LEARN THE SCALA LANGUAGE IN A PRACTICAL, PROJECT-BASED WAY

HANDS-ON SCALA
PROGRAMMING

LI HAOYI
Free Chapters

Table of Contents

Part I Introduction to Scala

Part II Local Development

Part III Web Services

Part IV Program Design

Foreword

Scala as a language delegates much to libraries. Instead of many primitive concepts and types it offers a few powerful abstractions that let libraries define flexible interfaces that are natural to use.

Haoyi's Scala libraries are a beautiful example of what can be built on top of these foundations. There's a whole universe he covers in this book: libraries for interacting with the operating system, testing, serialization, parsing, web-services to a full-featured REPL and build tool. A common thread of all these libraries is that they are simple and user-friendly.

Hands-On Scala is a great resource for learning how to use Scala. It covers a lot of ground with over a hundred mini-applications using Haoyi's Scala libraries in a straightforward way. Its code-first philosophy gets to the point quickly with minimal fuss, with code that is simple and easy to understand.

Making things simple is not easy. It requires restraint, thought, and expertise. Haoyi has laid out his approach in an illuminating blog post titled *Strategic Scala Style: The Principle of Least Power*, arguing that less power means more predictable code, faster understanding and easier maintenance for developers. I see *Hands-On Scala* as the *Principle of Least Power* in action: it shows that one can build powerful applications without needing complex frameworks.

The *Principle of Least Power* is what makes Haoyi's Scala code so easy to understand and his libraries so easy to use. *Hands-On Scala* is the best way to learn about writing Scala in this simple and straightforward manner, and a great resource for getting things done using the Scala ecosystem.

- **Martin Odersky, creator of the Scala Language**

Author's Note

I first used Scala in 2012. Back then, the language was young and it was a rough experience: weak tooling, confusing libraries, and a community focused more on fun experiments rather than serious workloads. But something caught my interest. Here was a programming language that had the convenience of scripting, the performance and scalability of a compiled language, and a strong focus on safety and correctness. Normally convenience, performance, and safety were things you had to trade off against each other, but with Scala for the first time it seemed I could have them all.

Since then, I've worked in a wide range of languages: websites in PHP and Javascript, software in Java or C# or F#, and massive codebases in Python and Coffeescript. Problems around convenience, performance, and safety were ever-present, no matter which language I was working in. It was clear to me that Scala has already solved many of these eternal problems without compromise, but it was difficult to reap these benefits unless the ecosystem of tools, libraries and community could catch up.

Today, the Scala ecosystem has caught up. Tooling has matured, simpler libraries have emerged, and the community is increasingly using Scala in serious production deployments. I myself have played a part in this, building tools and libraries to help push Scala into the mainstream. The Scala experience today is better in every way than my experience in 2012.

This book aims to introduce the Scala programming experience of today. You will learn how to use Scala in real-world applications like building websites, concurrent data pipelines, or programming language interpreters. Through these projects, you will see how Scala is the easiest way to tackle complex and difficult problems in an elegant and straightforward manner.

- **Li Haoyi, author of *Hands-on Scala Programming***
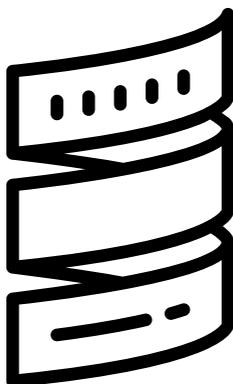
Part I: Introduction to Scala

The first part of this book is a self-contained introduction to the Scala language. We assume that you have some background programming before, and aim to help translate your existing knowledge and apply it to Scala. You will come out of this familiar with the Scala language itself, ready to begin using it in a wide variety of interesting use cases.

1

Hands-on Scala

```scala
package app
object MinimalApplication extends cask.MainRoutes {
  @cask.get("/")
  def hello() = {
    "Hello World!"
  }

  initialize()
}
```
</> 1.1.scala

Snippet 1.1: a tiny Scala web app, one of many example programs we will encounter in this book

Hands-on Scala teaches you how to use the Scala programming language in a practical, project-based fashion. Rather than trying to develop expertise in the deep details of the Scala language itself, *Hands-on Scala* aims to develop expertise using Scala in a broad range of practical applications. This book takes you from "hello world" to building interactive websites, parallel web crawlers, and distributed applications in Scala.

