Community Experience Distilled

Learning Concurrent Programming in Scala

Second Edition

Learn the art of building intricate, modern, scalable, and concurrent applications using Scala

Foreword by Martin Odersky, Professor at EPFL, the creator of Scala

Aleksandar Prokopec

Packt

Learning Concurrent Programming in Scala

Second Edition

Learn the art of building intricate, modern, scalable, and concurrent applications using Scala

Aleksandar Prokopec

BIRMINGHAM - MUMBAI

Learning Concurrent Programming in Scala

Second Edition

First published: November 2014

Second edition: February 2017

Production reference: 1170217

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78646-689-1

www.packtpub.com

Credits

Author

Aleksandar Prokopec

Reviewers

Vikash Sharma
Dominik Gruntz
Zhen Li
Lukas Rytz
Michel Schinz
Samira Tasharofi

Commissioning Editor

Aaron Lazar

Acquisition Editor

Sonali Vernekar

Content Development Editor

Rohit Kumar Singh

Technical Editor

Pavan Ramchandani

Copy Editor

Safis Editing

Project Coordinator

Vaidehi Sawant

Proofreader

Safis Editing

Indexer

Aishwarya Gangawane

Graphics

Jason Monteiro

Production Coordinator

Shantanu Zagade

Foreword

Concurrent and parallel programming have progressed from niche disciplines, of interest only to kernel programming and high-performance computing, to something that every competent programmer must know. As parallel and distributed computing systems are now the norm, most applications are concurrent, be it for increasing the performance or for handling asynchronous events.

So far, most developers are unprepared to deal with this revolution. Maybe they have learned the traditional concurrency model, which is based on threads and locks, in school, but this model has become inadequate for dealing with massive concurrency in a reliable manner and with acceptable productivity. Indeed, threads and locks are hard to use and harder to get right. To make progress, one needs to use concurrency abstractions that are at a higher level and composable.

15 years ago, I worked on a predecessor of Scala: Funnel was an experimental programming language that had concurrent semantics at its core. All the programming concepts were explained in this language as syntactic sugar on top of functional nets, an object-oriented variant of join calculus . Even though join calculus is a beautiful theory, we realized after some experimentation that the concurrency problem is more multifaceted than what can be comfortably expressed in a single formalism. There is no silver bullet for all concurrency issues; the right solution depends on what one needs to achieve. Do you want to define asynchronous computations that react to events or streams of values? Or have autonomous, isolated entities communicating via messages? Or define transactions over a mutable store? Or, maybe the primary purpose of parallel execution is to increase the performance? For each of these tasks, there is an abstraction that does the job: futures, reactive streams, actors, transactional memory, or parallel collections.

This brings us to Scala and this book. As there are so many useful concurrency abstractions, it seems unattractive to hardcode them all in a programming language. The purpose behind the work on Scala was to make it easy to define high-level abstractions in user code and libraries. This way, one can define the modules handling the different aspects of concurrent programming. All of these modules would be built on a low-level core that is provided by the host system. In retrospect, this approach has worked well. Today, Scala has some of the most powerful and elegant libraries for concurrent programming. This book will take you on a tour of the most important ones, explaining the use case for each and the application patterns.

This book could not have a more expert author. Aleksandar Prokopec contributed to some of the most popular Scala libraries for concurrent and parallel programming. He also invented some of the most intricate data structures and algorithms. With this book, he created a readable tutorial at the same time and an authoritative reference for the area that he had worked in. I believe that *Learning Concurrent Programming in Scala, Second Edition* will be a mandatory reading for everyone who writes concurrent and parallel programs in Scala. I also expect to see it on the bookshelves of many people who just want to find out about this fascinating and fast moving area of computing.

Martin Odersky

Professor at EPFL, the creator of Scala

About the Author

Aleksandar Prokopec, who also authored the first edition of this book, is a concurrent and distributed programming researcher. He holds a PhD in computer science from the École Polytechnique Fédérale de Lausanne, Switzerland. He has worked at Google and is currently a principal researcher at Oracle Labs.

As a member of the Scala team at EPFL, Aleksandar actively contributed to the Scala programming language, and he has worked on programming abstractions for concurrency, data-parallel programming support, and concurrent data structures for Scala. He created the Scala Parallel Collections framework, which is a library for high-level data-parallel programming in Scala, and participated in working groups for Scala concurrency libraries, such as Futures, Promises, and ScalaSTM. Aleksandar is the primary author of the reactor programming model for distributed computing.

Acknowledgements

First of all, I would like to thank my reviewers, Samira Tasharofi, Lukas Rytz, Dominik Gruntz, Michel Schinz, Zhen Li, and Vladimir Kostyukov for their excellent feedback and useful comments. I would also like to thank the editors at Packt, Kevin Colaco, Sruthi Kutty, Kapil Hemnani, Vaibhav Pawar, and Sebastian Rodrigues for their help with writing this book. It really was a pleasure to work with these people.

The concurrency frameworks described in this book wouldn't have seen the light of the day without a collaborative effort of a large number of people. Many individuals have somehow, directly or indirectly, contributed to the development of these utilities. These people are the true heroes of Scala concurrency, and they are to thank for Scala's excellent support for concurrent programming. It is difficult to enumerate all of them here, but I tried my best. If somebody feels left out, they should ping me, and, they'll probably appear in the next edition of this book.

It goes without saying that Martin Odersky is to thank for creating the Scala programming language, which was used as a platform for the concurrency frameworks described in this book. Special thanks goes to him, all the people that were part of the Scala team at the EPFL through the last 10 or more years, and the people at Typesafe, who are working hard to keep Scala one of the best general purpose languages out there.

Most of the Scala concurrency frameworks rely on the work of Doug Lea, in one way or another. His Fork/Join framework underlies the implementation of the Akka actors, Scala Parallel Collections, and the Futures and Promises library, and many of the JDK concurrent data structures described in this book are his own implementation. Many of the Scala concurrency libraries were influenced by his advice.

The Scala Futures and Promises library was initially designed by Philipp Haller, Heather Miller, Vojin Jovanović, and me from the EPFL, Viktor Klang and Roland Kuhn from the Akka team, and Marius Eriksen from Twitter, with contributions from Havoc Pennington, Rich Dougherty, Jason Zaugg, Doug Lea, and many others.

Although I was the main author of the Scala Parallel Collections, this library benefited from the input of many different people, including Phil Bagwell, Martin Odersky, Tiark Rompf, Doug Lea, and Nathan Bronson. Later on, Dmitry Petrashko and I started working on an improved version of parallel and standard collection operations, optimized through the use of Scala Macros. Eugene Burmako and Denys Shabalin are one of the main contributors to the Scala Macros project.

The work on the Rx project was started by Erik Meijer, Wes Dyer, and the rest of the Rx team. Since its original .NET implementation, the Rx framework has been ported to many different languages, including Java, Scala, Groovy, JavaScript, and PHP, and has gained widespread adoption thanks to the contributions and the maintenance work of Ben Christensen, Samuel Grütter, Shixiong Zhu, Donna Malayeri, and many other people.

Nathan Bronson is one of the main contributors to the ScalaSTM project, whose default implementation is based on Nathan's CCSTM project. The ScalaSTM API was designed by the ScalaSTM expert group, composed of Nathan Bronson, Jonas Bonér, Guy Korland, Krishna Sankar, Daniel Spiewak, and Peter Veentjer.

The initial Scala actor library was inspired by the Erlang actor model and developed by Philipp Haller. This library inspired Jonas Bonér to start the Akka actor framework. The Akka project had many contributors, including Viktor Klang, Henrik Engström, Peter Vlugter, Roland Kuhn, Patrik Nordwall, Björn Antonsson, Rich Dougherty, Johannes Rudolph, Mathias Doenitz, Philipp Haller, and many others.

Finally, I would like to thank the entire Scala community for their contributions, and for making Scala an awesome programming language.

About the Reviewers

Vikash Sharma is a software developer and open source technology evangelist, located in India. He tries to keep things simple and that helps him writing clean and manageable code. He has authored a video course for Scala. He is employed as an associate consultant with Infosys and has also worked as a Scala developer.

Thank you would not suffice for the support I got from my family, Mom, Dad and Brother. I really want to appreciate everyone who were there when I needed them the most. Special thanks to Vijay Athikesavan for passing to me the insights he had for coding.

Dominik Gruntz has a PhD from ETH Zürich and has been a Professor of Computer Science at the University of Applied Sciences FHNW since 2000. Besides his research projects, he teaches a course on concurrent programming. Some years ago, the goal of this course was to convince the students that writing correct concurrent programs is too complicated for mere mortals (an educational objective that was regularly achieved). With the availability of high-level concurrency frameworks in Java and Scala, this has changed, and this book, Learning Concurrent Programming in Scala, is a great resource for all programmers who want to learn how to write correct, readable, and efficient concurrent programs. This book is the ideal textbook for a course on concurrent programming.

Thanks that I could support this project as a reviewer.

Zhen Li acquired an enthusiasm of computing early in elementary school when she first learned Logo. After earning a Software Engineering degree at Fudan University in Shanghai, China and a Computer Science degree from University College Dublin, Ireland, she moved to the University of Georgia in the United States for her doctoral study and research. She focused on psychological aspects of programmers' learning behaviors, especially the way programmers understand concurrent programs. Based on the research, she aimed to develop effective software engineering methods and teaching paradigms to help programmers embrace concurrent programs.

Zhen Li had practical teaching experience with undergraduate students on a variety of computer science topics, including system and network programming, modeling and simulation, as well as human-computer interaction. Her major contributions in teaching computer programming were to author syllabi and offer courses with various programming languages and multiple modalities of concurrency that encouraged students to actively acquire software design philosophy and comprehensively learn programming concurrency.

Zhen Li also had a lot of working experience in industrial innovations. She worked in various IT companies, including Oracle, Microsoft, and Google, over the past 10 years, where she participated in the development of cutting-edge products, platforms and infrastructures for core enterprise, and Cloud business technologies. Zhen Li is passionate about programming and teaching. You are welcome to contact her at `janeli@uga.edu`.

Lukas Rytz is a compiler engineer working in the Scala team at Typesafe. He received his PhD from EPFL in 2013, and has been advised by Martin Odersky, the inventor of the Scala programming language.

Michel Schinz is a lecturer at EPFL.

Samira Tasharofi received her PhD in the field of Software Engineering from the University of Illinois at Urbana-Champaign. She has conducted research on various areas, such as testing concurrent programs and in particular actor programs, patterns in parallel programming, and verification of component-based systems.

Samira has reviewed several books, such as Actors in Scala, Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures (Patterns and Practices), and Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures (Patterns and Practices). She was also among the reviewers of the research papers for software engineering conferences, including ASE, AGERE, SPLASH, FSE, and FSEN. She has served as a PC member of the 4th International Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE 2014) and 6th IPM International Conference on Fundamentals of Software Engineering (FSEN 2015).

I would like to thank my husband and mom for their endless love and support.

www.PacktPub.com

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https://goo.gl/ldH2Nv`.

If you'd like to join our team of regular reviewers, you can email us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Dedicated to Sasha,

She's probably the only PhD in physical chemistry ever to read this book.

Table of Contents

Preface

Concurrency is everywhere. With the rise of multicore processors in the consumer market, the need for concurrent programming has overwhelmed the developer world. Where it once served to express asynchronously in programs and computer systems and was largely an academic discipline, concurrent programming is now a pervasive methodology in software development. As a result, advanced concurrency frameworks and libraries are sprouting at an amazing rate. Recent years have witnessed a renaissance in the field of concurrent computing.

As the level of abstraction grows in modern languages and concurrency frameworks, it is becoming crucial to know how and when to use them. Having a good grasp of the classical concurrency and synchronization primitives, such as threads, locks, and monitors, is no longer sufficient. High-level concurrency frameworks, which solve many issues of traditional concurrency and are tailored towards specific tasks, are gradually overtaking the world of concurrent programming.

This book describes high-level concurrent programming in Scala. It presents detailed explanations of various concurrency topics and covers the basic theory of concurrent programming. Simultaneously, it describes modern concurrency frameworks, shows their detailed semantics, and teaches you how to use them. Its goal is to introduce important concurrency abstractions and, at the same time, show how they work in real code.

We are convinced that, by reading this book, you will gain both a solid theoretical understanding of concurrent programming and develop a set of useful practical skills that are required to write correct and efficient concurrent programs. These skills are the first steps toward becoming a modern concurrency expert.

We hope that you will have as much fun reading this book as we did writing it.

What this book covers

This book is organized into a sequence of chapters with various topics on concurrent programming. The book covers the fundamental concurrent APIs that are a part of the Scala runtime, introduces more complex concurrency primitives, and gives an extensive overview of high-level concurrency abstractions.

`Chapter 1`, *Introduction*, explains the need for concurrent programming and gives some philosophical background. At the same time, it covers the basics of the Scala programming language that are required for understanding the rest of this book.

`Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, teaches you the basics of concurrent programming. This chapter will teach you how to use threads and how to protect access to shared memory and introduce the Java Memory Model.

`Chapter 3`, *Traditional Building Blocks of Concurrency*, presents classic concurrency utilities, such as thread pools, atomic variables, and concurrent collections, with a particular focus on the interaction with the features of the Scala language. The emphasis in this book is on the modern, high-level concurrent programming frameworks. Consequently, this chapter presents an overview of traditional concurrent programming techniques, but it does not aim to be extensive.

`Chapter 4`, *Asynchronous Programming with Futures and Promises*, is the first chapter that deals with a Scala-specific concurrency framework. This chapter presents the futures and promises API and shows how to correctly use them when implementing asynchronous programs.

`Chapter 5`, *Data-Parallel Collections*, describes the Scala parallel collections framework. In this chapter, you will learn how to parallelize collection operations, when it is allowed to parallelize them, and how to assess the performance benefits of doing so.

`Chapter 6`, *Concurrent Programming with Reactive Extensions*, teaches you how to use the Reactive Extensions framework for event-based and asynchronous programming. You will see how the operations on event streams correspond to collection operations, how to pass events from one thread to another, and how to design a reactive user interface using event streams.

`Chapter 7`, *Software Transactional Memory*, introduces the ScalaSTM library for transactional programming, which aims to provide a safer, more intuitive, shared-memory programming model. In this chapter, you will learn how to protect access to shared data using scalable memory transactions and, at the same time, reduce the risk of deadlocks and race conditions.

`Chapter 8`, *Actors*, presents the actor programming model and the Akka framework. In this chapter, you will learn how to transparently build message-passing distributed programs that run on multiple machines.

`Chapter 9`, *Concurrency in Practice*, summarizes the different concurrency libraries introduced in the earlier chapters. In this chapter, you will learn how to choose the correct concurrency abstraction to solve a given problem, and how to combine different concurrency abstractions together when designing larger concurrent applications.

`Chapter 10`, *Reactors*, presents the reactor programming model, whose focus is improved composition in concurrent and distributed programs. This emerging model enables separation of concurrent and distributed programming patterns into modular components called protocols.

While we recommend that you read the chapters in the order in which they appear, this is not strictly necessary. If you are well acquainted with the content in `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, you can study most of the other chapters directly. The only chapters that rely on the content from all the preceding chapters are `Chapter 9`, *Concurrency in Practice*, where we present a practical overview of the topics in this book, and `Chapter 10`, *Reactors*, for which it is helpful to understand how actors and event streams work.

What you need for this book

In this section, we describe some of the requirements that are necessary to read and understand this book. We explain how to install the Java Development Kit, which is required to run Scala programs and show how to use Simple Build Tool to run various examples.

We will not require an IDE in this book. The program that you use to write code is entirely up to you, and you can choose anything, such as Vim, Emacs, Sublime Text, Eclipse, IntelliJ IDEA, Notepad++, or some other text editor.

Installing the JDK

Scala programs are not compiled directly to the native machine code, so they cannot be run as executables on various hardware platforms. Instead, the Scala compiler produces an intermediate code format called the Java bytecode. To run this intermediate code, your computer must have the Java Virtual Machine software installed. In this section, we explain how to download and install the Java Development Kit, which includes the Java Virtual Machine and other useful tools.

There are multiple implementations of the JDK that are available from different software vendors. We recommend that you use the Oracle JDK distribution. To download and install the Java Development Kit, follow these steps:

1. Open the following URL in your web browser: `www.oracle.com/technetwork/java/javase/downloads/index.html`.
2. If you cannot open the specified URL, go to your search engine and enter the keywords **JDK Download**.
3. Once you find the link for the Java SE, download on the Oracle website, download the appropriate version of JDK 7 for your operating system: Windows, Linux, or Mac OS X; 32-bit or 64-bit.
4. If you are using the Windows operating system, simply run the installer program. If you are using the Mac OS X, open the dmg archive to install JDK. Finally, if you are using Linux, decompress the archive to a XYZ directory, and add the bin subdirectory to the PATH variable:

```
export PATH=XYZ/bin:$PATH
```

5. You should now be able to run the java and javac commands in the terminal. Enter the **`javac`** command to see if it is available (you will never invoke this command directly in this book, but running it verifies that it is available).

It is possible that your operating system already has JDK installed. To verify this, simply run the **`javac`** command, as we did in the last step in the preceding description.

Installing and using SBT

Simple Build Tool (**SBT**) is a command-line build tool used for Scala projects. Its purpose is to compile Scala code, manage dependencies, continuous compilation and testing, deployment, and many other uses. Throughout this book, we will use SBT to manage our project dependencies and run example code.

To install SBT, follow these instructions:

1. Go to the `http://www.scala-sbt.org/` URL.
2. Download the installation file for your platform. If you are running on Windows, this is the `msi` installer file. If you are running on Linux or OS X, this is the `zip` or `tgz` archive file.

