chapter8/cosmos.

Step 3. Invoke scaladoc passing the source files to process as arguments.

```
$ scaladoc -d output Universe.scala
model contains 7 documentable templates
```

Your Scaladoc is now generated and ready for viewing. List the output directory content first.

```
$ ls -lF output
com/
index/
index.html
index.js
lib/
package.html
```

SCALADOC COMMAND OPTIONS

The scaladoc tool shares many options with scalac tool. You will see this when displaying the scaladoc help,

```
$ scaladoc -help
```

If you look past the scaladoc section down to the scalac section you will find the -d option. It's not obvious.

Step 4. Now open index.html in a browser. You will be greeted with a complete set of Scaladoc for your one file project. This is shown in Figure 8-12.

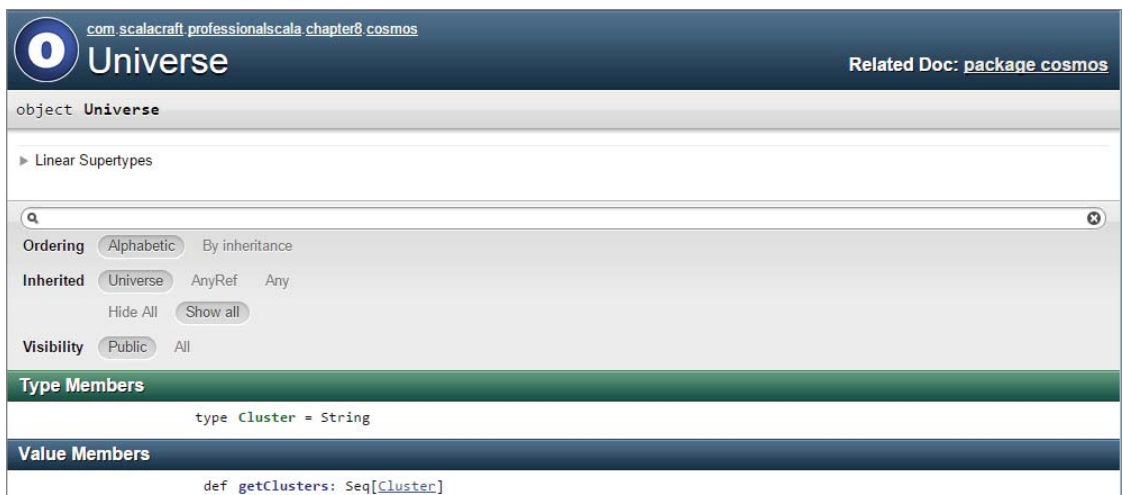


FIGURE 8-12

WIKI SYNTAX

You can use wiki syntax to provide some visual flair to your documentation. The wiki syntax options are few and simple, and can be divided into two groups:

- Inline formatting
- Block elements

The following examples will take you through the essential techniques, leaving you with practical experience applicable to your everyday documentation tasks.

Formatting with Inline Wiki Syntax

Follow these steps to start your journey into the world of Scaladoc formatting. Start with some inline formatting examples.

If you would prefer not to enter the examples, but would still like to follow along, then you can use the source code present in the Git repo identified at the introduction section of the book.

Step 1: Create a class `Galaxy` in the same package as `Universe`,

```
package com.scalacraft.professionalscala.chapter8.cosmos

class Galaxy
```

Step 2: Add a class comment using each of the inline wiki syntax options:

```
/**
 * All inline styles: 'bold', 'italic', `monospace`, __underline__,
 * ^superscript^, ,,subscript,,.
 */
class Galaxy
```

Step 3: Generate the Scaladoc from the command line, this time supplying a shell glob to identify all the source files in the current directory:

```
$ scaladoc -d output *.scala
model contains 8 documentable templates
```

Step 4: Load the Scaladoc for `Galaxy` in your browser and compare the outcome to Figure 8-13.

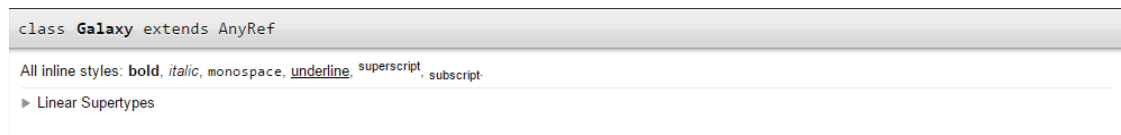


FIGURE 8-13

NOTE Start your block comment with `/**`. This is required for the comment to be processed as Scaladoc.

Table 8-2 details the syntax for inline styling.

TABLE 8-2: Wiki Syntax and Effects

WIKI SYNTAX	EFFECT
' ' ' — three single quotes	Embolden enclosed text
' ' — two single quotes	Italicize enclosed text
` — single backquote	Monospace enclosed text
__ — two underscores	Underline enclosed text
^ — single circumflex	Superscript enclosed text
, , — two commas	Subscript enclosed text

Nesting Inline Styles

Inline styles can be nested in the majority of cases as shown in Figure 8-14. One notable exception is the failure to nest bold inside italic where an extraneous single quote escapes into the page.

class InlineNesting extends AnyRef
<p>Nesting of inline styles.</p> <p>All inline styles: bold, <i>italic</i>, monospace, <u>underline</u>, ^{superscript}, _{subscript}.</p> <p>Bold, Italic: bold and italic, end</p> <p>Italic, Bold: <i>italic and 'bold</i>, end</p> <p>Bold, Monospace: bold and monospace, end</p> <p>Monospace, Bold: monospace and bold, end</p> <p>Bold, Underline: bold and underline, end</p> <p>Underline, Bold: <u>underline and bold</u>, end</p> <p>Italic, Monospace: <i>italic and monospace</i>, end</p> <p>Monospace, Italic: monospace and <i>italic</i>, end</p> <p>Italic, Underline: <i>italic and underline</i>, end</p> <p>Underline, Italic: <u>underline and italic</u>, end</p> <p>Monospace, Underline: monospace and <u>underline</u>, end</p> <p>Underline, Monospace, : <u>underline and monospace</u>, end</p> <p>Superscript, Underline: ^{superscript}, <u>underline</u></p> <p>Underline, Superscript: <u>underline and superscript</u>, end</p> <p>► Linear Supertypes</p>

FIGURE 8-14

If you are interested to learn the details of Scaladoc parsing, clone the Scala GitHub repo and look at CommentFactoryBase.

The source for nested formatting examples is available from the GitHub repo as misc-examples/InlineNesting.scala.

Structuring with Block Elements

Now that you have worked with inline formatting, you can move on to the block element, which occupies the next level of the Scaladoc food chain. Block elements allow you to structure your documentation in several ways to enhance the presentation at a higher level. Again, wiki syntax is used to facilitate this (see Table 8-3). In this section you will extend Galaxy to include examples of each of the five block elements that are listed here:

- Titles
- Paragraphs
- Code blocks
- Lists
- Horizontal rules

Step 1: Return to Galaxy and extend the Scaladoc comment to match this:

```
package com.scalacraft.professionalscala.chapter8.cosmos

/**
 * All inline styles: '''bold''', '''italic''', `monospace`, __underline__,
 *   ^superscript^, ,,subscript,,.
 *
 * =Title 1: Introduction=
 *
 * Paragraph: A galaxy is a system of stars, gas and dust.
 *
 * Create a galaxy using the `new` keyword,
 * {{{
 *   /* Code example */
 *   val zeta = new Galaxy
 * }}}
 *
 * ----
 *
 * ==Title 2: Types==
 *
 * There are different types of galaxy,
 *
 * - Elliptical
 *   A. Maffei 1
 *   A. Centaurus A
 * - Spiral
 *   1. M100
```

```

*      1. NGC 1365
*      - Barred Spiral
*      I. NGC 1300
*      I. NGC 1073
*      - Irregular
*      i. PGC 18431
*      i. IC 559
*      - Lenticular
*      a. Messier 84
*      a. Cartwheel Galaxy
*/
class Galaxy

```

Step 2: Generate the Scaladoc from the command line:

```

$ scaladoc -d output *.scala
model contains 8 documentable templates

```

Step 3: Load the Scaladoc for Galaxy in your browser and compare the outcome to Figure 8-15.

class **Galaxy** extends AnyRef

All inline styles: **bold**, *italic*, monospace, underline, ^{superscript}, _{subscript}

Title 1: Introduction

Paragraph: A galaxy is a system of stars, gas and dust.

Create a galaxy using the new keyword,

```
/* Code example */
val zeta = new Galaxy
```

Title 2: Types

There are different types of galaxy,

- Elliptical
 - A. Maffei 1
 - B. Centaurus A
- Spiral
 - 1. M100
 - 2. NGC 1365
 - Barred Spiral
 - I. NGC 1300
 - II. NGC 1073
- Irregular
 - i. PGC 18431
 - ii. IC 559
- Lenticular
 - a. Messier 84
 - b. Cartwheel Galaxy

► Linear Supertypes

FIGURE 8-15

TABLE 8-3: Wiki Syntax for Block Types

BLOCK TYPE	WIKI SYNTAX	DESCRIPTION	EFFECT
Title	<code>=Title=</code> <code>==Title==</code> <code>===Title===</code> <code>====Title====</code>	Balanced equal signs surrounding text. Maximum of four equal signs.	Produces a header. As more equal signs are added the header becomes smaller. Single equal signs produce a HTML h3 header, which is unfortunately hard to read, so please avoid. Use two, three, or four equal signs.
Paragraph	A blank line	-	Starts a new paragraph.
Code	<code>{{{code example}}}</code>	Three opening and closing curly braces with code examples inside.	Styles the code example to look like code using a monospace font and ignoring inline wiki syntax for the duration of the block.
List	<code>-</code> , <code>1.</code> , <code>A.</code> , <code>a.</code> , <code>i.</code> , <code>l.</code> ,	A list item prefix that includes a trailing whitespace.	This is described in Table 8-4.
Horizontal line	<code>----</code>	At least four hyphens on an otherwise blank line. Must be preceded by a blank line.	Produces a horizontal bar.

The wiki syntax options for lists deserves further explanation starting with whitespace indenting. In the context of lists, wiki syntax whitespace is significant for lists. You should use two spaces per level. Examine the source for Galaxy and you will note there are two spaces between the asterisk and the hyphen that introduces the first item as shown here.

```
*  - Elliptical
```

The subsequent indents also use two spaces. You can use other levels of indenting, but to avoid wasting time on a triviality, always use two spaces to indent lists. Finally, lists can have unindexed or indexed items depending the choice of item prefix. Table 8-4 catalogs the available wiki syntax for list prefixes.

TABLE 8-4: Wiki Syntax for Lists

PREFIX	DESCRIPTION	RESULT
<code>-</code>	A hyphen followed by a space	Bullets
<code>1.</code>	The digit one, a period, and a space	Numerical sequence: 1., 2., 3., ...

PREFIX	DESCRIPTION	RESULT
I.	The letter I, a period and a space	Uppercase Roman numerals: I., II., III., iv., v., vi., ...
i.	The letter i, a period and a space	Lowercase Roman numerals: i., ii., iii., iv., v., vi., ...
A.	The letter A, a period and a space	Uppercase letters: A., B., C., ..., Z., AA., AB., ...
a.	The letter a, a period and a space	Lowercase letters: a., b., c., ..., z., aa., ab., ...

WARNING *At the time of writing, IntelliJ IDEA 15.0.2 disrupts the whitespace used by the list wiki syntax when Reformat Code is used. As a workaround go into Settings, search for Formatter Control, and enable formatter markers. Then mark the comment as shown here with @formatter:off.*

```
// @formatter:off
/**
 * My carefully formatted Scaladoc
 */
```

Linking

Scaladoc supports a compact wiki syntax for generating links. To include a link to a page anywhere on the web, you should enclose the link in square brackets followed by an optional label that the reader will see. The Scaladoc generated from the code below is shown in Figure 8-16.

```
/**
 * An external link: [[http://science.nasa.gov/astrophysics/]].
 *
 * An external link with a label:
 * [[http://science.nasa.gov/astrophysics/ Astrophysics]].
 */
class ExternalLinks
```

```
class ExternalLinks extends AnyRef

An external link: http://science.nasa.gov/astrophysics/.
An external link with a label: Astrophysics.
```

FIGURE 8-16

You now have external links under your belt. Follow the next steps to learn how to link to objects, traits, classes, methods, specific overloads of methods, and types. These are known as entity links.

Step 1. Add the new class Starquake shown below. Make sure you use the correct package.

```
package com.scalacraft.professionalscala.chapter8.cosmos.phenom

import com.scalacraft.professionalscala.chapter8.cosmos.Magnetar
import Starquake.QuakeMagnitude

class Starquake(val magnitude: QuakeMagnitude)

object Starquake {
  /** The magnitude of a quake. */
  type QuakeMagnitude = Int
  def triggerStarquake(magnetar: Magnetar): Starquake = new Starquake(1)
}
```

Step 2. Add Magnetar in the package above with Scaladoc links to Starquake elements as shown here.

```
package com.scalacraft.professionalscala.chapter8.cosmos

/**
 * Magnetars are commonly found within a [[Galaxy]].
 *
 * Trigger a [[phenom.Starquake]] by calling
 * [[phenom.Starquake.triggerStarquake triggerStarquake on object Starquake]]
 */
class Magnetar(galaxy: Option[Galaxy])
```

Step 3. Generate the Scaladoc and confirm that it appears as shown in Figure 8-17. Note, this time it is necessary to extend the scaladoc command arguments to include the new package phenom. Use this list of arguments for the remainder of this chapter unless otherwise advised.

```
$ scaladoc -d output *.scala phenom/*.scala
```

Test each link.



FIGURE 8-17

Step 4. Now imagine you have a requirement to allow the magnitude of the Starquake to be passed in. This results in a method overload. Add the new method on the Starquake object as shown here:

```
object Starquake {
  /** The magnitude of a quake. */
  type QuakeMagnitude = Int
  def triggerStarquake(magnetar: Magnetar): Starquake = new Starquake(1)
  def triggerStarquake(magnetar: Magnetar, magnitude: QuakeMagnitude): Starquake =
    new Starquake(magnitude)
}
```

Step 5. Generate the Scaladoc. You will encounter this warning:

```
Magnetar.scala:3: warning: The link target "phenom.Starquake.triggerStarquake" is
  ambiguous. Several members fit the target:
(magnetar: Magnetar, magnitude: QuakeMagnitude):
  Starquake in object Starquake [chosen]
(magnetar: Magnetar): Starquake in object Starquake
```

The Scaladoc was created, but the method ambiguity is resolved using a strategy you were not consulted about.

Step 6. Fix this pernicious behavior by selecting the method overload precisely in Magnetar. Also add a link to the new method. See below for details.

```
/**
 * Magnetars are commonly found within a [[Galaxy]].
 *
 * Trigger a [[phenom.Starquake]] by calling
 * [[phenom.Starquake.triggerStarquake(magnetar:com\scalacraft\professionalscala
 * \chapter8\cosmos\Magnetar)* triggerStarquake on object Starquake]] or
 * specifying the quake magnitude with
 * [[phenom.Starquake.triggerStarquake(magnetar:com\scalacraft\professionalscala
 * \chapter8\cosmos\Magnetar,magnitude:com\scalacraft\professionalscala\
 * \chapter8\cosmos\phenom\Starquake\QuakeMagnitude)* triggerStarquake on object
 * Starquake]]
 */
```

Step 7. Generate the Scaladoc, confirm it resembles Figure 8-18, and check the links to the Starquake overloaded methods.

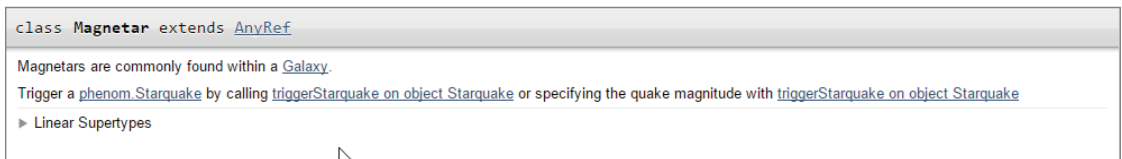


FIGURE 8-18

The Magnetars Scaladoc comment above requires some explanation. What is happening with those lengthy type references? You can use these rules to generate disambiguated links to your own overloaded methods,

- Take the method signature from the source and remove all spaces.
- Drop the result type.
- Add an asterisk.
- Convert the type to the fully qualified type, escaping periods with backslash.

NOTE *The fully qualified name is required when disambiguating link targets. This is not required when the method parameter doesn't require a package prefix, as is the case with `Int` and `String`. The result type is not required and the reference can be shortened if it remains unique. Do not forget the trailing asterisk.*

A general theme with the syntax for links is that Scaladoc will make a best effort to find a target for a link which is convenient, but this helpfulness may have surprising outcomes. Test your Scaladoc in a browser.

You will find that this crib sheet (Table 8-5) will handle the majority of link types typically found within a Scaladoc corpus.

TABLE 8-5: Link types

YOU WANT TO LINK TO...	FOLLOW THIS FORMAT	NOTE
A class or trait	<code>[[mypackage.Name! optional label]]</code>	The exclamation mark selects a class or trait over an object of the same name.
An object	<code>[[mypackage.Name\$ optional label]]</code>	The dollar selects an object over a class or trait of the same name.
Overloaded methods on an object	<code>[[Target\$.foo(z:Str* optional label]]</code> <code>[[Target\$.foo(z:Int* optional label]]</code>	<code>String</code> and <code>Int</code> do not require a full package prefix. The amount of method signature prefix included can be minimized while preserving uniqueness.
Overloaded methods on a class or trait	<code>[[Target!.foo(z:Str* optional label]]</code> <code>[[Target!.foo(z:Int* optional label]]</code>	The same as the object example but showing the use of the exclamation mark for classes and traits.
A method with type parameters	<code>[[[[Target\$. foo[A[_[_]]]*</code>	If square brackets are included in the method signature add additional layers of opening and closing square brackets to compensate.

Scaladoc links can be tricky to get right. To help with this, an extensive set of examples is provided in the code download for this chapter. Find the examples in `misc-examples/links.scala`. These examples have been adapted from a test case in the Scala source code. The original can be found at <https://github.com/scala/scala/blob/2.11.x/test/scaladoc/resources/links.scala>.

Locating Scaladoc

So far you have been adding and modifying Scaladoc on the top level type for the most part. Where else can Scaladoc be placed? Although Scaladoc comments can be added anywhere whitespace is allowed, comments will generate documentation if placed at these locations:

- Before a class, trait, or object declaration
- Before a package object declaration
- Before a method, value, or variable declaration
- Before an alias or abstract type declaration

See <https://wiki.scala-lang.org/display/SW/Syntax> for a more detailed account of this.

TAGGING

This section will take you through the documentation options realized through the use of tags and annotations. Because some tags can only apply to methods, the examples will be method-centric, although there will be exceptions.

Scaladoc tags naturally cluster into three groups that constitute the material you will explore in the following sections:

- Everyday Tagging
- Tagging for Groups
- Advanced Tagging

NOTE *Annotations are language features that are not embedded in comments. Scaladoc will enrich your documentation by processing `@deprecated` and `@migration` annotations. At the time of writing `@migration` is private to the `scala` package and is not of general utility. `@deprecation` will be covered later.*

Tags are implemented using a commercial at sign followed by a tag name. You have no doubt seen numerous usages of tags. `@return` is an example. You will meet the full set of tags over the course of the following sections.

Everyday Tagging

In this section you will add a class `Star`, apply relevant Scaladoc tags, and view the generated documentation. This will introduce you to the tags you will use most often in your day-to-day development activities.

Step 1. Add Star as shown here.

```
package com.scalacraft.professionalscala.chapter8.cosmos

/**
 * Luminous sphere of plasma held together by own gravity.
 * @see [[https://en.wikipedia.org/wiki/Plasma_(physics) Plasma (Wikipedia)]]
 * @see Found in a [[Galaxy]]
 * @author <Your Name Here>
 * @author [[https://github.com/janekdb Janek]]
 * @version 13.8
 * @since 0.7
 * @todo Add `+(other: Star)` to model stellar collisions
 * @todo Add magnetic field
 * @constructor Construct a [[Star]] with the given radius
 * @param radius Initial star radius in metres
 */
class Star(var radius: Double) {

  /** @return The mass of this star in kg */
  def mass: Long = ???


  /**
   * Trigger stellar collapse.
   * @param delay Seconds to wait until collapse
   * @param blackhole If true skip white dwarf and neutron star stages
   * @throws IllegalArgumentException if `delay` is negative
   * @throws IllegalStateException if already a blackhole
   */
  def collapse(delay: Int, blackhole: Boolean): Unit = ???
}
```

Step 2. Generate the Scaladoc and review the output in a browser. Figure 8-19 shows the expected output. The search pane has been removed from the figure to save space.

Tags fall into four groups with respect to the requirements of what must follow the tag. The simplest type of tag is the standalone tag, which is exemplified by `@documentable`. A standalone tag is essentially a flag. It is present or absent. You will cover an example of a standalone tag later in the form of `@documentable`. The next type of tag is the content tag. `@since` is a content tag. This type of tag expects content in the form of free text following the tag name. The penultimate type of tag is the symbol tag. This tag has a name followed by a symbol and then descriptive text. `@param` is an example of a symbol tag. For `@param` tags the expected symbol is the name of a method parameter. The last type of tag is a structured tag, exemplified by `@contentDiagram`. This type of tag imposes a syntax on the text that follows, which allows the effect of the tag to be controlled at a detailed level.

Table 8-6 explains the purpose of each tag you have used so far. You can use this as a quick reference.

In terms of deciding where to place tags, assume Scaladoc follows the principle of least surprise. Combine this with a tight review cycle on the generated Scaladoc and you won't often be surprised.


com.scalacraft.professionalscala.chapter8.cosmos

Star

Related Doc: [package cosmos](#)

class **Star** extends AnyRef

Luminous sphere of plasma held together by own gravity.

Version

13.8

Since

0.7

To do

Add magnetic field

See also

Add +(other: Star) to model stellar collisions
 Found in a [Galaxy](#)
[Plasma \(Wikipedia\)](#)

▶ Linear Supertypes

Instance Constructors

▼

new Star(radius: Double)
 Construct a [Star](#) with the given radius

radius Initial star radius in metres

Value Members

▼

def collapse(delay: Int, blackhole: Boolean): Unit
 Trigger stellar collapse.

delay

Seconds to wait until collapse

blackhole

If true skip white dwarf and neutron star stages

Exceptions thrown

IllegalArgumentException if delay is negative
 IllegalStateException if already a blackhole

▼

def mass: Long

returns

The mass of this star in kg

var radius: Double

Initial star radius in metres

FIGURE 8-19

TABLE 8-6: Tags and Their Purposes

TAG	TYPE	EXAMPLE	PURPOSE	CARDINALITY
@author	Content	@author Janek Bogucki	Identify author.	Multiple
@constructor	Content	@constructor Construct a Star	Document the primary constructor.	At most one
@param	Symbol	@param blackhole If true skip early stages	Describe the purpose of the named parameter.	At most one per method parameter
@return	Content	@return The predicted value	Document the return value.	At most one

continues

TABLE 8-6 (continued)

TAG	TYPE	EXAMPLE	PURPOSE	CARDINALITY
@see	Content	@see [[Starquake]]	Point to additional material including other classes or entities, or external links.	Multiple
@since	Content	@since 1.4.14	Version the entity was introduced in. Include @version if using this.	At most one
@throws	Symbol	@throws IndexOutOfBoundsException when an attempt was made to index outside of the allowed range	Surface the thrown exceptions a user might need to know about. The description is optional.	At most one per thrown exception type
@version	Content	@version 1.6.18	The version of a system or API this entity is part of.	At most one
@todo	Content	@todo Handle lower boundary case	Document gaps in implementation.	Multiple

WHERE ARE MY AUTHORS?

By default @author tags are not included in the generated Scaladoc. To include them, add the -author flag as shown here.

```
$ scaladoc -d output -author *.scala phenom/*.scala
```

This is a sensible default if you are using any kind of modern VCS. @author tags can be used on any entity: classes, traits, object, methods, val, vars, and types.

@THROWS OR @THROWS?

The use of the `@throws` annotation offers greater type safety but does not result in as clear Scaladoc, nor is it inherited. Generate the Scaladoc for this snippet to see the difference.

```
class Throws {
  /** @throws Exception always */
  def a = throw new Exception

  @throws[RuntimeException] ("always")
  def b = throw new RuntimeException
}
```

Before you learn how to group methods and other entities using the `@group` family of tags, there are a few more everyday tags to check over. Return to Star and follow these steps.

Step 1. Add the freeze method to the end of the class as shown here.

```
/**
 * Use some freezers to freeze sunspots.
 * @example
 * {{{
 *   val star = ...
 *   val freezers = List.fill(63)(new Icecube)
 *   val partiallySpentFreezers = star.freezeSunspots(freezers)
 * }}}
 * @note Do not call if collapsed
 * @note Following this the radius will be reduced
 * @tparam T A type that `freeze` can use to freeze a sunspot
 * @param freezers
 * @param freeze A function to freeze a sunspot on a star given a freezer
 * @return The freezers minus any used freezing capacity
 */
@deprecated("Use SolarKit instead", "14.0")
def freezeSunspots[T](freezers: List[T], freeze: (Star, T) => T): List[T] = ???
```

Step 2. Generate the Scaladoc and review the output in a browser. Figure 8-20 shows the expected output. Only the new method is shown to avoid repetition.

Notice the threefold impact of annotating with `@deprecated`. The method appears in the Deprecated Value Members section at the end of the page, the deprecation version and comment are shown, and as part of the generic handling of annotations, the annotation is listed in the Annotations section.

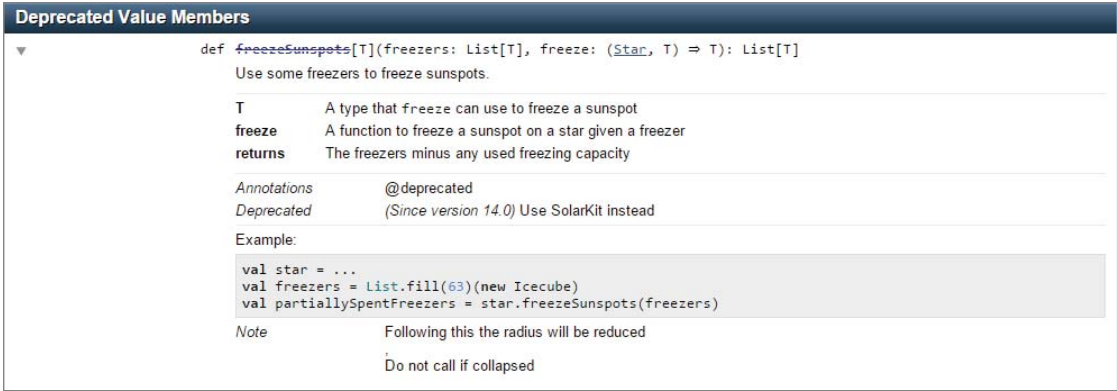


FIGURE 8-20

Table 8-7 explains the purpose of the newly introduced tags.