The book covers the concrete skills necessary for anyone using Scala professionally: handling files, data serializing, querying databases, concurrency, and so on. *Hands-on Scala* will guide you through completing several non-trivial projects which reflect the applications you may end up building as part of a software engineering job. This will let you quickly hit the ground running using Scala professionally.

Hands-on Scala assumes you are a software developer who already has experience working in another programming language, and want to quickly become productive working with Scala. If you fall into any of the following categories, this book is for you.

- Doing big data processing using software like Apache Spark which is written in Scala

- Joining one of the many companies using Scala, and need to quickly get up to speed

- Hitting performance limits in Ruby or Python, and looking for a faster compiled language

- Working with Java or Go, and looking for a language that allows more rapid development

- Already experienced working with Scala, but want to take your skills to the next level

- Founding a tech startup, looking for a language that can scale with your company as it grows

Note that this book is not targeted at complete newcomers to programming. We expect you to be familiar with basic programming concepts: variables, integers, conditionals, loops, functions, classes, and so on. We will touch on any cases where these concepts behave differently in Scala, but expect that you already have a basic understanding of how they work.

1.1 Why Scala?

Scala combines object-oriented and functional programming in one concise, high-level language. Scala's static types help avoid bugs in complex applications, and its JVM (Java Virtual Machine) runtime lets you build high-performance systems with easy access to a huge ecosystem of tools and libraries.

Scala is a general-purpose programming language, and has been applied to a wide variety of problems and domains:

- The Twitter social network has most of its backend systems written in Scala
- The Apache Spark big data engine is implemented using in Scala
- The Chisel hardware design language is built on top of Scala

While Scala has never been as mainstream as languages like Python, Java, or C++, it remains heavily used in a wide range of companies and open source projects.

1.1.1 A Compiled Language that feels Dynamic

Scala is a language that scales well from one-line snippets to million-line production codebases, with the convenience of a scripting language and the performance and scalability of a compiled language. Scala's conciseness makes rapid prototyping a joy, while its optimizing compiler and fast JVM runtime provide great performance to support your heaviest production workloads. Rather than being forced to learn a different language for each use case, Scala lets you re-use your existing skills so you can focus your attention on the actual task at hand.

1.1.2 Easy Safety and Correctness

Scala's functional programming style and type-checking compiler helps rule out entire classes of bugs and defects, saving you time and effort you can instead spend developing features for your users. Rather than fighting `TypeError`s and `NullPointerException`s in production, Scala surfaces mistakes and issues early on during compilation so you can resolve them before they impact your bottom line. Deploy your code with the confidence that you won't get woken up by outages caused by silly bugs or trivial mistakes.

1.1.3 A Broad and Deep Ecosystem

As a language running on the Java Virtual Machine, Scala has access to the large Java ecosystem of standard libraries and tools that you will inevitably need to build production applications. Whether you are looking for a Protobuf parser, a machine learning toolkit, a database access library, a profiler to find bottlenecks, or monitoring tools for your production deployment, Scala has everything you need to bring your code to production.

1.2 Why This Book?

The goal of *Hands-on Scala* is to make a software engineer productive using the Scala programming language as quickly as possible.

1.2.1 Beyond the Scala Language

Most existing Scala books focus on teaching you the language. However, knowing the minutiae of language details is neither necessary nor sufficient when the time comes to set up a website, integrate with a third-party API, or structure non-trivial applications. *Hands-on Scala* aims to bridge that gap.

This book goes beyond the Scala language itself, to also cover the various tools and libraries you need to use Scala for typical real-world work. *Hands-on Scala* will ensure you have the supporting skills necessary to use the Scala language to the fullest.

