Community Experience Distilled

Learning Concurrent Programming in Scala

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

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

Aleksandar Prokopec

[PACKT] PUBLISHING open source* community experience distilled

Learning Concurrent Programming in Scala

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

Aleksandar Prokopec

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Learning Concurrent Programming in Scala

First published: November 2014

Production reference: 1211114

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-141-1

www.packtpub.com

Credits

Author
Aleksandar Prokopec

Reviewers
Dominik Gruntz
Vladimir Kostyukov
Zhen Li
Lukas Rytz
Michel Schinz
Samira Tasharofi

Commissioning Editor
Kevin Colaco

Acquisition Editor
Kevin Colaco

Content Development Editor
Vaibhav Pawar

Technical Editor
Sebastian Rodrigues

Copy Editors
Rashmi Sawant
Stuti Srivastava

Project Coordinator
Kranti Berde

Proofreaders
Mario Cecere
Martin Diver
Ameesha Green

Indexer
Tejal Soni

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

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 a 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 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. Scala has today 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.

The 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* will be a mandatory reading for everyone who writes concurrent and parallel programs in Scala. I expect to also 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 is a software developer and a concurrent and distributed programming researcher. He holds an MSc in Computing from the Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia, and a PhD in Computer Science from the École Polytechnique Fédérale de Lausanne, Switzerland. As a doctoral assistant and member of the Scala team at EPFL, he actively contributed to the Scala programming language, and 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 and Promises and ScalaSTM.

Acknowledgments

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 valuable comments. They have shown exceptional dedication and expertise in improving the quality of this book. I would also like to thank the editors at Packt Publishing: Kevin Colaco, Sruthi Kutty, Kapil Hemnani, Vaibhav Pawar, and Sebastian Rodrigues for their help in writing this book. It was really a pleasure to work with these people.

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

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

Most of the Scala concurrency frameworks rely on the works 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 implementations. Many of the Scala concurrency libraries were influenced by his advice. Furthermore, I would like to thank the Java concurrency experts for the years of work they invested into making JVM a solid concurrency platform, and especially, Brian Goetz, whose book inspired our front cover.

The Scala Futures and Promises library was initially designed by Philipp Haller, Heather Miller, Vojin Jovanović, and myself, from the EPFL; Viktor Klang and Roland Kuhn from the Akka team; 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, which were optimized through the use of Scala Macros. Eugene Burmako and Denys Shabalin are among 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, which comprised 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

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 to Packt Publishing for giving me the opportunity to 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 2014, 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 accompanied her research with valuable practical experiences by working at several IT companies, such as Microsoft and LinkedIn during the past few years. 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 & Practices)*. She was also among the reviewers of the technical research papers for software engineering conferences and workshops, 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).

Thanks for giving me the opportunity to review this book and contribute to this project.

---

www.PacktPub.com

Support files, eBooks, discount offers, and more

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://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

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

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Dedicated to Sasha, she's probably the only PhD in physical chemistry who has 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 asynchrony 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.

How this book is organized

The primary goal of this book is to help you develop skills that are necessary to write correct and efficient concurrent programs. The best way to obtain a skill is to apply it in practice. When it comes to programming, the best way to learn it is to write programs. This book aims to teach you about concurrency in Scala through a sequence of example programs, each designed to show you a particular aspect of concurrent programming. The examples range from the simplest counterparts of a "Hello World" program to programs demonstrating advanced intricacies of concurrency.

What is common to most of the programs in this book is that they are short and self-contained. This has two benefits. First, you can study most of the examples in isolation. Although we recommend that you read the entire book in the order of the chapters, you should have no problem studying specific topics. Second, conciseness ensures that each new concept is easy to grasp and understand. It is much easier to comprehend concepts like atomicity, memory contention, or busy-waiting on simple programs. This does not mean that these programs are contrived or artificial; each example illustrates an effect present in real-world programs, although stripped of irrelevant nonessentials.

When reading this book, we strongly encourage you to write down and run these examples yourself, rather than just passively study them. Each example will teach you about a new concept, but you can only fully understand each of these concepts if you try them in practice. Witnessing a particular effect in a running concurrent program is a far more valuable experience than just reading about it. So, make sure that you download SBT, and create an empty project before starting to read this book, as described later in a subsequent section. The examples are made short so that you, the reader, can try them out with almost no hassle.

At the end of each chapter, you will find a list of programming exercises. These exercises are designed to test your understanding of the various topics that have been introduced. We recommend that you try to solve at least a few after completing a chapter.

In most cases, we avoid listing the API methods, or their exact signatures. There are several reasons for this. First, you can always study the APIs in the online ScalaDoc documentation. This book would not be particularly useful if it simply repeated the content that's already there. Second, software is in a constant state of change. Although the Scala concurrency framework designers strive to keep the APIs stable, the method names and signatures are occasionally changed. This book describes the semantics of the most important concurrency facilities that are sufficient to write concurrent programs and unlikely to change.

The goal of this book is not to give a comprehensive overview of every dark corner of the Scala concurrency APIs. Instead, this book will teach you the most important concepts of concurrent programming. By the time you are done reading this book, you will not just be able to find additional information in the online documentation; you will also know what to look for. Rather than serving as a complete API reference and feeding you the exact semantics of every method, the purpose of this book is to teach you how to fish. By the time you are done reading, you will not only understand how different concurrency libraries work, but you will also know how to think when building a concurrent program.

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, 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.

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 chapter that heavily relies on the content from all the preceding chapters is *Chapter 9*, *Concurrency in Practice*, where we present a practical overview of the topics in this book.

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 that 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 `javac` to see if it is available (you will never invoke this command directly in this book, but running it verifies that it is available):

   ```
   javac
   ```

It is possible that your operating system already has JDK installed. To verify this, simply run the `javac` command, as 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, please 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 styles of text 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: "Then, it calls the `square` method to compute the value for the local variable `s`."

A block of code is shown as follows:

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

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

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

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "After clicking on the **Thread Dump** button, Java VisualVM displays the stack traces of all the threads, as shown in the following screenshot:".

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 may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via 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 on `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 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. Alternatively, you can download the source code for this book at `https://github.com/concurrent-programming-in-scala/learning-examples/`.

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 would 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright 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

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

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 gained a lot of traction in recent years with the arrival of multicore 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.

Then, 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.

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**.