TABLE 8-7: Newly Introduced Tags and Their Purpose

TAG	TYPE	EXAMPLE	PURPOSE	CARDINALITY
@example	Content	@example {{{val zeta = ...}}}	Add an example into the examples section. The wiki syntax for code blocks in example is optional, but often appropriate.	Multiple
@note	Content	@note The option must be non empty.	Document requirements, restrictions, pre and post conditions.	Multiple
@tparam	Symbol	@tparam T Type of values handled by this pickler.	Document a type parameter.	At most one per type parameter

NOTE Always use the annotation form of `@deprecated`. Although `@deprecated` will be detected when used as a Scaladoc tag, the annotation form is more useful because it includes the version when the class or method was first deprecated. When both are present the annotation wins. There are other deprecation annotations you can use: `@deprecatedInheritance`, `@deprecatedName`, `@deprecatedOverriding`. Consult the Scala API documentation for further details.

Member Permalinks

Go to any member and hover over the top right of the documentation. Right click and copy to get a deep link directly to the method. This is illustrated in Figure 8-21.

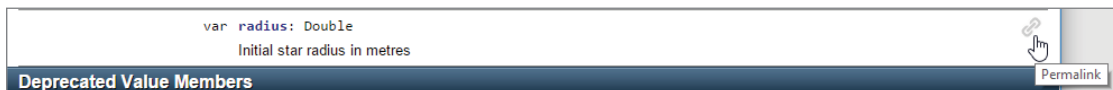


FIGURE 8-21

Tagging for Groups

Now you have mastered the common fare of Scaladoc tagging, so it is time to ascend to more rarefied strata. In this section you will create a class that demonstrates the use of Scaladoc groups that allow related members of an entity to be collected together.

Step 1. Add the class shown here to the project.

```
package com.scalacraft.professionalscala.chapter8.cosmos

/**
 * Star Types with examples.
 *
 * @groupname Type-O Star Type O
 * @groupdesc Type-O Blue, average solar mass: 60
 * @groupprio Type-O 10
 *
 * @groupname Type-B Star Type B
 * @groupdesc Type-B Blue, average solar mass: 18
 * @groupprio Type-B 20
 *
 * @groupname Type-K Star Type K
 * @groupdesc Type-K Orange to Red, average solar mass: 0.8
 * @groupprio Type-K 30
 */
trait StarTypes {

  /** @group Type-O */
  val `10 Lacertra`: Star

  /** @group Type-B */
  val Rigel: Star

  /** @group Type-B */
  val Spica: Star

  /** @group Type-K */
}
```



```
val Arcturus: Star

/** @group Type-K */
val Aldebaran: Star
}
```

Step 2. Generate the Scaladoc and review the output in a browser. To take maximum advantage of the group tags the `-groups` option must be supplied. Without this, the Grouped button will be missing from the Ordered section.

```
$ scaladoc" -d output -groups *.scala phenom/*.scala
```

Figure 8-22 shows the expected output. You can see vals have been listed under the group descriptions corresponding to the given `@group` tag. Try switching between Grouped and Alphabetic.

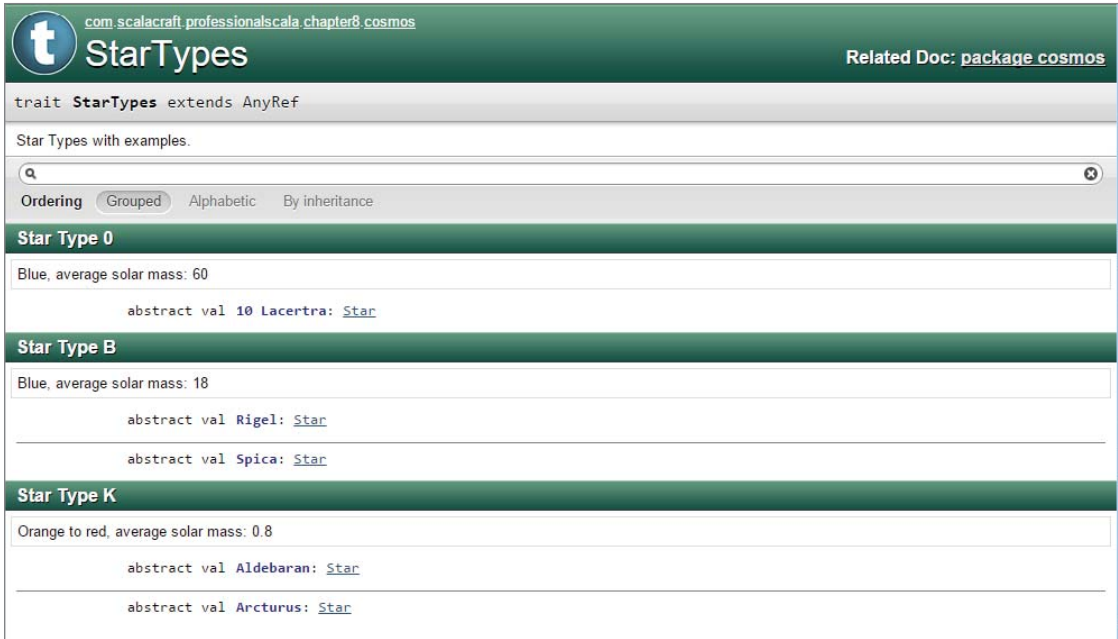


FIGURE 8-22

Table 8-8 explains the purpose of the group tags.

TABLE 8-8: The Group Tags

TAG	TYPE	EXAMPLE	PURPOSE	CARDINALITY
@group	Symbol	@group InfoSec	Indicate the tagged entity is in the named group.	At most one
@groupname	Symbol	@groupname InfoSec The information security API.	Provide a label of the group. Appears in green bar before group description.	At most one per group

TAG	TYPE	EXAMPLE	PURPOSE	CARDINALITY
@groupdesc	Symbol	@groupdesc InfoSec This API handles the security aspects of the system. For example: {{{ val pki = ... }}}}	Add descriptive text to be used under group name. Can include wiki syntax and extend over several lines.	At most one per group
@groupprio	Symbol	@groupprio InfoSec -100	Associate a relative position with a group. If group A has a lower priority value than group B, then group A appears before group B.	At most one per group

NOTE All @group* tags are optional. Use as many or as few of them as you wish. Scaladoc employs an unsurprising defaulting scheme to fill in the blanks where required. In particular, @groupprio has a default value of 0, while ungrouped elements have an implicit priority of 1000, which means they will always be at the bottom if you use priorities less than that.

Advanced Tagging

You have arrived at the advanced Scaladoc tags, some of which you will try out with a few more code additions. To start with you will see how to place a simplifying lens onto any method signature no matter how hieroglyphically rich it is, how to not repeat yourself, and how to elevate nested entities to first class citizens in the context of documentation. Following that, you will gain insight into tags that control documentation at a higher level.

“Seeing, hearing, feeling, are miracles, and each part and tag of me is a miracle.”

—“SONG OF MYSELF,” WALT WHITMAN

Follow these steps to get a feel for the possibilities of @define, @usecase, and @documentable.

Step 1. Add the code from the code listing below into Planet.scala. This is the Scaladoc before you fully augment it with the new tags. A macro expansion Body is defined in Terraformable and used in the @return tag. You will see the utility of this when you extend the Scaladoc with @inheritdoc.

```
package com.scalacraft.professionalscala.chapter8.cosmos

/**
 * @define Body Terraformable
 */
trait Terraformable[T] {

  /**
```

```

    * Create a terraformed copy of this $Body.
    * @param tfs A collection of [[Terraformer]]s to apply in order
    * @return A terraformed copy of this $Body
    */
  def terraform[P1 >: T, P2 <: T](implicit tfs: Seq[Terraformer[P1, P2]]): T
}

trait Terraformer[T, U] {
  def terraform[U <: T](body: T): U
}

class Planet extends Terraformable[Planet] {

  /**
   * A specialised kind of [[Planet]]
   */
  type SpecialisedPlanet <: Planet

  /**
   * Create a new world from this planet.
   * @note May result in mass extinction of existing life
   */
  def terraform[P1 >: Planet, SpecialisedPlanet]
    (implicit tfs: Seq[Terraformer[P1, SpecialisedPlanet]]): Planet = ???
}

```

Step 2. Generate the Scaladoc and review the output in a browser. Figure 8-23 shows the expected output.

Instance Constructors	
	new Planet()
Type Members	
	abstract type SpecialisedPlanet <: Planet A specialised kind of Planet
Value Members	
▼	def terraform[P1 >: Planet, SpecialisedPlanet](implicit tfs: Seq[Terraformer[P1, SpecialisedPlanet]]): Planet Create a new world from this planet.
	tfs A collection of Terraformers to apply in order returns A terraformed copy of this Terraformable
	Definition Classes Planet → Terraformable Note May result in mass extinction of existing life

FIGURE 8-23

You should make note of the following aspects of the terraform method documentation before you add the new tags, an action that will change each of these points.

- The @return and @param tags are inherited from the Terraformable trait.
- The @return comment mentions Terraformable.
- The terraform method comment from Terraformable trait is missing.

- The `SpecialisedPlanet` type alias is not a hyperlink.
- The `terraform` method signature is very busy with implicits and type parameters—not great if you have an entire planetary system to get done.

Step 3. Edit `Planet` by adding a redefinition of the `Body` macro into the class comment, a `@documentable` tag on the type alias, and `@inheritdoc` on the `terraform` method as shown below.

```
/**
 * @define Body "'candidate'" colony world
 */
class Planet extends Terraformable[Planet] {

  /**
   * A specialised kind of [[Planet]]
   * @documentable
   */
  type SpecialisedPlanet <: Planet

  /**
   * @inheritdoc
   * Create a new world from this planet.
   * @note May result in mass extinction of existing life
   */
  def terraform[P1 >: Planet, SpecialisedPlanet]
    (implicit tfs: Seq[Terraformer[P1, SpecialisedPlanet]]): Planet = ???
}
```

Step 4. Generate the Scaladoc. Figure 8-24 shows the expected documentation.

Instance Constructors					
new Planet()					
Type Members					
abstract type SpecialisedPlanet <: Planet A specialised kind of Planet					
Value Members					
▼	def terraform [P1 >: Planet , SpecialisedPlanet](implicit tfs: Seq[Terraformer [P1, SpecialisedPlanet]]): Planet Create a terraformed copy of this <i>candidate</i> colony world. Create a new world from this planet.				
	<table> <tr> <td>tfs</td><td>A collection of Terraformers to apply in order</td></tr> <tr> <td>returns</td><td>A terraformed copy of this <i>candidate</i> colony world</td></tr> </table>	tfs	A collection of Terraformers to apply in order	returns	A terraformed copy of this <i>candidate</i> colony world
tfs	A collection of Terraformers to apply in order				
returns	A terraformed copy of this <i>candidate</i> colony world				
	<table> <tr> <td>Definition Classes</td><td>Planet → Terraformable</td></tr> <tr> <td>Note</td><td>May result in mass extinction of existing life</td></tr> </table>	Definition Classes	Planet → Terraformable	Note	May result in mass extinction of existing life
Definition Classes	Planet → Terraformable				
Note	May result in mass extinction of existing life				

FIGURE 8-24

Now review the differences.

- The inherited `@return` comment now mentions “candidate colony world” instead of `Terraformable`.
- The `terraform` comment from `Terraformable` is now present beneath the comment defined in `Planet`. Remember, this was defined as “Create a terraformed copy of this `$Body`.” `@inheritdoc` contributed to this change.

- The effect of the redefinition of the Body macro can be seen in both the @return comment and the method comment.
- Macro definitions can contain wiki syntax.

The SpecialisedPlanet type alias is now a hyperlink.

Click SpecialisedPlanet to see that an entire page has been created as a consequence of placing @documentable tag on the type alias. The page header is shown in Figure 8-25.

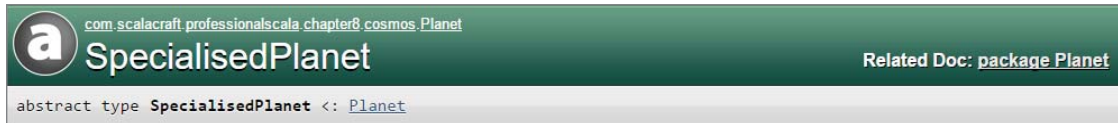


FIGURE 8-25

NOTE @documentable *is equivalent to and predated by* @template. *We prefer @documentable as this is clearer.*

Next, you will use @usecase to simplify the terraform method signature.

WARNING @usecase *will increase the maintenance effort you face when modifying your classes. You are also advised to check the generated Scaladoc carefully when using @usecase. Consult the Scala Standard Library for extensive examples of how to use @usecase at scale.*

Edit Planet once more.

Step 1. Modify the comment on the terraform method as shown here.

```
/**
 * @inheritdoc
 * @usecase def terraform: Planet
 * Create a new world from this planet.
 * @inheritdoc
 * @note May result in mass extinction of existing life
 */
def terraform[P1 >: Planet, SpecialisedPlanet]
  (implicit tfs: Seq[Terraformer[P1, SpecialisedPlanet]]): Planet = ???
```

Generate the Scaladoc. Figure 8-26 shows the new documentation of the terraform method.



FIGURE 8-26

The key differences are the introduction of the simplified method signature and the demotion of the full signature down into two collapsible sections. With details on demand your API documents will be simple to scan through while still providing the full signature for those sufficiently motivated to enquire more deeply.

NOTE *@inheritdoc appears twice in the Scaladoc for the terraform method above. Removing either results in the loss of the method comment from the mixed in Terraformable trait. You should experiment with @usecase to build a feel for its nuances before embarking on extensive application of it.*

To conclude this section you will briefly check out the diagramming related tags: @contentDiagram and @inheritanceDiagram. SVG-based inheritance and content diagrams are created by scaladoc when the -diagrams flag is supplied. You have two types of diagrams at your disposal:

- Content for showing contained entities
- Inheritance for showing inheritance relationships

Take a look at an example of a content diagram here: <http://www.scala-lang.org/api/2.11.7/scala-reflect/index.html#scala.reflect.api.Symbols>. The effects of using @contentDiagram in the main comment for Symbol are shown in Figure 8-27.

The diagram shows the type aliases contained in the Symbols trait. Here is an excerpt from Symbols corresponding to the types on display in Figure 8-27.

```
type Symbol >: Null <: AnyRef with SymbolApi
type TypeSymbol >: Null <: TypeSymbolApi with Symbol
type TermSymbol >: Null <: TermSymbolApi with Symbol
type MethodSymbol >: Null <: MethodSymbolApi with TermSymbol
type ModuleSymbol >: Null <: ModuleSymbolApi with TermSymbol
type ClassSymbol >: Null <: ClassSymbolApi with TypeSymbol
```

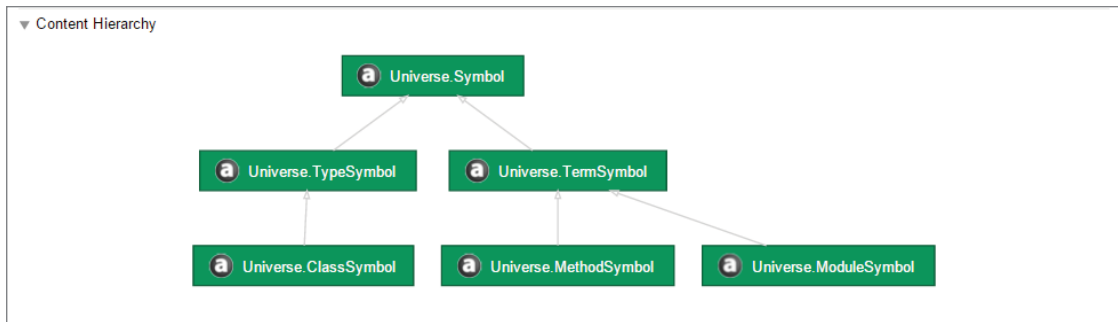


FIGURE 8-27

By default packages and objects will get content diagrams. For traits and classes you will need to add the tag.

NOTE For diagram creation to work scaladoc must be able to invoke the `dot` command. This is available from the `graphviz` package on Debian based distros.

Inheritance diagrams are generated by default for traits and classes, but when you require fine grained control of the inheritance diagrams, `@inheritanceDiagram` can be added to allow exclusionary specifiers to be added. These can be used to remove distracting elements for the diagram. As an example, suppose you wanted to omit superclasses from your diagrams. You can add `hideSuperclasses` to the tag as shown here to achieve that.

```

/**
 * @inheritanceDiagram hideSuperclasses
 */
class ExampleClass extends Example2 with Example3

```

You can use these specifiers with `@contentDiagrams` as well. The specifiers you can use are:

- `hideNodes`
- `hideDiagram`
- `hideOutgoingImplicits`
- `hideSubclasses`
- `hideEdges`
- `hideIncomingImplicits`

- `hideSuperclasses`
- `hideInheritedNodes`

Consult the Scala source code for examples of how to use these options.

Table 8-9 summarizes the purpose of the advanced tags.

TABLE 8-9: Advanced Tags

TAG	TYPE	EXAMPLE	PURPOSE	CARDINALITY
<code>@define</code>	Symbol	<code>@define sideEf-</code> fects This method has side effects. You have been __warned__! /** \$sideEffects */ def deleteOcean: Unit	Create a named block of wiki syntax text to expand when \$<name> is encountered.	At most one per comment per macro name. Can be rede- fined closer to point of use
<code>@inheritdoc</code>	Standalone	<code>@inheritdoc</code>	Extend the documenta- tion inheritance that is applied by default cover- ing certain tags to also include main comment inheritance.	At most one per comment
<code>@documentable</code>	Standalone	<code>@documentable</code>	Generate an additional documentation page for the tagged type alias.	At most one per type alias
<code>@template</code>	Standalone	<code>@template</code>	Identical to <code>@documentable</code> .	At most one per type alias
<code>@usecase</code>	Symbol	<code>@usecase def</code> terraform: Planet <Plus alternative versions of other tags>	Substitute a hand crafted, simplified account of the method signature. Nest full method definition down into method documenta- tion body.	At most once per method comment.

continues

TABLE 8-9 (continued)

TAG	TYPE	EXAMPLE	PURPOSE	CARDINALITY
@content-Diagram	Structured	@contentDiagram hideNodes <i>"*Api"</i>	Create content diagram for entities that do not have this type of diagram by default. Can also be used to prevent diagram creation.	At most one per containing entity
@inheritanceDiagram	Structured	@inheritanceDiagram hideEdges(<i>"*E"</i> -> <i>"*A"</i>)	Create an inheritance diagram for entities that do not have this type of diagram by default. Can also be used to prevent diagram creation.	At most one per entity

INVOKING SCALADOC: ADDITIONAL OPTIONS

You have seen how to specify Scaladoc within your source code. Your users are already lauding your handsomely documented project, but that is not the end of it. There are a number of useful scaladoc command line options you can incorporate into your tool chest to provide your efforts with a fine edge (see Table 8-10). This section picks out some of those options not already covered.

NOTE *To see the complete list of documented scaladoc options, invoke the tool with the -help option.*

```
$ scaladoc -help
```

TABLE 8-10: scaladoc Command-Line Options

OPTION	EXAMPLE	PURPOSE
-doc-footer <footer>	-doc-footer 'Copyright 2018 – H. W. Olbers'	Add a footer to each page. By default there is no footer.
-doc-root-content <path>	-doc-root-content docroot.txt	Provide content for the Scaladoc landing page. Can include wiki syntax.
-doc-title <title>	-doc-title 'Cosmic Toolkit'	Set title. Visible as browser window or tab title.

OPTION	EXAMPLE	PURPOSE
<code>-doc-version <version></code>	<code>-doc-version '13.8'</code>	Append the version to the title.
<code>-skip-packages</code> <code><<package1>:...:<packageN>></code>	<code>-skip-packages</code> <code>com.example.</code> <code>internal</code>	Do not include the nominated packages in the generated Scaladoc. Useful if you have packages users do not need to be aware of.

INTEGRATING SCALADOC CREATION WITH YOUR PROJECT

Although you have been generating Scaladoc by the direct execution of the scaladoc command line tool, which is fine for tutorial purposes, you can integrate documentation creation directly into your build tool or project management tool. In this section you will see how to configure Maven via a POM, and how to achieve the same thing with SBT. If you have not already studied the SBT and Maven chapters, now would be a good point to do so.

Configuring Maven

Provided you have already configured `scala-maven-plugin`, you can generate the Scaladoc via a Maven goal. Execute this code and then browse to `target/site/scaladocs/index.html`:

```
$ mvn scala:doc
```

If you need to elevate the structure around your build process, then adding a suitable plugin configuration will trigger Scaladoc generation during the Maven site phase. Take the configuration in the code below and add this into the `project/build/plugins` elements alongside the `scala-maven-plugin` configuration:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-site-plugin</artifactId>
  <version>3.4</version>
  <configuration>
    <reportPlugins>
      <plugin>
        <artifactId>maven-project-info-reports-plugin</artifactId>
        <version>2.8.1</version>
      </plugin>
    </reportPlugins>
    <plugin>
      <groupId>net.alchim31.maven</groupId>
      <artifactId>scala-maven-plugin</artifactId>
      <version>3.2.2</version>
      <configuration>
        <jvmArgs>
          <jvmArg>-Xms64m</jvmArg>
          <jvmArg>-Xmx1024m</jvmArg>
        </jvmArgs>
        <args>
          <arg>-doc-footer</arg>
        </args>
      </configuration>
    </plugin>
  </configuration>
</plugin>
```

```
        <arg>${copyright}</arg>
        <arg>-groups</arg>
        <arg>-doc-title</arg>
        <arg>Cosmic Toolkit</arg>
      </args>
    </configuration>
  </plugin>
</reportPlugins>
</configuration>
</plugin>
```

Now you can generate your Scaladoc with Maven, either from within the comfort of your IDE, or on the CLI as shown here.

```
$ mvn site
```

As before, the root page is located at `target/site/scaladocs/index.html`. The site goal will also generate additional project documentation unrelated to Scaladoc.

Configuring SBT

SBT supports a `doc` task that generates Scaladoc from your source files. If you are new to SBT, review the SBT chapter and an introduction to SBT in general. For now it's enough to add the minimal project file shown here to the module root level (chapter-8/):

```
lazy val root = (project in file(".")).
  settings(
    name := "Cosmic Toolkit",
    version := "0.1.0-SNAPSHOT",
    scalaVersion := "2.11.7"
  )

val copyright = "Janek Bogucki 2015"
scalacOptions in (Compile,doc) +=
  Seq("-doc-footer", copyright, "-groups", "-doc-title", "Cosmic Toolkit")
```

With that in place, call the `doc` task from SBT:

```
$ sbt doc
```

This will generate the same Scaladoc as the Maven configuration.

PUBLISHING SCALADOC

You have your Scaladoc sitting under `target/site/scaladocs`. The next step is to publish this on the web. By virtue of being a collection of static pages, there are few requirements on the web hosting solutions. If you already have a web site you can upload the pages to that. Amazon Web Services S3 is another way to publish static web sites with moderately little effort. In this section you will see how to use GitHub pages to bring your documentation to the wider world.

Step 1. Create the documentation using Maven.

```
$ mvn clean scala:doc
$ ls -xF target/site/scaladocs/
com/ deprecated-list.html index/ index.html
index.js lib/ package.html scala/
```

Step 2. Create and checkout a branch called gh-pages. The --orphan option creates a branch with no parent commit, which makes for a simpler publication history.

```
$ git checkout --orphan gh-pages
Switched to a new branch 'gh-pages'
```

Step 3. Replace the top level content of the repo with the generated Scaladocs.

```
$ mv * /tmp
$ cp -a /tmp/target/site/scaladocs/* .
```

Step 4. Commit and push the documentation to the remote repo:

```
$ git add --all .
$ git commit -m'Publish Scaladocs'
$ git push --set-upstream origin gh-pages
```

Step 5. Browse your new Scaladoc site. Navigate to <username>.github.io/<project-name>. You will be greeted by the index page.

In this example you have seen a minimized version of the process that covers the primitives required to get the documentation up onto GitHub pages.

As your project develops and goes through releases, you will want to consider a longer term strategy that allows different versions of the documentation to coexist. You can also take advantage of the open ended nature of GitHub pages to attain that goal. An example that accommodates multiple versions of documentation corresponding to different releases is shown in Figure 8-28. Going beyond that there is no restriction on just publishing documentation, but that is the focus here.

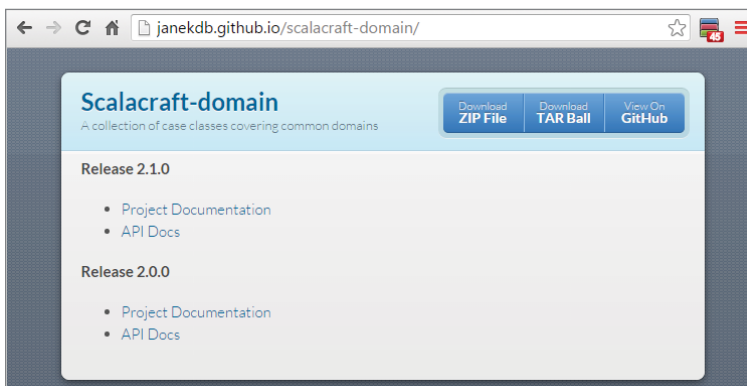


FIGURE 8-28

TABLES AND CSS

Wiki markdown syntax is sufficient for many presentation purposes, and when combined with headers, paragraphs, code blocks, and bulleted lists, the structural aspects of the documentation can be lifted considerably. However, there is no support for tabular presentation of information, and with this structural workhorse missing from the stables there is a big gap to fill. So saddle up, you will shortly be solving this problem.

The solution is actually straightforward. Scaladoc defines a subset of HTML tags that are passed though unchanged. The full list of supported tags is shown in Table 8-11.

TABLE 8-11: Supported Tags in Scaladoc

abbr	br	dd	ins	link	strong	tfoot
acronym	b	del	i	option	sub	th
address	caption	dfn	kbd	param	sup	thead
area	button	fieldset	label	pre	select	tr
a	cite	form	legend	q	table	tt
bdo	code	hr	em	samp	tbody	var
big	col	img	object	small	td	
blockquote	colgroup	input	optgroup	span	textarea	

Try out the tables tags by following the steps below.

Step 1. Add the StarSurvey object shown here.

```
package com.scalacraft.professionalscala.chapter8.cosmos

/**
 * <table style="border-collapse: collapse; border: 1px solid black">
 *   <caption>Star Data</caption>
 *   <colgroup>
 *     <col style="background-color:LemonChiffon"/>
 *     <col style="background-color:Gold"/>
 *     <col style="background-color:HoneyDew"/>
 *   </colgroup>
 *   <tr>
 *     <th style="background-color: #4CAF50; color: white">Name</th>
 *     <th style="background-color: #4CAF50; color: white">Absolute Magnitude</th>
 *     <th style="background-color: #4CAF50; color: white">Distance (parsecs)</th>
 *   </tr>
 *   <tr>
 *     <td>Arcturus</td>
 *     <td>-0.31</td>
 *     <td>11.25</td>
 *   </tr>
 * </table>
 */
```

```

*      <td>Vega</td>
*      <td>0.58</td>
*      <td>7.7561</td>
*    </tr>
*    <tr>
*      <td colspan="3" style="background-color: #CCCCCC">
*        Source: http://www.astronomynotes.com/starprop/s4.htm
*      </td>
*    </tr>
*  </table>
*/
object StarSurvey

```

Step 2. Build the Scaladoc.