1.2.2 Focused on Real Projects

The chapters in *Hands-on Scala* are project-based: every chapter builds up to a small project in Scala to accomplish something immediately useful in real-world workplaces. These are followed up with exercises (*1.5.3*) to consolidate your knowledge and test your intuition for the topic at hand.

In the course of *Hands-on Scala*, you will work through projects such as:

- An incremental static website generator
- A project migration tool using the Github API
- A parallel web crawler
- An interactive database-backed chat website
- A real-time networked file synchronizer
- A programming language interpreter

These projects serve dual purposes: to motivate the tools and techniques you will learn about in each chapter, and also to build up your engineering toolbox. The API clients, web scrapers, file synchronizers, static site generators, web apps, and other projects you will build are all based on real-world projects implemented in Scala. By the end of this book, you will have concrete experience in the specifics tasks common when doing professional work using Scala.

1.2.3 Code First

Hands-on Scala starts and ends with working code. The concepts you learn in this book are backed up by over 140 executable code examples that demonstrate the concepts in action, and every chapter ends with a set of exercises with complete executable solutions. More than just a source of knowledge, *Hands-on Scala* can also serve as a cookbook you can use to kickstart any project you work on in future.

Hands-on Scala acknowledges that as a reader, your time is valuable. Every chapter, section and paragraph has been carefully crafted to teach the important concepts needed and get you to a working application. You can then take your working code and evolve it into your next project, tool, or product.

1.3 How This Book Is Organized

This book is organized into four parts:

Part I Introduction to Scala is a self-contained introduction to the Scala language. We assume that you have some background programming before, and aim to help translate your existing knowledge and apply it to Scala. You will come out of this familiar with the Scala language itself, ready to begin using it in a wide variety of interesting use cases.

Part II Local Development explores the core tools and techniques necessary for writing Scala applications that run on a single computer. We will cover algorithms, files and subprocess management, data serialization, scripts and build pipelines. This chapter builds towards a capstone project where we write an efficient incremental static site generator using the Scala language.

Part III Web Services covers using Scala in a world of servers and clients, systems and services. We will explore using Scala both as a client and as a server, exchanging HTML and JSON over HTTP or Websockets. This part builds towards two capstone projects: a parallel web crawler and an interactive database-backed chat website, each representing common use cases you are likely to encounter using Scala in a networked, distributed environment.

Part IV Program Design explores different ways of structuring your Scala application to tackle real-world problems. This chapter builds towards another two capstone projects: building a real-time file synchronizer and building a programming-language interpreter. These projects will give you a glimpse of the very different ways the Scala language can be used to implement challenging applications in an elegant and intuitive manner.

Each part is broken down into five chapters, each of which has its own small projects and exercises. Each chapter contains both small code snippets as well as entire programs, which can be accessed via links (*1.5*) for copy-pasting into your editor or command-line.

The libraries and tools used in this book have their own comprehensive online documentation. *Hands-on Scala* does not aim to be a comprehensive reference to every possible topic, but instead will link you to the online documentation if you wish to learn more. Each chapter will also make note of alternate sets of libraries or tools that you may encounter using Scala in the wild.