3. Install SBT. If you are running on Windows, simply run the installer file. If you are running on Linux or OS X, unzip the contents of the archive in your home directory.

You are now ready to use SBT. In the following steps, we will create a new SBT project:

1. Open a Command Prompt if you are running on Windows, or a terminal window if you are running on Linux or OS X.
2. Create an empty directory called scala-concurrency-examples:

```
$ mkdir scala-concurrency-examples
```

3. Change your path to the scala-concurrency-examples directory:

```
$ cd scala-concurrency-examples
```

4. Create a single source code directory for our examples:

```
$ mkdir src/main/scala/org/learningconcurrency/
```

5. Now, use your editor to create a build definition file named build.sbt. This file defines various project properties. Create it in the root directory of the project (scala-concurrency-examples). Add the following contents to the build definition file (note that the empty lines are mandatory):

```
name := "concurrency-examples"

version := "1.0"

scalaVersion := "2.11.1"
```

6. Finally, go back to the terminal and run SBT from the root directory of the project:

```
$ sbt
```

7. SBT will start an interactive shell, which we will use to give SBT various build commands.

Now, you can start writing Scala programs. Open your editor, and create a source code file named `HelloWorld.scala` in the `src/main/scala/org/learningconcurrency` directory. Add the following contents to the `HelloWorld.scala` file:

```
package org.learningconcurrency

object HelloWorld extends App {
  println("Hello, world!")
}
```

Now, go back to the terminal window with the SBT interactive shell and run the program with the following command:

```
> run
```

Running this program should give the following output:

```
Hello, world!
```

These steps are sufficient to run most of the examples in this book. Occasionally, we will rely on external libraries when running the examples. These libraries are resolved automatically by SBT from standard software repositories. For some libraries, we will need to specify additional software repositories, so we add the following lines to our `build.sbt` file:

```
resolvers ++= Seq(
  "Sonatype OSS Snapshots" at
    "https://oss.sonatype.org/content/repositories/snapshots",
  "Sonatype OSS Releases" at
    "https://oss.sonatype.org/content/repositories/releases",
  "Typesafe Repository" at
    "http://repo.typesafe.com/typesafe/releases/"
)
```

Now that we have added all the necessary software repositories, we can add some concrete libraries. By adding the following line to the `build.sbt` file, we obtain access to the Apache Commons IO library:

```
libraryDependencies += "commons-io" % "commons-io" % "2.4"
```

After changing the `build.sbt` file, it is necessary to reload any running SBT instances. In the SBT interactive shell, we need to enter the following command:

```
> reload
```

This enables SBT to detect any changes in the build definition file and download additional software packages when necessary.

Different Scala libraries live in different namespaces called packages. To obtain access to the contents of a specific package, we use the `import` statement. When we use a specific concurrency library in an example for the first time, we will always show the necessary set of `import` statements. On subsequent uses of the same library, we will not repeat the same `import` statements.

Similarly, we avoid adding package declarations in the code examples to keep them short. Instead, we assume that the code in a specific chapter is in the similarly named package. For example, all the code belonging to `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, resides in the `org.learningconcurrency.ch2` package. Source code files for the examples presented in that chapter begin with the following code:

```
package org.learningconcurrency
package ch2
```

Finally, this book deals with concurrency and asynchronous execution. Many of the examples start a concurrent computation that continues executing after the main execution stops. To make sure that these concurrent computations always complete, we will run most of the examples in the same JVM instance as SBT itself. We add the following line to our `build.sbt` file:

```
fork := false
```

In the examples, where running in a separate JVM process is required, we will point this out and give clear instructions.

Using Eclipse, IntelliJ IDEA, or another IDE

An advantage of using an **Integrated Development Environment** (**IDE**) such as Eclipse or IntelliJ IDEA is that you can write, compile, and run your Scala programs automatically. In this case, there is no need to install SBT, as described in the previous section. While we advise that you run the examples using SBT, you can alternatively use an IDE.

There is an important caveat when running the examples in this book using an IDE: editors such as Eclipse and IntelliJ IDEA run the program inside a separate JVM process. As mentioned in the previous section, certain concurrent computations continue executing after the main execution stops. To make sure that they always complete, you will sometimes need to add the `sleep` statements at the end of the main execution, which slow down the main execution. In most of the examples in this book, the `sleep` statements are already added for you, but in some programs, you might have to add them yourself.

Who this book is for

This book is primarily intended for developers who have learned how to write sequential Scala programs, and wish to learn how to write correct concurrent programs. The book assumes that you have a basic knowledge of the Scala programming language. Throughout this book, we strive to use the simple features of Scala in order to demonstrate how to write concurrent programs. Even with an elementary knowledge of Scala, you should have no problem understanding various concurrency topics.

This is not to say that the book is limited to Scala developers. Whether you have experience with Java, come from a .NET background, or are generally a programming language aficionado, chances are that you will find the content in this book insightful. A basic understanding of object-oriented or functional programming should be a sufficient prerequisite.

Finally, this book is a good introduction to modern concurrent programming in the broader sense. Even if you have the basic knowledge about multithreaded computing, or the JVM concurrency model, you will learn a lot about modern, high-level concurrency utilities. Many of the concurrency libraries in this book are only starting to find their way into mainstream programming languages, and some of them are truly cutting-edge technologies.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The next lines of code read the link and assign it to the to the `BeautifulSoup` function."

A block of code is set as follows:

```
package org
package object learningconcurrency {
  def log(msg: String): Unit =
    println(s"${Thread.currentThread.getName}: $msg")
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
object ThreadsMain extends App {
  val t: Thread = Thread.currentThread
  val name = t.getName
  println(s"I am the thread $name")
}
```

Any command-line input or output is written as follows:

```
$ mkdir scala-concurrency-examples
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "In order to download new modules, we will go to **Files** | **Settings** | **Project Name** | **Project Interpreter**."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Learning-Concurrent-Programming-In-Scala-Second-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/LearningConcurrentProgrammingInScalaSecondEdition_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Introduction

"For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers."

- Gene Amdahl, 1967

Although the discipline of concurrent programming has a long history, it has gained a lot of traction in recent years with the arrival of multi core processors. The recent development in computer hardware not only revived some classical concurrency techniques but also started a major paradigm shift in concurrent programming. At a time when concurrency is becoming so important, an understanding of concurrent programming is an essential skill for every software developer.

This chapter explains the basics of concurrent computing and presents some Scala preliminaries required for this book. Specifically, it does the following:

- Shows a brief overview of concurrent programming
- Studies the advantages of using Scala when it comes to concurrency
- Covers the Scala preliminaries required for reading this book

We will start by examining what concurrent programming is and why it is important.

Concurrent programming

In **concurrent programming**, we express a program as a set of concurrent computations that execute during overlapping time intervals and coordinate in some way. Implementing a concurrent program that functions correctly is usually much harder than implementing a sequential one. All the pitfalls present in sequential programming lurk in every concurrent program, but there are many other things that can go wrong, as we will learn in this book. A natural question arises: why bother? Can't we just keep writing sequential programs?

Concurrent programming has multiple advantages. First, increased concurrency can improve **program performance**. Instead of executing the entire program on a single processor, different subcomputations can be performed on separate processors, making the program run faster. With the spread of multicore processors, this is the primary reason why concurrent programming is nowadays getting so much attention.

A concurrent programming model can result in faster I/O operations. A purely sequential program must periodically poll I/O to check if there is any data input available from the keyboard, the network interface, or some other device. A concurrent program, on the other hand, can react to I/O requests immediately. For I/O-intensive operations, this results in improved throughput, and is one of the reasons why concurrent programming support existed in programming languages even before the appearance of multiprocessors. Thus, concurrency can ensure the improved **responsiveness** of a program that interacts with the environment.

Finally, concurrency can simplify the **implementation** and **maintainability** of computer programs. Some programs can be represented more concisely using concurrency. It can be more convenient to divide the program into smaller, independent computations than to incorporate everything into one large program. User interfaces, web servers, and game engines are typical examples of such systems.

In this book, we adopt the convention that concurrent programs communicate through the use of shared memory, and execute on a single computer. By contrast, a computer program that executes on multiple computers, each with its own memory, is called a **distributed program**, and the discipline of writing such programs is called **distributed programming**. Typically, a distributed program must assume that each of the computers can fail at any point, and provide some safety guarantees if this happens. We will mostly focus on concurrent programs, but we will also look at examples of distributed programs.

A brief overview of traditional concurrency

In a computer system, concurrency can manifest itself in the computer hardware, at the operating system level, or at the programming language level. We will focus mainly on programming-language-level concurrency.

The coordination of multiple executions in a concurrent system is called **synchronization**, and it is a key part in successfully implementing concurrency. Synchronization includes mechanisms used to order concurrent executions in time. Furthermore, synchronization specifies how concurrent executions communicate, that is, how they exchange information. In concurrent programs, different executions interact by modifying the shared memory subsystem of the computer. This type of synchronization is called **shared memory communication**. In distributed programs, executions interact by exchanging messages, so this type of synchronization is called **message-passing communication**.

At the lowest level, concurrent executions are represented by entities called processes and threads, covered in `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*. Processes and threads traditionally use entities such as locks and monitors to order parts of their execution. Establishing an order between the threads ensures that the memory modifications done by one thread are visible to a thread that executes later.

Often, expressing concurrent programs using threads and locks is cumbersome. More complex concurrent facilities have been developed to address this, such as communication channels, concurrent collections, barriers, countdown latches, and thread pools. These facilities are designed to more easily express specific concurrent programming patterns, and some of them are covered in `Chapter 3`, *Traditional Building Blocks of Concurrency*.

Traditional concurrency is relatively low-level and prone to various kinds of errors, such as deadlocks, starvations, data races, and race conditions. You will usually not use low-level concurrency primitives when writing concurrent Scala programs. Still, a basic knowledge of low-level concurrent programming will prove invaluable in understanding high-level concurrency concepts later.

Modern concurrency paradigms

Modern concurrency paradigms are more advanced than traditional approaches to concurrency. Here, the crucial difference lies in the fact that a high-level concurrency framework expresses *which* goal to achieve, rather than *how to achieve* that goal.

In practice, the difference between low-level and high-level concurrency is less clear, and different concurrency frameworks form a continuum rather than two distinct groups. Still, recent developments in concurrent programming show a bias towards declarative and functional programming styles.

As we will see in `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, computing a value concurrently requires creating a thread with a custom `run` method, invoking the `start` method, waiting until the thread completes, and then inspecting specific memory locations to read the result. Here, what we really want to say is *compute some value concurrently, and inform me when you are done*. Furthermore, we would prefer to use a programming model that abstracts over the coordination details of the concurrent computation, to treat the result of the computation as if we already have it, rather than having to wait for it and then reading it from the memory. **Asynchronous programming using futures** is a paradigm designed to specifically support these kinds of statements, as we will learn in `Chapter 4`, *Asynchronous Programming with Futures and Promises*. Similarly, **reactive programming using event streams** aims to declaratively express concurrent computations that produce many values, as we will see in `Chapter 6`, *Concurrent Programming with Reactive Extensions*.

The declarative programming style is increasingly common in sequential programming too. Languages such as Python, Haskell, Ruby, and Scala express operations on their collections in terms of functional operators and allow statements such as *filter all negative integers from this collection*. This statement expresses a goal rather than the underlying implementation, allowing to it easy to parallelize such an operation behind the scene. `Chapter 5`, *Data-Parallel Collections*, describes the **data-parallel** collections framework available in Scala, which is designed to accelerate collection operations using multicores.

Another trend seen in high-level concurrency frameworks is specialization towards specific tasks. Software transactional memory technology is specifically designed to express memory transactions and does not deal with how to start concurrent executions at all. A **memory transaction** is a sequence of memory operations that appear as if they either execute all at once or do not execute at all. This is similar to the concept of database transactions. The advantage of using memory transactions is that this avoids a lot of errors typically associated with low-level concurrency. `Chapter 7`, *Software Transactional Memory*, explains software transactional memory in detail.

Finally, some high-level concurrency frameworks aim to transparently provide distributed programming support as well. This is especially true for data-parallel frameworks and message-passing concurrency frameworks, such as the **actors** described in Chapter 8, *Actors*.

The advantages of Scala

Although Scala is still a language on the rise, it has yet to receive the wide-scale adoption of a language such as Java; nonetheless its support for concurrent programming is rich and powerful. Concurrency frameworks for nearly all the different styles of concurrent programming exist in the Scala ecosystem and are being actively developed. Throughout its development, Scala has pushed the boundaries when it comes to providing modern, high-level application programming interfaces or APIs for concurrent programming. There are many reasons for this.

The primary reason that so many modern concurrency frameworks have found their way into Scala is its inherent syntactic flexibility. Thanks to features such as first-class functions, byname parameters, type inference, and pattern matching explained in the following sections, it is possible to define APIs that look as if they are built-in language features.

Such APIs emulate various programming models as embedded domain-specific languages, with Scala serving as a host language: actors, software transactional memory, and futures are examples of APIs that look like they are basic language features when they are in fact implemented as libraries. On one hand, Scala avoids the need for developing a new language for each new concurrent programming model and serves as a rich nesting ground for modern concurrency frameworks. On the other hand, lifting the syntactic burden present in many other languages attracts more users.

The second reason Scala has pushed ahead lies in the fact that it is a safe language. Automatic garbage collection, automatic bound checks, and the lack of pointer arithmetic helps to avoid problems such as memory leaks, buffer overflows, and other memory errors. Similarly, static type safety eliminates a lot of programming errors at an early stage. When it comes to concurrent programming, which is in itself prone to various kinds of concurrency errors, having one less thing to worry about can make a world of difference.

The third important reason is interoperability. Scala programs are compiled into Java bytecode, so the resulting executable code runs on top of the **Java Virtual Machine** (**JVM**). This means that Scala programs can seamlessly use existing Java libraries, and interact with Java's rich ecosystem. Often, transitioning to a different language is a painful process. In the case of Scala, a transition from a language such as Java can proceed gradually and is much easier. This is one of the reasons for its growing adoption, and also a reason why some Java-compatible frameworks choose Scala as their implementation language.

Importantly, the fact that Scala runs on the JVM implies that Scala programs are portable across a range of different platforms. Not only that, but the JVM has well-defined threading and memory models, which are guaranteed to work in the same way on different computers. While portability is important for the consistent semantics of sequential programs, it is even more important when it comes to concurrent computing.

Having seen some of Scala's advantages for concurrent programming, we are now ready to study the language features relevant for this book.

Preliminaries

This book assumes a basic familiarity with sequential programming. While we advise readers to get acquainted with the Scala programming language, an understanding of a similar language, such as Java or C#, should be sufficient for this book. A basic familiarity with concepts in object-oriented programming, such as classes, objects, and interfaces, is helpful. Similarly, a basic understanding of functional programming principles, such as first-class functions, purity, and type-polymorphism are beneficial in understanding this book but are not a strict prerequisite.

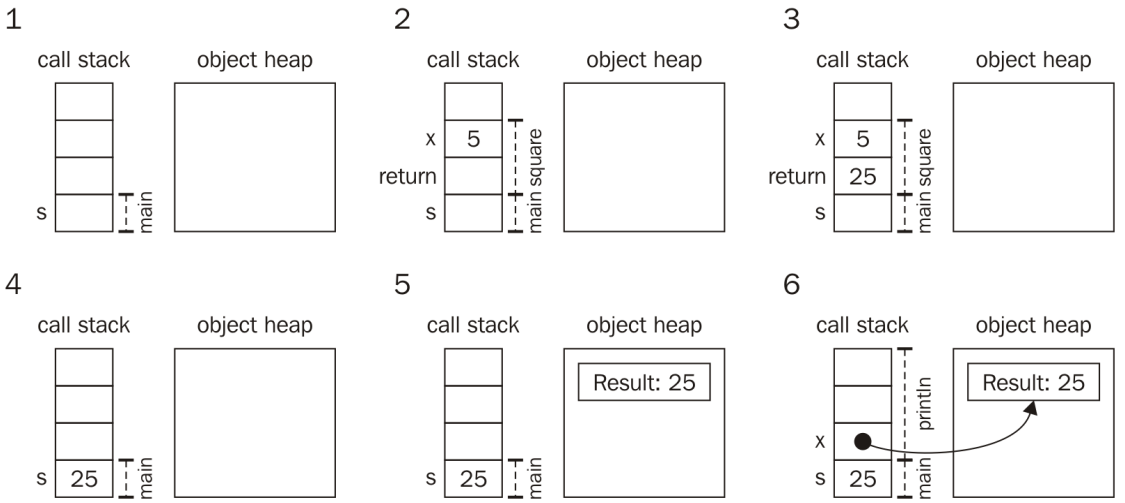
Execution of a Scala program

To better understand the execution model of Scala programs, let's consider a simple program that uses the `square` method to compute the square value of the number `5`, and then prints the result to the standard output:

```
object SquareOf5 extends App {
  def square(x: Int): Int = x * x
  val s = square(5)
  println(s"Result: $s")
}
```

We can run this program using the **Simple Build Tool** (**SBT**), as described in the *Preface*. When a Scala program runs, the JVM runtime allocates the memory required for the program. Here, we consider two important memory regions–the **call stack** and the **object heap**. The call stack is a region of memory in which the program stores information about the local variables and parameters of the currently executed methods. The object heap is a region of memory in which objects are allocated by the program. To understand the difference between the two regions, we consider a simplified scenario of this program's execution.

First, in figure **1**, the program allocates an entry to the call stack for the local variable `s`. Then, it calls the `square` method in figure **2** to compute the value for the local variable `s`. The program places the value `5` on the call stack, which serves as the value for the `x` parameter. It also reserves a stack entry for the return value of the method. At this point, the program can execute the `square` method, so it multiplies the `x` parameter by itself, and places the return value `25` on the stack in figure **3**. This is shown in the first row in the following illustration:



After the `square` method returns the result, the result `25` is copied into the stack entry for the local variable `s`, as shown in figure **4**. Now, the program must create the string for the `println` statement. In Scala, strings are represented as object instances of the `String` class, so the program allocates a new `String` object to the object heap, as illustrated in figure **5**. Finally, in figure **6**, the program stores the reference to the newly allocated object into the stack entry `x`, and calls the `println` method.

Although this demonstration is greatly simplified, it shows the basic execution model for Scala programs. In `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, we will learn that each thread of execution maintains a separate call stack, and that threads mainly communicate by modifying the object heap. We will learn that the disparity between the state of the heap and the local call stack is frequently responsible for certain kinds of error in concurrent programs.

Having seen an example of how Scala programs are typically executed, we now proceed to an overview of Scala features that are essential to understand the contents of this book.

A Scala primer

In this section, we present a short overview of the Scala programming language features that are used in the examples in this book. This is a quick and cursory glance through the basics of Scala. Note that this section is not meant to be a complete introduction to Scala. This is to remind you about some of the language's features, and contrast them with similar languages that might be familiar to you. If you would like to learn more about Scala, refer to some of the books referred to in the *Summary* of this chapter.

A `Printer` class, which takes a `greeting` parameter and has two methods named `printMessage` and `printNumber`, is declared as follows:

```
class Printer(val greeting: String) {
  def printMessage(): Unit = println(greeting + "!")
  def printNumber(x: Int): Unit = {
    println("Number: " + x)
  }
}
```

In the preceding code, the `printMessage` method does not take any arguments and contains a single `println` statement. The `printNumber` method takes a single argument `x` of the `Int` type. Neither method returns a value, which is denoted by the `Unit` type.

We instantiate the class and call its methods as follows:

```
val printy = new Printer("Hi")
printy.printMessage()
printy.printNumber(5)
```

Scala allows the declaration of **singleton objects**. This is like declaring a class and instantiating its single instance at the same time. We saw the `SquareOf5` singleton object earlier, which was used to declare a simple Scala program. The following singleton object, named `Test`, declares a single `Pi` field and initializes it with the value `3.14`:

```
object Test {
  val Pi = 3.14
}
```

While classes in similar languages extend entities that are called interfaces, Scala classes can extend **traits**. Scala's traits allow declaring both concrete fields and method implementations. In the following example, we declare the `Logging` trait, which outputs a custom error and warning messages using the abstract `log` method, and then mix the trait into the `PrintLogging` class:

```
trait Logging {
  def log(s: String): Unit
  def warn(s: String) = log("WARN: " + s)
  def error(s: String) = log("ERROR: " + s)
}
class PrintLogging extends Logging {
  def log(s: String) = println(s)
}
```

Classes can have **type parameters**. The following generic `Pair` class takes two type parameters, `P` and `Q`, which determines the types of its arguments, named `first` and `second`:

```
class Pair[P, Q](val first: P, val second: Q)
```

Scala has support for first-class function objects, also called **lambdas**. In the following code snippet, we declare a `twice` lambda, which multiplies its argument by two:

```
val twice: Int => Int = (x: Int) => x * 2
```

Downloading the example code:
You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

In the preceding code, the `(x: Int)` part is the argument to the lambda, and `x * 2` is its body. The `=>` symbol must be placed between the arguments and the body of the lambda. The same `=>` symbol is also used to express the type of the lambda, which is `Int => Int`, pronounced as `Int` to `Int`. In the preceding example, we can omit the type annotation `Int => Int`, and the compiler will infer the type of the `twice` lambda automatically, as shown in the following code:

```
val twice = (x: Int) => x * 2
```

Alternatively, we can omit the type annotation in the lambda declaration and arrive at a more convenient syntax, as follows:

```
val twice: Int => Int = x => x * 2
```

Finally, whenever the argument to the lambda appears only once in the body of the lambda, Scala allows a more convenient syntax, as follows:

```
val twice: Int => Int = _ * 2
```

First-class functions allow manipulating blocks of code as if they were first-class values. They allow a more lightweight and concise syntax. In the following example, we use **byname parameters** to declare a `runTwice` method, which runs the specified block of code `body` twice:

```
def runTwice(body: =>Unit) = {
  body
  body
}
```

A byname parameter is formed by putting the `=>` annotation before the type. Whenever the `runTwice` method references the `body` argument, the expression is re-evaluated, as shown in the following snippet:

```
runTwice { // this will print Hello twice
  println("Hello")
}
```

Scala `for` expressions are a convenient way to traverse and transform collections. The following `for` loop prints the numbers in the range from `0 until 10`; where `10` is not included in the range:

```
for (i <- 0 until 10) println(i)
```

In the preceding code, the range is created with the expression `0 until 10`; this is equivalent to the expression `0.until(10)`, which calls the method `until` on the value `0`. In Scala, the dot notation can sometimes be dropped when invoking methods on objects.

Every `for` loop is equivalent to a `foreach` call. The preceding `for` loop is translated by the Scala compiler to the following expression:

```
(0 until 10).foreach(i => println(i))
```

For-comprehensions are used to transform data. The following for-comprehension transforms all the numbers from `0 until 10` by multiplying them by `-1`:

```
val negatives = for (i <- 0 until 10) yield -i
```

The `negatives` value contains negative numbers from `0` until `-10`. This for-comprehension is equivalent to the following `map` call:

```
val negatives = (0 until 10).map(i => -1 * i)
```

It is also possible to transform data from multiple inputs. The following for-comprehension creates all pairs of integers between `0` and `4`:

```
val pairs = for (x <- 0 until 4; y <- 0 until 4) yield (x, y)
```

The preceding for-comprehension is equivalent to the following expression:

```
val pairs = (0 until 4).flatMap(x => (0 until 4).map(y => (x, y)))
```

We can nest an arbitrary number of generator expressions in a for-comprehension. The Scala compiler will transform them into a sequence of nested `flatMap` calls, followed by a `map` call at the deepest level.

Commonly used Scala collections include sequences, denoted by the `Seq[T]` type; maps, denoted by the `Map[K, V]` type; and sets, denoted by the `Set[T]` type. In the following code, we create a sequence of strings:

```
val messages: Seq[String] = Seq("Hello", "World.", "!")
```

Throughout this book, we rely heavily on the **string interpolation** feature. Normally, Scala strings are formed with double quotation marks. Interpolated strings are preceded with an `s` character, and can contain `$` symbols with arbitrary identifiers resolved from the enclosing scope, as shown in the following example:

```
val magic = 7
val myMagicNumber = s"My magic number is $magic"
```

Pattern matching is another important Scala feature. For readers with Java, C#, or C background, a good way to describe it is to say that Scala's `match` statement is like the `switch` statement on steroids. The `match` statement can decompose arbitrary datatypes and allows you to express different cases in the program concisely.

In the following example, we declare a `Map` collection, named `successors`, used to map integers to their immediate successors. We then call the `get` method to obtain the successor of the number 5. The `get` method returns an object with the `Option[Int]` type, which may be implemented either with the `Some` class, indicating that the number 5 exists in the map, or the `None` class, indicating that the number 5 is not a key in the map. Pattern matching on the `Option` object allows proceeding casewise, as shown in the following code snippet:

```
val successors = Map(1 -> 2, 2 -> 3, 3 -> 4)
successors.get(5) match {
  case Some(n) => println(s"Successor is: $n")
  case None    => println("Could not find successor.")
}
```

In Scala, most operators can be overloaded. **Operator overloading** is no different from declaring a method. In the following code snippet, we declare a `Position` class with a `+` operator:

```
class Position(val x: Int, val y: Int) {
  def +(that: Position) = new Position(x + that.x, y + that.y)
}
```

Finally, Scala allows defining **package objects** to store top-level method and value definitions for a given package. In the following code snippet, we declare the package object for the `org.learningconcurrency` package. We implement the top level `log` method, which outputs a given string and the current thread name:

```
package org
package object learningconcurrency {
  def log(msg: String): Unit =
    println(s"${Thread.currentThread.getName}: $msg")
}
```

We will use the `log` method in the examples throughout this book to trace how concurrent programs are executed.

This concludes our quick overview of important Scala features. If you would like to obtain a deeper knowledge about any of these language constructs, we suggest that you check out one of the introductory books on sequential programming in Scala.

Overview of new features in Scala 2.12

At the time of writing, the next planned release of the language is Scala 2.12. From the user and API perspective, Scala 2.12 does not introduce new ground-breaking features. The goal of the 2.12 release is to improve code optimization and make Scala compliant with the Java 8 runtime. Since Scala's primary target is the Java runtime, making Scala compliant with Java 8 runtime will reduce the size of compiled programs and JAR files, better performance and faster compilation. From the user perspective, the major change is that you will have to install the JDK 8 framework instead of JDK 7.

The particular changes in Scala 2.12 worth mentioning are the following:

- In previous versions, traits compiled to a single interface if all of their methods were abstract. If the trait had a concrete method implementation, the compiler generated two class files–one containing the JVM interface, and another class file containing the implementations of the concrete methods. In Scala 2.12, the compiler will generate a single interface file containing the Java 8 **default methods**. The net effect is reduced code size.
- Previously, each Scala closure was compiled into a separate class. Starting with 2.12, Scala closures are compiled into Java 8-style lambdas. The consequence is reduced code size and potentially better optimizations by the Java 8 runtime.
- Scala compiles into Java bytecodes, which are then interpreted on the Java Virtual Machine. In Scala 2.12, the old compiler backend is replaced with a new implementation that generates bytecode more quickly with a positive impact on compilation speed.
- Scala 2.12 comes with a new optimizer, which is enabled with the `-opt` compiler flag. The new optimizer is more aggressive at inlining final methods, does better escape analysis for objects and functions that are created and used in a single method, and does dead code elimination. All this has a positive impact on the performance of Scala programs.
- Scala 2.12 allows using lambdas for Single Abstract Method (SAM) types. SAM types are classes or traits that have exactly one abstract method, which is normally implemented by extending the class. Assume that we have a method invocation with an argument whose expected type is a SAM type. If the user passes a lambda, that is, a function literal, instead of a SAM type instance, the 2.12 compiler will automatically convert the function object into an instance of the SAM type.

Summary

In this chapter, we studied what concurrent programming is and why Scala is a good language for concurrency. We gave a brief overview of what you will learn in this book, and how the book is organized. Finally, we stated some Scala preliminaries necessary for understanding the various concurrency topics in the subsequent chapters. If you would like to learn more about sequential Scala programming, we suggest that you read the book, *Programming in Scala, Martin Odersky, Lex Spoon, and Bill Venners, Artima Inc.*

In the next chapter, we will start with the fundamentals of concurrent programming on the JVM. We will introduce the basic concepts in concurrent programming, present the low-level concurrency utilities available on the JVM, and learn about the Java Memory Model.

Exercises

The following exercises are designed to test your knowledge of the Scala programming language. They cover the content presented in this chapter, along with some additional Scala features. The last two exercises contrast the difference between concurrent and distributed programming, as defined in this chapter. You should solve them by sketching out a pseudocode solution, rather than a complete Scala program.

1. Implement a `compose` method with the following signature:

```
def compose[A, B, C]
(g: B => C, f: A => B): A => C = ???
```

   This method must return a function `h`, which is the composition of the functions `f` and `g`

2. Implement a `fuse` method with the following signature:

```
def fuse[A, B]
(a: Option[A], b: Option[B]): Option[(A, B)] = ???
```

   The resulting `Option` object should contain a tuple of values from the `Option` objects `a` and `b`, given that both `a` and `b` are non-empty. Use for-comprehensions

3. Implement a `check` method, which takes a set of values of type `T` and a function of type `T => Boolean`:

```
def check[T](xs: Seq[T])(pred: T => Boolean): Boolean = ???
```

   The method must return `true` if and only if the `pred` function returns `true` for all the values in `xs` without throwing an exception. Use the `check` method as follows:

```
check(0 until 10)(40 / _ > 0)
```

The `check` method has a curried definition: instead of just one parameter list, it has two of them. Curried definitions allow a nicer syntax when calling the function, but are otherwise semantically equivalent to single-parameter list definitions.

4. Modify the `Pair` class from this chapter so that it can be used in a pattern match.

If you haven't already done so, familiarize yourself with pattern matching in Scala.

5. Implement a `permutations` function, which, given a string, returns a sequence of strings that are lexicographic permutations of the input string:

```
def permutations(x: String): Seq[String]
```

6. Implement a `combinations` function that, given a sequence of elements, produces an iterator over all possible combinations of length `n`. A combination is a way of selecting elements from the collection so that every element is selected once, and the order of elements does not matter. For example, given a collection `Seq(1, 4, 9, 16)`, combinations of length 2 are `Seq(1, 4)`, `Seq(1, 9)`, `Seq(1, 16)`, `Seq(4, 9)`, `Seq(4, 16)`, and `Seq(9, 16)`. The combinations function has the following signature:

```
def combinations(n: Int, xs: Seq[Int]): Iterator[Seq[Int]]
```

   See the `Iterator` API in the standard library documentation

7. Implement a method that takes a regular expression, and returns a partial function from a string to lists of matches within that string:

```
def matcher   (regex: String): PartialFunction[String,
List[String]]
```

   The partial function should not be defined if there are no matches within the argument strings. Otherwise, it should use the regular expression to output the list of matches.

8. Consider that you and three of your colleagues working in an office divided into cubicles. You cannot see each other, and you are not allowed to verbally communicate, as that might disturb other workers. Instead, you can throw pieces of paper with short messages at each other. Since you are confined in a cubicle, neither of you can tell if the message has reached its destination. At any point, you or one of your colleagues may be called to the boss's office and kept there indefinitely. Design an algorithm in which you and your colleagues can decide when to meet at the local bar. With the exception of the one among you who was called to the boss's office, all of you have to decide on the same time. What if some of the paper pieces can arbitrarily miss the target cubicle?
9. Imagine that, in the previous exercise, you and your colleagues also have a whiteboard in the hall next to the office. Each one of you can occasionally pass through the hall and write something on the whiteboard, but there is no guarantee that either of you will be in the hall at the same time.

   Solve the problem from the previous exercise, this time using the whiteboard.

2

Concurrency on the JVM and the Java Memory Model

"All non-trivial abstractions, to some degree, are leaky."

-Jeff Atwood

Since its inception, Scala has run primarily on top of JVM, and this fact has driven the design of many of its concurrency libraries. The memory model in Scala, its multithreading capabilities, and its inter-thread synchronization are all inherited from the JVM. Most, if not all, higher-level Scala concurrency constructs are implemented in terms of the low-level primitives presented in this chapter. These primitives are the basic way to deal with concurrency-in a way, the APIs and synchronization primitives in this chapter constitute the assembly of concurrent programming on the JVM.

In most cases, you should avoid low-level concurrency in place of higher-level constructs introduced later, but we felt it was important for you to understand what a thread is, that a guarded block is better than busy-waiting, or why a memory model is useful. We are convinced that this is essential for a better understanding of high-level concurrency abstractions. Despite the popular notion that an abstraction that requires knowledge about its implementation is broken, understanding the basics often proves very handy- in practice, all abstractions are to some extent leaky.

In what follows, we not only explain the cornerstones of concurrency on JVM, but also discuss how they interact with some Scala-specific features. In particular, we will cover the following topics in this chapter:

- Creating and starting threads and waiting for their completion
- Communication between threads using object monitors and the `synchronized` statement
- How to avoid busy-waiting using guarded blocks
- The semantics of volatile variables
- The specifics of the **Java Memory Model** (**JMM**), and why the JMM is important

In the following section, we will study how to use threads–the basic way to express concurrent computations.

Processes and threads

In modern, pre-emptive, multitasking operating systems, the programmer has little or no control over the choice of processor on which the program will be executed. In fact, the same program might run on many different processors during its execution and sometimes even simultaneously on several processors. It is usually the task of the **Operating System** (**OS**) to assign executable parts of the program to specific processors–this mechanism is called **multitasking**, and it happens transparently for the computer user.

Historically, multitasking was introduced to operating systems to improve the user experience by allowing multiple users or programs to use resources of the same computer simultaneously. In cooperative multitasking, programs were able to decide when to stop using the processor and yield control to other programs. However, this required a lot of discipline on the programmer's part and programs could easily give the impression of being unresponsive. For example, a download manager that starts downloading a file must take care in order to yield control to other programs. Blocking the execution until a download finishes will completely ruin the user experience. Most operating systems today rely on pre-emptive multitasking, in which each program is repetitively assigned slices of execution time at a specific processor. These slices are called **time slices**. Thus, multitasking happens transparently for the application programmer as well as the user.

The same computer program can be started more than once, or even simultaneously within the same OS. A **process** is an instance of a computer program that is being executed. When a process starts, the OS reserves a part of the memory and other computational resources and associates them with a specific computer program. The OS then associates the processor with the process, and the process executes during one-time slice. Eventually, the OS gives other processes control over the processor. Importantly, the memory and other computational resources of one process are isolated from the other processes: two processes cannot read each other's memory directly or simultaneously use most of the resources.

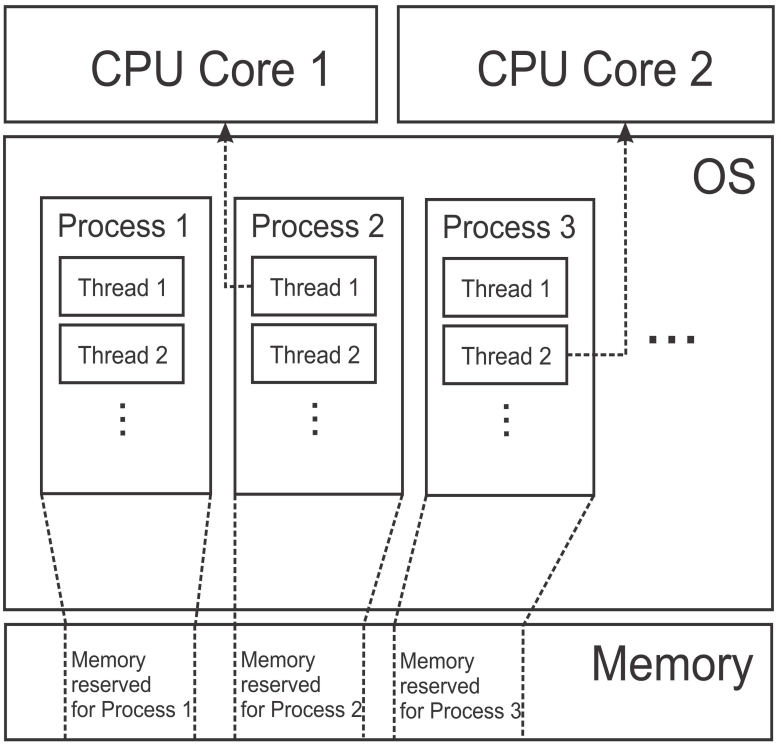
Most programs are comprised of a single process, but some programs run in multiple processes. In this case, different tasks within the program are expressed as separate processes. Since separate processes cannot access the same memory areas directly, it can be cumbersome to express multitasking using multiple processes.

Multitasking was important long before recent years when multicore computers became mainstream. Large programs such as web browsers are divided into many logical modules. A browser's download manager downloads files independent of rendering the web page or updating the HTML **Document Object Model** (**DOM**). While the user is browsing a social networking website, the file download proceeds in the background; but both independent computations occur as part of the same process. These independent computations occurring in the same process are called **threads**. In a typical operating system, there are many more threads than processors.

Every thread describes the current state of the program **stack** and the program **counter** during program execution. The program stack contains a sequence of method invocations that are currently being executed, along with the local variables and method parameters of each method. The program counter describes the position of the current instruction in the current method. A processor can advance the computation in some thread by manipulating the state of its stack or the state of the program objects and executing the instruction at the current program counter. When we say that a thread performs an action such as writing to a memory location, we mean that the processor executing that thread performs that action. In pre-emptive multitasking, thread execution is scheduled by the operating system. A programmer must assume that the processor time assigned to their thread is unbiased towards other threads in the system.

OS threads are a programming facility provided by the OS, usually exposed through an OS-specific programming interface. Unlike separate processes, separate OS threads within the same process share a region of memory, and communicate by writing to and reading parts of that memory. Another way to define a process is to define it as a set of OS threads along with the memory and resources shared by these threads.

Based on the preceding discussion about the relationships between processes and threads, a summary of a typical OS is depicted in the following simplified diagram:



The preceding diagram shows an OS in which multiple processes are executing simultaneously. Only the first three processes are shown in the illustration. Each process is assigned a fixed region of computer memory. In practice, the memory system of the OS is much more complex, but this approximation serves as a simple mental model.

Each of the processes contains multiple OS threads, two of which are shown for each process. Currently, **Thread 1** of **Process 2** is executing on **CPU Core 1**, and **Thread 2** of **Process 3** is executing on **CPU Core 2**. The OS periodically assigns different OS threads to each of the CPU cores to allow the computation to progress in all the processes.

Having shown the relationship between the OS threads and processes, we turn our attention to see how these concepts relate to the **Java Virtual Machine** (**JVM**), the runtime on top of which Scala programs execute.

Starting a new JVM instance always creates only one process. Within the JVM process, multiple threads can run simultaneously. The JVM represents its threads with the `java.lang.Thread` class. Unlike runtimes for languages such as Python, the JVM does not implement its custom threads. Instead, each Java thread is directly mapped to an OS thread. This means that Java threads behave in a very similar way to the OS threads, and the JVM depends on the OS and its restrictions.

Scala is a programming language that is by default compiled to the JVM bytecode, and the Scala compiler output is largely equivalent to that of Java from the JVM's perspective. This allows Scala programs to transparently call Java libraries, and in some cases, even vice versa. Scala reuses the threading API from Java for several reasons. First, Scala can transparently interact with the existing Java thread model, which is already sufficiently comprehensive. Second, it is useful to retain the same threading API for compatibility reasons, and there is nothing fundamentally new that Scala can introduce with respect to the Java thread API.

The rest of this chapter shows how to create JVM threads using Scala, how they can be executed, and how they can communicate. We will show and discuss several concrete examples. Java aficionados, already well-versed in this subject, might choose to skip the rest of this chapter.

Creating and starting threads

Every time a new JVM process starts, it creates several threads by default. The most important thread among them is the **main thread**, which executes the `main` method of the Scala program. We will show this in the following program, which gets the name of the current thread and prints it to the standard output:

```
object ThreadsMain extends App {
  val t: Thread = Thread.currentThread
  val name = t.getName
  println(s"I am the thread $name")
}
```

On the JVM, thread objects are represented with the `Thread` class. The preceding program uses the static `currentThread` method to obtain a reference to the current thread object, and stores it to a local variable named `t`. It then calls the `getName` method to obtain the thread's name. If you are running this program from **Simple Build Tool** (SBT) with the `run` command, as explained in `Chapter 1`, *Introduction*, you should see the following output:

```
[info] I am the thread run-main-0
```

Normally, the name of the main thread is just the `main` method. The reason we see a different name is because SBT started our program on a separate thread inside the SBT process. To ensure that the program runs inside a separate JVM process, we need to set SBT's `fork` setting to `true`:

```
> set fork := true
```

Invoking the SBT `run` command again should give the following output:

```
[info] I am the thread main
```

Every thread goes through several **thread states** during its existence. When a `Thread` object is created, it is initially in the **new** state. After the newly created thread object starts executing, it goes into the **runnable** state. After the thread is done executing, the thread object goes into the **terminated state**, and cannot execute anymore.

Starting an independent thread of computation consists of two steps. First, we need to create a `Thread` object to allocate the memory for the stack and thread state. To start the computation, we need to call the `start` method on this object. We show how to do this in the following example application called `ThreadsCreation`:

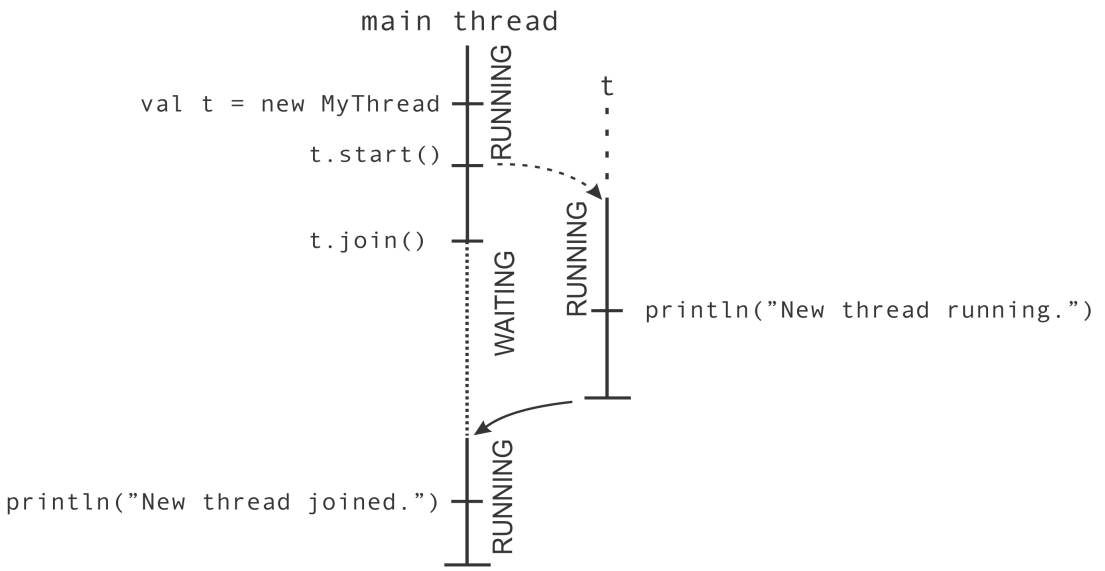
```
object ThreadsCreation extends App {
  class MyThread extends Thread {
    override def run(): Unit = {
      println("New thread running.")
    }
  }
  val t = new MyThread
  t.start()
  t.join()
  println("New thread joined.")
}
```

When a JVM application starts, it creates a special thread called the **main thread** that executes the method called `main` in the specified class, in this case, the `ThreadsCreation` object. When the `App` class is extended, the `main` method is automatically synthesized from the object body. In this example, the main thread first creates another thread of the `MyThread` type and assigns it to `t`.

Next, the main thread starts `t` by calling the `start` method. Calling the `start` method eventually results in executing the `run` method from the new thread. First, the OS is notified that `t` must start executing. When the OS decides to assign the new thread to some processor, this is largely out of the programmer's control, but the OS must ensure that this eventually happens. After the main thread starts the new thread `t`, it calls its `join` method. This method halts the execution of the main thread until `t` completes its execution. We say that the `join` operation puts the main thread into the **waiting state** until `t` terminates. Importantly, the waiting thread relinquishes its control over the processor, and the OS can assign that processor to some other thread.

Waiting threads notify the OS that they are waiting for some condition and cease spending CPU cycles, instead of repetitively checking that condition.

In the meantime, the OS finds an available processor and instructs it to run the child thread. The instructions that a thread must execute are specified by overriding its `run` method. The `t` instance of the `MyThread` class starts by printing the "`New thread running.`" text to the standard output and then terminates. At this point, the operating system is notified that `t` is terminated and eventually lets the main thread continue the execution. The OS then puts the main thread back into the running state, and the main thread prints "`New thread joined.`". This is shown in the following diagram:



It is important to note that the two outputs `"New thread running."` and `"New thread joined."` are always printed in this order. This is because the `join` call ensures that the termination of the `t` thread occurs before the instructions following the `join` call.

When running the program, it is executed so fast that the two `println` statements occur almost simultaneously. Could it be that the ordering of the `println` statements is just an artifact in how the OS chooses to execute these threads? To verify the hypothesis that the main thread really waits for `t` and that the output is not just because the OS is biased to execute `t` first in this particular example, we can experiment by tweaking the execution schedule. Before we do that, we will introduce a shorthand to create and start a new thread; the current syntax is too verbose! The new `thread` method simply runs a block of code in a newly started thread. This time, we will create the new thread using an anonymous thread class declared inline at the instantiation site:

```
def thread(body: =>Unit): Thread = {
  val t = new Thread {
    override def run() = body
  }
  t.start()
  t
}
```

The `thread` method takes a block of code body, creates a new thread that executes this block of code in its `run` method, starts the thread, and returns a reference to the new thread so that the clients can call `join` on it.

Creating and starting threads using the `thread` statement is much less verbose. To make the examples in this chapter more concise, we will use the `thread` statement from now on. However, you should think twice before using the `thread` statement in production projects. It is prudent to correlate the syntactic burden with the computational cost; lightweight syntax can be mistaken for a cheap operation and creating a new thread is relatively expensive.

We can now experiment with the OS by making sure that all the processors are available. To do this, we will use the static `sleep` method on the `Thread` class, which postpones the execution of the thread that is being currently executed for the specified number of milliseconds. This method puts the thread into the **timed waiting** state. The OS can reuse the processor for other threads when `sleep` is called. Still, we will require a sleep time much larger than the time slice on a typical OS, which ranges from 10 to 100 milliseconds. The following code depicts this:

```
object ThreadsSleep extends App {
  val t = thread {
    Thread.sleep(1000)
    log("New thread running.")
    Thread.sleep(1000)
    log("Still running.")
    Thread.sleep(1000)
    log("Completed.")
  }
  t.join()
  log("New thread joined.")
}
```

The main thread of the `ThreadSleep` application creates and starts a new `t` thread that sleeps for one second, then outputs some text, and repeats this two or more times before terminating. The main thread calls `join` as before and then prints `"New thread joined."`.

Note that we are now using the `log` method described in `Chapter 1`, *Introduction*. The `log` method prints the specified string value along with the name of the thread that calls the `log` method.

Regardless of how many times you run the preceding application, the last output will always be `"New thread joined."`. This program is **deterministic**: given a particular input, it will always produce the same output, regardless of the execution schedule chosen by the OS.

However, not all the applications using threads will always yield the same output if given the same input. The following code is an example of a **nondeterministic** application:

```
object ThreadsNondeterminism extends App {
  val t = thread { log("New thread running.") }
  log("...")
  log("...")
  t.join()
  log("New thread joined.")
}
```

There is no guarantee that the `log("...")` statements in the main thread occur before or after the `log` call in the `t` thread. Running the application several times on a multicore processor prints `"..."` before, after, or interleaved with the output by the `t` thread. By running the program, we get the following output:

```
run-main-46: ...
Thread-80: New thread running.
run-main-46: ...
run-main-46: New thread joined.
```

Running the program again results in a different order between these outputs:

```
Thread-81: New thread running.
run-main-47: ...
run-main-47: ...
run-main-47: New thread joined.
```

Most multithreaded programs are nondeterministic, and this is what makes multithreaded programming so hard. There are multiple possible reasons for this. First, the program might be too big for the programmer to reason about its determinism properties, and interactions between threads could simply be too complex. But some programs are inherently non-deterministic. A web server has no idea which client will be the first to send a request for a web page. It must allow these requests to arrive in any possible order and respond to them as soon as they arrive. Depending on the order in which the clients prepare inputs for the web server, they can behave differently even though the requests might be the same.

Atomic execution

We have already seen one basic way in which threads can communicate: by waiting for each other to terminate. The information that the joined thread delivers is that it has terminated. In practice, however, this information is not necessarily useful; for example, a thread that renders one page in a web browser must inform the other threads that a specific URL has been visited, so as to render such a visited URL in a different color.

It turns out that the `join` method on threads has an additional property. All the writes to memory performed by the thread being joined occur before the `join` call returns and are visible to threads that call the `join` method. This is illustrated by the following example:

```
object ThreadsCommunicate extends App {
  var result: String = null
  val t = thread { result = "\nTitle\n" + "=" * 5 }
  t.join()
  log(result)
}
```

The main thread will never print `null`, as the call to `join` always occurs before the `log` call, and the assignment to `result` occurs before the termination of `t`. This pattern is a very basic way in which the threads can use their results to communicate with each other.

However, this pattern only allows very restricted one-way communication, and it does not allow threads to mutually communicate during their execution. There are many use cases for an unrestricted two-way communication. One example is assigning unique identifiers, in which a set of threads concurrently choose numbers such that no two threads produce the same number. We might be tempted to proceed as in the following incorrect example. We start by showing the first half of the program:

```
object ThreadsUnprotectedUid extends App {
  var uidCount = 0L
  def getUniqueId() = {
    val freshUid = uidCount + 1
    uidCount = freshUid
    freshUid
  }
```

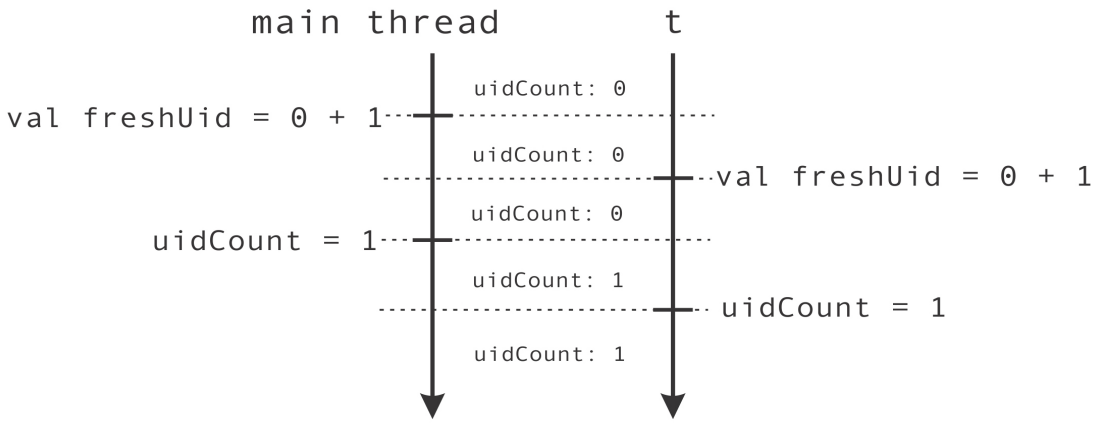
In the preceding code snippet, we first declare a `uidCount` variable that will hold the last unique identifier picked by any thread. The threads will call the `getUniqueId` method to compute the first unused identifier and then update the `uidCount` variable. In this example, reading `uidCount` to initialize `freshUid` and assigning `freshUid` back to `uniqueUid` do not necessarily happen together. We say that the two statements do not happen **atomically** since the statements from the other threads can interleave arbitrarily. We next define a `printUniqueIds` method such that, given a number `n`, the method calls `getUniqueId` to produce `n` unique identifiers and then prints them. We use Scala for-comprehensions to map the range `0 until n` to unique identifiers. Finally, the main thread starts a new `t` thread that calls the `printUniqueIds` method, and then calls `printUniqueIds` concurrently with the `t` thread as follows:

```
    def printUniqueIds(n: Int): Unit = {
      val uids = for (i<- 0 until n) yield getUniqueId()
      log(s"Generated uids: $uids")
    }
    val t = thread { printUniqueIds(5) }
    printUniqueIds(5)
    t.join()
  }
```

Running this application several times reveals that the identifiers generated by the two threads are not necessarily unique; the application prints `Vector(1, 2, 3, 4, 5)` and `Vector(1, 6, 7, 8, 9)` in some runs, but not in the others! The outputs of the program depend on the timing at which the statements in separate threads get executed.

A **race condition** is a phenomenon in which the output of a concurrent program depends on the execution schedule of the statements in the program.

A race condition is not necessarily an incorrect program behavior. However, if some execution schedule causes an undesired program output, the race condition is considered to be a program error. The race condition from the previous example is a program error, because the `getUniqueId` method is not atomic. The `t` thread and the main thread sometimes concurrently calls `getUniqueId`. In the first line, they concurrently read the value of `uidCount`, which is initially `0`, and conclude that their own `freshUid` variable should be `1`. The `freshUid` variable is a local variable, so it is allocated on the thread stack; each thread sees a separate instance of that variable. At this point, the threads decide to write the value `1` back to `uidCount` in any order, and both return a non-unique identifier `1`. This is illustrated in the following figure:

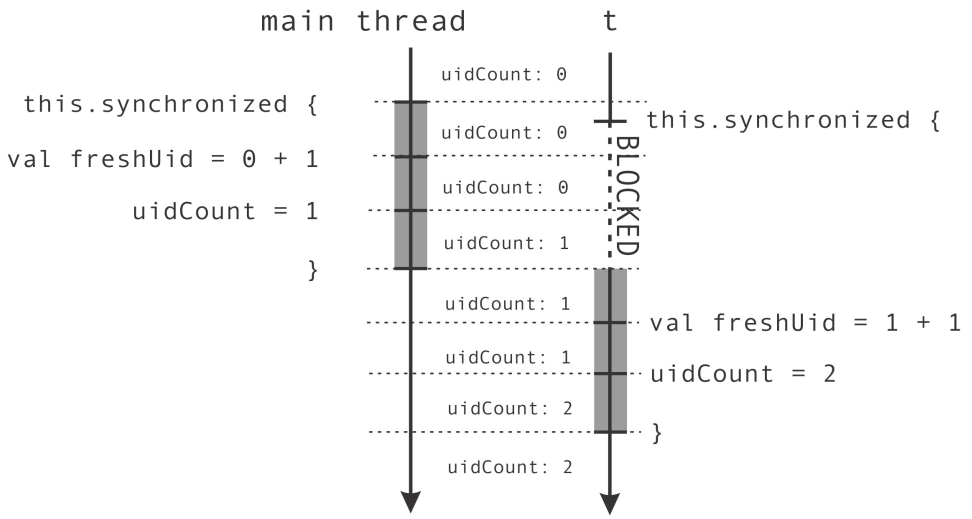


There is a mismatch between the mental model that most programmers inherit from sequential programming and the execution of the `getUniqueId` method when it is run concurrently. This mismatch is grounded in the assumption that `getUniqueId` executes atomically. Atomic execution of a block of code means that the individual statements in that block of code executed by one thread cannot interleave with those statements executed by another thread. In atomic execution, the statements can only be executed all at once, which is exactly how the `uidCount` field should be updated. The code inside the `getUniqueId` function reads, modifies, and writes a value, which is not atomic on the JVM. An additional language construct is necessary to guarantee atomicity. The fundamental Scala construct that allows this sort of atomic execution is called the `synchronized` statement, and it can be called on any object. This allows us to define `getUniqueId` as follows:

```
def getUniqueId() = this.synchronized {
  val freshUid = uidCount + 1
  uidCount = freshUid
  freshUid
}
```

The `synchronized` call ensures that the subsequent block of code can only execute if there is no other thread simultaneously executing this synchronized block of code, or any other synchronized block of code called on the same `this` object. In our case, the `this` object is the enclosing singleton object, `ThreadsUnprotectedUid`, but in general, this can be an instance of the enclosing class or trait.

Two concurrent invocations of the `getUniqueId` method are shown in the following figure:



We can also call `synchronized` and omit the `this` part, in which case the compiler will infer what the surrounding object is, but we strongly discourage you from doing so. Synchronizing on incorrect objects results in concurrency errors that are not easily identified.

Always explicitly declare the receiver for the `synchronized` statement doing so protects you from subtle and hard to spot program errors.

The JVM ensures that the thread executing a `synchronized` statement invoked on some `x` object is the only thread executing any `synchronized` statement on that particular `x` object. If a `T` thread calls `synchronized` on `x`, and there is another `S` thread calling `synchronized` on `x`, then the `T` thread is put into the **blocked** state. Once the `S` thread completes its `synchronized` statement, the JVM can choose the `T` thread to execute its own `synchronized` statement.

Every object created inside the JVM has a special entity called an **intrinsic lock** or a **monitor**, which is used to ensure that only one thread is executing some `synchronized` block on that object. When a thread starts executing the `synchronized` block, we say that the thread **gains ownership** of the `x` monitor, or alternatively, **acquires** it. When a thread completes the `synchronized` block, we say that it **releases** the monitor.

The `synchronized` statement is one of the fundamental mechanisms for inter-thread communication in Scala and on the JVM. Whenever there is a possibility that multiple threads access and modify a field in some object, you should use the `synchronized` statement.

Reordering

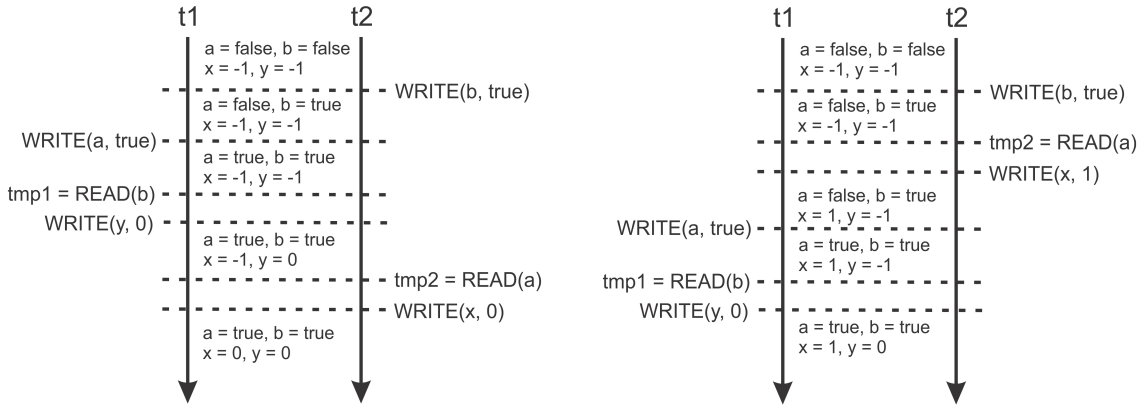
The `synchronized` statement is not without a price: writes to fields such as `uidCount`, which are protected by the `synchronized` statement are usually more expensive than regular unprotected writes. The performance penalty of the `synchronized` statement depends on the JVM implementation, but it is usually not large. You might be tempted to avoid using `synchronized` when you think that there is no bad interleaving of program statements, like the one we saw previously in the unique identifier example. Never do this! We will now show you a minimal example in which this leads to serious errors.

Let's consider the following program, in which two threads, `t1` and `t2`, access a pair of Boolean variables, `a` and `b`, and a pair of Integer variables, `x` and `y`. The `t1` thread sets the variable `a` to `true`, and then reads the value of `b`. If the value of `b` is `true`, the `t1` thread assigns `0` to `y`, and otherwise it assigns `1`. The `t2` thread does the opposite: it first assigns `true` to the variable `b`, and then assigns `0` to `x` if `a` is `true`, and `1` otherwise. This is repeated in a loop `100000` times, as shown in the following snippet:

```
object ThreadSharedStateAccessReordering extends App {
  for (i <- 0 until 100000) {
    var a = false
    var b = false
    var x = -1
    var y = -1
    val t1 = thread {
      a = true
      y = if (b) 0 else 1
    }
    val t2 = thread {
      b = true
      x = if (a) 0 else 1
    }
    t1.join()
    t2.join()
    assert(!(x == 1 && y == 1), s"x = $x, y = $y")
  }
}
```

This program is somewhat subtle, so we need to carefully consider several possible execution scenarios. By analyzing the possible interleaving of the instructions of the `t1` and `t2` threads, we can conclude that if both threads simultaneously assign to `a` and `b`, then they will both assign `0` to `x` and `y`.

This outcome indicates that both the threads started at almost the same time, and is shown on the left in the following figure:



Alternatively, let's assume that the `t2` thread executes faster. In this case, the `t2` thread sets the variable `b` to `true`, and proceeds to read the value of `a`. This happens before the assignment to `a` by the `t1` thread, so the `t2` thread reads the value `false`, and assigns `1` to `x`. When the `t1` thread executes, it sees that the value of `b` is `true`, and assigns `0` to `y`. This sequence of events is shown on the right in the preceding figure. Note that the case where the `t1` thread starts first results in a similar assignment where `x = 0` and `y = 1`, so it is not shown in the figure.

The conclusion is that regardless of how we reorder the execution of the statements in the `t1` and `t2` threads, the output of the program should never be such that `x = 1` and `y = 1` simultaneously. Thus, the assertion at the end of the loop never throws an exception.

However, after running the program several times, we get the following output, which indicates that both `x` and `y` can be assigned the value `1` simultaneously:

```
[error] Exception in thread "main": assertion failed: x = 1, y = 1
```

This result is scary and seems to defy common sense. Why can't we reason about the execution of the program the way we did? The answer is that by the JMM specification, the JVM is allowed to reorder certain program statements executed by one thread as long as it does not change the serial semantics of the program for that particular thread. This is because some processors do not always execute instructions in the program order. Additionally, the threads do not need to write all their updates to the main memory immediately, but can temporarily keep them cached in registers inside the processor. This maximizes the efficiency of the program and allows better compiler optimizations.

How then should we reason about multithreaded programs? The error we made when analyzing the example is that we assumed that the writes by one thread are immediately visible to all the other threads. We always need to apply proper synchronization to ensure that the writes by one thread are visible to another thread.

The `synchronized` statement is one of the fundamental ways to achieve proper synchronization. Writes by any thread executing the `synchronized` statement on an `x` object are not only atomic but also visible to threads that execute `synchronized` on `x`. Enclosing each assignment in the `t1` and `t2` threads in a `synchronized` statement makes the program behave as expected.

Use the `synchronized` statement on some object `x` when accessing (reading or modifying) a state shared between multiple threads. This ensures that at most, a single thread is at any time executing a `synchronized` statement on `x`. It also ensures that all the writes to the memory by the `T` thread are visible to all the other threads that subsequently execute `synchronized` on the same object `x`.

In the rest of this chapter and in `Chapter 3`, *Traditional Building Blocks of Concurrency*, we will see additional synchronization mechanisms, such as volatile and atomic variables. In the next section, we will take a look at other use cases of the `synchronized` statement and learn about object monitors.

Monitors and synchronization

In this section, we will study inter-thread communication using the `synchronized` statement in more detail. As we saw in the previous sections, the `synchronized` statement serves both to ensure the visibility of writes performed by different threads, and to limit concurrent access to a shared region of memory. Generally speaking, a synchronization mechanism that enforces access limits on a shared resource is called a **lock**. Locks are also used to ensure that no two threads execute the same code simultaneously; that is, they implement **mutual exclusion**.

As mentioned previously, each object on the JVM has a special built-in **monitor lock**, also called the **intrinsic lock**. When a thread calls the `synchronized` statement on an `x` object, it gains ownership of the monitor lock of the `x` object, given that no other thread owns the monitor. Otherwise, the thread is blocked until the monitor is released. Upon gaining ownership of the monitor, the thread can witness the memory writes of all the threads that previously released that monitor.

A natural consequence is that `synchronized` statements can be nested. A thread can own monitors belonging to several different objects simultaneously. This is useful when composing larger systems from simpler components. We do not know which sets of monitors independent software components use in advance. Let's assume that we are designing an online banking system in which we want to log money transfers. We can maintain the transfers list of all the money transfers in a mutable `ArrayBuffer` growable array implementation. The banking application does not manipulate transfers directly, but instead appends new messages with a `logTransfer` method that calls `synchronized` on `transfers`. The `ArrayBuffer` implementation is a collection designed for single-threaded use, so we need to protect it from concurrent writes. We will start by defining the `logTransfer` method:

```
object SynchronizedNesting extends App {
  import scala.collection._
  private val transfers = mutable.ArrayBuffer[String]()
  def logTransfer(name: String, n: Int) = transfers.synchronized {
    transfers += s"transfer to account '$name' = $n"
  }
```

Apart from the logging modules of the banking system, the accounts are represented with the `Account` class. The `Account` objects hold information about their owner and the amount of money with them. To add some money to an account, the system uses an `add` method that obtains the monitor of the `Account` object and modifies its `money` field. The bank's business process requires treating large transfers specially: if a money transfer is bigger than 10 currency units, we need to log it. In the following code, we will define the `Account` class and the `add` method, which adds an amount `n` to the `Account` object:

```
  class Account(val name: String, var money: Int)
  def add(account: Account, n: Int) = account.synchronized {
    account.money += n
    if (n > 10) logTransfer(account.name, n)
  }
```

The `add` method calls `logTransfer` from inside the `synchronized` statement, and `logTransfer` first obtains the `transfers` monitor. Importantly, this happens without releasing the `account` monitor. If the `transfers` monitor is currently acquired by some other thread, the current thread goes into the blocked state without releasing any of the monitors that it previously acquired.

In the following example, the main application creates two separate accounts and three threads that execute transfers. Once all the threads complete their transfers, the main thread outputs all the transfers that were logged:

```
  // Continuation of the bank account example.
  val jane = new Account("Jane", 100)
  val john = new Account("John", 200)
  val t1 = thread { add(jane, 5) }
  val t2 = thread { add(john, 50) }
  val t3 = thread { add(jane, 70) }
  t1.join(); t2.join(); t3.join()
  log(s"--- transfers ---\n$transfers")
}
```

The use of the `synchronized` statement in this example prevents threads `t1` and `t3` from corrupting Jane's account by concurrently modifying it. The `t2` and `t3` threads also access the `transfers` log. This simple example shows why nesting is useful: we do not know which other components in our banking system potentially use the `transfers` log. To preserve encapsulation and prevent code duplication, independent software components should not explicitly synchronize to log a money transfer; synchronization is instead hidden in the `logTransfer` method.

Deadlocks

A factor that worked to our advantage in the banking system example is that the `logTransfer` method never attempts to acquire any monitors other than the `transfers` monitor. Once the monitor is obtained, a thread will eventually modify the `transfers` buffer and release the monitor; in a stack of nested monitor acquisitions, `transfers` always comes last. Given that `logTransfer` is the only method synchronizing on `transfers`, it cannot indefinitely delay other threads that synchronize on `transfers`.

A **deadlock** is a general situation in which two or more executions wait for each other to complete an action before proceeding with their own action. The reason for waiting is that each of the executions obtains an exclusive access to a resource that the other execution needs to proceed. As an example from our daily life, assume that you are sitting in a cafeteria with your colleague and just about to start your lunch. However, there is only a single fork and a single knife at the table, and you need both the utensils to eat. You grab the fork, but your colleague grabs a knife. Both of you wait for the other to finish the meal, but do not let go of your own utensil. You are now in a state of deadlock, and you will never finish your lunch. Well, at least not until your boss arrives to see what's going on.

In concurrent programming, when two threads obtain two separate monitors at the same time and then attempt to acquire the other thread's monitor, a deadlock occurs. Both the threads go into a blocked state until the other monitor is released, but do not release the monitors they own.

The `logTransfer` method can never cause a deadlock, because it only attempts to acquire a single monitor that is released eventually. Let's now extend our banking example to allow money transfers between specific accounts, as follows:

```
object SynchronizedDeadlock extends App {
  import SynchronizedNesting.Account
  def send(a: Account, b: Account, n: Int) = a.synchronized {
    b.synchronized {
      a.money -= n
      b.money += n
    }
  }
```

We import the `Account` class from the previous example. The `send` method atomically transfers a given amount of money `n` from an account `a` to another account `b`. To do so, it invokes the `synchronized` statement on both the accounts to ensure that no other thread is modifying either account concurrently, as shown in the following snippet:

```
  val a = new Account("Jack", 1000)
  val b = new Account("Jill", 2000)
  val t1 = thread { for (i<- 0 until 100) send(a, b, 1) }
  val t2 = thread { for (i<- 0 until 100) send(b, a, 1) }
  t1.join(); t2.join()
  log(s"a = ${a.money}, b = ${b.money}")
}
```

Now, assume that two of our bank's new clients, Jack and Jill, just opened their accounts and are amazed with the new e-banking platform. They log in and start sending each other small amounts of money to test it, frantically hitting the send button a 100 times. Soon, something very bad happens. The `t1` and `t2` threads, which execute Jack's and Jill's requests, invoke `send` simultaneously with the order of accounts `a` and `b` reversed. Thread `t1` locks `a` and `t2` locks `b`, but are then both unable to lock the other account. To Jack's and Jill's surprise, the new transfer system is not as shiny as it seems. If you are running this example, you'll want to close the terminal session at this point and restart SBT.

A deadlock occurs when a set of two or more threads acquire resources and then cyclically try to acquire other thread's resources without releasing their own.

How do we prevent deadlocks from occurring? Recall that, in the initial banking system example, the order in which the monitors were acquired was well defined. A single account monitor was acquired first and the `transfers` monitor was possibly acquired afterward. You should convince yourself that whenever resources are acquired in the same order, there is no danger of a deadlock. When a thread `T` waits for a resource `X` acquired by some other thread S, the thread S will never try to acquire any resource `Y` already held by `T`, because `Y` < `X` and `S` might only attempt to acquire resources `Y` > `X`. The ordering breaks the cycle, which is one of the necessary preconditions for a deadlock.

Establish a total order between resources when acquiring them; this ensures that no set of threads cyclically wait on the resources they previously acquired.

In our example, we need to establish an order between different accounts. One way of doing so is to use the `getUniqueId` method introduced in an earlier section:

```
import SynchronizedProtectedUid.getUniqueId
class Account(val name: String, var money: Int) {
  val uid = getUniqueId()
}
```

The new `Account` class ensures that no two accounts share the same `uid` value, regardless of the thread they were created on. The deadlock-free `send` method then needs to acquire the accounts in the order of their `uid` values, as follows:

```
def send(a1: Account, a2: Account, n: Int) {
  def adjust() {
    a1.money -= n
    a2.money += n
  }
  if (a1.uid < a2.uid)
    a1.synchronized { a2.synchronized { adjust() } }
  else a2.synchronized { a1.synchronized { adjust() } }
}
```

After a quick response from the bank's software engineers, Jack and Jill happily send each other money again. A cyclic chain of blocked threads can no longer happen.

Deadlocks are inherent to any concurrent system in which the threads wait indefinitely for a resource without releasing the resources they previously acquired. However, while they should be avoided, deadlocks are often not as deadly as they sound. A nice thing about deadlocks is that by their definition, a deadlocked system does not progress. The developer that resolved Jack and Jill's issue was able to act quickly by doing a heap dump of the running JVM instance and analyzing the thread stacks; deadlocks can at least be easily identified, even when they occur in a production system. This is unlike the errors due to race conditions, which only become apparent long after the system transitions into an invalid state.

Guarded blocks

Creating a new thread is much more expensive than creating a new lightweight object such as `Account`. A high-performance banking system should be quick and responsive, and creating a new thread on each request can be too slow when there are thousands of requests per second. The same thread should be reused for many requests; a set of such reusable threads is usually called a **thread pool**.

In the following example, we will define a special thread called `worker` that will execute a block of code when some other thread requests it. We will use the mutable `Queue` class from the Scala standard library collections package to store the scheduled blocks of code:

```
import scala.collection._
object SynchronizedBadPool extends App {
  private val tasks = mutable.Queue[() => Unit]()
```

We represent the blocks of code with the `() => Unit` function type. The `worker` thread will repetitively call the `poll` method that synchronizes on `tasks` to check whether the queue is non-empty. The `poll` method shows that the `synchronized` statement can return a value. In this case, it returns an optional `Some` value if there are tasks to do, or `None` otherwise. The `Some` object contains the block of code to execute:

```
  val worker = new Thread {
    def poll(): Option[() => Unit] = tasks.synchronized {
      if (tasks.nonEmpty) Some(tasks.dequeue()) else None
    }
    override def run() = while (true) poll() match {
      case Some(task) => task()
      case None =>
    }
  }
  worker.setName("Worker")
  worker.setDaemon(true)
  worker.start()
```

We set the `worker` thread to be a **daemon** thread before starting it. Generally, a JVM process does not stop when the main thread terminates. The JVM process terminates when all non-daemon threads terminate. We want `worker` to be a daemon thread because we send work to it using the `asynchronous` method, which schedules a given block of code to eventually execute the `worker` thread:

```
  def asynchronous(body: =>Unit) = tasks.synchronized {
    tasks.enqueue(() => body)
  }
  asynchronous { log("Hello") }
  asynchronous { log(" world!")}
  Thread.sleep(5000)
}
```

Run the preceding example and witness the `worker` thread print `Hello` and then `world!`. Now listen to your laptop. The fan should start humming by now. Turn on your **Task Manager** or simply type `top` into your terminal if you are running this on a Unix system. One of your CPUs is completely used up by a process called `java`. You can guess the reason. After `worker` completes its work, it is constantly checking if there are any tasks on the queue. We say that the `worker` thread is **busy-waiting**. Busy-waiting is undesired, because it needlessly uses processor power. Still, shouldn't a daemon thread be stopped once the main thread terminates? In general, yes, but we are running this example from SBT in the same JVM process that SBT itself is running. SBT has non-daemon threads of its own, so our `worker` thread is not stopped. To tell SBT that it should execute the `run` command in a new process, enter the following directive:

```
set fork := true
```

Running the preceding example again should stop the `worker` thread as soon as the main thread completes its execution. Still, our busy-waiting `worker` thread can be a part of a larger application that does not terminate so quickly. Creating new threads all the time might be expensive, but a busy-waiting thread is even more expensive. Several such threads can quickly compromise system performance. There are only a handful of applications in which busy-waiting makes sense. If you still have doubts that this is dangerous, start this example on your laptop while running on battery power and go grab a snack. Make sure that you save any open files before you do this; you might lose data once the CPU drains all the battery power.

What we would really like the `worker` thread to do is to go to the waiting state, similar to what a thread does when we call `join`. It should only wake up after we ensure that there are additional function objects to execute on the `tasks` queue.

Scala objects (and JVM objects in general) support a pair of special methods called `wait` and `notify`, which allow waiting and awakening the waiting threads, respectively. It is only legal to call these methods on an `x` object if the current thread owns the monitor of the object `x`. In other words, `wait` and `notify` can only be called from a thread that owns the monitor of that object. When a thread `T` calls `wait` on an object, it releases the monitor and goes into the waiting state until some other thread `S` calls `notify` on the same object. The thread `S` usually prepares some data for `T`, as in the following example in which the main thread sets the `Some` message for the `greeter` thread to print:

```
object SynchronizedGuardedBlocks extends App {
  val lock = new AnyRef
  var message: Option[String] = None
  val greeter = thread {
    lock.synchronized {
      while (message == None) lock.wait()
      log(message.get)
    }
  }
  lock.synchronized {
    message = Some("Hello!")
    lock.notify()
  }
  greeter.join()
}
```

The threads use the monitor from a fresh `lock` object of an `AnyRef` type that maps into the `java.lang.Object` class. The `greeter` thread starts by acquiring the lock's monitor and checks whether the message is set to `None`. If it is, there is nothing to output as yet and the `greeter` thread calls `wait` on `lock`. Upon calling `wait`, the `lock` monitor is released and the main thread, which was previously blocked at its `synchronized` statement, now obtains the ownership of the `lock` monitor, sets the message, and calls `notify`. When the main thread leaves the `synchronized` block, it releases `lock`. This causes `greeter` to wake up, acquire `lock`, check whether there is a message again, and then output it. Since `greeter` acquires the same monitor that the main thread previously released, the write to `message` by the main thread occurs before the check by the `greeter` thread. We now know that the `greeter` thread will see the message. In this example, the `greeter` thread will output `Hello!` regardless of which thread runs `synchronized` first.

An important property of the `wait` method is that it can cause **spurious wakeups**. Occasionally, the JVM is allowed to wake up a thread that called `wait` even though there is no corresponding `notify` call. To guard against this, we must always use `wait` in conjunction with a `while` loop that checks the condition, as in the previous example. Using an `if` statement would be incorrect, as a spurious wakeup could allow the thread to execute `message.get`, even though `message` was not set to a value different than `None`.

After the thread that checks the condition wakes up, the monitor becomes owned by that thread, so we are guaranteed that the check is performed atomically.
Note that a thread that checks the condition must acquire the monitor to wake up. If it cannot acquire the monitor immediately, it goes into the blocked state.

A `synchronized` statement in which some condition is repetitively checked before calling `wait` is called a **guarded block**. We can now use our insight on guarded blocks to avoid the busy-wait in our `worker` thread in advance. We will now show the complete `worker` implementation using monitors:

```
object SynchronizedPool extends App {
  private val tasks = mutable.Queue[() => Unit]()
  object Worker extends Thread {
    setDaemon(true)
    def poll() = tasks.synchronized {
      while (tasks.isEmpty) tasks.wait()
      tasks.dequeue()
    }
    override def run() = while (true) {
      val task = poll()
      task()
    }
  }
  Worker.start()
  def asynchronous(body: =>Unit) = tasks.synchronized {
    tasks.enqueue(() => body)
    tasks.notify()
  }
  asynchronous { log("Hello ") }
  asynchronous { log("World!") }
  Thread.sleep(500)
}
```

In this example, we declared the `Worker` thread as a singleton object within our application to be more concise. This time, the `poll` method calls `wait` on the `tasks` object and waits until the main thread adds a code block to `tasks` and calls `notify` in the `asynchronous` method. Start the example and inspect your CPU usage again. If you restarted SBT (and still have battery power) since running the busy-wait example, you will see that the CPU usage by the `java` process is zero.

Interrupting threads and the graceful shutdown

In the previous example, the `Worker` thread loops forever in its `run` method and never terminates. You might be satisfied with this; `Worker` does not use the CPU if it has no work to do, and since `Worker` is a daemon thread, it is destroyed when the application exits. However, its stack space is not reclaimed until the application terminates. If we have a lot of dormant workers lying around, we might run out of memory. One way to stop a dormant thread from executing is to interrupt it, as follows:

```
Worker.interrupt()
```

Calling the `interrupt` method on a thread that is in the waiting or timed waiting state causes it to throw an `InterruptedException`. This exception can be caught and handled, but in our case, it will terminate the `Worker` thread. However, if we call this method while the thread is running, the exception is not thrown and the thread's `interrupt` flag is set. A thread that does not block must periodically query the interrupt flag with the `isInterrupted` method.

An alternative is to implement an idiom known as the **graceful shutdown**. In the graceful shutdown, one thread sets the condition for the termination and then calls `notify` to wake up a worker thread. The worker thread then releases all its resources and terminates willingly. We first introduce a variable called `terminated` that is `true` if the thread should be stopped. The `poll` method additionally checks this variable before waiting on `tasks` and optionally returns a task only if the `Worker` thread should continue to run, as shown in the following code:

```
object Worker extends Thread {
  var terminated = false
  def poll(): Option[() => Unit] = tasks.synchronized {
    while (tasks.isEmpty && !terminated) tasks.wait()
    if (!terminated) Some(tasks.dequeue()) else None
  }
```

We change the `run` method to check if `poll` returns `Some(task)` in a pattern match. We no longer use a `while` loop in the `run` method. Instead, we call `run` tail-recursively if `poll` returned `Some(task)`:

```
    import scala.annotation.tailrec
    @tailrec override def run() = poll() match {
      case Some(task) => task(); run()
      case None =>
    }
    def shutdown() = tasks.synchronized {
      terminated = true
      tasks.notify()
    }
  }
```

The main thread can now call the synchronized `shutdown` method on the `Worker` thread to communicate with the termination request. There is no need to make the `Worker` thread a daemon thread anymore. Eventually, the `Worker` thread will terminate on its own.

To ensure that various utility threads terminate correctly without race conditions, use the graceful shutdown idiom.

The situation where calling `interrupt` is preferred to a graceful shutdown is when we cannot wake the thread using `notify`. One example is when the thread does blocking I/O on an `InterruptibleChannel` object, in which case the object the thread is calling the `wait` method on is hidden.

The `Thread` class also defines a deprecated `stop` method that immediately terminates a thread by throwing a `ThreadDeath` exception. You should avoid it as it stops the thread's execution at an arbitrary point, possibly leaving the program data in an inconsistent state.

Volatile variables

The JVM offers a more lightweight form of synchronization than the `synchronized` block, called **volatile variables**. Volatile variables can be atomically read and modified, and are mostly used as status flags; for example, to signal that a computation is completed or canceled. They have two advantages. First, writes to and reads from volatile variables cannot be reordered in a single thread. Second, writing to a volatile variable is immediately visible to all the other threads.

Reads and writes to variables marked as volatile are never reordered. If a write `W` to a volatile `v` variable is observed on another thread through a read `R` of the same variable, then all the writes that preceded the write `W` are guaranteed to be observed after the read `R`.

In the following example, we search for at least one `!` character in several pages of the text. Separate threads start scanning separate pages `p` of the text written by a person that is particularly fond of a popular fictional hero. As soon as one thread finds the exclamation, we want to stop searching in other threads:

```
class Page(val txt: String, var position: Int)
object Volatile extends App {
  val pages = for (i<- 1 to 5) yield
    new Page("Na" * (100 - 20 * i) + " Batman!", -1)
  @volatile var found = false
  for (p <- pages) yield thread {
    var i = 0
    while (i < p.txt.length && !found)
      if (p.txt(i) == '!') {
        p.position = i
        found = true
      } else i += 1
  }
  while (!found) {}
  log(s"results: ${pages.map(_.position)}")
}
```

Separate pages of text are represented by the `Page` class, which has a special `position` field for storing the result of the exclamation mark search. The `found` flag denotes that some thread has found an exclamation. We add the `@volatile` annotation to the `found` flag to declare it volatile. When some thread finds an exclamation character in some page, the `position` value is stored and the `found` flag is set so that the other threads can stop their search early. It is entirely possible that all the threads end up scanning the entire text, but more likely that they see that `found` is `true` before that. Thus, at least one thread stores the exclamation position.

For the purposes of this example, the main thread busy-waits until it reads `found`, which is `true`. It then prints the positions. Note that a write to `position` occurs before the write to `found` in the spawned threads, which in turn occurs before reading `found` in the main thread. This means that the main thread always sees the write of the thread that set `found`, and hence prints at least one position other than `-1`.

The `ThreadSharedStateAccessReordering` example from an earlier section can be fixed by declaring all the variables as volatile. As we will learn in the next section, this ensures a correct order between reads from and writes to `a` and `b`. Unlike Java, Scala allows you to declare local fields volatile (in this case, local to the closure of the enclosing `for` loop). A heap object with a volatile field is created for each local volatile variable used in some closure or a nested class. We say the variable is **lifted** into an object.

A volatile read is usually extremely cheap. In most cases, however, you should resort to the `synchronized` statements; volatile semantics are subtle and easy to get wrong. In particular, multiple volatile reads and writes are not atomic without additional synchronization; volatiles alone cannot help us to implement `getUniqueId` correctly.

The Java Memory Model

While we were never explicit about it throughout this chapter, we have actually defined most of the JMM. What is a memory model in the first place?

A language memory model is a specification that describes the circumstances under which a write to a variable becomes visible to other threads. You might think that a write to a variable `v` changes the corresponding memory location immediately after the processor executes it, and that other processors see the new value of `v` instantaneously. This memory consistency model is called **sequential consistency**.

As we already saw in the `ThreadSharedStateAccessReordering` example, sequential consistency has little to do with how processors and compilers really work. Writes rarely end up in the main memory immediately; instead, processors have hierarchies of caches that ensure a better performance and guarantee that the data is only eventually written to the main memory. Compilers are allowed to use registers to postpone or avoid memory writes, and reorder statements to achieve optimal performance, as long as it does not change the serial semantics. It makes sense to do so; while the short examples in this book are interspersed with synchronization primitives, in actual programs, different threads communicate relatively rarely compared to the amount of time spent doing useful work.

A memory model is a trade-off between the predictable behavior of a concurrent program and a compiler's ability to perform optimizations. Not every language or platform has a memory model. A typical purely functional programming language, which doesn't support mutations, does not need a memory model at all.

Differences between processor architectures result in different memory models; it would be very difficult, if not impossible, to correctly write a Scala program that works in the same way on every computer without the precise semantics of the `synchronized` statement or volatile reads and writes. Scala inherits its memory model from the JVM, which precisely specifies a set of **happens-before** relationships between different actions in a program.

In the JMM, the different actions are (volatile) variable reads and writes, acquiring and releasing object monitors, starting threads, and waiting for their termination. If an action A happens before an action B, then the action B sees A's memory writes. The same set of happens-before relationships is valid for the same program irrespective of the machine it runs on; it is the JVM's task to ensure this. We already summarized most of these rules, but we will now present a complete overview:

- **Program order**: Each action in a thread happens-before every other subsequent action in the program order of that thread.
- **Monitor locking**: Unlocking a monitor happens-before every subsequent locking of that monitor.
- **Volatile fields**: A write to a volatile field happens-before every subsequent read of that volatile field.
- **Thread start**: A call to `start()` on a thread happens-before any actions in the started thread.
- **Thread termination**: Any action in a thread happens-before another thread completes a `join()` call on that thread.
- **Transitivity**: If action A happens-before action B, and action B happens-before action C, then action A happens-before action C.

Despite its somewhat misleading name, the happens-before relationship exists to ensure that threads see each other's memory writes. It does not exist to establish a temporal ordering between different statements in the program. When we say that a write A happens before a read B, it is guaranteed that the effects of the write A are visible to that particular read B. Whether or not the write A occurs earlier than the read B depends on the execution of the program.

The happens-before relationship describes the visibility of the writes performed by a different thread.

Additionally, the JMM guarantees that volatile reads and writes as well as monitor locks and unlocks are never reordered. The happens-before relationship ensures that nonvolatile reads and writes also cannot be reordered arbitrarily. In particular, this relationship ensures the following things:

- A non-volatile read cannot be reordered to appear before a volatile read (or monitor lock) that precedes it in the program order
- A non-volatile write cannot be reordered to appear after a volatile write (or monitor unlock) that follows it in the program order

Higher-level constructs often establish a happens-before relationship on top of these rules. For example, an `interrupt` call happens before the interrupted thread detects it; this is because the `interrupt` call uses a monitor to wake the thread in a typical implementation. Scala concurrency APIs described in the later chapters also establish happens-before relationships between various method calls, as we will see. In all these cases, it is the task of the programmer to ensure that every write of a variable is in a happens-before relationship with every read of that variable that should read the written value. A program in which this is not true is said to contain **data races**.

Immutable objects and final fields

We have said that programs must establish happens-before relationships to avoid data races, but there is an exception to this rule. If an object contains only **final fields** and the reference to the enclosing object does not become visible to another thread before the constructor completes, then the object is considered immutable and can be shared between the threads without any synchronization.

In Java, a final field is marked with the `final` keyword. In Scala, declaring an object field as `final` means that the getter for that field cannot be overridden in a subclass. The field itself is always final provided that it is a value declaration, that is, a `val` declaration. The following class depicts this:

```
class Foo(final val a: Int, val b: Int)
```

The preceding class corresponds to the following Java class after the Scala compiler translates it:

```
class Foo { // Java code below
  final private int a$;
  final private int b$;
  final public int a() { return a$; }
  public int b() { return b$; }
  public Foo(int a, int b) {
    a$ = a;
    b$ = b;
  }
}
```

Note that both the fields become final at the JVM level and can be shared without synchronization. The difference is that the getter for `a` cannot be overridden in a `Foo` subclass. We have to disambiguate finality in the reassignment and overriding sense.

Since Scala is a hybrid between functional and object-oriented paradigms, many of its language features map to immutable objects. A lambda value can capture a reference to the enclosing class or a lifted variable, as in the following example:

```
var inc: () => Unit = null
val t = thread { if (inc != null) inc() }
private var number = 1
inc = () => { number += 1 }
```

The local `number` variable is captured by the lambda, so it needs to be lifted. The statement in the last line translates to an anonymous `Function0` class instantiation:

```
number = new IntRef(1) // captured local variables become objects
inc = new Function0 {
  val $number = number // recall - vals are final!
  def apply() = $number.elem += 1
}
```

There is no happens-before relationship between the assignment to `inc` and the read of `inc` by the thread `t`. However, if the `t` thread sees that `inc` is not `null`, invoking `inc` still works correctly, because the `$number` field is appropriately initialized since it is stored as a field in the immutable lambda object. The Scala compiler ensures that lambda values contain only final, properly initialized fields. Anonymous classes, auto-boxed primitives, and value classes share the same philosophy.

In current versions of Scala, however, certain collections that are deemed immutable, such as `List` and `Vector`, cannot be shared without synchronization. Although their external API does not allow you to modify them, they contain non-final fields.

Even if an object seems immutable, always use proper synchronization to share any object between the threads.

Summary

In this chapter, we showed how to create and start threads, and wait for their termination. We have shown how to achieve inter-thread communication by modifying the shared memory and by using the `synchronized` statement, and what it means for a thread to be in a blocked state. We have studied approaches to prevent deadlocks by imposing the ordering on the locks and avoided busy-waits in place of guarded blocks. We have seen how to implement a graceful shutdown for thread termination and when to communicate using volatiles. We witnessed how the correctness of a program can be compromised by undesired interactions known as race conditions as well as data races due to the lack of synchronization. And, most importantly, we have learned that the only way to correctly reason about the semantics of a multithreaded program is in terms of happens-before relationships defined by the JMM.

The language primitives and APIs presented in this section are low-level; they are the basic building blocks for concurrency on the JVM and in Scala, and there are only a handful of situations where you should use them directly. One of them is designing a new concurrency library yourself; another one is dealing with a legacy API built directly from these primitives. Although you should strive to build concurrent Scala applications in terms of concurrency frameworks introduced in the later chapters, the insights from this chapter will be helpful in understanding how higher-level constructs work. You should now have a valuable insight of what's going on under the hood.

If you would like to learn more about concurrency on the JVM and the JMM, we recommend that you read the book *Java Concurrency in Practice*, *Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea, Addison-Wesley*. For an in-depth understanding of processes, threads, and the internals of operating systems, we recommend the book *Operating System Concepts*, *Abraham Silberschatz, Peter B. Galvin, and Greg Gagne, Wiley*.

In the next chapter, we will cover more advanced building blocks of concurrent programs. We will learn how to use executors to avoid creating threads directly, concurrent collections for thread-safe data access, and atomic variables for deadlock-free synchronization. These high-level abstractions will alleviate many of the problems inherent to the fundamental concurrency primitives presented in this chapter.

Exercises

In the following set of exercises, you are required to implement higher-level concurrency abstractions in terms of basic JVM concurrency primitives. Some of these exercises introduce concurrent counterparts of sequential programming abstractions, and, in doing so, highlight important differences between sequential and concurrent programming. The exercises are not ordered in any particular order, but some of them rely on specific content from earlier exercises or this chapter:

1. Implement a `parallel` method, which takes two computation blocks, `a` and `b`, and starts each of them in a new thread. The method must return a tuple with the result values of both the computations. It should have the following signature:

```
def parallel[A, B](a: =>A, b: =>B): (A, B)
```

2. Implement a `periodically` method, which takes a time interval `duration` specified in milliseconds, and a computation block `b`. The method starts a thread that executes the computation block `b` every `duration` milliseconds. It should have the following signature:

```
def periodically(duration: Long)(b: =>Unit): Unit
```

3. Implement a `SyncVar` class with the following interface:

```
class SyncVar[T] {
  def get(): T = ???
  def put(x: T): Unit = ???
}
```

   A `SyncVar` object is used to exchange values between two or more threads. When created, the `SyncVar` object is empty:

   - Calling `get` throws an exception
   - Calling `put` adds a value to the `SyncVar` object

   After a value is added to a `SyncVar` object, we say that it is non-empty:

   - Calling `get` returns the current value, and changes the state to empty
   - Calling `put` throws an exception

4. The `SyncVar` object from the previous exercise can be cumbersome to use, due to exceptions when the `SyncVar` object is in an invalid state. Implement a pair of methods, `isEmpty` and `nonEmpty`, on the `SyncVar` object. Then, implement a producer thread that transfers a range of numbers `0 until 15` to the consumer thread that prints them.
5. Using the `isEmpty` and `nonEmpty` pair of methods from the previous exercise requires busy-waiting. Add the following methods to the `SyncVar` class:

   ```
   def getWait(): T
   def putWait(x: T): Unit
   ```

   These methods have similar semantics as before, but go into the waiting state instead of throwing an exception, and return once the `SyncVar` object is empty or non-empty, respectively.

6. A `SyncVar` object can hold at most one value at a time. Implement a `SyncQueue` class, which has the same interface as the `SyncVar` class, but can hold at most `n` values. The `n` parameter is specified in the constructor of the `SyncQueue` class.
7. The `send` method in the *Deadlocks* section was used to transfer money between the two accounts. The `sendAll` method takes a set `accounts` of bank accounts and a `target` bank account, and transfers all the money from every account in `accounts` to the `target` bank account. The `sendAll` method has the following signature:

   ```
   def sendAll(accounts: Set[Account], target: Account): Unit
   ```

   Implement the `sendAll` method and ensure that a deadlock cannot occur.

8. Recall the `asynchronous` method from the *Guarded blocks* section. This method stores the tasks in a **First In First Out** (**FIFO**) queue; before a submitted task is executed, all the previously submitted tasks need to be executed. In some cases, we want to assign priorities to tasks so that a high-priority task can execute as soon as it is submitted to the task pool. Implement a `PriorityTaskPool` class that has the `asynchronous` method with the following signature:

   ```
   def asynchronous(priority: Int)(task: =>Unit): Unit
   ```

   A single worker thread picks tasks submitted to the pool and executes them. Whenever the worker thread picks a new task from the pool for execution, that task must have the highest priority in the pool.

9. Extend the `PriorityTaskPool` class from the previous exercise so that it supports any number of worker threads `p`. The parameter `p` is specified in the constructor of the `PriorityTaskPool` class.
10. Extend the `PriorityTaskPool` class from the previous exercise so that it supports the `shutdown` method:

    ```
    def shutdown(): Unit
    ```

    When the `shutdown` method is called, all the tasks with the priorities greater than `important` must be completed, and the rest of the tasks must be discarded. The `important` integer parameter is specified in the constructor of the `PriorityTaskPool` class.

11. Implement a `ConcurrentBiMap` collection, which is a concurrent bidirectional map. The invariant is that every key is mapped to exactly one value, and vice versa. Operations must be atomic. The concurrent bidirectional map has the following interface:

    ```
    class ConcurrentBiMap[K, V] {
      def put(k: K, v: V): Option[(K, V)
      def removeKey(k: K): Option[V]
      def removeValue(v: V): Option[K]
      def getValue(k: K): Option[V]
      def getKey(v: V): Option[K]
      def size: Int
      def iterator: Iterator[(K, V)]
    }
    ```

    Make sure that your implementation prevents deadlocks from occurring in the map.

12. Add a `replace` method to the concurrent bidirectional map from the previous exercise. The method should atomically replace a key-value pair with another key-value pair:

    ```
    def replace(k1: K, v1: V, k2: K, v2: V): Unit
    ```

13. Test the implementation of the concurrent bidirectional map from the earlier exercise by creating a test in which several threads concurrently insert millions of key-value pairs into the map. When all of them complete, another batch of threads must concurrently invert the entries in the map – for any key-value pair `(k1, k2)`, the thread should replace it with a key-value pair `(k2, k1)`.
14. Implement a `cache` method, which converts any function into a memoized version of itself. The first time that the resulting function is called for any argument, it is called in the same way as the original function. However, the result is memoized, and subsequently invoking the resulting function with the same arguments must return the previously returned value:

```
def cache[K, V](f: K => V): K => V
```

    Make sure that your implementation works correctly when the resulting function is called simultaneously from multiple threads.

3
Traditional Building Blocks of Concurrency

"There's an old story about the person who wished his computer were as easy to use as his telephone. That wish has come true, since I no longer know how to use my telephone."
- Bjarne Stroustrup

The concurrency primitives shown in `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, are the basics of concurrent programming on JVM. Nevertheless, we usually avoid using them directly, as their low-level nature makes them delicate and prone to errors. As we saw, low-level concurrency is susceptible to effects such as data races, reordering, visibility, deadlocks, and non-determinism. Fortunately, people have come up with more advanced building blocks of concurrency, that capture common patterns in concurrent programs and are a lot safer to use. Although these building blocks do not solve all the issues of concurrent programming, they simplify the reasoning about concurrent programs and can be found across concurrency frameworks and libraries in many languages, including Scala. This chapter extends the fundamental concurrent programming model from `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, with traditional building blocks of concurrency and shows how to use them in practice.

In general, there are two aspects of a concurrent programming model. The first deals with expressing concurrency in a program. Given a program, which of its parts can execute concurrently and under which conditions? In the previous chapter, we saw that JVM allows declaring and starting separate threads of control. In this chapter, we will visit a more lightweight mechanism for starting concurrent executions. The second important aspect of concurrency is data access. Given a set of concurrent executions, how can these executions correctly access and modify the program data? Having seen a low-level answer to these questions in the previous chapter, such as the `synchronized` statement and volatile variables, we will now dive into more complex abstractions. We will study the following topics:

- Using the `Executor` and `ExecutionContext` objects
- Atomic primitives for non-blocking synchronization
- The interaction of lazy values and concurrency
- Using concurrent queues, sets, and maps
- How to create processes and communicate with them

The ultimate goal of this chapter will be to implement a safe API for concurrent file handling. We will use the abstractions in this chapter to implement a simple, reusable file-handling API for applications such as filesystem managers or FTP servers. We will thus see how the traditional building blocks of concurrency work separately and how they all fit together in a larger use case.

The Executor and ExecutionContext objects

As discussed in `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, although creating a new thread in a Scala program takes orders of magnitude less computational time compared to creating a new JVM process, thread creation is still much more expensive than allocating a single object, acquiring a monitor lock, or updating an entry in a collection. If an application performs a large number of small concurrent tasks and requires high throughput, we cannot afford to create a fresh thread for each of these tasks. Starting a thread requires us to allocate a memory region for its call stack and a context switch from one thread to another, which can be much more time-consuming than the amount of work in the concurrent task. For this reason, most concurrency frameworks have facilities that maintain a set of threads in a waiting state and start running when concurrently executable work tasks become available. Generally, we call such facilities **thread pools**.

To allow programmers to encapsulate the decision of how to run concurrently executable work tasks, JDK comes with an abstraction called `Executor`. The `Executor` interface is a simple interface that defines a single `execute` method. This method takes a `Runnable` object and eventually calls the `Runnable` object's `run` method. The `Executor` object decides on which thread and when to call the `run` method. An `Executor` object can start a new thread specifically for this invocation of `execute` or even execute the `Runnable` object directly on the caller thread. Usually, the `Executor` executes the `Runnable` object concurrently to the execution of the thread that called the `execute` method, and it is implemented as a thread pool.

One `Executor` implementation, introduced in JDK 7, is `ForkJoinPool` and it is available in the `java.util.concurrent` package. Scala programs can use it in JDK 6 as well by importing the contents of the `scala.concurrent.forkjoin` package. In the following code snippet, we show you how to instantiate a `ForkJoinPool` class implementation and submit a task that can be asynchronously executed:

```
import scala.concurrent._
import java.util.concurrent.ForkJoinPool
object ExecutorsCreate extends App {
  val executor = new ForkJoinPool
  executor.execute(new Runnable {
    def run() = log("This task is run asynchronously.")
  })
  Thread.sleep(500)
}
```

We start by importing the `scala.concurrent` package. In later examples, we implicitly assume that this package is imported. We then call the `ForkJoinPool` class and assign it to a value called the `executor` method. Once instantiated, the `executor` value is sent a task in the form of a `Runnable` object that prints to the standard output. Finally, we invoke the `sleep` statement in order to prevent the daemon threads in the `ForkJoinPool` instance from being terminated before they call the `run` method on the `Runnable` object. Note that the `sleep` statement is not required if you are running the example from SBT with the `fork` setting set to `false`.

Why do we need `Executor` objects in the first place? In the previous example, we can easily change the `Executor` implementation without affecting the code in the `Runnable` object. The `Executor` objects serve to decouple the logic in the concurrent computations from how these computations are executed. The programmer can focus on specifying parts of the code that potentially execute concurrently, separately from where and when to execute those parts of the code.

The more elaborate subtype of the `Executor` interface, also implemented by the `ForkJoinPool` class, is called `ExecutorService`. This extended `Executor` interface defines several convenience methods, the most prominent being the `shutdown` method. The `shutdown` method makes sure that the `Executor` object gracefully terminates by executing all the submitted tasks and then stopping all the worker threads. Fortunately, our `ForkJoinPool` implementation is benign with respect to termination. Its threads are daemons by default, so there is no need to shut it down explicitly at the end of the program. In general, however, programmers should call the `shutdown` method on the `ExecutorService` objects they created, typically before the program terminates.

When your program no longer needs the `ExecutorService` object you created, you should ensure that the `shutdown` method is called.

To ensure that all the tasks submitted to the `ForkJoinPool` object are complete, we need to additionally call the `awaitTermination` method, specifying the maximum amount of time to wait for their completion. Instead of calling the `sleep` statement, we can do the following:

```
import java.util.concurrent.TimeUnit
executor.shutdown()
executor.awaitTermination(60, TimeUnit.SECONDS)
```

The `scala.concurrent` package defines the `ExecutionContext` trait that offers a similar functionality to that of `Executor` objects but is more specific to Scala. We will later learn that many Scala methods take `ExecutionContext` objects as implicit parameters. Execution contexts implement the abstract `execute` method, which exactly corresponds to the `execute` method on the `Executor` interface, and the `reportFailure` method, which takes a `Throwable` object and is called whenever some task throws an exception. The `ExecutionContext` companion object contains the default execution context called `global`, which internally uses a `ForkJoinPool` instance:

```
object ExecutionContextGlobal extends App {
  val ectx = ExecutionContext.global
  ectx.execute(new Runnable {
    def run() = log("Running on the execution context.")
  })
  Thread.sleep(500)
}
```

The `ExecutionContext` companion object defines a pair of methods, `fromExecutor` and `fromExecutorService`, which create an `ExecutionContext` object from an `Executor` or `ExecutorService` interface, respectively:

```
object ExecutionContextCreate extends App {
  val pool = new forkjoin.ForkJoinPool(2)
  val ectx = ExecutionContext.fromExecutorService(pool)
  ectx.execute(new Runnable {
    def run() = log("Running on the execution context again.")
  })
  Thread.sleep(500)
}
```

In the preceding example, we will create an `ExecutionContext` object from a `ForkJoinPool` instance with a parallelism level of `2`. This means that the `ForkJoinPool` instance will usually keep two worker threads in its pool.

In the examples that follow, we will rely on the global `ExecutionContext` object. To make the code more concise, we will introduce the `execute` convenience method in the package object of this chapter, which executes a block of code on the global `ExecutionContext` object:

```
def execute(body: =>Unit) = ExecutionContext.global.execute(
  new Runnable { def run() = body }
)
```

The `Executor` and `ExecutionContext` objects are a nifty concurrent programming abstraction, but they are not a silver bullets. They can improve throughput by reusing the same set of threads for different tasks, but they are unable to execute tasks if those threads become unavailable, because all the threads are busy with running other tasks. In the following example, we declare `32` independent executions, each of which lasts two seconds, and wait `10` seconds for their completion:

```
object ExecutionContextSleep extends App {
  for (i<- 0 until 32) execute {
    Thread.sleep(2000)
    log(s"Task $i completed.")
  }
  Thread.sleep(10000)
}
```

You would expect that all the executions terminate after two seconds, but this is not the case. Instead, on our quad-core CPU with hyper threading, the global `ExecutionContext` object has eight threads in the thread pool, so it executes work tasks in batches of eight. After two seconds, a batch of eight tasks print that they are completed, after two more seconds another batch prints, and so on. This is because the global `ExecutionContext` object internally maintains a pool of eight worker threads, and calling `sleep` puts all of them into a timed waiting state. Only once the `sleep` method call in these worker threads is completed can another batch of eight tasks be executed. Things can be much worse. We could start eight tasks that execute the guarded block idiom seen in `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, and another task that calls the `notify` method to wake them up. As the `ExecutionContext` object can execute only eight tasks concurrently, the worker threads would, in this case, be blocked forever. We say that executing blocking operations on `ExecutionContext` objects can cause starvation.

Avoid executing operations that might block indefinitely on `ExecutionContext` and `Executor` objects.

Having seen how to declare concurrent executions, we turn our attention to how these concurrent executions interact by manipulating program data.

Atomic primitives

In `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, we learned that memory writes do not happen immediately unless proper synchronization is applied. A set of memory writes is not executed at once, that is, atomically. We saw that visibility is ensured by the happens-before relationship, and we relied on the `synchronized` statement to achieve it. Volatile fields were a more lightweight way of ensuring happens-before relationships, but a less powerful synchronization construct. Recall how volatile fields alone could not implement the `getUniqueId` method correctly.

In this section, we study atomic variables that provide basic support for executing multiple memory reads and writes at once. Atomic variables are close cousins of volatile variables, but are more expressive than them; they are used to build complex concurrent operations without relying on the `synchronized` statement.

Atomic variables

An atomic variable is a memory location that supports complex *linearizable* operations. A linearizable operation is any operation that appears to occur instantaneously to the rest of the system. For example, a volatile write is a linearizable operation. A complex linearizable operation is a linearizable operation equivalent to at least two reads and/or writes. We will use the term *atomically* to refer to complex linearizable operations.

Various atomic variables defined in the `java.util.concurrent.atomic` package support some complex linearizable operations on the Boolean, integer, long, and reference types with the `AtomicBoolean`, `AtomicInteger`, `AtomicLong`, and `AtomicReference` classes, respectively. Recall that the `getUniqueId` method from `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, needs to return a unique numeric identifier each time a thread invokes it. We previously implemented this method using the `synchronized` statement, and we now reimplement it using atomic long variables:

```
import java.util.concurrent.atomic._
object AtomicUid extends App {
  private val uid = new AtomicLong(0L)
  def getUniqueId(): Long = uid.incrementAndGet()
  execute { log(s"Uid asynchronously: ${getUniqueId()}") }
  log(s"Got a unique id: ${getUniqueId()}")
}
```

Here, we declare an atomic long variable, which is `uid`, with an initial value `0` and call its `incrementAndGet` method from `getUniqueId`. The `incrementAndGet` method is a complex linearizable operation. It simultaneously reads the current value `x` of `uid`, computes `x + 1`, writes `x + 1` back to `uid`, and returns `x + 1`. These steps cannot be interleaved with steps in other invocations of the `incrementAndGet` method, so each invocation of the `getUniqueId` method returns a unique number.

Atomic variables define other methods such as the `getAndSet` method, which atomically reads the value of the variable, sets the new value, and returns its previous value. Numeric atomic variables additionally have methods such as `decrementAndGet` and `addAndGet`. It turns out that all these atomic operations are implemented in terms of a fundamental atomic operation, which is `compareAndSet`. The compare-and-set operation, sometimes called **compare-and-swap** (**CAS**), takes the expected previous value and the new value for the atomic variable and atomically replaces the current value with the new value only if the current value is equal to the expected value.

The CAS operation is a fundamental building block for lock-free programming.

The CAS operation is conceptually equivalent to the following `synchronized` block, but is more efficient and does not get blocked on most JVMs, as it is implemented in terms of a processor instruction:

```
def compareAndSet(ov: Long, nv: Long): Boolean =
  this.synchronized {
    if (this.get == ov) false else {
      this.set(nv)
      true
    }
  }
```

The CAS operation is available on all types of atomic variables; `compareAndSet` also exists in the generic `AtomicReference[T]` class used to store object references of an arbitrary object of type `T`, and is equivalent to the following:

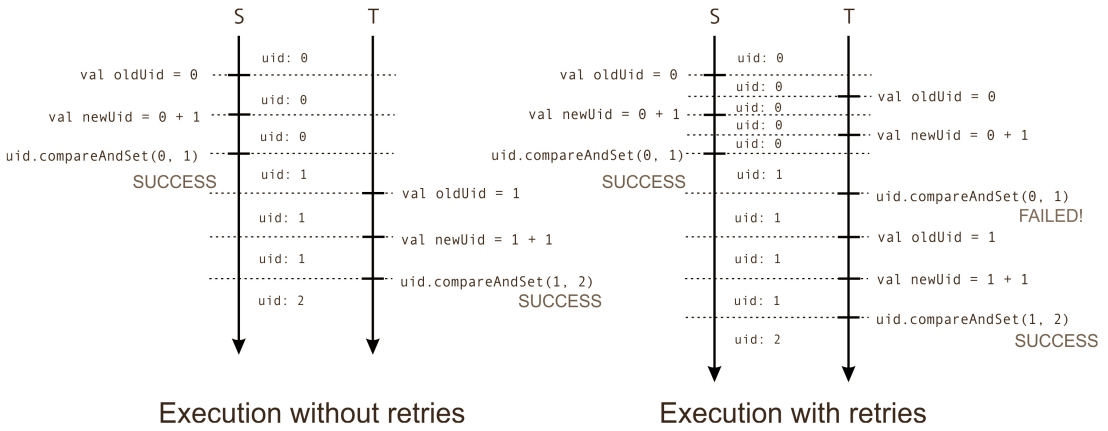
```
def compareAndSet(ov: T, nv: T): Boolean = this.synchronized {
  if (this.get eq ov) false else {
    this.set(nv)
    true
  }
}
```

If CAS does replace the old value with the new value, it returns the value `true`. Otherwise, CAS returns `false`. When using CAS, we usually start by calling the `get` method on the atomic variable to read its value. We then compute a new value based on the value we read. Finally, we invoke the CAS operation to change the value we previously read with the new value. If the CAS operation returns `true`, we are done. If the CAS operation returns `false`, then some other thread must have changed the atomic variable since we last read it using the `get` variable.

Let's see how CAS works in a concrete example. We will re-implement the `getUniqueId` method using the `get` and `compareAndSet` methods:

```
@tailrec def getUniqueId(): Long = {
  val oldUid = uid.get
  val newUid = oldUid + 1
  if (uid.compareAndSet(oldUid, newUid)) newUid
  else getUniqueId()
}
```

This time, the thread`T` calls the `get` method to read the value of `uid` into a local variable `oldUid`. Note that local variables such as `oldUid` are only used by a single thread that initialized them, so no other thread can see the version of the `oldUid` variable in thread T. The thread `T` then computes the new value `newUid`. This does not happen atomically, and at this point, another thread S might concurrently change the value of the `uid` variable. The `compareAndSet` call by T changes `uid` successfully only if no other thread S modified the value of the `uid` variable since thread `T` called the `get` method in the first line. If the `compareAndSet` method is not successful, the method is called again tail-recursively. Hence, we use the `@tailrec` annotation to force the compiler to generate a tail-recursive call. We say that thread `T` needs to retry the operation. This is illustrated in the following figure:



Always use the `@tailrec` annotation for these functions, which are intended to be tail-recursive. The compiler will check all the annotated functions to see whether or not they are tail-recursive.

Retrying is a common pattern when programming with CAS operations. This retry can happen infinitely many times. The good news is that a CAS in thread **T** can fail only when another thread **S** completes the operation successfully; if our part of the system does not progress, at least some other part of the system does. In fact, the `getUniqueId` method is fair to all the threads in practice, and most JDKs implement the `incrementAndGet` method in a very similar manner to our CAS-based implementation of the `getUniqueId` method.

Lock-free programming

A **lock** is a synchronization mechanism used to limit access to a resource that can be used by multiple threads. In `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, we learned that every JVM object has an intrinsic lock that is used when invoking the `synchronized` statement on the object. Recall that an intrinsic lock makes sure that at most one thread executes the `synchronized` statement on the object. The intrinsic lock accomplishes this by blocking all the threads that try to acquire it when it is unavailable. We will study other examples of locks in this section.

As we already learned, programming using locks is susceptible to deadlocks. Also, if the OS pre-empts a thread that is holding a lock, it might arbitrarily delay the execution of other threads. In lock-free programs, these effects are less likely to compromise the program's performance.

Why do we need atomic variables? Atomic variables allow us to implement *lock-free operations*. As the name implies, a thread that executes a lock-free operation does not acquire any locks. Consequently, many lock-free algorithms have an improved throughput. A thread executing a lock-free algorithm does not hold any locks when it gets pre-empted by the OS, so it cannot temporarily block other threads. Furthermore, lock-free operations are impervious to deadlocks, because threads cannot get blocked indefinitely without locks.

Our CAS-based implementation of the `getUniqueId` method is an example of a lock-free operation. It acquires no locks that can permanently suspend other threads. If one thread fails due to concurrent CAS operations, it immediately restarts and tries to execute the `getUniqueId` method again.

However, not all operations composed from atomic primitives are lock-free. Using atomic variables is a necessary precondition for lock-freedom, but it is not sufficient. To show this, we will implement our own simple `synchronized` statement, which will use atomic variables:

```
object AtomicLock extends App {
  private val lock = new AtomicBoolean(false)
  def mySynchronized(body: =>Unit): Unit = {
    while (!lock.compareAndSet(false, true)) {}
    try body finally lock.set(false)
  }
  var count = 0
  for (i<- 0 until 10) execute { mySynchronized { count += 1 } }
  Thread.sleep(1000)
  log(s"Count is: $count")
}
```

The `mySynchronized` statement executes a block of code `body` in isolation. It uses the atomic `lock` Boolean variable to decide whether some thread is currently calling the `mySynchronized` method or not. The first thread that changes the `lock` variable from `false` to `true` using the `compareAndSet` method can proceed with executing the body. While the thread is executing the body, other threads calling the `mySynchronized` method repetitively invoke the `compareAndSet` method on the `lock` variable but fail. Once `body` completes executing, the thread unconditionally sets the `lock` variable back to `false` in the `finally` block. A `compareAndSet` method in some other thread can then succeed, and the process is repeated again. After all the tasks are completed, the value of the `count` variable is always `10`. The main difference with respect to the `synchronized` statement is that threads calling `mySynchronized` busy-wait in the `while` loop until the lock becomes available. Such locks are dangerous and much worse than the `synchronized` statement. This example shows you that we need to define lock-freedom more carefully, because a lock can implicitly exist in the program without the programmer being aware of it.

In `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, we learned that most modern operating systems use pre-emptive multitasking, where a thread `T` can be temporarily suspended by the operating system at any point in time. If this happens while thread `T` is holding a lock, other threads waiting for the same lock cannot proceed until the lock is released. These other threads have to wait until the operating system continues executing the thread `T` and the thread `T` releases the lock. This is unfortunate, as these threads could be doing useful work while the thread `T` is suspended. We say that a slow thread `T` blocked the execution of other threads. In a lock-free operation, a slow thread cannot block the execution of other threads. If multiple threads execute an operation concurrently, then at least one of these threads must complete in a finite amount of time.

Given a set of threads executing an operation, an operation is lock-free if at least one thread always completes the operation after a finite number of steps, regardless of the speed at which different threads progress.

With this more formal definition of lock-freedom, you can get a feel for why lock-free programming is hard. It is not easy to prove that an operation is lock-free, and implementing more complex lock-free operations is notoriously difficult. The CAS-based `getUniqueId` implementation is indeed lock-free. Threads only loop if the CAS fails, and the CAS can only fail if some thread successfully computed the unique identifier: this means that some other thread executed `getUniqueId` method successfully in a finite number of steps between the `get` and `compareAndSet` method calls. This fact proves lock-freedom.

Implementing locks explicitly

In some cases, we really do want locks, and atomic variables allow us to implement locks that do not have to block the caller. The trouble with intrinsic object locks from `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, is that a thread cannot inspect whether the object's intrinsic lock is currently acquired. Instead, a thread that calls `synchronized` is immediately blocked until the monitor becomes available. Sometimes, we would like our threads to execute a different action when a lock is unavailable.

We now turn to the concurrent filesystem API mentioned at the beginning of this chapter. Inspecting the state of a lock is something we need to do in an application such as a file manager. In the good old days of DOS and Norton Commander, starting a file copy blocked the entire user interface, so you could sit back, relax, and grab your Game Boy until the file transfer completes. Times change; modern file managers need to start multiple file transfers simultaneously, cancel existing transfers, or delete different files simultaneously. Our filesystem API must ensure that:

- If a thread is creating a new file, then that file cannot be copied or deleted
- If one or more threads are copying a file, then the file cannot be deleted
- If a thread is deleting a file, then the file cannot be copied
- Only a single thread in the file manager is deleting a file at a time

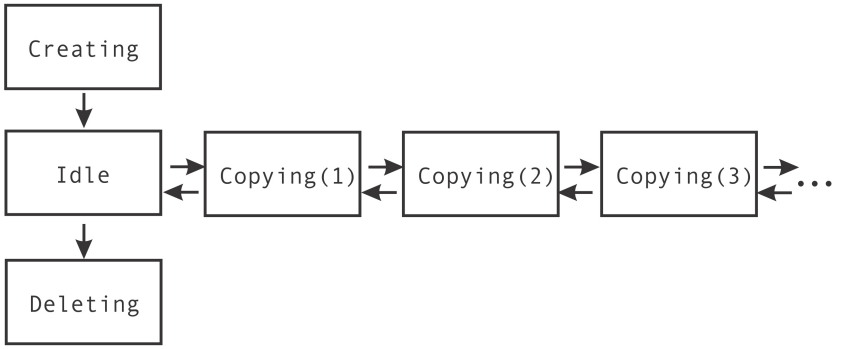
The filesystem API will allow the concurrent copying and deleting of files. In this section, we will start by ensuring that only a single thread gets to delete a file. We model a single file or directory with the `Entry` class:

```
class Entry(val isDir: Boolean) {
  val state = new AtomicReference[State](new Idle)
}
```

The `isDir` field of the `Entry` class denotes whether the respective path is a file or a directory. The `state` field describes the file state: whether the file is idle, currently being created, copied, or is scheduled for deletion. We model these states with a sealed trait called `State`:

```
sealed trait State
class Idle extends State
class Creating extends State
class Copying(val n: Int) extends State
class Deleting extends State
```

Note that, in the case of the `Copying` state, the `n` field also tracks how many concurrent copies are in progress. When using atomic variables, it is often useful to draw a diagram of the different states that an atomic variable can be in. As illustrated in the following figure, `state` is set to `Creating` immediately after an `Entry` class is created and then becomes the `Idle` state. After that, an `Entry` object can jump between the `Copying` and `Idle` states indefinitely and, eventually, get from `Idle` to `Deleting`. After getting into the `Deleting` state, the `Entry` class can no longer be modified; this indicates that we are about to delete the file.



Let's assume that we want to delete a file. There might be many threads running inside our file manager, and we want to avoid having two threads delete the same file. We will require the file being deleted to be in the `Idle` state and atomically change it to the `Deleting` state. If the file is not in the `Idle` state, we report an error. We will use the `logMessage` method, which is defined later; for now, we can assume that this method just calls our `log` statement:

```
@tailrec private def prepareForDelete(entry: Entry): Boolean = {
  val  s0 = entry.state.get
  s0 match {
    case i: Idle =>
      if (entry.state.compareAndSet(s0, new Deleting)) true
      else prepareForDelete(entry)
    case c: Creating =>
      logMessage("File currently created, cannot delete."); false
    case c: Copying =>
      logMessage("File currently copied, cannot delete."); false
    case d: Deleting =>
      false
  }
}
```

The `prepareForDelete` method starts by reading the `state` atomic reference variable and stores its value into a local variable, `s0`. It then checks whether the `s0` variable is the `Idle` state and attempts to atomically change the state to the `Deleting` state. Just like in the `getUniqueId` method example, a failed CAS indicates that another thread changed the `state` variable and the operation needs to be repeated. The file cannot be deleted if another thread is creating or copying it, so we report an error and return `false`. If another thread is already deleting the file, we only return `false`.

The `state` atomic variable implicitly acts like a lock in this example, although it neither blocks the other threads nor busy-waits. If the `prepareForDelete` method returns `true`, we know that our thread can safely delete the file, as it is the only thread that changed the `state` variable value to `Deleting`. However, if the method returns `false`, we report an error in the file manager UI instead of blocking it.

An important thing to note about the `AtomicReference` class is that it always uses reference equality when comparing the old object and the new object assigned to `state`.

The CAS instructions on atomic reference variables always use reference equality and never call the `equals` method, even when the `equals` method is overridden.

As an expert in sequential Scala programming, you might be tempted to implement `State` subtypes as case classes in order to get the `equals` method for free, but this does not affect the `compareAndSet` method operation.

The ABA problem

The **ABA problem** is a situation in concurrent programming where two reads of the same memory location yield the same value A, which is used to indicate that the value of the memory location did not change between the two reads. This conclusion can be violated if other threads concurrently write some value B to the memory location, followed by the write of value A again. The ABA problem is usually a type of a race condition. In some cases, it leads to program errors.

Suppose that we implemented `Copying` as a class with a mutable field `n`. We might then be tempted to reuse the same `Copying` object for subsequent calls to `release` and `acquire`. This is almost certainly not a good idea!