```
$ mvn clean scala:doc
```

Step 3. Confirm your Scaladoc now includes the table shown in Figure 8-29.

object StarSurvey		
Star Data		
Name	Absolute Magnitude	Distance (parsecs)
Arcturus	-0.31	11.25
Vega	0.58	7.7561
Source: http://www.astronomynotes.com/starprop/s4.htm		

FIGURE 8-29

With your documentation now enriched with tables, your users will be delighted, but there is a price to pay. Your Scaladoc is now considerably noisier than when it only used markdown. This has a development cost that you should evaluate when deciding to break free from markdown.

CUSTOM CSS

A significant amount of the markup in this example is devoted to style attributes. It is possible to add user defined CSS to the Scaladoc CSS files, thereby allowing the inline styles to be replaced by class and id attributes. The mechanics of achieving this are beyond the scope of this chapter. If you are interested, clone *ScalaTest* (<https://github.com/scalatest/scalatest>), and review the build configuration. *ScalaTest* is also a fine example of Scaladoc pushed to the extremes.

NOTE *Some HTML tags are passed through unchanged when included in Scaladoc. Add this to a class or method and take a look at the result, but avoid staring directly at it.*

```
/**  
 *   
 */
```

SUMMARY

“Voluminous documentation is part of the problem, not part of the solution”

—TOM DEMARCO

Although it is possible to produce and consume Scaladoc without a user guide, there are aspects that are not easily discovered through everyday use. Lifting the lid on Scaladoc reveals powerful features and options for both the author and the reader. There is no Scaladoc specification to peruse. Scaladoc is defined through implementation, allowing for rapid innovation but bringing the perils of never quite knowing for sure what should and should not be possible.

The Scala Language source code is the primary source of exemplar Scaladoc. You should clone it and take a look. If you are working with SBT, then `ScalaTest` is a project that demonstrates the levers to pull to coerce Scaladoc to your exact needs.

With out-of-the-box support for grouping methods and values, and factoring out repetition with macros, you can produce extensive documentation that is also well structured and low maintenance. By reviewing your generated Scaladoc as you author it, you can strengthen your documentation prowess and find surprising outcomes earlier rather than later. Always test your Scaladoc links.

Maven and SBT have support for Scaladoc generation. Adding a documentation generation mode to your build allows you to concentrate on the more interesting activities and avoid documenting the process for generating the documentation.

With generated Scaladoc in hand you can publish to the web with negligible effort using GitHub pages. This will expose your inline markdown outcomes to the world, or if you elected to whip out the power tools, your glorious HTML and CSS tables.



Type System

WHAT'S IN THIS CHAPTER?

- Understanding Scala's type system
- Getting to know the different types of polymorphisms
- Understanding bounds and variance
- Using other niceties of the type system

This chapter begins with an overview of type systems in general and highlights the main difference between static and dynamic typing. Then it shows the main features of the powerful Scala type system. The different types of polymorphisms are examined showing how Scala excels compared to Java thanks to ad hoc polymorphism through type classes. Type bounds is another powerful concept shown in this chapter. Roughly speaking, a bound lets you restrict the set of possible types that can be used in your data structures.

Due to the presence of both subtype and parametric polymorphism, you are forced to face, sooner or later, the concept of variance. You'll find out how to define your data structures as covariant, contravariant or invariant, and use bounds to satisfy the compiler complaints. You'll also meet other Scala type system niceties such as: self-type annotations, self-recursive types and abstract type members. Finally, you'll see how Scala let you *simulate* dynamic typing that allows you to deal with some situations where static typing is not a feasible solution.

Take into account that Scala's type system is a big subject that cannot be fully covered in a chapter or two. You can easily write an entire book about it, and the shapeless project (<https://github.com/milessabin/shapeless>) is a proof of how complex and powerful it can be. In this book you'll learn about Scala's type system features that will help you survive in your day-by-day coding sessions.

WHAT IS A TYPE SYSTEM?

“A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.” This is the definition given by Benjamin C. Pierce in his excellent book *Types and Programming Languages*.

Put simply, a type is something that describes a set of values that have operations in common. For instance, the type `Int` represents a set of integer numbers that support operations like addition, multiplication and so on.

A good type system gives you guarantees about the soundness (correctness) of your programs. It does not let you apply an operation to a value of the wrong type. For example, you cannot pass a `String` to a function that expects a `List[Int]` and see it fail at runtime; a thing that can instead happen in dynamically typed languages. For example, the following code wouldn't compile:

```
def headOrZero(xs: List[Int]): Int = xs.headOption.getOrElse(0)

headOrZero("hello") // compile error
```

You get an error similar to the following:

```
[error] type mismatch;
[error]   found   : String("hello")
[error]   required: List[Int]
[error] headOrZero("hello")
[error]             ^
[error] one error found
[error] (core/compile:compileIncremental) Compilation failed
```

In the type system jargon you say that that code doesn't *type check*. The most important thing to understand about this simple example is that, in a statically typed language, the failure happens at compile-time while in a dynamically typed one you run across this at runtime. This is something you, hopefully, always want to avoid.

Static versus Dynamic Typing

In this section you'll see the advantages of both type systems. There's a war going on between people of both factions. We don't like wars, but both systems have advantages and disadvantages just like everything in life. We cannot deny, however, that from a typing point of view we're more about static typing. However, we understand that, sometimes, dynamic typing can be the right choice for the problem at hand. So, in this context, our motto is: “static typing whenever possible, dynamic typing only when really needed.”

In a statically typed language, type errors are caught by a compiler module, called type checker, prior to running the program. A type checker just makes a conservative approximation and gives error messages for anything that might cause a type error. This is because, analogously to the halting problem (https://en.wikipedia.org/wiki/Halting_problem), it's not possible to build a type checker that can exactly predict which programs will surely result in type errors.

In a dynamically typed language, type checking is performed at runtime, immediately before the application of each operation, to make sure that the operand type is suitable for the operation. In a dynamic type system values have types but variables don't. That is, you can have a variable and assign it a string first and an int successively.

In the dynamic world, many type errors that can be caught at compile-time arise at runtime, which is not what you really want. Imagine being called at 3:00 a.m. by your manager since your application, after six months of being in production, broke because, due to a combination of user actions, it fell into that branch of the programs where you have the type error.

In a static type system that simply can't happen.

What Static Type Systems Are Good For

These are the main advantages of a statically typed language:

- **Error detection.** A static type system catches type errors at compile-time.
- **Performance.** Statically typed languages provide better performance than dynamically typed ones. This is because the compiler can generate optimized code, plus there's no need to perform type check at runtime.
- **Abstractions.** A good type system lets you build very reusable, polymorphic and type safe components. This improves the modularity of your systems.
- **Documentation.** Last but not least, types are very good for auto documenting your code. They let you reason about your code in terms of types, which is pretty good. Correct implementations, then, are also easier because you just need to follow the types.

What Dynamic Type Systems Are Good For

Here are the main advantages of the dynamic counterpart:

- **Fast prototyping.** Undoubtedly, dynamically typed languages allow faster prototyping than static ones.
- **Dealing with values whose types depends on runtime information.** In these cases dynamic typing is a huge win instead of resorting to reflection and other convoluted tricks used in static languages. That said, toward the end of this chapter, you'll see what Scala has to offer in this regard.

SCALA'S UNIFIED TYPE SYSTEM

What makes Scala stand apart from Java is, without a doubt, its powerful type system. Even if you look at it from the only object-oriented perspective, Scala's type system is much better than Java's. In this regard, Scala has a unified type system in that there's a top type, `Any`, and a bottom type, `Nothing`. There are no *orphans* like the Java's primitive types. Figure 9-1 shows Scala's class hierarchy.

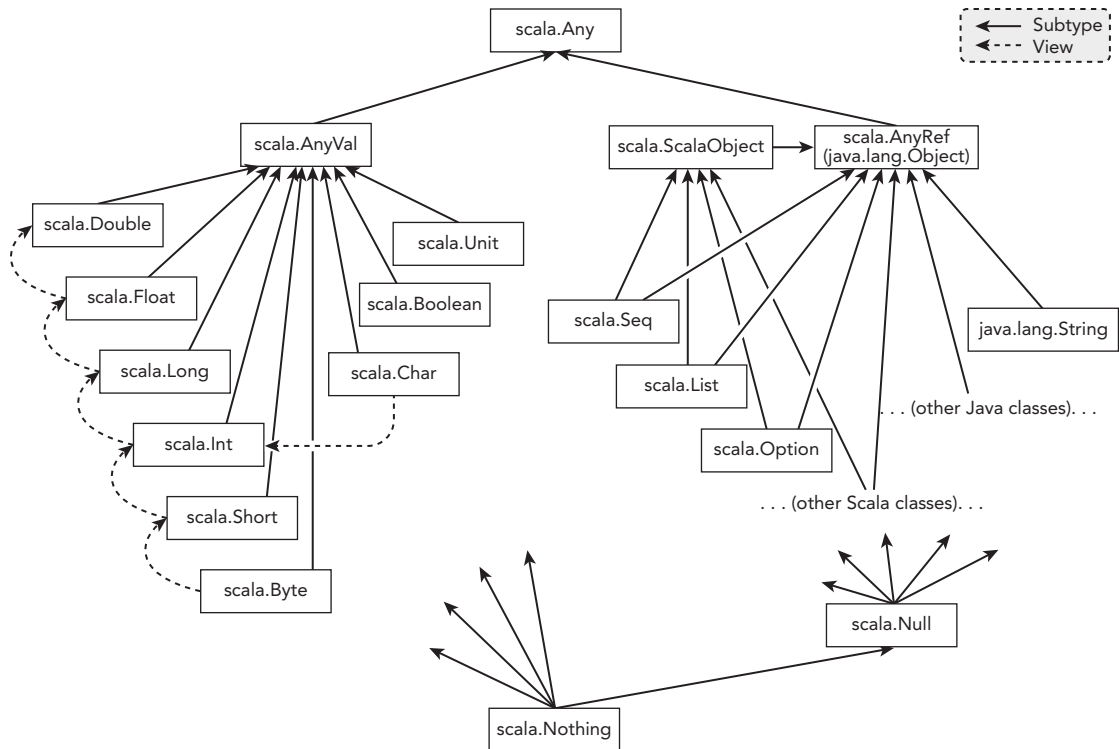


FIGURE 9-1

In Scala all values are instances of a class, included functions. For instance, the following two definitions of a function from `Int` to `Int` are equivalent. The first is just syntactic sugar for the second:

```
scala> val f: Int => Int = _ + 1
f: Int => Int = <function1>
scala> f(42)
res1: Int = 43

scala> val f: Function1[Int, Int] = new Function1[Int, Int] {
  |   override def apply(x: Int): Int = x + 1
  | }
f: Function1[Int,Int] = <function1>

scala> f(42)
res2: Int = 43
```

As you can see, a one-parameter function is just an instance of the `Function1[A, B]` class where `A` is the type of the parameter and `B` the return type.

The superclass of all classes is `Any` and it has two direct subclasses. `AnyVal` and `AnyRef` represent two different class worlds: value classes and reference classes, respectively. Value classes correspond

to the primitive types of Java. So Java's primitives are, in Scala, classes. You don't need boilerplate machinery to wrap primitives in wrapper classes when needed. Scala does it for you. You don't even have to worry about performance because, at the JVM level, Scala uses primitives wherever possible.

Scala also has the `Unit` type which you use in place of `void`. However, this is not the full story. Scala promotes functional programming where expressions are preferred to statements. To make a long story short, expressions always return a value, statements don't.

For example, curly braces always wrap expressions whose value is that of the last expression or `Unit`. This is important to know because it can save you from some pitfalls. For instance, the following line of code looks like a statement but it's actually an expression:

```
{ val a = 42 }
```

Its type is `Unit`, and the return value is the only inhabitant of the `Unit` type, that is `()`. Here's the proof:

```
scala> val b: Unit = { val a = 42 }
b: Unit = ()
```

Here's a common pitfall:

```
scala> val b = if (condition) "a"
```

Here, `condition` is some boolean value. What's the type of `b`? Since the compiler cannot know whether `condition` will be true until runtime, it is not able to infer `String` as the type of `b`. Indeed, its type is `Any`. This is because, if the condition is false, the returned value is `()`, of type `Unit`, and the least upper bound class between `String` and `Unit` is `Any`.

All other classes define reference types. User-defined classes define reference types by default and they always, indirectly, subclass `AnyRef`. Roughly speaking, `AnyRef` corresponds to `java.lang.Object`.

Figure 9-1 also shows implicit conversions, called *views*, between the value classes.

```
scala> val x: Int = 'a' // implicit conversion from Char to Int
x: Int = 97
```

The char `a` is implicitly converted to `Int`. Since Scala 2.10, you can also write your own value classes, which has some important consequences as you'll see in the next section.

Value Classes

Value classes are classes that extend `AnyVal` instead of the default `AnyRef`. There are two main advantages to this:

- **Type safety.** By using a value class, you introduce a new type instead of utilizing a mere `Int`, for example, and the compiler can be by your side from a type safety point of view.
- **No overhead.** No object is allocated for value classes.

You can see it by implementing a simple value class such as the following:

```
class Meter(val value: Double) extends AnyVal {  
  def add(m: Meter): Meter = new Meter(value + m.value)  
}
```

If you decompile this class using `javap`, you will see the following signature for the `add` method:

```
public double add(double);
```

Hold on! In the original signature the `add` method takes a `Meter` type, where is it now? Basically, the `Meter` class is not used in the final bytecode. It uses, instead, the primitive type it is wrapping; `double` in this case. On the other hand, if you declare your class without extending `AnyVal`, it will default to implicitly extend `AnyRef`, which will make it a reference type:

```
class Meter(val value: Double) {  
  def add(m: Meter): Meter = new Meter(value + m.value)  
}
```

Indeed, if you decompile it, this is what the `add` method looks like:

```
public Meter add(Meter);
```

There you go! The class `Meter` gets back in the final bytecode. So you get better type safety because you use an ad hoc type, thus restricting the set of acceptable values, at no performance cost because of the no allocation *thingy*.

The natural question that arises at this point is “Why don’t you make every class a value class?” The answer is that you may not. There are very strict restrictions to what classes are good candidates to become value classes:

- A value class must have only a primary constructor with exactly one `val` parameter whose type is not a value class. Since Scala 2.11 the parameter can also be `private val`.
- It may not have specialized type parameters.
- You may not define classes, traits or objects inside a value class. Also you may not define the `equals` and `hashCode` method.
- A value class must be a top-level class or a member of a statically accessible object.
- It may have only `defs` as members.
- It may not be extended by another class.

The most common use case for a value class is in the pimp-my-library pattern, which is when you extend an existing type through an implicit class. For example, you can think of adding the method `stars` to the `Int` class, which builds a string composed of `N` star characters, where `N` is the `Int`:

```
// private val allowed since Scala 2.11.0  
implicit class IntOps(private val x: Int) extends AnyVal {  
  def stars: String = "*" * x  
}
```

```
val review: String = 5 stars

println(s"review: $review")
```

The output is:

```
review: *****
```

Although value classes are a huge win, what makes Scala stand, as a language, is that it provides three types of polymorphism, which is the subject of the next section.

POLYMORPHISM

Basically there are three types of polymorphism: Subtype, Parametric, and Ad hoc. Scala offers all of them. Java, on the other hand, has just subtype and parametric polymorphism. The king of statically typed functional programming languages, Haskell, has only parametric and ad hoc polymorphism. Even if this could sound more limiting than Scala you'll see that not having subtype polymorphism makes your life easier as a developer because you don't need to worry about variance (subject of the next section). Furthermore, ad hoc polymorphism is much more powerful than subtyping, as you'll see when you meet type classes.

Subtype Polymorphism

This type of polymorphism is typical for object-oriented languages. The traditional example is an abstract superclass, `Shape`, with subclasses `Rectangle`, `Square` and `Circle`:

```
trait Shape {
  def area: Double
}

class Rectangle(val width: Double, val height: Double) extends Shape {
  override def area: Double = width * height
}

class Square(val width: Double) extends Shape {
  override def area: Double = width * width
}

class Circle(val radius: Double) extends Shape {
  override def area: Double = math.Pi * radius * radius
}

val shape: Shape = new Rectangle(10, 5)

println(s"Area: ${shape.area}")
```

The output is:

```
Area: 50
```

Basically, you have a common interface and each implementation provides its own meaning for the methods exposed by the interface—trait in Scala jargon. We won't spend too much time on subtype polymorphism since you are already used to it from other object-oriented languages, such as Java, C#, C++ and so on.

Parametric Polymorphism

If subtype polymorphism is canonical in the object-oriented world, the parametric one is typical in functional languages. Indeed, when programmers talk about polymorphism, without adding any detail, object-oriented developers mean subtyping while functional programmers mean parametric polymorphism.

NOTE *From now on, whenever you see functional languages in the text, we actually mean statically typed functional languages unless otherwise specified.*

Parametric polymorphism also exists in some object-oriented languages and sometimes it is referred to as *generic programming*. For example, Java added parametric polymorphism through generics in version 5.

Here is an example of a method that uses parametric polymorphism:

```
def map[A, B](xs: List[A])(f: A => B): List[B] = xs map f
```

The `map` method takes a `List[A]` and a function from `A` to `B` as input and returns a `List[B]`. As you can see you don't mention concrete types but rather use type parameters—hence parametric polymorphism—to abstract over types. For example you can use that method to transform a `List[Int]` into a `List[String]` or, analogously, a `List[String]` into a `List[Int]`:

```
val stringList: List[String] = map(List(1, 2, 3))(_.toString)

val intList: List[Int] = map(List("1", "2", "3"))(_.toInt)
```

Roughly speaking, whenever your methods have type parameters, you're using parametric polymorphism.

Ad Hoc Polymorphism

Type classes come from the Haskell world and are a very powerful technique used to achieve ad hoc polymorphism.

First of all, a type class has nothing to do with the concept of class of object-oriented languages. In this regard a type class is best seen as a set of types that adhere to a given contract specified by the type class. If this is not very clear, don't worry; an example is worth more than a thousand words.

Consider the equality concept. You know that in order to compare two instances of a given class for equality, in Java, you need to override `equals` for that class. It turns out that writing a correct equality method is surprisingly difficult in object-oriented languages.

You may know that `equals` belongs to `Object`, the superclass of all Java classes. One of the problems is that the signature of `equals` is the following:

```
public boolean equals(Object other)
```

This means that you need to check, among other things, that `other` is an instance of the class you're overriding `equals` for, proceed by doing ugly casts and so on. Furthermore, if you override `equals` you also need to override `hashCode`, as it's brilliantly explained by Joshua Bloch in his book *Effective Java*.

Analyzing all the problems regarding `equals` is out of scope, obviously. The complexity in defining object equality using subtyping emerges in Chapter 30 of *Programming in Scala: A Comprehensive Step-by-Step Guide*, 2nd Edition by Odersky, Spoon, and Venner. Its title is Object Equality and it's about 25 pages long! The authors claim that, after studying a large body of Java code, the authors of a 2007 paper concluded that almost all implementations of `equals` methods are faulty.

So, that said, do you have a better alternative to compare two objects for equality? Yes, you guessed it: type classes. The recipe of a type class is not complicated. It consists of just three ingredients. First of all, capture the concept into a trait:

```
trait Equal[A] {
  def eq(a1: A, a2: A): Boolean
}
```

The second thing you need to do is define a method that takes, implicitly, an instance of that trait. Typically you do it within the companion object:

```
object Equal {
  def areEqual[A](a1: A, a2: A)(implicit equal: Equal[A]): Boolean =
    equal.eq(a1, a2)
}
```

The `areEqual` method says: “Give me two instances of any class `A` for which exists an implicit instance of `Equal[A]` in the current scope and I'll tell you if they are equal.” So, the last ingredient is the definition of the instances of `Equal[A]`. For example, suppose you have the `Person` class and you want a case-insensitive comparison between first and last names:

```
case class Person(firstName: String, lastName: String)
```

In order to be able to compare two instances of this class you just need to implement `Equal[Person]` and make it available in the current scope. We'll do it in the `Person` companion object because that's one of the places that are searched when looking for `Equal[Person]` instances in scope:

```
object Person {
  implicit object PersonEqual extends Equal[Person] {
    override def eq(a1: Person, a2: Person): Boolean =
      a1.firstName.equalsIgnoreCase(a2.firstName) &&
      a1.lastName.equalsIgnoreCase(a2.lastName)
  }
}
```


You have defined the `Equal` type class and provided an implementation for the class `Person`. It sounds like a pattern and, actually, it is. In fact, in Scala, the type class concept is not first class as in Haskell. It's just a pattern that is possible thanks to implicits.

Here you can see it in use:

```
val p1 = Person("John", "Doe")
val p2 = Person("john", "doe")

val comparisonResult = Equal.areEqual(p1, p2)
```

The value of `comparisonResult` is `true`. So, type classes let you model orthogonal concerns in a very elegant way. Indeed, the equality concern has nothing to do with the model, `Person`, if you think about it. Furthermore, the solution provided by the type class is more type safe than that provided by the `equals` method. That is, the `areEqual` method takes two instances of the same class and not `Object`. You don't need ugly downcasts or other abominations.

Talking about implicit resolution, another place inspected when looking for implicit values is the companion object of the type class. So you could have defined the instance of `Equal[Person]` also in the `Equal` companion object. A thorough analysis of implicit resolution policy is out of scope, but can be found here <http://eed3si9n.com/implicit-parameter-precedence-again>. In order to have a direct comparison with the object-oriented solution you can reimplement the shape example, seen in the subtype polymorphism section, using type classes.

First of all, the trait that capture the concept:

```
trait AreaComputer[T] {
  def area(t: T): Double
}
```

This trait means: “Given a type `T` you can compute its area, which is of type `Double`.” You don't say anything about `T`. The following case classes, instead, represent the model:

```
case class Rectangle(width: Double, height: Double)

case class Square(width: Double)

case class Circle(radius: Double)
```

At this point you need to provide the method that takes a `T` and, implicitly, an implementation of the previous trait, plus the implementations for your model:

```
object AreaComputer {
  def areaOf[T](t: T)(implicit computer: AreaComputer[T]): Double =
    computer.area(t)

  implicit val rectAreaComputer = new AreaComputer[Rectangle] {
    override def area(rectangle: Rectangle): Double =
      rectangle.width * rectangle.height
  }
}
```

```

implicit val squareAreaComputer = new AreaComputer[Square] {
  override def area(square: Square): Double =
    square.width * square.width
}

implicit val circleAreaComputer = new AreaComputer[Circle] {
  override def area(circle: Circle): Double =
    math.Pi * circle.radius * circle.radius
}

```

Here is a usage example:

```

import AreaComputer._
val square = Square(10)
val area = areaOf(square)

```

It seems like type classes are a nicer and more powerful alternative to subtyping and, indeed, they are. Just think that, using type classes, you don't need to access the source code of a class to add a behavior. You just provide an implementation of the type class for that class and you're done.

Type classes are so important that very famous Scala libraries you may already have heard of, such as scalaz, cats, shapeless and so on couldn't even exist without them. You also saw that the concept is not that complex after all, it's just a pattern with the same three steps applied over and over.

BOUNDS

Bounds in Scala are used for two main purposes:

- Provide evidence in the context of implicits
- Restrict the set of acceptable types

The former is served by context bounds—the latter by upper and lower type bounds.

Context Bounds

Context bounds are just syntactic sugar that lets you provide the implicit parameter list. So, a context bound describes an implicit value. It is used to declare that for some type A , there is an implicit value of type $B[A]$ available in scope.

As an example consider the `areEqual` method defined earlier in this chapter when you saw type classes:

```
def areEqual[A](a1: A, a2: A)(implicit equal: Equal[A]): Boolean = equal.eq(a1, a2)
```

You can, automatically, translate it to the analogous one that uses a context bound:

```
def areEqual[A: Equal](a1: A, a2: A): Boolean = implicitly[Equal[A]].eq(a1, a2)
```

You just need to:

1. Change the type `A` to `A: Equal`.
2. Remove the second parameter list where you defined your expected implicit values.
3. Retrieve, in the body of your method, the implicit value through the `implicitly` keyword.

So, yes, context bound is just syntactic sugar you can use in place of explicitly defining the implicits required (no pun intended). It's just a matter of taste choosing one syntax over the other.

However, there are cases where you may not use context bounds and you're forced to fall back on the explicit syntax. For instance, when your implicit type depends on more than one type you're out of luck with the context bound approach. Take for example the concept of serialization:

```
trait Serializer[A, B] {  
  def serialize(a: A): B  
}  
  
object Serializer {  
  def serialize[A, B](a: A)(implicit s: Serializer[A, B]): B = s.serialize(a)  
  
  // implementations of Serializer[A, B]  
}
```

The `Serializer` trait encapsulates the concept of taking a type `A` and serializing it into the `B` type. As you can see, the `serialize` method of the `Serializer` companion object could not use the context bound syntax for `Serializer[A, B]` since it takes two type parameters.

Before closing the section on context bounds let's show you a common trick used to allow easy access to type class instances. Basically, you just need to define an `apply` method on the companion object:

```
object Equal {  
  def apply[A: Equal]: Equal[A] = implicitly[Equal[A]]  
}
```

This will let you rewrite the `areEqual` method as follows:

```
def areEqual[A: Equal](a1: A, a2: A): Boolean = Equal[A].eq(a1, a2)
```

Instead of using the `implicitly[Equal[A]].eq(a1, a2)` you just write `Equal[A].eq(a1, a2)`, which is cleaner and clearer. You can do that because, as you may already know, `Equal[A]` is the same as `Equal[A].apply`.

Scala also has the concept of *view bounds* but since they were deprecated in 2.11 we won't cover them here.

Upper and Lower Bounds

In Scala, type parameters may be constrained by a type bound. If you define your type without using a bound there will be no restriction on the possible types one can use. Whenever you want to restrict the type you need a type bound.

If you think about it, it makes sense to desire a restriction over the type. After all, we love static typing also because it lets us restrict the function domain. On the other hand, in dynamic typing a function takes zero or more parameters of any type and returns a value of any type.

Without further ado, here is an example of upper bound:

```
sealed trait Animal {
  def name: String
}

case class Cat(name: String, livesLeft: Int) extends Animal

case class Dog(name: String, bonesHidden: Int) extends Animal

def name[A <: Animal](animal: A): String = animal.name
```

The name method is where upper bound comes into play. The `A <: Animal` type signature means: “Accept any type that is an `Animal` or any of its subclasses.” Lower bounds work in a similar fashion in that they restrict the type to its superclasses instead of its subclasses. The syntax is the following:

```
A >: Animal
```

This means that the type `A` must be of type `Animal` or any of its superclasses. In this particular case the superclass of `Animal` is `AnyRef`. While the usefulness of upper bounds is obvious, a lower bound might sound useless at first sight. However, when you meet variance in the next section, you’ll see that you’re forced to use lower bounds to make your code work in some scenarios.

Variance

Since Scala has both subtype and parametric polymorphism you’re forced to face the concept of variance sooner or later. Consider the following scenario:

```
sealed trait Fruit {
  def describe: String
}

class Orange extends Fruit {
  override def describe: String = "Orange"
}

class Apple extends Fruit {
  override def describe: String = "Apple"
}

class Delicious extends Apple {
  override def describe: String = "Apple Delicious"
}

class Box[A]

def describeContent(box: Box[Fruit]): String = ???
```

```
val oranges = new Box[Orange]

describeContent(oranges) // does not compile
```

When you try to compile this code you get an error similar to the following:

```
type mismatch;
[error] found   : Box[Orange]
[error] required: Box[Fruit]
...
```

What’s the problem? Basically, even if `Orange` is a subclass of `Fruit`, there’s no such relationship between `Box[Orange]` and `Box[Fruit]`. This came out of mixing subtype polymorphism—the `Fruit` class hierarchy—with parametric polymorphism, `Box[A]`. The previous error message is not the full story though. Indeed, the Scala compiler is kind enough to tell you what a possible fix could be. Indeed it continued with:

```
[error] Note:Orange <:Fruit, but class Box is invariant in type A.
[error] You may wish to define A as +A instead.
```

In this specific case, it actually tells you what to do. Before following its suggestion let’s see, in Table 9-1, what are the available options when variance comes into play.

TABLE 9-1: Variance Types

VARIANCE TYPE	SYNTAX	MEANING
Covariant	Box[+A]	If B is a subtype of A, then Box[B] is also a subtype of Box[A]
Contravariant	Box[-A]	If B is a subtype of A, then Box[A] is also a subtype of Box[B]
Invariant	Box[A]	Even if B is a subtype of A, Box[A] and Box[B] are unrelated

```
class Box[+A]
```

You just need to put the plus sign before the type, that’s all. Well, not really, since nothing comes for free. Indeed, try to make `Box` more interesting by adding a method to it:

```
class Box[+A] {
  def describe(a: A): String = ???
}
```

If you try to compile this code this time you’ll get the following error:

```
covariant type A occurs in contravariant position in type A of value a
[error]   def describe(a: A): String = ???
[error]                   ^
```

This time the compiler does not tell you what to do. The very reason behind this error has its roots in Category Theory, a branch of mathematics, which is, obviously, out of scope in a pragmatic book

like this one. However you can, almost automatically, fix this type of error by following some rules. For example, you can fix the previous compilation error by changing the code as follows:

```
class Box[+A] {
  def describe[AA >: A] (a: AA): String = ???
}
```

What are you doing here? Basically you are saying to the compiler: “Hey, I promise that my type parameter is not simply A but AA, which is an A or one of its superclasses.” This way you satisfied the contravariant position the compiler was talking about.

Now, in order to better understand variance, consider the (simplified) signature of the `Function1` class of the standard library:

```
trait Function1[-T1, +R] {
  def apply(t : T1) : R
  ...
}
```

As you can see it’s contravariant for its input type parameter and covariant for the output one. Some examples will hopefully make the reason for this clearer.

Suppose you have the following function and an instance of `Apple`:

```
def f(apple: Apple, g: Apple => Apple): Fruit = g(apple)

val apple = new Apple
```

It’s a simple higher-order function that applies the `g` function to the `apple` object passed in.

Now, given that variance declaration for the `Function1` type, experiment a bit to understand what type of functions you can pass as `g` by playing with the input and output type. Of course the implementations of the examples are deliberately simple because I want you to concentrate on the variance subject, not on understanding complex implementations.

First things first, consider a function that has exactly the signature required by `g`, that is `Apple => Apple`:

```
val appleFunc: Apple => Apple = x => identity(x)
```

Of course, you can pass `appleFunc` to `f` since it matches exactly the `g` signature:

```
f(apple, appleFunc)
```

No problem; it compiles as expected. Now consider the following function:

```
val fruitFunc: Fruit => Apple = x =>
  if (x.describe == "Apple") new Apple
  else new Delicious
```

It goes from `Fruit` to `Apple`. Should the function `f` accept this type of function as `g`? Yes, absolutely! Since `Function1` is contravariant in its input type parameter this means that it can accept the `Apple` type and all its superclasses and `Fruit`, obviously, respects this rule.

It makes perfect sense if you think about it for a moment. What can a function, which goes from `Fruit` to `Apple`, do on the input parameter of type `Fruit`? For sure—less specific things than on a subtype of `Fruit`, such as `Apple`. So it's safe passing a function with a less specific input type parameter or, said differently, with a contravariant type parameter.

Now that we justified the contravariant type let's try to do the same with the covariant one. Take a look again at the `fruitFunc` function. If you look closely, you'll notice you are already exploiting the covariance of the output type parameter for functions. Indeed, `fruitFunc` returns an instance of `Apple` in one case and an instance of `Delicious`, `Apple`'s subclass, in all other cases. This is possible due to the covariance of the output type.

It's plausible for a very simple reason, that is the caller of the function expects all the methods on `Apple` to be available. Now, you have the guarantee that `Apple`'s subclasses have, at least, all `Apple`'s methods implemented. That's the reason why the only logical choice for the output type parameter is to be covariant.

At this point you could say: "OK, now I'm convinced that `Function1` must be contravariant in its input type parameter and covariant in its output one. But what does this have to do with the trick used in the class `Box` to make it compile?" It does if you look at the class `Box` in a more abstract way. Indeed, its `describe` method takes an input type and returns an output type. Well, you can say that it is isomorphic to `Function1`! As a matter of fact consider the following code:

```
val box = new Box[Apple]
val f: Apple => String = box.describe
```

As you can see it transforms the `Box`'s `describe` method to a function through a process called *eta-expansion*. If you don't know it don't worry about its details; just consider it a means of coercing (converting) methods into functions.

At this point the reason the Scala compiler complained when you tried to use a covariant type in a contravariant position should start to make sense.

Of course you could constrain `A` to be a subclass of `Fruit` and delegate the `Box`'s `describe` method to `Fruit`:

```
class Box[+A <: Fruit] {
  def describe[AA >: A <: Fruit](a: AA): String = a.describe
}
```

As you can see, again, bounds to the rescue! In this case, for `AA`, you have a multiple bound: one to satisfy the variance and the other to constrain it to a `Fruit` type.

Before closing this paragraph here is the other common type of error you can get using variance. Consider the following class:

```
class ContraBox[-A] {
  def get: A = ???
}
```

If you try to compile it you'll get the infamous error:

```
contravariant type A occurs in covariant position in type => A of method get
```

After having reasoned about `Function1` and why the return type of it makes sense to be covariant you may already know the reason behind the previous error.

Just try to see the `get` method as a function `() => A`, that is a zero-parameter function. Can you spot the problem now? The type returned by a function should be covariant, while `A` is contravariant. Don't worry you can fix this too, this time using an upper bound:

```
class ContraBox[-A] {
  def get[AA <: A]: AA = ???
}
```

There you go; this time the code will compile.

Of course, given the practical approach that a programming book must have, we tried to oversimplify some concepts but, from a pragmatic point of view, we think this is more than acceptable.

If this is the first time you've encountered the concept of variance, don't worry if something is not completely clear now. Working with it will become natural for you.

At this point you've seen enough about Scala's type system to *survive* while coding in Scala. Actually bounds, variance and ad hoc polymorphism, through type classes, are enough knowledge to let you write very abstract and polymorphic Scala code. In the following sections you'll see other features of the powerful Scala type system.

OTHER NICETIES

As stated at the beginning of this chapter, Scala's type system is a subject too complex and powerful to be covered in a single chapter of a book. However in the following sections you'll see other niceties of the type system.

Take into account that in Scala you can do the same thing in many different ways. For example, given a problem to model, you can do it using type classes. On the other hand, someone else may prefer self-recursive types—you'll see them in a bit. It depends both on the problem at hand and the taste of the programmer. That said, it's important to know the tools you have and equally important to choose the right one. However this last peculiarity cannot be taught; you just learn it through practice.

Self-Type Annotations

Self-type annotations allow you to serve, mainly, two purposes:

- Provide an alias for the `this` keyword.
- Impose dependencies over a class so that, in order to be instantiated, its client needs to satisfy them otherwise the compiler will complain.

Here is an example for the first case:

```
trait Foo { self =>
  def message: String

  private trait Bar {
    def message: String = this.message + " and Bar"
  }

  val fullMessage = {
    object bar extends Bar

    bar.message
  }
}

object FooImpl extends Foo {
  override def message: String = "Hello from Foo"
}

println(s"Message: ${FooImpl.fullMessage}")
```

First of all, you can see the self-type annotation right after the `Foo` trait declaration. You'll see how to use the `self` keyword in a bit.

Indeed, the previous code compiles but there's an insidious bug that will make your stack explode because of a nasty recursion. Look at the `Bar` trait. The programmer's intention here was to implement the `Bar`'s `message` method as the concatenation of the `Foo`'s `message` method and the " and Bar" string. However, `this.message` refers, recursively, to `Bar`'s `message` and not to `Foo`'s. You can easily fix this by using the `self` alias as follows:

```
private trait Bar {
  def message: String = self.message + " and Bar"
}
```

The rest of the code remains unchanged.

The second and more important use of the self-type annotation is to provide dependencies among types. Consider this simple example:

```
trait Foo {
  def message: String
}

trait Bar { self: Foo =>
  def fullMessage: String = message + " and Bar"
}
```

Here you're saying: "Dear compiler, the `Bar` trait depends on the `Foo` trait. This is pretty obvious since we're using the `message` method belonging to `Foo` within the `fullMessage` method implementation. Now, if a client of this API tries to implement the `Bar` trait without providing also an

implementation of the `Foo` trait, would you be so kind to raise a compilation error?” Since the compiler is kind it will fulfill your request, indeed the following attempt wouldn’t compile:

```
object BarImpl extends Bar
```

The error message is something like:

```
illegal inheritance;
[error] self-type BarImpl.type does not conform to Bar's selftype Bar with Foo
```

That basically means: “Where is my `Foo` implementation?” Here’s how you can fix it:

```
trait MyFoo extends Foo {
  override def message: String = "Hello from Foo"
}

object BarImpl extends Bar with MyFoo

println(s"Message: ${BarImpl.fullMessage}")
```

Now the compiler is happy and if you run the code you get the following string printed to the console:

```
Message: Hello from Foo and Bar
```

So, here is a little recap. The syntax for self-type annotations can be of two types:

- `self =>`
- `self: YourType =>`

The former is just an alias for the `this` keyword and can be useful in some situations. The latter is a constraint that won’t make the code compile if the client does not satisfy it.

You can also impose multiple dependencies using the syntax:

```
self: Type1 with Type2 with Type3 ...
```

This means that the client needs to provide implementations for `Type1`, `Type2`, `Type3` and so on in order to make things work.

Self-type annotations are used as the foundation of the Cake Pattern, brilliantly described here <http://jonasboner.com/2008/10/06/real-world-scala-dependency-injection-di/>. To tell you the truth, we’re not the biggest fans of the Cake Pattern when it comes to dependency injection. We prefer to use type classes or other techniques. Nevertheless, it can be useful in some situations so we suggest you go through that article.

Self-type annotations also serve as a basis for self-recursive types, which are the subject of the next section.

Self-Recursive Types

One of the advantages of using a statically typed language is that you can use the type system to enforce some constraints. Scala provides *self-recursive types*, also known as *F-bounded polymorphic types* that—along with self types—let you put powerful constraints to your type definitions.

Terminology apart, here is one of the use cases where this could be useful. Consider the following example which does not use a self-recursive type:

```
trait Doubler[T] {  
  def double: T  
}  
  
case class Square(base: Double) extends Doubler[Square] {  
  override def double: Square = Square(base * 2)  
}
```

So far so good; the compiler will not complain. The problem is that it won't complain even if you write something outrageous like the following code:

```
case class Person(firstname: String, lastname: String, age: Int)  
  
case class Square(base: Double) extends Doubler[Person] {  
  override def double: Person = Person("John", "Smith", 42)  
}
```

You want to avoid something like that by enforcing a compile-time check. Enter a self-recursive type:

```
trait Doubler[T <: Doubler[T]] {  
  def double: T  
}
```

By using this definition of `Doubler` you're saying: "Hey, if someone tries to extends `Doubler` with a type that doesn't extend `Doubler` in turn (hence self-recursive), do not compile it." In this case the previous definition of `Square`, which extends `Doubler[Person]`, wouldn't compile.

Note that self-recursive types are not specific to Scala. Indeed Java uses them too. Take, for example, the `Enum` definition:

```
public abstract class Enum<E> extends Enum<E> {  
  implements Comparable<E>, Serializable {  
    ...  
  }
```

`E extends Enum<E>` in *Javanesque* means exactly `E <: Enum[E]`.

F-bounded polymorphic types are of great help, but sometimes they are not enough to enforce the constraints you need. Indeed, the previous definition of `Doubler` still has one problem. Consider the next code:

```
trait Doubler[T <: Doubler[T]] {  
  def double: T  
}
```

```

case class Square(base: Double) extends Doubler[Square] {
  override def double: Square = Square(base * 2)
}

case class Apple(kind: String) extends Doubler[Square] {
  override def double: Square = Square(5)
}

```

Can you spot the problem? Look at the `Apple` definition, which extends `Doubler[Square]` instead of `Doubler[Apple]`.

This code compiles because it respects the constraint put by the `Doubler` definition. Indeed `Square` extends `Doubler` so it can be used in `Apple`. Sometimes this is what you want in which case the self-recursive type will do. In cases when you don't want this to happen a self type can work this out:

```

trait Doubler[T <: Doubler[T]] { self: T =>
  def double: T
}

```

Now if you try to compile the previous definition of `Apple`, the compiler will complain by saying something like:

```

error: illegal inheritance;
  self-type Apple does not conform to Doubler[Square]'s selftype Square
    case class Apple(kind: String) extends Doubler[Square] {
                                   ^

```

Abstract Type Members

In Scala, besides abstract methods and fields, you can also have abstract types within your trait or abstract classes. Here is a simple example:

```

trait Food

class Grass extends Food {
  override def toString = "Grass"
}

class Fish extends Food {
  override def toString = "Fish"
}

trait Animal {
  type SuitableFood <: Food

  def eat(food: SuitableFood): Unit = println(s"Eating $food...")
}

class Cow extends Animal {
  type SuitableFood = Grass
}

class Cat extends Animal {
  type SuitableFood = Fish
}

```

Look at the definition of `SuitableFood` within the `Animal` trait. Using that declaration you're just saying: "The type `SuitableFood` is abstract and it's a subclass of `Food`." The classes that extend `Animal` are responsible for *refining* the type definition. For example, `Cow` defines the `Grass` type as `SuitableFood`. Similarly, the `Cat` class refines `SuitableFood` using the `Fish` type. Now, the following code compiles as:

```
val grass = new Grass
val cow = new Cow

val fish = new Fish
val cat = new Cat
```

On the other hand, if you try to feed a cow with fish and/or a cat with grass it won't work:

```
cow.eat(fish) // won't compile

cat.eat(grass) // won't compile
```

At this point you can object: "Hey, I could have done the same thing using a type parameter instead of an abstract type member." You're right. Indeed, the `Animal` hierarchy could have been implemented as follows:

```
trait Animal[SuitableFood <: Food] {
  def eat(food: SuitableFood): Unit = println(s"Eating $food...")
}

class Cow extends Animal[Grass]

class Cat extends Animal[Fish]
```

The result is the same. At this point the question is: "When to prefer abstract type members to type parameters?" Well, many times it's just a matter of taste. However, when the number of type parameters is not just one, the abstract type approach could make your code easier to read.

Furthermore, we use the following rule of thumb, but take it with a grain of salt and evaluate case by case to choose which technique is best suited for the problem at hand.

We tend to use type parameters if we find the type does not make sense without citing its type parameter; otherwise we could opt for abstract types. For example, `List`, `Option`, `Set` and so on do not make much sense without citing their contained type. List of what? `List[Int]`, `List[String]`, and so on. On the other hand, in the previous example, `Animal` makes perfect sense without citing the type. It's more elegant even in code. You notice it more if you explicitly use a type annotation. Compare these two declarations:

```
val animal: Animal = new Cow

val animal: Animal[Grass] = new Cow
```

Moreover, there are corner cases where abstract type members could greatly simplify the implementation of the API and its client code. For example, Bill Venners used it for `ScalaTest`'s fixtures, as he explains here: <http://www.artima.com/weblogs/viewpost.jsp?thread=270195>. Also, the very famous `shapeless` library, where the Scala type system is pushed to the limit, makes extensive use

of abstract types. In this excellent post, Travis Brown explains brilliantly a corner case where using abstract type members made the difference: <http://stackoverflow.com/questions/34544660/why-is-the-aux-technique-required-for-type-level-computations/34548518#34548518>.

Dynamic Programming

Scala is a statically typed language, and this is a good thing, as you’ve seen so far. However, there are times when being able to use the peculiarities of dynamically typed languages can be a big plus for some types of problems.

In this regard, Scala provides two interesting mechanisms through which you can emulate dynamic programming for those parts of your application that need it. The techniques we’re referring to go by the names of Structural Types and the Dynamic trait.

Structural Types

Structural types let you accomplish the so-called *duck typing*, typically found in dynamic languages. It can be summarized with a sentence: “If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.”

In duck typing, a programmer is only concerned with making sure that objects behave as demanded of them in a given context, rather than ensuring that they are of a specific class.

For example, you could say that a resource is closable if its class contains the `close` method. Here is how you could model this requirement using structural types:

```
def closeResource(resource: { def close(): Unit }): Unit = {
  println("Closing resource...")

  resource.close()
}
```

The duck typing incantation happens by defining the resource type as:

```
{ def close(): Unit }
```

It basically means: “Any type that has a zero-parameter `close` method, which returns `Unit`, is suitable to be passed to the `closeResource` method.” You can see it in action in the following example:

```
class Foo {
  def close(): Unit = println("Foo closed")
}

class Bar {
  def close(): Unit = println("Bar closed")
}

val foo = new Foo
closeResource(foo)

val bar = new Bar
closeResource(bar)
```

The output is:

```
Closing resource...
Foo closed
Closing resource...
Bar closed
```

Even if structural types give you the power of duck typing, they have the advantage, over a dynamically typed language, of providing some form of type safety at compile time. For example, the following code wouldn't compile:

```
val baz = new Baz
closeResource(baz)
```

The error you get is something like:

```
[error] ...: type mismatch;
[error]   found: baz.type (with underlying type Baz)
[error]   required: AnyRef{def close(): Unit}
[error]   closeResource(baz)
[error]                   ^
```

This is pretty self-explanatory.

You can also require the existence of more than one method. In this case it's cleaner to define a type alias as in:

```
type Resource = {
  def open(): Unit
  def close(): Unit
}

def useResource(resource: Resource): Unit = {
  resource.open()

  // do what you need to do

  resource.close()
}
```

Well, it seems like structural types have no downside, but we know that all that glitters is not gold. Indeed, the machinery behind structural types is the reflection and, as such, it has a non-trivial runtime cost. This is one of the reasons why you should strive to avoid structural types as much as you can. For instance, both of the previous examples can be elegantly solved using type classes and, by now, you should also know how to do it.

Dynamic Trait

Structural types let you write generic code that will work, provided that a class has the given methods. The Dynamic marker trait, on the other hand, addresses the somewhat dual problem. It lets you

pretend that an object has fields and/or methods that actually do not exist at declaration time. An example will make this clear:

```
import scala.language.dynamics

class Magic extends Dynamic {
  def selectDynamic(field: String): Unit = println(s"You called $field")
}

val magic = new Magic

magic.foo
magic.bar
```

The output of the previous code is:

```
You called foo
You called bar
```

The first thing to do is import the feature, since it's disabled by default. Alternatively, you can add it to the `scalacOptions` key in your SBT build file as follows:

```
scalacOptions += "-language:dynamics"
```

As you can see, even if the `Magic` class does not contain the `foo` and `bar` fields, you can still call them because it extends `Dynamic`. The `selectDynamic` method is where the calls to fields are rooted. The string passed as parameter is the name of the field you called.

You can also easily chain calls:

```
class Magic extends Dynamic {
  def selectDynamic(field: String): Magic = {
    println(s"You called $field")

    this
  }
}

val magic = new Magic

magic.foo.bar
```

The output will be the same as the previous example. As you may have guessed the trick here is that, instead of `Unit`, the method returns `this`.

Apart from fields, you can also fake methods:

```
class Magic extends Dynamic {
  def applyDynamic(name: String)(args: Any*): Unit =
    println(s"method '$name' called with arguments: ${args.mkString(", ")}")
}
```



```
val magic = new Magic  
  
magic.someMethod("foo", 42, List(1, 2, 3))
```

The output is:

```
method 'someMethod' called with arguments: foo, 42, List(1, 2, 3)
```

The name parameter of the first section of `applyDynamic` is the name of the invoked method. The `args` parameter of its second section is a `Varargs` of type `Any`, that is any number and type of arguments.

There are other methods you can use to build full incantations using the `Dynamic` trait, but we won't cover them for space's sake so please refer to the Scala API for more info.

SUMMARY

It's been a long road. The Scala type system is a very hard topic, and in this chapter you've seen its most used features. Don't worry if, at this time, something is not crystal clear. You'll digest these concepts while working with them.

Even from an OOP perspective, the Scala type system is superior to the Java one, since it has no distinction between primitives and reference types. This makes your code more coherent and with less boilerplate to go back and forth from the primitive world.

You've also seen that Scala offers the very powerful ad hoc polymorphism through type classes. After talking about bounds you've met the concept of variance that was introduced using a different and, maybe, more friendly approach.

In the end, after covering other goodies of the type system, you've seen what the language has to offer when it comes to dynamic typing. Take your deserved rest, then a deep breath. See you in the next chapter with an advanced type system concept and a demystification of the most common functional design patterns.

10

Advanced Functional Programming

WHAT'S IN THIS CHAPTER?

- Understanding advanced features of the type system
- Providing functional design pattern knowledge
- Mastering the relationship between programming and algebraic structures

The previous chapter gave you solid foundations to move with agility among Scala's type system. This chapter starts by introducing an advanced type system concept that goes by the name of higher-kinded types.

It proceeds with an overview of the most common functional design patterns, such as functor, applicative and monad. As you'll see, these concepts are much easier than you might think.

The chapter ends with an analysis of two very simple algebraic structures, showing how they can be exploited to write very elegant code.

HIGHER-KINDED TYPES

Option or List are not proper types, but they are kinds. A kind is the type of a type constructor. Simply speaking, it means that, in order to be constructed, it needs another type.

In type theory, kinds like `List` and `Option` are indicated using the following symbology: `* -> *`. The star symbol stands for type. So, `* -> *` means: "Give me a type and I construct another type." Indeed, given the `List` kind, if you provide the `Int` type then your final type will be `List[Int]`. If you furnish a `String` then it'll be a `List[String]` and so on. Table 10-1 shows the most commonly used kinds.

TABLE 10-1: Kinds in Type Theory

SYMBOL	KIND	EXAMPLES
*	Simple type. Also known as nullary type constructor or proper type.	Int, String, Double, ...
* -> *	Unary type constructor.	List, Option, Set, ...
* -> * -> *	Binary type constructor.	Either, Function1, Map, ...
(* -> *) -> *	Higher-order type operator, higher-kinded type for friends.	Foo[F[_]], Bar[G[_]], Functor[F[_]], Monad[M[_]], ...

So, a higher-kinded type is a type that, in order to be constructed, needs a kind that is a type constructor. It sounds complex, but it really isn't. An example should help.

Suppose you want to capture, in a type class, the concept represented by the `map` method you've met in `Option`, `List`, `Set` and so on. For this purpose, you can write the following abstraction:

```
import scala.language.higherKinds

trait Mapper[M[_]] {
  def map[A, B] (ma: M[A]) (f: A => B): M[B]
}
```

The first thing to do, in order to avoid a compiler warning, is import the higher-kinded type's feature. Alternatively, you can add it to the `scalacOptions` key in your SBT build file as follows:

```
scalacOptions += "-language:higherKinds"
```

In the previous example, `Mapper` is a higher-kinded type. Indeed, given `M[_]`, which is a unary type constructor, you can construct the `Mapper` type.

You've seen, from Table 10-1, that unary type constructors are, for example, `List` and `Option`, so let's go ahead and implement `mapper` for them. Furthermore, we'll use the type class pattern in order to define a method that will take an instance of `Mapper` implicitly, just to make the things more interesting:

```
object Mapper {
  def apply[M[_]: Mapper]: Mapper[M] = implicitly[Mapper[M]]

  def map[A, B, M[_]: Mapper] (ma: M[A]) (f: A => B): M[B] = Mapper[M].map(ma) (f)

  implicit val optionMapper = new Mapper[Option] {
    override def map[A, B] (ma: Option[A]) (f: A => B): Option[B] = ma.map(f)
  }

  implicit val listMapper = new Mapper[List] {
    override def map[A, B] (ma: List[A]) (f: A => B): List[B] = ma.map(f)
  }
}
```

Look at the `map` method. As you can see, the context bound can also be used with higher-kinded types.

The `Mapper` companion object also contains two implicit instances, one for `Option` and the other for `List`. The implementation is very trivial since both `Option` and `List` already have an implementation of the `map` method.

Follow a couple of examples of the `map` method in action:

```
import Mapper._

val as: List[Int] = List(1, 2, 3)
val bs: List[Int] = map(as) (_ + 1)

println(s"bs: $bs") // prints List(2, 3, 4)

val a: Option[String] = Some("1")
val b: Option[Int] = map(a) (_.toInt + 1)

println(s"b: $b") // prints Some(2)
```

As you can see, the `map` method can be applied, seamlessly, both to `Option` and `List` instances. Ad hoc polymorphism, for the win.

Don't worry if, at this stage, something is not completely clear. Working with higher-kinded types will become second nature to you because they let you build very powerful abstractions such as functors, applicative functors, monads and so on, which are the subjects of the next sections.

FUNCTIONAL DESIGN PATTERNS

If you've been using functional programming for a while you may have heard about one or all of the following funny-looking terms: functor, applicative functor and monad.

These concepts can be examined from two different approaches: using a very abstract mathematical branch that goes by the name of Category Theory or opting for a more pragmatic approach and leaving the theory for a later moment when you're prepared to dig deeper into the subject.

In this book we'll use the practical approach, trying to demystify these concepts that are, as you'll see, not too complex.

Functor

The first evidence that these concepts are not so complex is given by the functor. Do you remember the `Mapper` type class seen in the previous section? That's a Functor. We called it `Mapper` just to arrive at this demystifying moment. So, you can now rewrite the `Mapper` type class and companion object as follows:

```
trait Functor[F[_]] {
  def map[A, B] (fa: F[A]) (f: A => B): F[B]
}
```

```
object Functor {
  def apply[F[_]: Functor]: Functor[F] = implicitly[Functor[F]]

  def map[A, B, F[_]: Functor] (fa: F[A]) (f: A => B): F[B] = Functor[F].map(fa)(f)

  implicit val optionFunctor = new Functor[Option] {
    override def map[A, B] (fa: Option[A]) (f: A => B): Option[B] = fa.map(f)
  }

  implicit val listFunctor = new Functor[List] {
    override def map[A, B] (fa: List[A]) (f: A => B): List[B] = fa.map(f)
  }
}
```

Using a functor you can apply a given function to a value that is inside a *context*. Examples of contexts are: `Option`, `List`, `Try`, `Future`, and so on.

For example, the `Option` type represents the context of an optional value that eschews `null` to handle the no value case. The `List` context is that of representing no value or, at least, one value. The `Try` type represents the context of a possible failure.

`Future`, on the other hand, represents the context where a value may be available in the future. For instance, suppose you have `Some(42)` and you want to:

1. Extract the value 42 outside of the context (`Some`)
2. Apply a function to it
3. Put the result inside the context again

Well, that's basically what a functor is for.

Also, you may have heard of the *lift* word associated with the functor concept. A functor lifts a simple function that goes from `A` to `B` to one that goes from `F[A]` to `F[B]`, where `F` is the functor, e.g. `List` or `Option`. Figure 10-1 illustrates lift, which is just the effect of `map`.

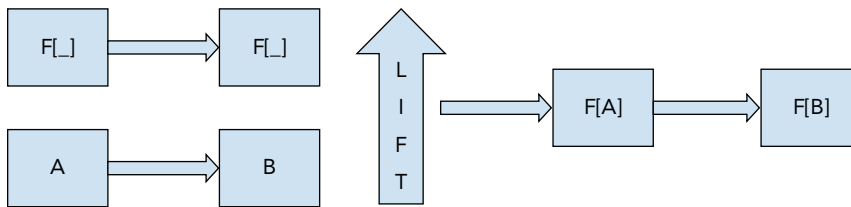


FIGURE 10-1

Actually, you could have written the `map` method of `Functor`, equivalently, as follows:

```
trait Functor[F[_]] {
  def map[A, B] (f: A => B): F[A] => F[B]
}

object Functor {
  def apply[F[_]: Functor]: Functor[F] = implicitly[Functor[F]]
}
```

```
def map[A, B, F[_] : Functor] (f: A => B): F[A] => F[B] = fa =>
  Functor[F].map(f) (fa)

  // ...
}
```

Can you see now why they say that, through a functor, you can lift a function of type $A \Rightarrow B$ to one of type $F[A] \Rightarrow F[B]$? Again, lift, another buzzword demystified. The latter is the signature used in Haskell, by the way. The problem with this signature in Scala is that its type inference is not as good as Haskell's. Easy up to now, isn't it? This is not the full story, though.

In order for something to be a functor, it should satisfy a couple of laws. In formulating and elaborating the laws, let's use the latter `map` function signature, because it makes the reasoning easier to follow. However, in *real life*, you'd stick to the previous signature because of the type inference problem aforementioned. Let's see these laws then.

Given `fa: F[A]` and the identity function defined as follows:

```
def identity[A] (a: A): A = a
```

The following laws must hold (in pseudo-code):

```
Law I:   map(identity) (fa) = fa
Law II: map(f compose g) (fa) = map(f) (map(g) (fa))
```

The first law states that if you map the identity function over a functor, the returned functor should be the same as the original one. The second law states that if you have two functions, `f` and `g`, the result of mapping their composition over `fa` must be the same as mapping `g` to `fa` obtaining, say, `fb` and then mapping `f` to `fb`.

If you had to read Law I as a question, it would be: "What's the function that applied to `fa` gives back `fa` unmodified?" If you thought of the identity function you guessed it. So, this means that a potential functor satisfies the first law if and only if:

```
map(identity) = identity
```

You can easily verify that this is true both for `Option` and `List`. For `Option` it means that, for example:

```
Some(42).map(identity) == Some(42)
None.map(identity) == None
```

This is the same reasoning for `List`. So they respect the first law. Now the second. For `Option` it means:

```
Some(42).map(f compose g) == Some(42).map(g).map(f)
None.map(f compose g) == None.map(g).map(f)
```

The second law is also respected—by `List` too. You can say there exists a functor instance for both `Option` and `List`.

Note that the compiler cannot enforce these laws, so you have to test them out yourself to ensure the *functoriness*. However, if you use functors written by others, you don't even need to worry about

these laws, since they should have done it on their side. For example, popular libraries such as scalaz (<https://github.com/scalaz/scalaz>) and cats (<https://github.com/non/cats>) make sure that every functor instance satisfies the aforementioned laws.

Now that we have demystified the functor concept let's proceed with the applicative functor.

Applicative Functor

Before introducing the Applicative Functor (simply Applicative from now on) we want to expose the problem it resolves.

Using a functor you can apply a given function to a value that is inside a context: `Option`, `List`, `Try`, `Future`, and so on. Now, what if the function you want to apply is within a context as well? An example will make this clear.

Suppose you want to write a very trivial method, based on string interpretation, that returns a function to be applied to an `Int` value. For instance, something like the following:

```
def interpret(str: String): Option[Int => Int] = str.toLowerCase match {
  case "incr" => Some(_ + 1)
  case "decr" => Some(_ - 1)
  case "square" => Some(x => x * x)
  case "halve" => Some(x => x / 2)
  case _ => None
}
```

The `interpret` method takes a `String` and returns an `Option[Int => Int]`. If the command is not recognized it returns `None`. Pretty fair.

It would be nice being able to apply the function returned by `interpret`, which is inside the `Option`, to a value wrapped in an `Option` too. Of course, as a result, we still want an `Option`. Stated differently, given:

```
val func: Option[Int => Int] = interpret("incr")

val v: Option[Int] = Some(42)
```

We want to apply the wrapped function represented by `func` to the wrapped value represented by `v`. Applicative solves this type of problem. Here's the definition of the Applicative type class:

```
trait Applicative[F[_]] extends Functor[F] {
  def pure[A](a: A): F[A]

  def ap[A, B](fa: F[A])(fab: F[A => B]): F[B]

  override def map[A, B](fa: F[A])(fab: A => B): F[B] = ap(fa)(pure(fab))
}
```

The methods exposed are `pure` and `ap`. The first is needed to wrap a value into a context. The `ap` method, instead, is exactly what we needed to combine `func` and `v` from the previous example.

Moreover, Applicative extends Functor. Indeed, as you can see from the previous code, by providing `pure` and `ap` you can derive *for free* an implementation of the `map` method, required by the functor.

NOTE *If you have some difficulty understanding the implementation of `map` in terms of `pure` and `ap`, remember to always follow the types. For example, `map` requires an `F[A]` and a function `A => B`. On the other hand, `ap` requires an `F[A]` and an `F[A => B]`. Is there a way to transform `A => B` to `F[A => B]` just to reuse `ap` and make the types align? Sure, `pure` is of the type `A => F[A]`, for all types `A`. The function `A => B` is a type, much like `Int`, `Double` and `String`. So, the solution is to use `pure` to transform `A => B` to `F[A => B]`, and you're done.*

Now, let's provide the applicative instances for `Option` and `List`:

```
object Applicative {
  def apply[F[_]: Applicative]: Applicative[F] = implicitly[Applicative[F]]

  def pure[A, F[_]: Applicative](a: A): F[A] = Applicative[F].pure(a)

  def ap[A, B, F[_]: Applicative](fa: F[A])(fab: F[A => B]): F[B] =
    Applicative[F].ap(fa)(fab)

  implicit val optionApplicative = new Applicative[Option] {
    override def pure[A](a: A): Option[A] = Option(a)

    override def ap[A, B](fa: Option[A])(fab: Option[A => B]): Option[B] = for {
      a <- fa
      f <- fab
    } yield f(a)
  }

  implicit val listApplicative = new Applicative[List] {
    override def pure[A](a: A): List[A] = List(a)

    override def ap[A, B](fa: List[A])(fab: List[A => B]): List[B] = for {
      a <- fa
      f <- fab
    } yield f(a)
  }
}
```

Remember to always *follow the types*.

Now you can finally combine `func` and `v` thanks to `Applicative`:

```
val func: Option[Int => Int] = interpret("incr")

val v: Option[Int] = Some(42)

val result: Option[Int] = ap(v)(func)
```

`Applicative`, like `Functor` and all other concepts coming from Category Theory, come with some laws that must be respected. For a matter of space, we won't cover the applicative laws here. For the moment, you don't need to worry about them either. At this stage of things, you'll use applicatives

exposed by, say, `scalaz` or `cats`, as a user. When and if you need to write your own applicative instances, you can then dig deeper into the applicative laws.

Monad

The moment to talk about monads arrived. By now, however, you should already know that it is a type class with some laws to respect.

Prof. Eugenio Moggi, an Italian professor of computer science, was the first to describe the general use of monads to structure programs.

Monads can be seen as beefed up applicatives, much in the way applicatives can be seen as beefed up functors. Indeed, here is the definition of the `Monad` type class:

```
trait Monad[M[_]] extends Applicative[M] {
  def unit[A] (a: A): M[A]

  def flatMap[A, B] (ma: M[A]) (f: A => M[B]): M[B]

  override def pure[A] (a: A): M[A] = unit(a)

  override def ap[A, B] (fa: M[A]) (fab: M[A => B]): M[B] = flatMap(fab) (map(fa))
}
```

The primitives required by the `Monad` type class are `unit` and `flatMap`.

NOTE *In the Haskell literature you may find the primitives under the names `return` and `bind`, respectively. The symbol used to denote `bind` is `>=>=`. Just in case you happen to read an article about monads in Haskell, this should help.*

Once you have an implementation for `unit` and `flatMap`, you get for free, the implementation of `pure` and `ap` of the `Applicative` type class. Here is the `Monad` companion object that contains the implementation of `Monad` for `Option` and `List`:

```
object Monad {
  def apply[M[_] : Monad]: Monad[M] = implicitly[Monad[M]]

  def flatMap[M[_] : Monad, A, B] (ma: M[A]) (f: A => M[B]): M[B] =
    Monad[M].flatMap(ma) (f)

  implicit val optionMonad = new Monad[Option] {
    override def unit[A] (a: A): Option[A] = Option(a)

    override def flatMap[A, B] (ma: Option[A]) (f: A => Option[B]): Option[B] =
      ma.flatMap(f)
  }
```

```
implicit val listMonad = new Monad[List] {
  override def unit[A] (a: A): List[A] = List(a)

  override def flatMap[A, B] (ma: List[A]) (f: A => List[B]): List[B] =
    ma.flatMap(f)
}
```

Both `Option` and `List` expose the `flatMap` method. Now let's have a look at an example. Suppose you have the following function and value:

```
val sqrt: Double => Option[Double] = { value =>
  val result = math.sqrt(value)

  if (result.toInt.toDouble == result) Some(result) else None
}

val perfectSquare: Option[Double] = Some(49)
```

The `sqrt` function takes a `Double` and, if it's a perfect square, returns the result of applying the square root to the number wrapped in a `Some`; otherwise it just returns `None`. You want to apply the `sqrt` function to the `perfectSquare` value.

Follow the types. You have an `Option[Double]` and a function `Double => Option[Double]`. The `flatMap` method has the following signature:

```
def flatMap[M[_] : Monad, A, B] (ma: M[A]) (f: A => M[B]): M[B]
```

Specifically, if you replace `A` and `B` with `Double` and `M` with `Option` you'll soon realize you have everything you need. Indeed:

```
val res: Option[Double] = flatMap(perfectSquare) (sqrt) // Some(7.0)
```

Again, monads have some laws that must be respected too, but we won't cover them here for space's sake. Moreover, you don't need to know them to exploit the already existent monad instances provided by `scalaz` and `cats`.

The concepts of functor, applicative, monad and many others, useful in the functional programming world, are taken very seriously by pretty famous libraries such as `scalaz` and `cats`. The latter is younger than `scalaz`, but better organized and documented. We recommend that you may start with `cats` and then, once you have mastered some concepts, take a look at the more mature `scalaz` library.

Semigroup

Another concept you may have heard of is the semigroup. It comes from mathematics.

A semigroup is an algebraic structure consisting of a set together with an associative binary operation. So, a semigroup involves a set `S` and a binary operation.

The following properties must hold for a semigroup:

- **Closure:** Given two elements of `S` and applying the binary operation to them, the result must also be in `S`.

- **Associativity:** When combining more than two elements of the set, it doesn't matter which pairwise combination you do first.

An example will make it clearer. Consider the set of integers along with the $+$ binary operation. Does it form a semigroup? Yes. The closure law is satisfied since, by summing two integers, you get back an integer again. Also the associativity holds. Indeed, for example:

$$(4 + 10) + 2 = 4 + (10 + 2)$$

In order to see how these mathematical structures can be of help in your day-by-day coding sessions, let's introduce the monoid, which is just a beefed up semigroup.

Monoid

A semigroup with the identity element is called a monoid. Here's the formal identity element definition:

Identity element: There exists an element e in S such that for every element a in S , $e \bullet a = a \bullet e = a$.

Again, does the set of integers along with the $+$ operation form a monoid? You've already seen that it forms a semigroup. If you can find the identity element then it's also a monoid. For the plus operation, the identity element, or the neutral element, is the zero. Indeed:

$$x + 0 = 0 + x = x, \text{ for all } x \text{ being an integer.}$$

Note that the set of integers with the $*$ operation forms a monoid too. In this case the neutral element is the unit. Indeed:

$$x * 1 = 1 * x = x, \text{ for all } x \text{ being an integer}$$

OK, enough math; let's get back to coding and see how these mathematical concepts can make your code cleaner and more elegant.

First, let's capture the semigroup and monoid concepts:

```
trait Semigroup[A] {
  def append(a1: A, a2: A): A
}
trait Monoid[A] extends Semigroup[A] {
  def zero: A
}
```

Now, suppose that you want to abstract the concept of summation. That is, given a `List[A]`, you want to be able to sum all its elements. Scala already exposes the `sum` method on `List`, but, unfortunately, it works only with numbers. So the following attempts will work:

```
scala> List(1, 2, 3).sum
res0: Int = 6

scala> List(3.14, 42.6, 3).sum
res1: Double = 48.74
```

However, if you try to use `sum` on a `List[String]` it won't work because there's no instance of the `Numeric` type class for `String`:

```
scala> List("hello", " ", " ", "world") .sum
<console>:8: error: could not find implicit value for parameter
num: Numeric[String]
      List("hello", " ", " ", "world") .sum
                                   ^
```

Can you write a more generic `sum` method that will also work for strings? Yes, thanks to the monoid structure. Here is the method definition:

```
def sum[A](elements: List[A])(implicit M: Monoid[A]): A =
  elements.foldLeft(M.zero)(M.append)
```

Look how beautiful and elegant that is! As you already know, `foldLeft` expects an initial element of type `A` and a binary function of type `(A, A) => A`. That's what we are providing through the monoid structure.

In order to make it work for integers you just need to provide an implicit instance of `Monoid[Int]` in scope. For `String`, you implement `Monoid[String]` and make sure it's in scope, implicitly. Let's do it:

```
object Monoid {
  def apply[A: Monoid]: Monoid[A] = implicitly[Monoid[A]]

  implicit val intMonoid = new Monoid[Int] {
    override def zero: Int = 0

    override def append(a1: Int, a2: Int): Int = a1 + a2
  }

  implicit val stringMonoid = new Monoid[String] {
    override def zero: String = ""

    override def append(a1: String, a2: String): String = a1 + a2
  }
}
```

As you can see, the implementation isn't fancy. The identity element for the `String` type is, naturally, the empty string.

Here's the `sum` method in action:

```
val intResult: Int = sum(List(1, 2, 3))
val stringResult: String = sum(List("hello", " ", " ", "world"))

println(intResult)
println(stringResult)
```

The output is:

```
6
hello, world
```

It seems to work like a charm. Now, suppose you want your `sum` function to work with `Option` types too. Do you need to change anything from the previous code? No, you don't. You just need to provide an implementation of `Monoid[Option[A]]`:

```
implicit def optionMonoid[A: Semigroup]: Monoid[Option[A]] =
  new Monoid[Option[A]] {
    def zero: Option[A] = None

    def append(f1: Option[A], f2: Option[A]): Option[A] = (f1, f2) match {
      case (Some(a1), Some(a2)) => Some(Semigroup[A].append(a1, a2))
      case (Some(a1), None) => f1
      case (None, Some(a2)) => f2
      case (None, None) => None
    }
  }
```

What follows is an example of its use:

```
val optionResult: Option[String] =
  sum(List(Some("hello"), Some(", "), None, Some("world")))

println(optionResult)
```

The output is:

```
Some(hello, world)
```

Now some words about one interesting part of the implementation. Notice that we require an implementation of `Semigroup` for `A` so that you can use it to append the two values of type `A` when both values are of type `Some`.

You might wonder why we require it to be a `Semigroup` and not a `Monoid`. The reason is that you only need the `append` method, and it's a good development practice to use the least powerful abstraction that will get the job done. This makes your code more reusable. Furthermore, there could be plausible implementations of `Semigroup` for some types, but not valid `Monoid` implementations since the `zero` might not make any sense in some contexts. Similarly, if your method can get away with an applicative, don't require a monad as evidence.

SUMMARY

Concepts such as `Functor`, `Applicative Functor` and `Monad` are very useful to provide a common interface to very different and, sometimes, unrelated context. Pure algebraic structures can help you write very elegant and reusable code.

Object-oriented design patterns let you use the same vocabulary with other OO programmers. Now functors, applicatives, monads, and so on let you do the same thing among functional programmers. Some object-oriented design patterns reflect the limitation of a language/paradigm, in terms of expressivity. Just think that many of the classic twenty-three design patterns used in OOP are easily implemented in functional programming through HOFs and type classes.

Finally, take into account that we approached the concepts in this chapter very pragmatically. Of course, in doing so, we had to gloss over some details and use some terminology improperly. From a Category Theory point of view this may not be acceptable, but we hope that this served to help you understand the part of these concepts that you need as a programmer. This chapter was meant as an introduction to these concepts. The field is so big you can easily write an entire book about it. Actually, if you want to deepen your knowledge about these and other advanced functional programming patterns you should definitely read the *red book* that is Functional Programming in Scala—Paul Chiusano and Rúnar Bjarnason.

Oh, and remember:

- Never be scared by the buzzwords. Sometimes, the concepts behind them are not as hard as some people love to make you believe.
- Always *follow the types*! They'll never disappoint you.

11

Concurrency

WHAT'S IN THIS CHAPTER?

- Understanding why concurrent and parallel programming is hard and why it needs a special attention
- Learning about asynchronous structures and the best practices for dealing with concurrency
- Getting to know the ways to benchmark your solution to see whether or not the concurrent version is faster
- Having a way of creating and understanding distributed applications with Akka framework

There is a great chance that you've heard about the old Moore's law, which states that processing power will double every two years. It was handy for quite some time: you could develop your applications and if the actual performance of the software was poor, you could just wait for a few months so that everyone would have a better computer to handle the appetites of your application. At that time concurrency was only used to handle multitasking, as in a web browser: when you click on a download link, it should be going in parallel, without blocking out the main window.

Sadly, the “free lunch” is over and today to increase the fluidity of your software it is necessary to know how to handle more than one CPU core at the same time. This task is not easy, and the concurrency problem is considered as one of the most difficult in today developer's craft.

But wait, there is more! The long-time reign of mainframes is nearly nonexistent today, and now we are dealing with many small servers instead of a lonely large one. Distributed systems are no longer a scientific subject. Today they are used to solve complex problems by many of the world's top 500 technological companies. All of this is due to the fact that building one great computer (called “vertical scaling”) is much, much more expensive than connecting a cluster of a bunch of cheap ones (also called “horizontal scaling”). That's why developers who

understand and know how to handle distributed (or, at least, multi core) programming are very valued today.

Java was created in 1995 when there was no widespread multi core or distributed applications. Even if it is true that Java has evolved during the last years and nearly everything is available as a linkable library, Scala has some of these tools directly out of the box. Things like monadic Future, Parallel collections, Akka, and Spark were specifically made to be as readable and expressive as possible when using Scala.

So why bother learning concurrency? As an example, consider that you need to query a distant database. Why not just wait for a reply from the server and then proceed with computations? Take a look at this schema in Figure 11-1.



FIGURE 11-1

Here you have two processes concurrently fighting for CPU's time. It's normal, on a low level, so the scheduler handles those things by giving each one of the processes some time according to their priority. Now let's see what happens when a process makes a call to a remote MongoDB database (Figure 11-2).

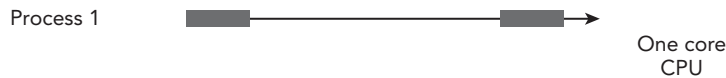


FIGURE 11-2

The second process took some of its time to make a request, but after that it did nothing but wait for the reply. If the MongoDB is located on a separate server, it can mean that it is blocking the thread for a quite some time as the request latency is an order of magnitude larger than the time to prepare the request.

Now here is what it looks like when using a non-blocking, reactive MongoDB driver (Figure 11-3).



FIGURE 11-3

Quite an improvement, don't you think so? You are using all of the power of the processor without blocking it. Now imagine for a second that you have thousands of threads that are doing asynchronous requests on many cores. This is where the reactive (or asynchronous) programming may be something that separates a system with non-viable performances on an expensive server from the one that will handle nearly any number of requests on a few cheap machines.

In this chapter we discuss the main ways to handle concurrency in Scala. We start with a small reminder of how this problem was solved originally, and how the concurrency model looks on a low level. Then we gradually proceed to more advanced techniques available in which the concurrency is handled behind powerful abstractions.

Each of the parts in this chapter may take a whole book to be described in detail, so instead we will analyze where to use the described tool and where it should be avoided at all costs. With this in mind, to have the best results from this chapter, don't be afraid to experiment and launch the examples yourself, since it's one of the best ways to feel the unpredictability and the power of the concurrent programming.

SYNCHRONIZE/ATOMIC VARIABLES

Java 1.5 introduced the concurrency model that became the building bricks of today's concurrent abstractions in JVM languages. Even if it is rarely used in Scala (as there are better alternatives), you should still understand how it works under the hood. As you may already know, every time you need to execute a concurrent process, you need to create a thread:

```
class CustomThread extends Thread {
  override def run(): Unit = {
    println("Custom thread is running.")
  }
}
val thread = new CustomThread
thread.start()
thread.join()
println("Custom thread has joined.")
```

In this code you create a custom class that extends `Thread`. The `start()` function effectively executes the `run()` method of the `CustomThread` class on a different system's thread. It is the OS's job to find an available CPU core to run the method. The `join()` method, on the other hand, notifies the main thread (the one on which the program is executed) to wait for the end of thread's execution. As a consequence, the string "Custom thread has joined" will be always shown after "Custom thread is running." It's worth noting that the main thread is not busy waiting while doing the `join()`, so no additional CPU-cycles are consumed.

Now, let's say you have a variable that two threads should modify at the same time:

```
var a = 0
class CustomThread extends Thread {
  override def run(): Unit = {
    a += 1
  }
}
val thread1 = new CustomThread
val thread2 = new CustomThread
thread1.start()
thread2.start()

thread1.join()
thread2.join()
println(a)
```

After executing this code several times, you may notice that the last line does not always show “2”, but sometimes it’s “1” instead. What is happening here is that two threads are modifying the variable at the same time: `thread1` reads the value that is “0” and before it modifies it, `thread2` also reads the value “0” of the variable. So both threads see “0” and they both add one to it to save the final value as “1”. This phenomenon is called “race conditions”: when the result of a concurrent program depends on the order of the execution of its statements.

Luckily there is a way to handle this situation, and it’s called synchronization. Let’s explain it with an analogy.

A while back there was a version control system called RCS. It worked as follows: when you want to modify a file you should have locked it so that nobody other than you can modify it until you’re done. What happens when you want to modify a file that is locked by another person? Well, you have to wait some time, probably doing nothing but waiting. Here is how it looks applied to code with an incrementing variable:

```
var a = 0
val obj = new Object
class CustomThread extends Thread {
  override def run(): Unit = {
    obj.synchronized {
      a += 1
    }
  }
}
val thread1 = new CustomThread
val thread2 = new CustomThread
thread1.start()
thread2.start()

thread1.join()
thread2.join()
println(a)
```

In this case the last line will always print “2” no matter how many times you execute the program. It is not possible to do a `synchronize` on `a` as it is an integer, and that’s why we introduced the new `obj` value so that you could have a lock for the modifications on `a`.

The problem with this model is that you could easily stumble upon a deadlock. Consider this code:

```
var a = 0
val obj1 = new Object
val obj2 = new Object
class CustomThread1 extends Thread {
  override def run(): Unit = {
    obj1.synchronized {
      obj2.synchronized {
        a += 1
      }
    }
  }
}
```

```

class CustomThread2 extends Thread {
  override def run(): Unit = {
    obj2.synchronized {
      obj1.synchronized {
        a += 1
      }
    }
  }
}

(1 to 100).foreach(i => {
  println(s"current iteration: $i")
  val thread1 = new CustomThread1
  val thread2 = new CustomThread2
  thread1.start()
  thread2.start()
  thread1.join()
  thread2.join()
})
println(a)

```

If you execute this code, there is a very high chance that it will just stack on some iteration (just re-launch it a few times if that's not the case for you). You will have to halt the execution. What happens here is that at some moment `thread1` synchronizes on `obj1` when, at the same time, `thread2` synchronizes on `obj2`. Then `thread1` needs a synchronization on `obj2` that is locked and `thread2` needs it on `obj1`. Nobody can move, since we are in deadlock!

This is why the low-level model is better to be avoided. But especially for these cases SUN introduced so called Atomic Variables that, in some cases, may help avoid these problems with concurrent access. As an example, let's rewrite the “counter” example where each thread tries to increment the variable:

```

import java.util.concurrent.atomic.AtomicInteger
var a = new AtomicInteger()
class CustomThread extends Thread {
  override def run(): Unit = {
    a.getAndIncrement()
  }
}

val thread1 = new CustomThread
val thread2 = new CustomThread
thread1.start()
thread2.start()

thread1.join()
thread2.join()
println(a)

```

Now no matter how many times you execute this code, it will always print “2” as the output. Sure, you may no longer use symbolic methods like “+” or “—” when working with `AtomicInteger`, but that's what the low-level concurrent integer primitive looks like. There are other “atomic” variables like `AtomicLong`, `AtomicBoolean`, or even `AtomicRef`, and they all have their specific use cases, but these are outside of the scope of this chapter.

There should be a small word about “volatile” variables, as they are quite common in Java concurrency programming. Take a look at this application:

```
class CustomThread extends Thread {
  var flag = true
  override def run(): Unit = {
    while(flag) { }
    println("Thread terminated")
  }
}
val thread = new CustomThread
thread.start()
Thread.sleep(2000)
thread.flag = false
println("App terminated")
```

Here you have a main application thread that launches another custom thread. In the `run()` method of the custom thread you may notice that it does a `while` loop infinitely, until someone tells it to stop. That is called “busy-waiting” and it also locks the CPU core, so never use `while` in a thread to wait for something!

The problem in this application is that when you execute `thread.flag = false`, the thread won’t see the change and will continue to “busy-wait” for something to change. It will work as expected if, instead, you mark that variable with `@volatile` annotation, like the following:

```
@volatile var flag = true
```

In this case, the custom thread will successfully see the change of the `flag` variable and will terminate the execution.

To conclude, in this section you saw why the low-level concurrent programming is considered hard and very dangerous. In the following sections of the chapter you will see how this problem can be solved with powerful abstractions that Scala brings to the table.

FUTURE COMPOSITION

Nearly all modern languages have some means of an asynchronous call. For example, promises exist in JavaScript, goroutines exist in Go, and fibers exist in Ruby. Scala has its own asynchronous structure called `Future` that is directly included in the standard library. This way other libraries that need to use it don’t need to include other external implementations, as in Java.

`Future` is a container for asynchronous computations. It’s like a magic box that promises you an object, but you must wait for some time until you open it. Consider the following code:

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
Future {
  print("World ")
}
print("Hello ")
```

Here you import the `Future` class that will hold your computations and you also import the global execution context that will be discussed in a few moments. If you execute it, you will print

“Hello World.” While it is not guaranteed that the sequence will be printed in that order and not in the order of “World Hello,” asynchronous computations need a separate thread from a thread pool as well as some time scheduled by the CPU. As this manipulation takes some time, the “Hello” gets printed before. Let’s spice it up a little bit:

```
for(i <- 1 to 30){
  Future{
    print("World ")
  }
  print("Hello ")
}
```

Now, you can’t predict anything about the resulting string except, maybe, that the first word will be “Hello.” This word continues to be printed with every loop, but the “World” word will be printed when the `Future` is executed, which isn’t predictable. For an even more unpredictable output, wrap the second `print()` in a `Future`, but be careful because your `for` loop may end before all of the “futures” are finished, effectively halting your application with all of the uncompleted `Futures`.