1.3.1 Chapter Dependency Graph

While *Hands-on Scala* is intended to be read cover-to-cover, you can also pick and choose which specific topics you want to read about. The following diagram shows the dependencies between chapters, so you can chart your own path through the book focusing on the things you are most interested in learning.
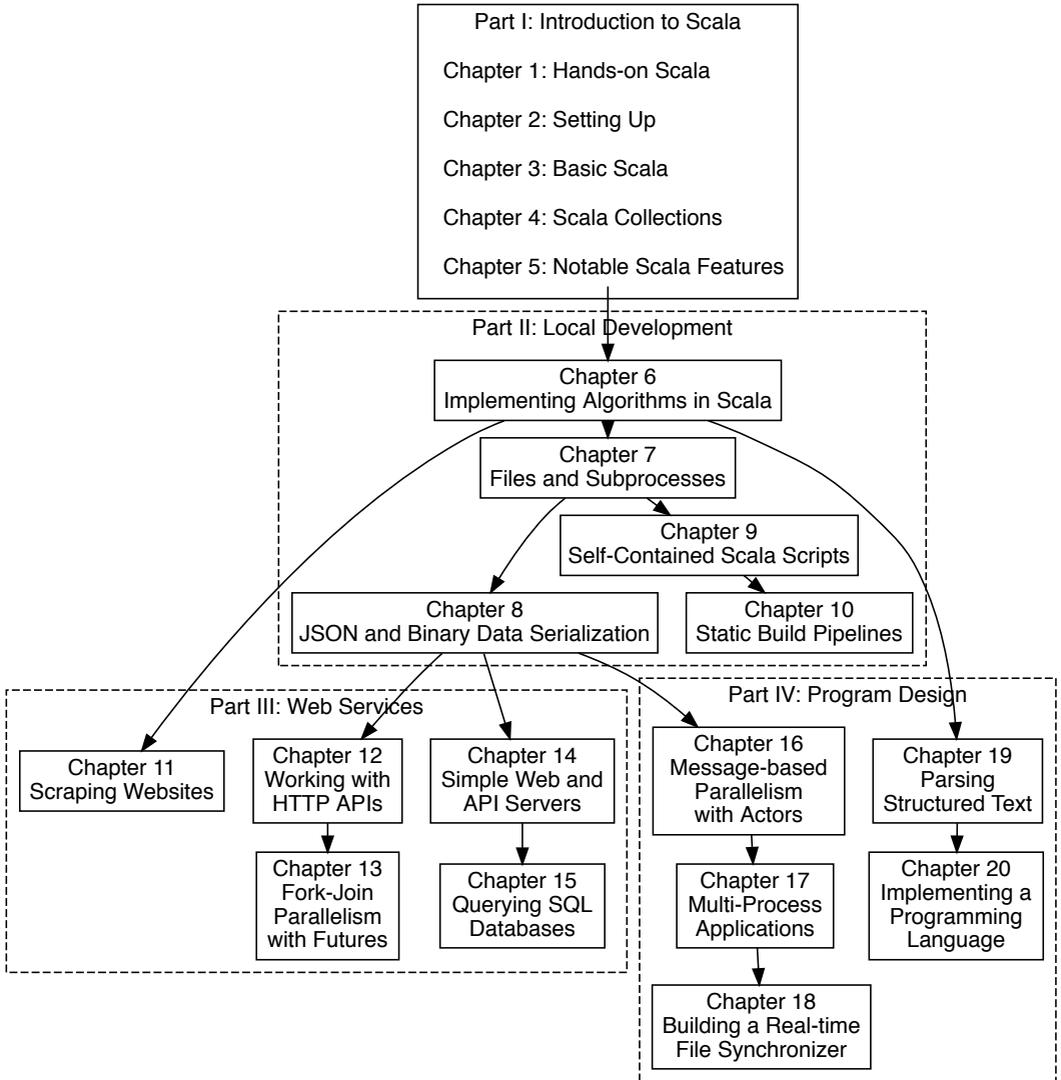
1.4 Code Snippet and Examples

In this book, we will be going through a lot of code. As a reader, we expect you to follow along with the code throughout the chapter: that means working with your terminal and your editor open, entering and executing the code examples given. Make sure you execute the code and see it working, to give yourself a feel for how the code behaves. This section will walk through what you can expect from the code snippets and examples.

1.4.1 Command-Line Snippets

Our command-line code snippets will assume you are using `bash` or a compatible shell like `sh` or `zsh`. On Windows, the shell can be accessed through Windows Subsystem for Linux. All these shells behave similarly, and we will be using code snippets prefixed by a `$` to indicate commands being entered into the Unix shell:

```
$ ls
build.sc
foo
mill

$ find . -type f
.
./build.sc
./foo/src/Example.scala
./mill
```
</> *1.2.bash*

In each case, the command entered is on the line prefixed by `$`, followed by the expected output of the command, and separated from the next command by an empty line.

1.4.2 Scala REPL Snippets

Within Scala, the simplest way to write code is in the Scala REPL (Read-Eval-Print-Loop). This is an interactive command-line that lets you enter lines of Scala code to run and immediately see their output. In this book, we will be using the Ammonite Scala REPL, and code snippets to be entered into the REPL are prefixed by @:

```
@ 1 + 1
res0: Int = 2

@ println("Hello World")
Hello World
```
</> *1.3.scala*

In each case, the command entered is on the line prefixed by @, with the following lines being the expected output of the command. The value of the entered expression may be implicitly printed to the terminal, as is the case for the `1 + 1` snippet above, or it may be explicitly printed via `println`.

The Ammonite Scala REPL also supports multi-line input, by enclosing the lines in a curly brace `{}` block:

```
@ {
  println("Hello" + (" " * 5) + "World")
  println("Hello" + (" " * 10) + "World")
  println("Hello" + (" " * 15) + "World")
  }
Hello      World
Hello           World
Hello                World
```
</> 1.4.scala

This is useful when we want to ensure the code is run as a single unit, rather than in multiple steps with a delay between them while the user is typing. Installation of Ammonite will be covered in **Chapter 2: Setting Up**.

1.4.3 Source Files

Many examples in this book require source files on disk: these may be run as scripts, or compiled and run as part of a larger project. All such snippets contain the name of the file in the top-right corner:

```
import mill._, scalalib._                                            build.sc

object foo extends ScalaModule {
  def scalaVersion = "2.13.2"
}
```
</> 1.5.scala

```
package foo                                            foo/src/Example.scala
object Example {
  def main(args: Array[String]): Unit = {
    println("Hello World")
  }
}
```
</> 1.6.scala

1.4.4 Diffs

We will illustrate changes to the a file via *diffs*. A diff is a snippet of code with + and - indicating the lines that were added and removed:

```scala
    def hello() = {
-       "Hello World!"
+       doctype("html")(
+         html(
+           head(),
+           body(
+             h1("Hello!"),
+             p("World")
+           )
+         )
+       )
    }                                                          </> 1.7.scala
```

The above diff represents the removal of one line - `"Hello World!"` - and the addition of 9 lines of code in its place. This helps focus your attention on the changes we are making to a program. After we have finished walking through a set of changes, we will show the full code for the files we were modifying, for easy reference.

1.5 Online Materials

The following Github repository acts as an online hub for all *Hands-on Scala* notes, errata, discussion, materials, and code examples:

- https://github.com/handsonscala/handsonscala

1.5.1 Code Snippets

Every code snippet in this book is available in the `snippets/` folder of the *Hands-on Scala* online repository:

- https://github.com/handsonscala/handsonscala/blob/v1/snippets

For example the code snippet with the following tag:

- </> `1.1.scala`

Is available at the following URL:

- https://github.com/handsonscala/handsonscala/blob/v1/snippets/1.1.scala

This lets you copy-paste code where convenient, rather than tediously typing out the code snippets by hand. Note that these snippets may include diffs and fragments that are not executable on their own. For executable examples, this book also provides complete Executable Code Examples (*1.5.2*).

1.5.2 Executable Code Examples

The code presented in this book is executable, and by following the instructions and code snippets in each chapter you should be able to run and reproduce all examples shown. *Hands-on Scala* also provides a set of complete executable examples online at:

- https://github.com/handsonscala/handsonscala/blob/v1/examples

Each of the examples in the `handsonscala/handsonscala` repository contains a `readme.md` file containing the command necessary to run that example. Throughout the book, we will refer to the online examples via callouts such as:

> **See example 6.1 - MergeSort**

As we progress through each chapter, we will often start from an initial piece of code and modify it via Diffs (*1.4.4*) or Code Snippets (*1.5.1*) to produce the final program. The intermediate programs would be too verbose to show in full at every step in the process, but these executable code examples give you a chance to see the complete working code at each stage of modification.

Each example is fully self-contained: following the setup in **Chapter 2: Setting Up**, you can run the command in each folder's `readme.md` and see the code execute in a self-contained fashion. You can use the working code as a basis for experimentation, or build upon it to create your own programs and applications. All code snippets and examples in this book are MIT licensed.