Now let’s talk about why we imported the global `ExecutionContext`. Execution contexts are used to handle a threads pool in your application that also depends on a number of cores that your CPU has. You may create your own `ExecutionContext` to handle threads in another manner or to handle just your database threads, but for these examples the default global one will be just fine. Where is it used? Take a look at the definition of the `Future`’s factory method:

```
object Future {
  def apply[T](body: => T)(implicit executor: ExecutionContext): Future[T]
}
```

It needs an implicit executor somewhere in the scope and the import `scala.concurrent.ExecutionContext.Implicits.global` does exactly that. Note that this provides a pool of a finite numbers of threads, so if there are for example, 8 threads available and you create 8 `Futures` that will block all of those threads, subsequent `Futures` will not be executed. This is called “thread starvation.” To avoid it, when you know that you will be creating many blocking `Futures`, wrap the contents into a blocking statement:

```
for (i <- 1 to 30) {
  Future{
    blocking {
      Thread.sleep(10000)
      println("Done")
    }
  }
}
```

Now let’s discuss how to work with a method that returns a `Future`. Imagine you have a DAO layer in your application:

```
object Dao {
  def findUserById(id: Int): Future[User] = Future {
    User("Alex", 26)
  }
}
```

We defined a DAO with a single method returning a user by id. For the sake of simplicity, it will return the same user every time. Normally, this method should have a type `Future[Option[User]]`, and as a user with a specific Id that may not exist, but, again, to avoid complications let's imagine that it will always return a user.

OK, now we have a task to show a user's name for a specific id. Here's how to do it:

```
import scala.util.{Success, Failure}
Dao.findUserId(1).onComplete {
  case Success(user) => println(s"User's name is ${user.name}")
  case Failure(ex) => println("An error has occurred!")
}
```

A callback is attached to the future, and when it's complete, we will immediately execute either the “success” case or the “failure” one (this happens if something throws an exception inside of a `Future`). As you may notice, the `onComplete()` method returns a `Unit` type, which means that the result of the computation in the `Success` case will be left there and nobody will be able to use it. But what if you need to create a “service layer” with a method having following signature:

```
def retrieveUserNameById(id: Int): Future[String]
```

In this case you won't do the fetch in the database, because it's Dao's job. But this means that you need to find a way to manipulate `Future` to transform it from `Future[User]` into `Future[String]`. You already know what's a `Monad` from the previous chapter on advanced types, and `Future` also has all of those useful methods like `foreach()`, `map()`, `fold()`, and others. The transformation from `Future[User]` into `c` can be handled by a `map()` method:

```
object UserService {
  def retrieveUserNameById(id: Int): Future[String] =
    Dao.findUserId(id).map { user =>
      user.name
    }
}
```

Just as with `Collections`, `map()` method transforms the element inside of the `Future` container and returns the wrapper with a new content. Now the top-level function will be able to use the `Service` layer without knowing how the `DAO` is implemented:

```
def sayHello(): Unit = UserService.retrieveUserNameById(1).onComplete {
  case Success(name) => println(s"Hello $name !")
  case Failure(ex) => println("An error has occurred!")
}
```

Or its more functional version:

```
def sayHello(): Unit = UserService.retrieveUserNameById(1).foreach { name =>
  println(s"Hello $name !")
}
```

The `filter()` method returns the `Future` if its element satisfies the predicate. If the element does not satisfy the predicate, this method returns a failed `Future`:

```
Dao.findUserId(1).filter(_.age > 18) // Successful, with the user
Dao.findUserId(1).filter(_.name == "Sam") // Failed Future
```

You may be sometimes tempted to use `Await.result` to wait for the `Future`'s result like this:

```
val userName = Await.result(UserService.retrieveUserNameById(1), 1 second)
```

Don't do it, since this will effectively block the current thread as well as the already blocked thread that is doing the computation, thus losing all of the benefits of the `Future`! The only place it may be appropriate is during your unit tests because you need to make sure the execution order is predictable.

And last but not least, consider using the library called “scala-async” because it may make your code more readable, so instead of:

```
def calculate: Future[Int] = {
  val future1 = Future(1 + 2)
  val future2 = Future(3 + 4)
  for {
    result1 <- future1
    result2 <- future2
  } yield result1 + result2
}
```

You may do an, arguably, more readable version:

```
def calculate: Future[Int] = async {
  val future1 = Future(1 + 2)
  val future2 = Future(3 + 4)

  await(future1) + await(future2)
}
```

But don't forget: these are just macros that are transformed into a version with `Future` upon the compilation. In addition, `scala-async` has some limitations: for example, you may not use a closure inside of an `async{}` statement.

This all just barely scratches the surface of the work with the `Future` structure. You should now have a sufficient knowledge of how to work with functions that compute an asynchronous result, or how to initialize an asynchronous computation yourself.

PARALLEL COLLECTIONS

So far you've learned about the ways to launch an asynchronous task. Now let's talk about data structures that have parallelism inside of their bones. Parallel collections were introduced in Java 8 (for streams), but they already existed in Scala even before that (since version 2.9). Parallel collections may be a great way to improve the performance of your application, but they may also make it worse. You may think “Well, our CPU is multi-core, so let's make all of the collection operations parallel! What can go wrong?” Now consider following code:

```
(1 to 10000000).filter(_ % 2 == 0)
```

Would it improve the performance if you transform it into parallel computation?

```
(1 to 10000000).par.filter(_ % 2 == 0)
```

The small `.par` there does exactly that: it transforms a collection into its “parallel” counterpart. It’s easy to see an improvement if the operation takes several dozens of minutes (in that case, parallel or normal collections may not be the best tool to use, and you may want to consider learning about Spark). But in this case the execution takes somewhere near 250 milliseconds. That’s too fast for humans, so we will need to test it through micro benchmarks.

Usually, to measure how fast an expression executes, you may consider using `System.nanoTime`:

```
val start = System.nanoTime
val result = (1 to 10000000).filter(_ % 2 == 0)
println(((System.nanoTime - start) / 1000000) + " milliseconds")
```

Let’s see the results. Ah, a whole 4 seconds and 293 milliseconds! Quite a long time for such a small application, so there must be an error somewhere here. Let’s execute the microbenchmark again: 2 seconds 759 milliseconds! That’s strange, the code is the same, but the execution time has 30% difference between the two runs. We may try a version with `.par`, so maybe it will be considerably faster:

```
val start = System.nanoTime
val result = (1 to 10000000).par.filter(_ % 2 == 0)
println(((System.nanoTime - start) / 1000000) + " milliseconds")
```

And now it’s 6 seconds and 438 milliseconds. At this stage you may say that the parallel collection is clearly slower and thus, useless in this situation. But instead, let’s explain why there is such a difference between the results for three separate runs.

Scala’s code is not directly compiled into machine code as it’s done for C. Instead, it is compiled into an intermediate state that is called a “byte-code.” That byte code is then interpreted by a JVM machine that is different for each platform, effectively making Scala’s (and also Java’s) code cross-platform. You should know that the JVM is intelligent and sees when you are executing the same code more than once; in that case, it optimizes it so that subsequent executions don’t take so long. This is called “Just In Time” compilation, or simply JIT. To take this optimization into account, use “warmup” cycles:

```
var start = 0L

for (i <- 1 to 10){
  start = System.nanoTime
  (1 to 10000000).par.filter(_ % 2 == 0)
  println(((System.nanoTime - start) / 1000000) + " milliseconds")
}
```

And the execution will output:

```
$ scala chap11.scala
1891 milliseconds
628 milliseconds
1262 milliseconds
393 milliseconds
348 milliseconds
330 milliseconds
```



```
361 milliseconds
324 milliseconds
332 milliseconds
347 milliseconds
```

You may clearly see that the first execution is not representative at all. It also takes some time for the code to get warmed up.

There are numerous other things to keep in mind when doing micro benchmarks: run-specific JVM optimizations that may not be representative at all, Garbage Collector may kick in during the execution, affecting the results, shared state problems, and so on. But it is possible to avoid some of these problems by using a specialized library.

At the moment, the JDK 9 is not officially released, but we already know what new features it will have. Among numerous improvements, there will be a micro benchmark tool called JMH (Java Microbenchmark Harness). You may find a description with quite a few examples for Java here: <http://openjdk.java.net/projects/code-tools/jmh/>. But we obviously want to use it with Scala, and there is a handy version of JMH adapted for this language called `sbt-jmh`: <https://github.com/ktoso/sbt-jmh>. Consider learning the basics of this tool so that you won't do optimizations without some real performance numbers.

With this in mind, let's see if the parallel version of the `filter()` method is better (Figure 11-4). Both versions were executed in JMH with 20 warm up iterations, 20 measurement iterations, and 20 JVM forks (so that there is no run-specific JVM optimization), and here are my results on an i7-4700HQ quad core processor with JVM version 1.8.0_66.

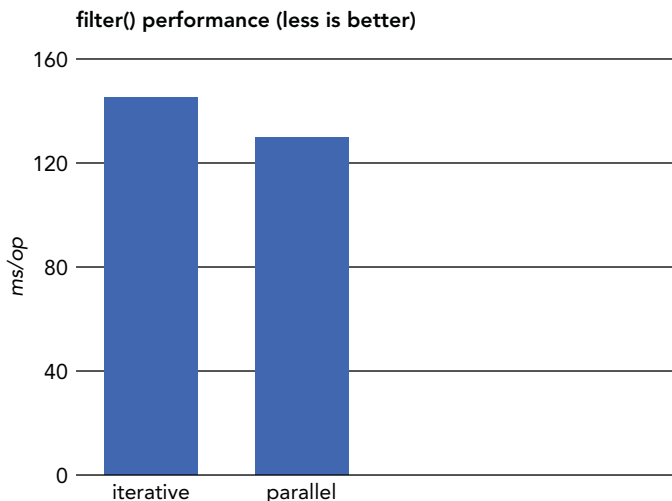


FIGURE 11-4

So now you know that the parallel version is faster, but not by much, and it's up to you to decide if the optimization is worthwhile. It is faster because the `filter()` operation is parallelizable: the collection is split into several chunks, and each chunk is filtered separately so the results are

concatenated together. Now, imagine you have a list of elements (integers in our case) and you need to apply a parallelizable operation to it (find a maximum):

```
val list = Random.shuffle(Vector.tabulate(5000000)(i => i)).toList
val max = list.max
```

This will produce a list with elements from 1 to 5000000 in a random order to find the maximum value. You might think that this is great, so why not throw a `.par` into it? This way you can use the power of the multi core CPU to suit your needs:

```
val max = list.par.max
```

With happy thoughts about gained optimizations, let's launch the same micro benchmark used for the `filter()` operation (Figure 11-5).

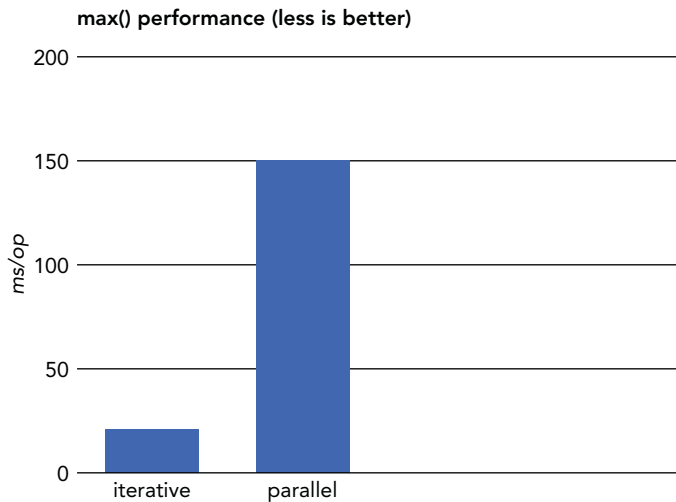


FIGURE 11-5

Now that is very strange: the parallel operation is nearly six times slower than the iterative one! Here is why that is the case: To execute methods in parallel, the collection, as mentioned before, needs to be split into several chunks. If the collection cannot do it in a constant time $O(1)$ (as in the case when we are using a `List`), it will first be converted to such a collection (`Array`, `Vector`, `Range`, etc.). The important thing is that the conversion is not parallelized, so it takes quite some time to convert the list of five million elements. Let's see what performances you have if you use a `Vector` type:

```
val vector = Random.shuffle(Vector.tabulate(5000000)(i => i))
val max = vector.max
val maxPar = vector.par.max
```

The results are shown in Figure 11-6.

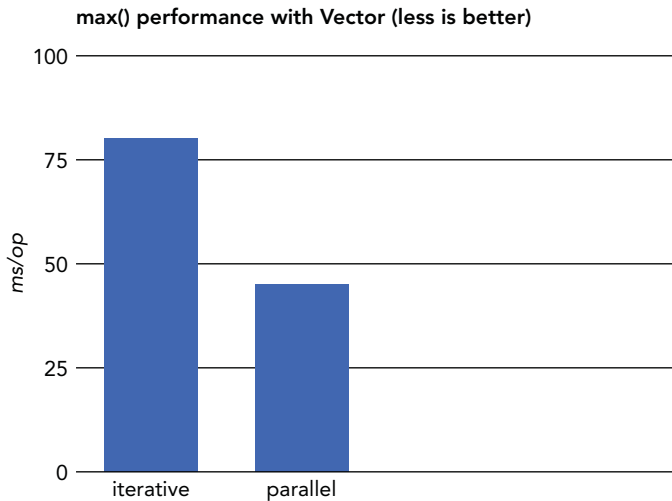


FIGURE 11-6

Notice that the parallel version is almost two times faster than the iterative one, but the version with a non-parallelized list is even two times faster than that. Always micro benchmark your code before doing `.par` optimizations!

Let's talk about another pitfall. As you know, parallel operations are executed more or less at the same time, so consider the following code:

```
var a = 0

(1 to 100).par.foreach(_ => a += 1)

println(a)
```

Can you predict what number this code will show on a multi-core CPU? It may be “100,” but this is highly improbable, so you may bet that the number will be anything but “100” and be sure to win it. This example looks almost the same as the one at the beginning of the chapter, when we talked about `Threads`: two or more different threads are reading the same value of `a` (for example, 41), then they increment it by 1 (for example, 42) and store it into the `a`. In the end, two or more threads applied an increment, but the result is still the same (42). That's why it is very important to keep the functions side-effect free for your parallel methods, (such functions are described in detail in Chapter 2).

As a rule of thumb, don't blindly add `.par` to a collection operation in hopes that it will improve the performances. Instead create a micro benchmark with a tool like “JMH” to measure the potential gains. Also, be aware of the problem with “side effects” when working with parallel collections.

REACTIVE STREAMS

In a chapter about concurrency it would be a shame not to talk about reactive programming. What is it? Imagine a simple operation:

```
var a = 1
var b = a + 1
```

In normal programming languages, `b` will be evaluated to 2, but what if we change the code a little bit by adding another operation:

```
var a = 1
var b = a + 1
a = 2
```

In non-reactive programming, the value of `b` is still 2, no matter how much you change `a`, `b` will remain the same. `b` does not depend on `a`. But in reactive programming, the value of `b` does depend on the value of `a` and reacts to each one of its changes. It's more or less like Excel's tables where changing a value in one cell may modify everything else.

One of the most mature libraries for reactive programming is RxJava, it is maintained by Netflix, and often considered a de facto standard when dealing with implementations for reactive extensions. One of the alternatives is Akka-streams, but it is still in an experimental state at the time of writing. We will use RxJava's Scala-adaption, RxScala (<https://github.com/ReactiveX/RxScala>). To include the library in your project, add this line to the SBT file:

```
libraryDependencies += "io.reactivex" %% "rxscala" % "0.25.1"
```

Here is an example of a trivial application:

```
println("Start")
val observable = Observable.from(1 to 100)
observable.subscribe(println(_))
println("End")
```

Here we may witness a simple implementation of a “Publisher/Subscriber” pattern. The code is simply printing numbers from 1 to 100. In this case the “publisher” is the `Observable` and the “subscriber” is the `println()` function. The interesting thing is that this output will be printed between the words “Start” and “End.” Not quite asynchronous as we would expect, this is the case because the `Observable` created with `from()` is synchronous by default. To make it execute on another thread, you should provide a scheduler:

```
println("Start")
val observable = Observable.from(1 to 100).observeOn(IOScheduler())
observable.subscribe(println(_))
println("End")
```

And now the “End” word should be printed long before the number “100.”

By now you may say that observables look like a way to process a collection of elements asynchronously. But is it better? As we learned from the previous part about parallel collections, we should never trust our feelings when dealing with parallel computations; instead we should be wise and

create a micro-benchmark that tests our assumptions. As a test subject we will take a `filter()` and `sum()` operations:

```
val observable = Observable.from(1 to 1000000)
observable.filter(_ % 2 == 0).sum.toBlocking.first
```

Here we transform a Range of one to a million into an `Observable`, then we filter the even numbers from it, and finally, we are calculating the sum of them. `toBlocking` and `first` are needed here to get the result of the computation without creating a separate subscription on the `Observable`.

In the other corner you have a standard `List` operation that you already saw for uncountable amounts of time:

```
val list = (1 to 1000000).toList
list.filter(_ % 2 == 0).sum
```

It is the same operation as with `Observable`, but with a simpler, more readable language. We will now benchmark those two blocks of code with JMH, using 20 warmup iterations, 20 measurement iterations, and 20 JVM forks to get the most representative results as possible. Here they are, as shown in Figure 11-7.

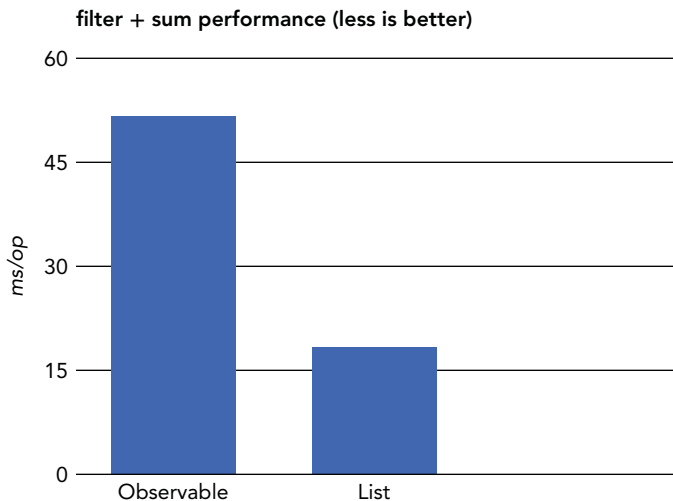


FIGURE 11-7

What a result! `Observable` is nearly 2.5 times slower than a `List`! So, with performances like that, why should anyone consider using RxScala? In fact, there are a few things that are possible with observables but are impossible with collections or streams. For instance, with `Observable` you may do the following:

```
import scala.concurrent.duration._
println(s"(Thread: ${Thread.currentThread().getId}) Start")
val observable = Observable.interval(1 second).observeOn(IOScheduler()).take(5)
```

```
observable.subscribe(x => println(s"(Thread: ${Thread.currentThread().getId}) $x"))

Thread.sleep(6000)
println(s"(Thread: ${Thread.currentThread().getId}) End")
```

In this code the created `Observable` will emit a number every second so that a subscriber can print it. Here is what the output will look like (thread's id may be different for you):

```
(Thread: 70) Start
(Thread: 74) 0
(Thread: 74) 1
(Thread: 74) 2
(Thread: 74) 3
(Thread: 74) 4
(Thread: 70) End
```

As you can see, the subscription operations are done on a separate thread. The `take(5)` is necessary because otherwise the subscriber won't stop printing numbers even after the main thread has terminated. As for the intervals, you are not limited to seconds, and it may be anything from nanoseconds to hours. With RxScala it becomes easy to manipulate events that are timed in a certain manner.

The other area where RxScala shines is in merging streams. Let's say you have several observables coming from different sources, and you need to merge them into one to create an observer, instead of creating separate observers for each of the `Observables`:

```
import scala.concurrent.duration._
println(s"(Thread: ${Thread.currentThread().getId}) Start")
val observable1 = Observable.interval(1 second).observeOn(IOScheduler()).take(3)
val observable2 = Observable.interval(700 millis).observeOn(IOScheduler()).take(4)
val observable3 = Observable.interval(300 millis).observeOn(IOScheduler()).take(6)
val mainObservable = Observable.from(Array(observable1, observable2,
  observable3)).flatten
mainObservable.subscribe(x => println(s"(Thread: ${Thread.currentThread().getId}) $x"))
Thread.sleep(4000)
println(s"(Thread: ${Thread.currentThread().getId}) End")
```

And here is its output:

```
(Thread: 100) Start
(Thread: 108) 0
(Thread: 108) 1
(Thread: 106) 0
(Thread: 108) 2
(Thread: 104) 0
(Thread: 108) 3
(Thread: 106) 1
(Thread: 108) 4
(Thread: 108) 5
(Thread: 104) 1
(Thread: 106) 2
(Thread: 106) 3
(Thread: 104) 2
(Thread: 100) End
```

This is not readable at all! But there is a way to make it more explicit, if you use a “marble diagram” to express what is happening (Figure 11-8).

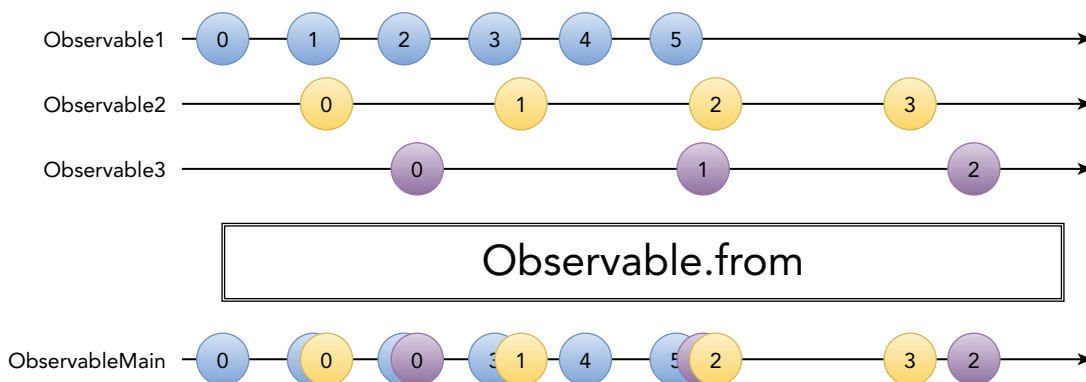


FIGURE 11-8

The small marbles from observables are placed on the “time arrow” according to the time when they were observed. After the merging operation, `Observable.from()`, you are only subscribed to the `ObservableMain` value that is the combination of the 3 separate `Observables`. It’s like combining three lists into one, but the elements are placed according to their execution time. You may make yourself more familiar with this kind of diagram by visiting this site: <http://rxmarbles.com/>. Nearly all of `Observable`’s methods are described in a form of a marble diagram.

There are quite a few things to say about reactive programming with RxScala, but we are here to decide if it is the right tool for the applications or not. If the logic of your application needs timed executions, instead of using `Thread.sleep()`, use this library. Also, when you need to merge a few streams into one, or you may imagine a marble diagram that represents events in your application, consider trying the RxScala library because it may make your application’s domain logic more natural and readable.

STM

It would be strange not to talk about Software Transactional Memory in a chapter about concurrency. This technique is less known than others, but it is worth mentioning because it may solve a particular problem you may have in your application.

Software Transactional Memory has much in common with database transactions. It is an alternative to the synchronization that “commits” a block of code potentially accessed concurrently and rolls back the execution if two threads are trying to change the variables in the “transaction” at the same time. The history of STM started in 1986 when Tom Knight proposed transactions on a hardware level. In 1995 the STM began to live as software-only transactions. Today it is still an area of research, but the practical implementations are widely used in frameworks such as Akka.

In this section of the chapter we will talk about a particular implementation called `ScalaSTM`. This library was created by experts in STM and it is the one that is currently used in the Akka framework. It will soon be introduced in Scala’s standard library. Do you remember the example of

`AtomicInteger` from the beginning of the chapter? All of the operations with a single instance are atomic and, thus, will work well in concurrent environments. But if you introduce another variable, things might get complicated. And it doesn't matter if the other variable is also `AtomicInteger`, the sequence of operations where we use them both is not considered atomic itself without a proper synchronization. Take a look at this code:

```
import java.util.concurrent.atomic.AtomicInteger
var a = new AtomicInteger()
var b = new AtomicInteger()
class CustomThread extends Thread {
  override def run(): Unit = {
    for (_ <- 1 to 30) {
      a.getAndIncrement()
      b.addAndGet(a.get())
    }
  }
}
val thread1 = new CustomThread
val thread2 = new CustomThread
thread1.start()
thread2.start()
thread1.join()
thread2.join()
println(a)
println(b)
```

It will print “60” for a and “1840” for b. Or “1834,” or “1844,” or even “1839.” You can't predict the result, because our computations are not atomic! One of the ways to handle this situation is to use the `synchronize` method that was used earlier in this chapter. But this, as you already saw, could easily lead to a deadlock. Also, programming with `synchronized` is quite difficult without any substantial advantages. A better way to handle the situation would be to use Scala STM. To include Scala STM in your project, add the following line to your dependencies:

```
"org.scala-stm" %% "scala-stm" % "0.7"
```

With that done, you now can modify your code so that it works as expected:

```
import scala.concurrent.stm._
var a = Ref(0)
var b = Ref(0)
class CustomThread extends Thread {
  override def run(): Unit = {
    for (_ <- 1 to 30) {
      atomic{ implicit txn =>
        a() = a() + 1
        b() = b() + a()
      }
    }
  }
}
val thread1 = new CustomThread
val thread2 = new CustomThread
thread1.start()
```



```

thread2.start()
thread1.join()
thread2.join()
println(a.single())
println(b.single())

```

Great! Now in the end, you always get “60” for a and “1830” for b. So, how does it work? Notice that instead of a plain Integer or an atomic one, you have a Ref at the variable definition, which stands for “transactional reference.” A Ref variable has to be used inside a transaction (the atomic statement), as its `apply()` and `update()` methods are requiring an implicit `InTxn` value:

```

override def apply()(implicit txn: InTxn): T = impl.get(handle)
override def update(v: T)(implicit txn: InTxn) { impl.set(handle, v) }

```

The way that ScalaSTM implements atomic transactions is quite complex: it keeps a log of every “write” and “read” inside of the atomic statement and once it gets to the end of the block, it “commits” the result. But it is highly possible that someone else tries to read or write the value at the same time. In this case there is a “transactional conflict,” and when this happens, both transactions are canceled and executed in sequential order (you can’t predict in what exact order). Doesn’t it look the same as database transactions?

At the last line with `println(a.single())` we printed the value of a without using any atomic statement. It is called “single-operation transaction” and it doesn’t need to be inside a transaction.

Be aware that the transaction may be rolled back and re-executed, so any side effects inside of the atomic statement will be re-executed as well. Change the contents of the `CustomThread`’s `run()` method to the following:

```

override def run(): Unit = {
  for (_ <- 1 to 10) {
    atomic{ implicit txn =>
      println(s"The value of a is ${a()}")
      a() = a() + 1
      b() = b() + a()
    }
  }
}

```

Here you have a `println()` statement that is, obviously, a side-effect. In the output of your application you may find lines that are repeated:

```

The value of a is 2
The value of a is 2
The value of a is 3

```

This means that both threads accessed the value at the same time, so the transaction was rolled back and executed in sequential order. In this particular case, the side effect is not critical, but may not always be like that. If you really need to use side effects inside of atomic statements, you may use `Txn.afterCommit` or `Txn.afterRollback` statements:

```

override def run(): Unit = {
  for (_ <- 1 to 10) {

```

```

    atomic { implicit txn =>
      Txn.afterRollback { _ => println(s"Rollback!") }
      a() = a() + 1
      b() = b() + a()
      val aValue = a()
      Txn.afterCommit { _ => println(s"The value of a after commit is $aValue") }
    }
  }
}

```

The output may contain the following lines:

```

The value of a after commit is 10
The value of a after commit is 11
Rollback!
The value of a after commit is 12
Rollback!
The value of a after commit is 13

```

Notice that you didn't directly use `a()` inside of `println()` because you can't use references outside of transactions. Instead, you need to create an intermediate value that will be used for the `println()`.

To conclude, you saw how it is possible to create thread-safe, deadlock-free mutable state between threads. This is a great replacement for the `synchronized` statement, not only because it is safer, but also because it is more readable. The developers of the framework did some benchmarking here: <https://nbronson.github.io/scala-stm/benchmark.html>, which states that even if STM is a tiny bit slower than tricky locks, it is much safer to use.

ACTORS (AKKA)

Most of the Scala developers come to the language because they need to use Akka or Spark in their project and Java is just not good enough for this job. Only a few of us are in the domain because of Scala itself, and not because of its ecosystem. So, let's see why those frameworks are so popular that they make people learn a new language just to use them efficiently.

Akka was first created in 2009 and today it is more popular than Playframework, Typesafe company, or even Scala itself (judging by the number of Akka's stickers taken at a typical programming conference where Typesafe is participating). This framework may be used with Java, but it becomes apparent that Akka was not created for this language, and only has some adapters making it possible to use it by Java-only developers.

Akka is one of the most popular frameworks for Actor programming and, by chance, it's made in Scala. Actor programming is a more natural, human way to approach concurrency problems: imagine you are a manager and you are given a task to calculate Pi number up to a 1,000th decimal. You also have 10 people under your subordination who crave to do some work for you. One of the options could be to give all the work to your favorite co-worker and to wait quite some time until the result is done. Another option is to use a special algorithm that precisely calculates Pi using a `sum` (see Figure 11-9):

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{\pi}{4}$$

FIGURE 11-9

As you may see, the fact that Pi's calculation may be done as a form of a simple sum lets you distribute your work in individual chunks. The "worker#1" will calculate $n=0$ to $n=19$, the "worker#2" will do the calculations from $n=20$ to $n=39$, etc. In the end you just get the result from each of the workers, add them up, and multiply the sum by four! You may even employ a worker who is not in your office, where a result is sent remotely (via TCP/IP or any other protocol).

How do Akka's actors work? Each actor receives messages in its mail box and treats these messages strictly in the order they were received. Not all messages are treated, but only the ones that have a name known to the receiver. Let's start with an elementary application that is using Akka to solve a problem we had at the beginning of this chapter: a concurrent counter. First you need to include the dependency into your build.sbt:

```
"com.typesafe.akka" %% "akka-actor" % "2.4.1"
```

Following that, you will need to add two files, one for the actor:

```
class CountingActor extends Actor with ActorLogging {
  var counter = 0
  def receive = {
    case "+1" =>
      counter += 1
      log.info(s"Current count is $counter")
      if (counter == 3) context.system.terminate()
  }
}
```

and one for the Application that will manipulate the Actor:

```
object ApplicationMain extends App {
  val system = ActorSystem("MyActorSystem")
  val countingActor = system.actorOf(Props[CountingActor], "pingActor")
  countingActor ! "+1"
  countingActor ! "+1"
  countingActor ! "+1"
  Await.result(system.whenTerminated, 10 seconds)
}
```

Pretty simple, so let's start with the Actor's file. The CountingActor has a mutable field counter and an overridden receive method. The method is of the type PartialFunction[Any, Unit], and it means that you may send any type of message to the actor: String, Integer, a Case Class, or anything else, and it will still receive it and try to react to it. In this case, the only message that it may react to is a string "+1"; after receiving it, the actor will increment the counter by one, log the result, and check if the counter is already equal to 3 to shut down the Actor platform. Notice that the actor won't react to a "+2" message as it, obviously, does not understand it.

In the application class a new actor system can be created, which initializes an instance of `CountingActor` and sends a few messages. You can terminate the actor system either when the actor increments the counter three times, or after 10 seconds. The `countingActor` is not actually an instance, but a reference to the actor because it may be located on a remote machine, just as an IP address is to an online server.

In this example, to keep it simple, we send a message to the actor from the main application (! means “send” coming from the Actor system in the Erlang language). In real life it’s mostly actors that communicate with other actors. What’s important here is that all communications are asynchronous: actor doesn’t wait for a reply to his message. If the other actor wants to send a reply, it will be a normal message with a reply data. The whole system is created to be scalable, so you can initialize thousands of actors that are located on the same machine or distributed over a cluster of servers. It’s amazing!

When working with actors, there are a few things to keep in mind, so take a look at the following schema (Figure 11-10).

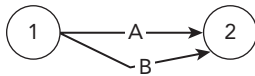


FIGURE 11-10

In this case, if the message “A” was sent before the message “B” it is guaranteed that the actor “2” will receive the messages in that exact order. Now here is another schema shown in Figure 11-11.

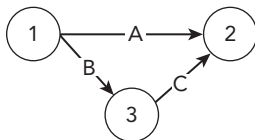


FIGURE 11-11

In this case, even if the message “A” was sent before the message “B,” we don’t have any guarantee that the message “A” will arrive before the message “C,” and it may also be the other way around. This way the system becomes nondeterministic and you may need to add some code to make it more predictable.

The Akka framework is a great way to abstract mutable state inside an isolated entity called Actor. Everything that happens there is sequential and the information is shared through messages, asynchronously. If your application may be imagined as a group of separate workers, Akka may be a great way to improve performance as well as add a possibility of an easy horizontal scaling.

SPARK

It’s worth mentioning Spark, which is a wonderful framework that is very helpful for those who are working with Big Data assets. As you may know, terabytes of data can’t be treated with a simple, well known “for loop,” since you need either a very powerful mainframe (vertical scaling) or a

bunch of low-cost servers connected together (horizontal scaling), as well as some specific tools to work with them. The former approach is, often, too expensive to be used in the Big Data context, which is why the instruments for horizontal scaling are so popular today.

Spark was created as an alternative to Hadoop's map-reduce. From the high level standpoint, this framework looks just like collection of methods that are executed on a distributed environment. If applied properly, the projects using this framework are much faster and use less code than the projects that were built with plain Hadoop. For the first example you will need the library to be included in the `build.sbt` file:

```
"org.apache.spark" %% "spark-core" % "1.6.0"
```

and this code:

```
val conf = new SparkConf().setAppName("Simple Application").setMaster("local[4]")
val sc = new SparkContext(conf)
val readme: RDD[String] = sc.textFile("README.md")
val numAs = readme.filter(line => line.contains("a")).count()
val numBs = readme.filter(line => line.contains("b")).count()
println("Lines with a: %s, Lines with b: %s".format(numAs, numBs))
```

First, you need to initialize your Spark instance with the provided configuration. `"local[4]"` means that it will be executed locally with four parallel threads. The `SparkContext` is ready and you need to provide it with something to compute. For this example you will read the `readme` file from the root of the project and feed it to some filters. But before that, let's talk a little bit about the `RDD[String]` type.

The "RDD" it is translated as "resilient distributed dataset" and you work with it just as with a simple collection of lines from the file. The distribution is transparent to you just as parallelism is in Parallel collections. If you apply some operation to it like, for instance, the `filter()`, it will still return an RDD. The important thing here is that the result of the operation is not computed until you need to return something that isn't an RDD. Such operations are called "actions," and by contrast, the operations that return RDD are called "transformations."

As you know, in the case of Scala's collections even simple "transformation" operations like `filter()` or `map()` will need a full traversal. If the collection contains a terabyte worth of elements, it would be a disaster to do it each time you add a new operation. Instead, Spark postpones all of the transformations until an "action" operation is applied, such as `count()`, `collect`, or `reduce()`.

By default, each time you apply an "action" operation, the RDD is re-computed (in this case it is re-read from the file). You can use the value two times, so you need some way to speed it up. One of the ways to do it is to cache it and keep the data mounted in memory. Don't worry, if your operational memory is not sufficient, Spark will handle it by storing the remaining data inside of a storage file:

```
val readme: RDD[String] = sc.textFile("README.md").cache()
```

This way the data will be kept on the cluster (in this case, in the local memory) for faster access. Note that the type is left the same, but the operations are quicker! But remember, only use it if the RDD is manipulated more than once, because it is better for memory management and time execution.

This is just a simple example of how Spark works. The data file may be located on an HDFS filesystem, in a database, as a CSV file, or even as a JSON extract from your MongoDB. In addition to the problems we talked about in the section in this chapter about Parallel Collections, now you need to worry about data locality, so that transformations don't need to transfer a lot of data between the servers in the cluster.

Another way of doing distributed data analysis with Spark is the Spark SQL package. If your data can be represented as a table (it was read from an actual database table, CSV file, JSON file, etc.), you may query it as you do in relational databases: the SQL statement will be automatically translated as a sequence of Spark's operations. For example, analyze the following .csv file:

```
name,age
Alex,26
Sam,24
Bob,15
```

It will look like the following:

```
val conf = new SparkConf().setAppName("Simple Application").setMaster("local[4]")
val sc = new SparkContext(conf)
val sqlContext = new SQLContext(sc)
val df = sqlContext.read
    .format("com.databricks.spark.csv")
    .option("header", "true") // Use first line of all files as header
    .option("inferSchema", "true") // Automatically infer data types
    .load("src/main/resources/users.csv")
df.show() // will show current DataFrame as a table
df.registerTempTable("users")
val adults = sqlContext.sql("SELECT name FROM users WHERE age > 18")
adults.show()
```

The first 8 lines are just Spark context initialization and file reading. The `show()` method applied to a `DataFrame` prints its content in a nice table-like format. `registerTempTable()` makes it possible to apply SQL queries to your dataset, so you can select adults within our users.

This was a fraction of Spark's capabilities, but it should give you the idea if it is the tool you need for your application. It is easy to use, fast, reliable, and it has a wonderful ecosystem that, in addition to traditional batches, let's you work with streams of data, graphs, or even machine learning algorithms.

SUMMARY

Concurrent, parallel, or even distributed programming is considered to be one of the most complex topics in today's programming. In this chapter you saw what tools Scala has up its sleeve to deal with it and in what situations they can be applied. Even though we only touched the surface on all of these topics, you should be able to choose the right tools to work with your concurrent problems.

We started from the basics, where the roots of concurrent programming in Java introduced in version 1.5 were discussed. You saw why it is considered complicated and why it is better to use more high-level abstractions instead of pure threads and synchronization. You also learned that if

your needs are limited to a single variable, then there is no need to use `synchronize`, and you should use Atomic variables instead!

Following that we covered how you can work with an asynchronous monad called `Future` and how it is convenient to work with a method that is returning it. You also saw parallel collections available in Java 8 and how it is hard to predict if adding `.par` can improve performance. Always benchmarking the optimization before adding it!

We covered more exotic libraries like `RxScala` and `ScalaSTM`, both of which are useful in some specific situations.

Last but not least, we covered the basics of developing with `Akka` and `Spark` frameworks. `Akka` can improve your application with asynchronous messaging and easier horizontal scaling if only your domain logic may be imagined as a work of a group of actors, communicating with messages. And `Spark` is a tool of choice when dealing with large-scale, distributed data.

All in all, this chapter has provided you with the fundamentals of how to choose the right tool for the right problem. If you are not sure whether the concurrent optimization helps, pass it through a JMH micro benchmark so you can be certain that there are some actual benefits.

12

Scala.js

WHAT'S IN THIS CHAPTER?

- Understanding Scala.js
- Using Webjars
- Interopting with the current JavaScript ecosystem

Scala.js is a relatively new feature in Scala that allows you to compile Scala code straight to JavaScript, providing you with the full power of the Scala language, while being able to target JavaScript. It also allows you to share code between the server and the client, which can dramatically reduce boilerplate, especially if you use one of the many pickling/serialization libraries available in Scala.

This chapter also covers Webjars, which are Maven packages with installed resources. Using Webjars, plus the powerful interoperability features that Scala.js provides, provides you with a compelling and fresh way to tackle the highly complex area of web programming.

SCALA.JS AND ITS DESIGN

Scala.js, foremost, is a Scala compiler. That is, it takes `.scala` files, and compiles them to an intermediate bytecode specifically designed for JavaScript (`.sjsir`) files. When combined with `.sbt`, you can then produce an actual `.js` file.

This means that Scala.js does not work with Java source code, nor does it work with JVM bytecode. This means that when using Scala.js, you must use pure Scala code and any dependencies that you may have also have to be written in pure Scala. For this reason, you may have difficulties in using Scala.js in a project that heavily uses Java (either directly or indirectly). Strong integration with Scala.js and SBT means that it's quite easy to cross compile Scala projects. Although, you may think that this limitation is quite severe, it also means that Scala.js provides many benefits that are critical in providing practical success in modern web programming.

One of these advantages is that Scala.js uses a very accurate DCE, which is a requirement for minimizing assets for the client. The reason for this accuracy is that since it works on Scala source code, Scala.js is able to determine which methods are being used and which methods aren't, with a much higher dividend compared to just dealing with pure bytecode. The usage of DCEs is nothing new for web programming (or programming in general); however how much code can be eliminated depends on how strongly statically typed the language is and since idiomatic Scala is very strongly typed, this is a very big benefit. On top of this, Scala.js also uses the Google Closure Compiler for production output. This combination means that it's possible to reduce very large codebases (with a lot of dependencies) to sizes that are smaller than what is possible in JavaScript.

The above however also means that Scala.js isn't completely suited to lazy modules (i.e. modules that are loaded lazily for a specific section of the site). Due to how the Scala DCE works, one would have to include the Scala library (such as the collections library) in each lazy module you would define. Scala.js is much more suited to creating a single .js file, which encompasses your project, rather than many separate .js files.

The usage of Scala source files also means that Scala.js combined with SBT also provides a module system, which is exactly the same module system that is used for using Scala normally with the JVM. This means that Scala.js provides an established solution to a problem that plagues the JavaScript fragmentation in module systems, due to there not being an established standard (AMD/Common.js/npm/bower/require.js etc). Libraries created in Scala.js are packaged in JAR files, and they use the same directory layout as the standard JVM jars. You don't need to worry about how to load files with their dependency trees (a problem that Require.js resolves), because this is all handled by SBT.

Arguably the strongest strength of Scala.js, which is a notable difference from many other language -> JavaScript compilers (particularly static languages), is that Scala.js has an easy-to-use FFI for libraries written in JavaScript. This means that you don't have to reimplement a huge amount of the JavaScript ecosystem in Scala. There are Scala.js bindings for many popular web frameworks, such as React and Angular. This is possible due to `scala.dynamic` (which allows you to dynamically define classes and methods) along with Scala facades (this allows you to enforce types on current JavaScript libraries).

This combination allows you to provide a Scala experience while still being able to interoperate with the current JavaScript ecosystem, as well as offering benefits that aren't available in other languages.

GETTING STARTED: SCALA.JS WITH SBT

One of the easiest ways to learn how Scala.js works is to see how a cross build is set up from scratch. Scala.js hosts a bare repo on Github at <https://github.com/scala-js/scala.js-cross-compile-example>, which shows the basic structure of a Scala.js application, so let's get started by cloning it:

```
git clone https://github.com/scala-js/scala.js-cross-compile-example
```

Before starting, it's important to note that Scala.js is implemented as an SBT plugin, which can be seen if you open `project/plugins.sbt`:

```
addSbtPlugin("org.scala-js" % "sbt-scalajs" % "0.6.3")
```

The first thing you see is that the structure is slightly different compared to a typical project. In this case, you have a root project and two subprojects (one for JVM and one for Scala.js). Code that is within the JVM directory is only compiled for the JVM, and code that is contained within the `js` directory is only compiled for JavaScript, and code in `shared` is compiled for all targets. If you examine the `build.sbt`, you can see how this is set up.

```
name := "Foo root project"

lazy val root = project.in(file(".")).
  aggregate(fooJS, fooJVM).
  settings(
    publish := {},
    publishLocal := {}
  )

lazy val foo = crossProject.in(file(".")).
  settings(
    name := "foo",
    version := "0.1-SNAPSHOT",
    scalaVersion := "2.11.6"
  ).
  jvmSettings(
    // Add JVM-specific settings here
  ).
  jsSettings(
    // Add JS-specific settings here
  )

lazy val fooJVM = foo.jvm
lazy val fooJS = foo.js
```

This template makes use of the `sbt aggregate` feature (<http://www.scala-sbt.org/0.13/docs/Multi-Project.html>). The interesting thing to note here is that you can override `publish` and `publishLocal`. This is because publishing the “root” project doesn’t make sense. When you publish, you want to publish only the subprojects in root (which is the JVM and JavaScript build). Beneath this, you see `jsSettings` and `jvmSetting`, which respectively allow you to specify settings for only JavaScript, or for only the JVM. JVM settings such as “-target:jvm-1.8” when used with `scalacOptions` are pointless when compiling for JavaScript, whereas options such as “-Ywarn-dead-code” can be placed in shared settings.

The most obvious usecase for splitting out the settings is for dependencies. In Scala.js, dependencies for JavaScript are packaged separately from the standard JVM Maven packages (this is also due to the fact that Scala.js has its own notion of binary compatibility, which is separate from JVM binary compatibility). The easiest way to see this is to run `publishLocal` on this current project.

```
[info] Done packaging.
[info]   published foo_2.11 to /Users/matthewdedetrich/.ivy2/local/foo/
      foo_2.11/0.1-SNAPSHOT/poms/foo_2.11.pom
[info]   published foo_2.11 to /Users/matthewdedetrich/.ivy2/local/foo/
      foo_2.11/0.1-SNAPSHOT/jars/foo_2.11.jar
[info]   published foo_2.11 to /Users/matthewdedetrich/.ivy2/local/foo/
      foo_2.11/0.1-SNAPSHOT/srcs/foo_2.11-sources.jar
```

```
[info]      published foo_2.11 to /Users/matthewdedetrich/.ivy2/local/foo/
foo_2.11/0.1-SNAPSHOT/docs/foo_2.11-javadoc.jar
[info]      published ivy to /Users/matthewdedetrich/.ivy2/local/foo/
foo_2.11/0.1-SNAPSHOT/ivys/ivy.xml
[info]      published foo_sjs0.6_2.11 to /Users/matthewdedetrich/.ivy2/local/foo/
foo_sjs0.6_2.11/0.1-SNAPSHOT/poms/foo_sjs0.6_2.11.pom
[info]      published foo_sjs0.6_2.11 to /Users/matthewdedetrich/.ivy2/local/foo/
foo_sjs0.6_2.11/0.1-SNAPSHOT/jars/foo_sjs0.6_2.11.jar
[info]      published foo_sjs0.6_2.11 to /Users/matthewdedetrich/.ivy2/local/
foo/foo_sjs0.6_2.11/0.1-SNAPSHOT/srcs/foo_sjs0.6_2.11-sources.jar
[info]      published foo_sjs0.6_2.11 to /Users/matthewdedetrich/.ivy2/local/
foo/foo_sjs0.6_2.11/0.1-SNAPSHOT/docs/foo_sjs0.6_2.11-javadoc.jar
[info]      published ivy to /Users/matthewdedetrich/.ivy2/local/foo/
foo_sjs0.6_2.11/0.1-SNAPSHOT/ivys/ivy.xml
```

As you can see, the main JVM package is specified as `/foo/foo_2.11/0.1-SNAPSHOT/`, and then for the JavaScript you have `foo/foo_sjs0.6_2.11/0.1-SNAPSHOT`. The `foo_sjs0.6_2.11` means that this package is binary compatible with Scala.js version 0.6.x.

For the example here, let's add Scalatags as a dependency. Scalatags is a library for HTML templating; however the main interest is that it is cross-compiled for JVM and Scala.js. The documentation states that the JVM dependency is `"com.lihaoyi" %% "scalatags" % "0.5.4"` whereas the Scala.js dependency is `"com.lihaoyi" %%% "scalatags" % "0.5.4"`. Let's update `build.sbt` to reflect this (as well as add some common `scalacOptions` as mentioned previously):

```
lazy val foo = crossProject.in(file(".")).
  settings(
    name := "foo",
    version := "0.1-SNAPSHOT",
    scalaVersion := "2.11.6",
    scalacOptions ++= Seq(
      "-encoding", "UTF-8",
      "-deprecation", // warning and location for usages of deprecated APIs
      "-feature", // warning and location for usages of features that should
        be imported explicitly
      "-unchecked", // additional warnings where generated code depends
        on assumptions
      "-Xlint", // recommended additional warnings
      "-Xcheckinit", // runtime error when a val is not initialized due to
        trait hierarchies (instead of NPE somewhere else)
      "-Ywarn-adapted-args", // Warn if an argument list is modified to match
        the receiver
      "-Ywarn-value-discard", // Warn when non-Unit expression results are
        unused
      "-Ywarn-inaccessible",
      "-Ywarn-dead-code"
    )
  ).
  jvmSettings(
    libraryDependencies += "com.lihaoyi" %% "scalatags" % "0.5.4",
    scalacOptions ++= Seq(
      "-target:jvm-1.8"
    )
  ).
```

```
jsSettings(
  libraryDependencies += "com.lihaoyi" %% "scalatags" % "0.5.4"
)

lazy val fooJVM = foo.jvm
lazy val fooJS = foo.js
```

In the preceding example, Scalatags is added as a dependency for both JVM and Scala.js, and specific settings for just compiling on JVM (in our case, we are setting the target to be JDK 1.8) have also been added. This is easy to do because Scalatags is a cross compiled project. If you open up `shared/src/main/scala/MyLibrary.scala`, you will see a trivial implementation.

```
class MyLibrary {
  def sq(x: Int): Int = x * x
}
```

Let's extend this, and generate some HTML using ScalaTags:

```
import scalatags.Text.all._

object MyLibrary {
  def template: String = (
    html(
      div(
        p("This is my template")
      )
    )
  ).render
}
```

Since this is contained in `shared`, it will be compiled for both targets. You also have to make this visible to both targets. Let's modify `js/src/main/scala/Main.scala` to look like this:

```
import scala.scalajs.js

object Main extends js.JSApp {
  def main(): Unit = {
    val lib = new MyLibrary
    println(lib.sq(2))
    println(MyLibrary.template)
  }
}
```

And let's modify `jvm/src/main/scala/Main.scala` to look like this:

```
object Main extends App {
  def main(): Unit = {
    val lib = new MyLibrary
    println(lib.sq(2))
    println(MyLibrary.template)
  }
}
```

What you have just done is modify the entry points for the project. Just as Scala has `App` to specify how the application is run when you execute the `.jar`, `Scala.js` has a `js.JsApp` to specify the entry point for the JavaScript application.

In SBT, if you `fooJS/run` you will see the output for `js/src/main/scala/Main.scala`, whereas running `fooJVM/run` will provide the output for `jvm/src/main/scala/Main.scala`. Extending `js.JsApp` also dictates the entry point for JavaScript, and hence how the `.js` file is actually created. Running `fastOptJS` will create the `.js` file in `js/target/scala-2.11/foo-fastopt.js`, which you can directly include in the header in any HTML. `fullOptJS` runs a full optimization that creates a file in `js/target/scala-2.11/foo-opt.js`. This takes much longer than `fastOptJS`; however it also generates a much smaller file. It's recommended to use `fastOptJS` when developing locally, since file size is not a concern there.

At this point, you now know the basics of how `Scala.js` works. We have set up an example that uses cross compilation technique (using SBT's `aggregate`), and we have also learned how to separate and share SBT settings (including dependencies). In addition we have shown how to run the program (both in JVM and JavaScript), as well as how to generate the `.js` file, which is what is included in the site.

SCALA.JS PECULIARITIES

`Scala.js` does a fantastic job in compiling most Scala code; however, there are corner cases that must be investigated. These corner cases exist because JavaScript isn't a very low level VM, and hence there are restrictions on how certain code can be produced if you also want to maintain reasonable performance.

One very good example is how numbers are treated. In the JavaScript specification, there is only one number type. However in the JVM (and hence Scala), there are numerous number types, such as `Long`/`Int`/`Float`/`Double`. In order to avoid boxing and its associated performance penalties, `Scala.js` has had to make compromises in this area. With the previous examples of numbers, `Scala.js` will map most of the Scala primitive number types to the same JavaScript number type, which means precision (and hence certain math operations) may be defined differently on separate platforms.

`Scala.js` also handles opaque types. These are types that have no actual representation in JavaScript (the most common example of this is `Char`). Opaque types are essentially types that can only be exported to JavaScript. They can't be directly used unless you explicitly code conversions and there also isn't any way to manipulate the type in JavaScript. More information on this can be found here <http://www.scala-js.org/doc/interoperability/types.html>.

Another area in which a difference can be found is exceptions. The JavaScript platform doesn't provide native support for all of the types of exceptions that can be found on the JVM, so implementing them manually in JavaScript is very expensive from a performance perspective. `Scala.js` provides three ways of dealing with this issue: `Compliant`, `Unchecked` and `Fatal`. `Compliant` provides a full JVM specification; however it is very slow. `Unchecked` means that exceptions will be thrown, so the runtime behavior is completely undefined. `Fatal` will throw an exception; however it will be an `UndefinedBehaviourError`, instead of the original exception. More details about dealing with exceptions can be found at <http://www.scala-js.org/doc/semantics.html>. The link also describes instances where pattern matching is different among other semantic differences.

WEBJARS AND DEALING WITH THE FRONTEND ECOSYSTEM

As mentioned previously, the current web ecosystem is heavily fragmented, especially when it comes to dealing with JavaScript modules, and their representation in files. It isn't unusual for web frameworks to create their own asset management systems. As an example, Ruby on Rails has already gone through two different ways of managing assets. There are also competing ways to load files (manually, using tools like `require.js/webpack`).

This problem is further complicated when having to deal with other JavaScript style languages such as CoffeeScript, TypeScript or newer versions of ECMAScript. The combination of all of these has also created quite complex tooling/build tools (Grunt.js, yeoman, NPM)

In Java/Scala with Webjars, these issues are practically nonexistent. Maven/Ivy already provides all of the tooling and dependency management necessary for working with build/module management.

Just as with Scala.js, Webjars are just Maven packages. More specifically, they are Maven packages with resources that are installed with specific paths (so that they can be discovered by various methods). Due to this simplicity, Webjars are very easy to work with. They just contain the compiled assets that are needed, and any dependencies are treated just as normal Maven dependencies. Furthermore, since Maven relies on immutable artifacts as part of its core design, it avoids a lot of issues that happen when using repositories as packages (which is common in frontend development).

The most powerful feature of Webjars, however, is the fact that they can be automatically generated from Bower packages (as well as being manually created, or what is referred to as classic Webjars). Webjars has a website that will convert, on demand, valid Bower packages to a Maven style package, with all of its dependencies defined and deploy it onto Bintray. The website (see Figure 12-1) for converting bower packages to Webjars can be found at <http://www.webjars.org/bower>.

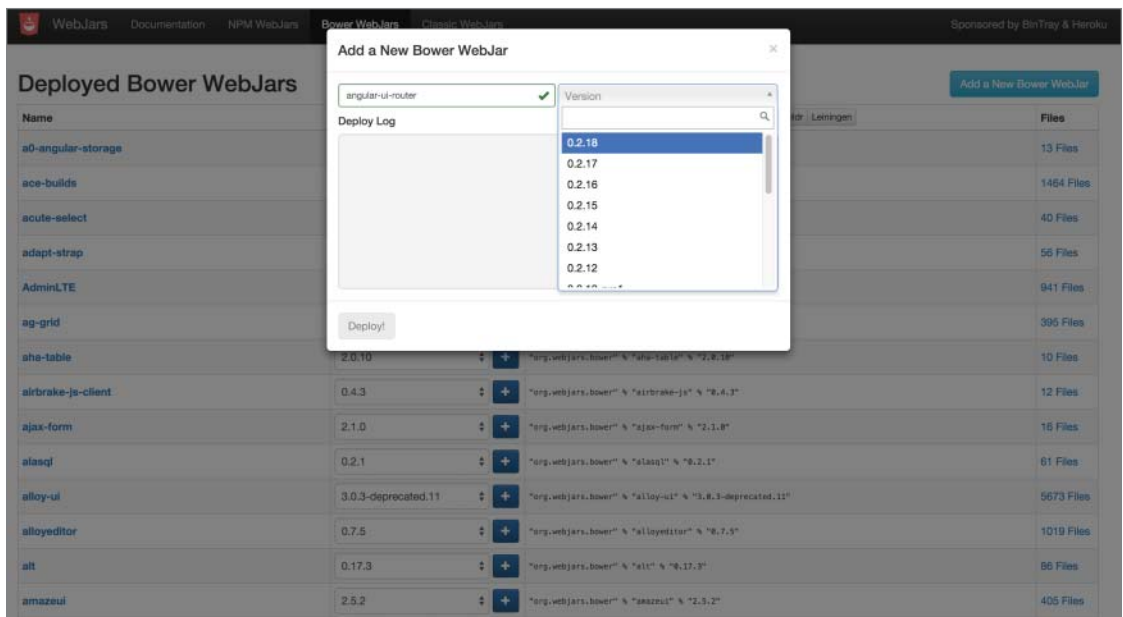


FIGURE 12-1

As can be seen by the screenshot, using the Webjars service to convert bower packages is very easy. To include a Webjar into a project (regardless of whether it's a bower Webjar or a normal classic Webjar) you simply need to add it as a dependency in `libraryDependencies`. As an example, if you want to add the latest current version of AngularJS, you simply need to add “org.webjars.bower” % “angularjs” % “1.4.8” in your `libraryDependencies` in `build.sbt`.

At its lowest level, you can access the contents of a Webjar using `.getResource`. To load the `angular.js` file from the above example, you can do the following:

```
scala.io.Source.fromURL(getClass
  .getResource("/META-INF/resources/webjars/angularjs/1.4.8/angular.js")
).mkString
```

This is quite a manual process and wouldn't typically be done (it's to demonstrate that Webjars are just standard resources). You should use one of the helpers that are provided by various web frameworks, and the list can be seen at <http://www.webjars.org/documentation>. It is recommended that you use the web framework integration if it's available, since the helpers often provide integrations and tooling features like caching and minification.

One caveat that you need to be aware of is that sometimes the dependencies that automatically get added into the Maven package may not actually be available. This can happen due to several reasons. One is that a dependency is not available on Bower because it hasn't been converted yet. The solution to this issue is simple; you just need to convert the dependency to a bower Webjar using the same process. The other cause could be that the dependency could not be converted due to a licensing issue. Since Bower Webjars deploy onto Bintray, the system requires a valid license to be defined. Bower package management allows you to specify Github repos as dependencies (and it doesn't mandate that a proper license be provided).

There are a couple of ways to resolve this issue. The most immediate one is to make an issue on the relevant project's website and ask them to provide a valid Bower license file. Another is to have a look if a classic Webjar exists for that dependency. Often people use Bower dependencies that just point to a Github repo that happens to be missing the license (even though the dependency in question has a valid license). Thankfully, SBT gives us the tools to deal with this situation.

If you have a look at the bower Webjar for foundation, you will see that it depends on `modernizr`, which isn't available as a bower Webjar (however, it is available as a classic Webjar).

```
libraryDependencies += Seq(
  "org.webjars.bower" % "foundation" % "5.5.4" exclude
    ("org.webjars.bower", "modernizr"),
  "org.webjars" % "modernizr" % "2.8.3"
)
```

As you can see, we use the `exclude` option to prevent a dependency from being loaded by Webjars. You can then manually load that dependency as a classic Webjar. Although this process is reminiscent of manual dependency management, it's also quite rare.

The last resort is to create your own Webjar and deploy it either locally or to your own Maven repository. Since Webjars are just Maven projects, creating them is very easy, and templates for

the current class Webjars can be found on the github organization page: <https://github.com/webjars>.

SUMMARY

For those of you who love to use JavaScript, this chapter has detailed some ways that you can interact with Scala using the JavaScript language. By using Scala.js you now know that you can compile Scala code straight to JavaScript. This allows you to share code between the server and the client, which can dramatically reduce boilerplate.

This also happens to be the last chapter in this book. We hope that it has proven useful for you and that it helps you program with Scala.

INDEX

Symbols

\$
 for printing variable value, 9–10
 in Scaladoc query, 99
... (wildcard operator), 8
@ (at sign), for tags, 117
-> syntax, 11

A

abstract type members, 159–161
abstract value members, 105
Acceptance Test-Driven Development (ATDD),
 86
acceptance tests, 85, 90–92
actors (Akka), 198–200
ad hoc polymorphism, 145, 146–149
add() function, 33
ADTs (Algebraic Data Types), case classes for,
 6–7
Akka (actors), 198–200
Akka-streams, 192
Algebraic Data Types (ADTs), case classes for,
 6–7
alias, for tuple construction, 11
Amazon Web Services S3, 134
ammonite-repl, 62
annotations, 117
 self-type, 155–157
anonymous function, .map to apply to collection
 elements, 10
Any class, 142
Any type, 17
AnyRef class, 17, 18, 142, 143

AnyVal class, 16–17, 18, 142
 value classes and, 143–145
ApiKey class, 16
Applicative Functor, 170–172
areEqual method, 147, 149–150
ArrayBuffer, implicit conversions into, 37–38
asynchronous call, 185
at sign (@), for tags, 117
ATDD (Acceptance Test-Driven Development),
 86
Atomic Variables, 183
atto library, 70
@author, 119, 120
automatic conversions between types, implicits
 for, 4

B

BDD (Business-Driven Development), 86
Big Data assets, Spark for, 200–202
Bloch, Joshua, *Effective Java*, 147
block elements, for documentation structure,
 110–113
Boolean, methods returning, 31
bounds, 149–155
Bower dependencies, 212
Bower packages, 211
Brown, Travis, 160
build tool, integrating Scaladoc creation with, 133
build.sbt files, 47
 dependencies for, 86
 locations of, 52
 specifying sub-projects, 52–53
buildSettings, breakup of, 48
bytecode, 188

C

Cake Pattern, self-type annotations and, 157
 call-by-name parameters, 35–36
 case classes, 5, 18
 for ADTs, 6–7
 destructive assignment of, pattern matching and, 8
 classes, inheritance diagrams for, 130
`clean` command, 60
 Code block type, wiki syntax for, 112
 code formatting utility, 81
 code repetitions, higher-order function for, 27
 code smell, 79
 CoffeeScript, 211
`collect` method, 9
 Collection API, 29
 collections, 18
 immutable and mutable, 10–12
 Java, 37–40
 methods, 27–29
 methods returning, 29–32
 ridding of unwanted elements, 28
`CommentFactoryBase`, 109
 companion objects, 14–15
 compilation, Zinc for accelerating, 76–77
`compile` command, 60
 compiler generated phase diagram, 69
 compile-time check, enforcing, 158
 concurrency, 179–203
 actors, 198–200
 Future composition, 185–187
 parallel collections, 187–191
 reactive streams, 192–195
 Spark, 200–202
 STM (Software Transactional Memory), 195–198
 synchronize/atomic variables, 181–184
`<configuration>` element, for plugin, 67
 console command, 60
`consoleProject` command, 61
`consoleQuick` command, 61
`@constructor`, 119
`contains()` method, 31
 content pane in Scaladoc, 97–98, 100–106
 entity member details, 105–106
 filtering and ordering options, 103–105
 top-level information, 101–103
 content tag, 118
`@contentDiagram`, 118, 129–131, 132

context bounds, 149–150
 contravariant type `A occurs...` error, 155
 contravariant variance type, 152, 154
 conversions, implicit, 143
 converting types, 10–11
`CoreProbe`, adding to Java source directory, 75
`count()` method, 31
 covariant variance type, 152, 154
 CPU core, handling multiple, 179
 CSS, in Scaladoc, 136–138
 currying, 32–34
 custom tasks, 50

D

data-driven tests, 88–89
 DDD (Domain-Driven Development), 86
 deadlock, 182
 Debian package, creating, 58–59
`debian:packageBin` command, 59
`@define`, 125–132
 dependencies
 for JavaScript in `Scala.js`, 207
 library, 70
 in multi-project builds, 53–55
 in SBT, 50–51
 self-type annotations for, 156
 specifying for build, 59
`Dependencies.scala` file, 86
`@deprecated`, 117, 121–122
 deprecated value members, 106
 DI (Dependency Injection), 7
 distributed systems, 179
`doc` command, 61
 Docker image, creating, 59–60
`@documentable`, 118, 125–132
 documentation
 creating objectives for, 96–97
 generating, 55–56
 locations for comments, 117
 reasons for, 96–97
 with Scaladoc, 95–138
 structure, 97–106
 tables and CSS, 136–138
 tagging, 117–132
 for groups, 123–125
 wiki syntax, 108–117

- block elements for structure, 110–113
 - formatting with inline syntax, 108–109
 - linking, 113–116
 - nesting inline styles, 109
- Domain-Driven Development (DDD), 86
- dot command, 130
- dpkg, 59
- DSLs, implicit conversions in, 4
- duck typing, 161–162
- dynamic programming, 161–164
- dynamic trait, 162–164
- dynamic typing
 - advantages, 141
 - vs. static, 140–141

E

- EMCAScript, 211
- empty list, error from, 32
- entities, by category, 105–106
- entity links, 113–114
- entity signature, in Scaladoc, 103
- enumerations, 42–43
- equality concept, 146–149
 - comparison, pattern matching and, 8
- errors
 - detecting, 18
 - from empty list, 32
 - testing for thrown, 87
- eta-expansion, 154
- Eventually trait, 92
- @example, 122
- exceptions, Scala.js and, 210
- ExecutionContext, 185
- exists() method, 31
- explicit values, 3
- expressions
 - measuring execution speed, 188
 - vs. statements, 143
 - statements as, 9
- extractAdultUserNames method, 23

F

- factorial computation, 46–47
- failure with certainty, testing, 74
- fastOptJS, 210
- F-bounded polymorphic types, 158

- Fibonacci function, 25–26
- filtering options, in Scaladoc content pane, 103–105
- find() method, 31
- flatMap() function, 29
 - for monads, 172
- fold() method, 31–32
- footer, for documentation page, 132
- for comprehension, 12
- forall() method, 31
- foreach(), 31
- functional design patterns, 167–171
 - Applicative Functor, 170–172
 - functor, 167–170
- functional programming, 19–36, 165–177
- functions
 - higher-order, 26–27
 - partial, 9
 - partially applied, 32–34
 - pure, 22–23
 - return types of, 2–3
- functor, 167–170
- Future composition, 168, 185–187

G

- Gaitling, 93
- Garbage Collector, and micro benchmarks, 189
- generic programming, 146
- get() method, 35
- .getResource, to access Webjar contents, 212
- Git BASH, 65
- Github, Scala.js on, 206
- global settings, storage locations, 46
- Google Closure Compiler, and Scala.js, 206
- @group, 124
- groupBy() method, 29–30
- @groupdesc, 125
- grouped data, tuples to store multiple types, 5
- @groupname, 124
- @groupprio, 125
- groups, Scaladoc tagging for, 104, 123–125

H

- hashCode, 17
- Haskell developers, 20

help, for scala-maven-plugin, 72
 higher-order functions, 26–27
 Horizontal line, wiki syntax for, 112
 horizontal scaling, 179
 HTTP call, method to make, 3–4

I

identity element, semigroup with, 174–176
 identity function, 169
 illegal inheritance error message, 157
 immutability, 20–22, 36
 immutable collections, 10–12
 implicit conversions, 143
 implicit parameters, 3–5
 index, of collection element, 30
 index pane, in Scaladoc, 97–100
 @inheritanceDiagram, 129–131, 132
 @inheritdoc, 129, 131
 initialization, strict vs. non-strict, 35–36
 inline wiki syntax, 108–109
 InputKey, 49
 instance constructors, 105
 instance of class, values as, 142
 integration tests, 85, 87–93
 interfaces, 40–42
 invariant variance type, 152
 inversion of control, 7
 Is method, 4

J

JAR files, in maven repositories, 51
 Java, 180
 joint compilation with, 74–76
 Java collections, 37–40
 Java developers, 20
 Java libraries, in Scala projects, 40
 JavaConversions object, 37–40
 JavaConverters object, 37–40
 java.lang.SuppressWarning annotation, 82
 JavaScript, compiling Scala code to, 205–213
 Jenkins, 48–49
 JMH (Java Microbenchmark Harness), 189
 Jrebel, 62
 “Just In Time” compilation (JIT), 188–189
 JVM machine, 188

K

kind filter, 100
 kinds, 165–167
 Knight, Tom, 195

L

%lambda expression%, 28
 lazy val, 35–36, 48
 libraries, for testing, 55
 library dependencies, adding, 70
 lift by functor, 168–169
 Linear Supertypes, 103
 linking, wiki syntax for, 113–116
 List kind, 165, 168
 List types, 10
 wiki syntax for, 112–113
 load testing, 93–94
 login and logout, implementations for, 7
 lower bounds, 150–151

M

main method, 47
 mainframes, 179
 MainWithArgsInFile, 68–69
 .map, to apply anonymous function to collection
 element, 10
 map() function, 28
 map method, 37
 Mapper type, constructing, 166–167
 map-reduce model, 32
 Maven, 63–77
 basics, 64–67
 central repository, search for dependencies, 51
 configuring for Scaladoc, 133–134
 dependencies, 212
 external installations, 66
 publishing to repo, 57
 max() method, 31
 Maybe type, 34
 measure() function, 36
 measure method, 90
 member permalinks, 123
 members, documentation, 105–106
 memory leaks, finding, 93

- merging streams, 194–195
- methods, static, 41–42
- micro benchmarking, 89
 - and Garbage Collector, 189
- micro management, 28
- @migration annotation, 117
- min() method, 31
- mkString() method, 31
- mockito library, 92
- mocks, in testing, 92–93
- modernizr, 212
- Moggi, Eugenio, 172
- monads, 172–173
- monkey patching, 4
- monoids, 174–176
- mutable collections, 10–12

N

- nesting inline styles, in documentation, 109
- non-strict initialization, 35–36
- NoSymbol, 83
- @note, 122
- null handling, 34–35
- NullPointerException, 34
- number literals, 3
- numbers, Scala.js and, 210

O

- Object type, 16
- object-oriented programming, 19
- objects, 13
 - content diagrams for, 130
- Observable, 192–195
- Odersky, Martin, 18
 - Programming in Scala*, 147
- Option, 165, 168
 - for Maybe type, 34–35
 - sum function for, 176
- ordering options, in Scaladoc content pane, 103–105

P

- package command, 61
- package objects, 15

- packageBin command, 61
- packages
 - content diagrams for, 130
 - controlling scoping, 13–17
 - sbt-native-packager for, 58–59
- packageSRC command, 61
- Paragraph block type, wiki syntax for, 112
- parallel collections, 187–191
- parallel execution, and testing, 90
- @param, 118, 119, 126
- parametric polymorphism, 146
- ParMap, documentation, 102
- partial functions, 9
- partition() method, 30
- pattern matching, 8–12, 18, 34
- performance, 18
 - parallel collections and, 187
 - parallel operations and conversion, 190–191
 - testing, 89–90
- PGP signature, 56
- Pi, calculating, 198–200
- Pierce, Benjamin, *Types and Programming Languages*, 140
- Play framework, 58
- plugins
 - <configuration> element for, 67
 - for SBT, 61–62
 - storage locations, 46
- polymorphism, 145–149
- POM (Project Object Model), 64–65
 - in maven repositories, 51
 - metadata, 57
 - source directory layout, 65–66
- primitive types, and references in type system, 16
- println() statement, 181–186, 197–198
- private keyword, 15–16
- Probe, creating instance, 66
- project management, integrating Scaladoc creation with, 133
- property tests, 85, 88–89
- protected variables, in Scaladoc, 100
- publish command, 61
- “Publisher/Subscriber” pattern, 192
- publishing Scaladoc, 134–135
- publishLocal command, 61
- publish-m2 command, 56
- publishM2 command, 61
- publishSigned task, 57
- pure functions, 22–23

Q

query box, ordering options, 104

R

race conditions, 182
RCS version control system, 182
RDD[String] type, 201
reactive streams, 192–195
recursion, 23–26
reduce() method, 31–32
references types, 143
registerTempTable() method (Spark), 202
release management, 56–62
Reload command, 61
REPL, 71
 and companion object, 15
replaceSpaceWithUnderScore method, 4–5
repositories, specifying third-party, 51
repository manager, Sonatype Nexus as, 56–57
@return, 119, 126, 127
return types
 of functions, 2–3
 of statements, 9
reverse method, 30
root project, Docker image for, 60
Ruby on Rails, 211
run command, 48, 61
runtime, type checking at, 141
RxJava library, 192

S

SBT. . *See* Simple Build Tool (SBT)
sbt publish-local, 56
sbt-assembly plugin, 56
sbt-dependency-graph, 61
sbt-git, 62
sbt-musical, 62
sbt-native-packager, 58–59
sbt-pgp plugin, 56
sbt-release, 62
sbt-revolver, 61–62
SbtScalariform.scalariformSettings, 81
sbt-updates, 62
Scala

 implicits, 3–5
 and Java collections, 37–40
 joint compilation with Java, 74–76
 syntax, 1
 unified type system, 141–145
Scala Standard Library Regex class,
 documentation for, 96
scala-async library, 187
scalacOptions key, 166
 in SBT build file, 163
Scaladoc
 content for landing page, 132
 documentation with, 95–138
 HTML tags supported, 136–138
 integrating creation with project, 133–134
 location for, 117
 options, 132–133
 publishing, 134–135
 structure, 97–106
 content pane, 100–106
 index pane, 98–100
 layout, 97–98
scaladoc command line tool, 97
 invoking, 106–107
 additional options, 132–133
 options, 107
scala Enumeration abstract class, extending,
 42–43
Scala.js, 205–213
 peculiarities, 210
scala-maven-plugin, 67–70
 <executions> element, 75
 help for, 72
ScalaMeter, 89–90
ScalaSTM, 195
scalastyle, 79–80
scalastyle-config.xml file, generating, 80
Scalatags, 208–209
ScalaTest, 72–74, 86
ScalaTrait interface, 41
ScalaTrait\$class.class, 41
scalaz, 170
Scaliform, 81
Scapegoat, 82
Schönfinkelization, 32
Scoverage, 84
sealed keyword, 6
searching, in Scaladoc, 98–99
@see, 120

`selectDynamic` method, 163
 Selenium, 90–92
`self` keyword, 156
 self-recursive types, 158–159
 self-type annotations, 155–157
 semigroup, 173–174
 with identity element, 174–176
`Seq` constructor, 10
 serialization, 150
`SettingKey`, 49
 shadowed implicit value members, 106
 shared library, folder structure, 52
`show` command, 50
`show()` method (Spark), 202
 Simple Build Tool (SBT), 18, 45
 advanced usage, 52–56
 basic usage, 46–51
 commands, 60–61
 configs, 46
 configuring for doc task, 134
 dependencies, 50–51
 doc task to generate Scaladoc, 134
 plugins, 61–62
 project structure, 47–49
 resolvers, 51
 scalacOptions key, 163
 Scala.js as plugin, 206–210
 scopes, 49
 testing in console, 55–56
`@since`, 118, 120
 single-operation transaction, 197
 singletons, 13
`.sjsir` files, 205–213
 Software Transactional Memory (STM), 195–198
 Sonarqube, 84
 Sonatype, deploying to, 56–57
`sortBy()` method, 30
`sortWith()` method, 30
 source code, locating, 67
 Spark, 200–202
`sqrt` function, 173
 stack overflow, 24
 standalone tag, 118
 statement coverage, 84
 static methods, 41–42
 static types, 2–7
 static typing
 advantages, 141
 vs. dynamic, 140–141

STM (Software Transactional Memory), 195–198
`StringLike` trait, 106
 strings, interpolation, 9–10
 structural types, 161–162
 structured tag, 118
 stubs, 92–93
 subtype polymorphism, 145–146
`sum()` method, 31
 summation, abstracting concept of, 174–176
 symbol tag, 118
 synchronize/atomic variables, 181–184
`System.nanoTime`, 188

T

tables, in Scaladoc, 136–138
 tagging, in documentation, 117–132
 tail recursion, 24–25
`@tailrec` annotation, 25–26
`TaskKey`, 49, 50
`@template`, 131
`test` command, 61
 test-driven development (TDD), 85
 testing, 85–94
 acceptance tests, 90–92
 data-driven, 88–89
 load, 93–94
 mocks in, 92–93
 performance, 89–90
 running, 72–74
 ScalaTest, 86
 terminology, 85–86
`testOnly` command, 61
`testQuick` command, 61
`test-quick` task, 55
`this` keyword, alias for, 157
 thread, for concurrent process, 181
 threads pool, 185
`@throws`, 120, 121
 title, for Scaladoc, 132
 Title block type, wiki syntax for, 112
`@todo`, 120
`toString` method, in case classes, 5
`touchdown` method, 74
`@tparam`, 122
 traits, 6–7, 40–42
 defining, 14
 inheritance diagrams for, 130
 instantiating, 14

- transaction, single-operation, 197
- transform() method, 33
- transformations, RDD and, 201
- transparency, 20
- Try type, 168
- tuples, 5, 11
- tut, 62
- type bound, 150–151
- type checker (compiler module), 140
- type class, 146, 147–148
- type members, 105
- type mismatch error, 151–155
- type system, 18, 139–164. *See also* kinds
 - abstract type members, 159–161
 - automatic conversions, implicits for, 4
 - basics, 140–141
 - dynamic vs. static, 140–141
 - List, 10
 - mandatory signatures, 2
 - self-recursive types, 158–159
 - self-type annotations, 155–157
 - testing for inheritance, 87
 - unified type system, 141–145
- typesafe forced casting, pattern matching and, 8
- TypeScript, 211

U

- unit primitive, for monads, 172
- unit tests, 85, 87
- Unit type, 143
- update command, 61
- upper bounds, 150–151
- @usecase, 128–129, 131
- userAwareAction() method, 27
- user-defined classes, 143
- uuids, generator for, 50

V

- val database: Database, 7
- val keyword, 20–22
- value classes, 142–145

- restrictions for, 144
- value members, 106
- values
 - as instance of class, 142
 - methods returning, 31–32
- var keyword, 20–22
- variables
 - \$ for printing value, 9–10
 - Atomic Variables, 183
 - declarations, 2
 - protected, in Scaladoc, 100
 - synchronize/atomic, 181–184
 - volatile, 184
- variance, 151–155
- Venners, Bill, 160
- @version, 120
- vertical scaling, 179
- view bounds, 150
- views, 143
- visitor pattern in Java, 6
- void, 143
- volatile variables, 184

W

- WartRemover, 82–84
- Webjars, 211–213
- while, 23–24
- white box testing technique, 84
- wiki syntax, for documentation, 108–117
- wildcard operator (...), 8

Z

- Zinc, 76–77
- zipWithIndex() method, 30