1.5.3 Exercises

Starting from chapter 5, every chapter come with some exercises at the end:

> **Exercise:** Tries can come in both mutable and immutable variants. Define an `ImmutableTrie` class that has the same methods as the `Trie` class we discussed in this chapter, but instead of a `def add` method it should take a sequence of strings during construction and construct the data structure without any use of `var`s or `mutable` collections.
>
> **See example 6.7 - ImmutableTrie**

The goal of these exercises is to synthesize what you learned in the chapter into useful skills. Some exercises ask you to make use of what you learned to write new code, others ask you to modify the code presented in the chapter, while others ask you to combine techniques from multiple chapters to achieve some outcome. These will help you consolidate what you learned and build a solid foundation that you can apply to future tasks and challenges.

The solutions to these exercises are also available online as executable code examples.

1.5.4 Resources

The last set of files in the `handsonscala/handsonscala` Github respository are the *resources*: sample data files that are used to exercise the code in a chapter. These are available at:

- https://github.com/handsonscala/handsonscala/blob/v1/resources

For the chapters that make use of these resource files, you can download them by going to the linked file on Github, clicking the `Raw` button to view the raw contents of the file in the browser, and then `Cmd-S`/`Ctrl-S` to save the file to your disk for your code to access.

1.5.5 Online Discussion

For further help or discussion about this book, feel free to visit our online chat room below. There you may be able to find other readers to compare notes with or discuss the topics presented in this book:

- http://www.handsonscala.com/chat

There are also chapter-specific discussion threads, which will be linked to at the end of each chapter. You can use these threads to discuss topics specific to each chapter, without it getting mixed up in other discussion. A full listing of these chapter-specific discussion threads can be found at the following URLs:

- http://www.handsonscala.com/discuss (listing of all chapter discussions)
- http://www.handsonscala.com/discuss/2 (chapter 2 discussion)

1.6 Conclusion

This first chapter should have given you an idea of what this book is about, and what you can expect working your way through it. Now that we have covered how this book works, we will proceed to set up your Scala development environment that you will be using for the rest of this book.

> **Discuss Chapter 1 online at https://www.handsonscala.com/discuss/1**

2

Setting Up

```
$ amm
Loading...
Welcome to the Ammonite Repl 2.2.0 (Scala 2.13.2 Java 11.0.7)
@ 1 + 1
res0: Int = 2

@ println("hello world" + "!" * 10)
hello world!!!!!!!!!!                                          </> 2.1.scala
```

Snippet 2.1: getting started with the Ammonite Scala REPL

In this chapter, we will set up a simple Scala programming environment, giving you the ability to write, run, and test your Scala code. We will use this setup throughout the rest of the book. It will be a simple setup, but enough so you can get productive immediately with the Scala language.

Setting up your development environment is a crucial step in learning a new programming language. Make sure you get the setup in this chapter working. If you have issues, come to the online chat room https://www.handsonscala.com/chat to get help resolving them so you can proceed with the rest of the book in peace without tooling-related distractions.

We will be installing following tools for writing and running Scala code:

- Java, the underlying runtime which Scala runs on
- Ammonite, a lightweight REPL and script runner
- Mill, a build tool for larger Scala projects
- Intellij IDEA, an integrated development environment that supports Scala
- VSCode, a lightweight text editor with support for Scala

These tools will be all you need from your first line of Scala code to building and deploying production systems in Scala.

2.1 Windows Setup (Optional)

If you are on Windows, the easiest way to get started using Scala is to use the *Windows Subsystem for Linux 2* (WSL2) to provide a unix-like environment to run your code in. This can be done by following the documentation on the Microsoft website:

- https://docs.microsoft.com/en-us/windows/wsl/wsl2-install

WSL2 allows you to choose which Linux environment to host on your Windows computer. For this book, we will be using Ubuntu 18.04 LTS.

Completing the setup, you should have a Ubuntu terminal open with a standard linux filesystem and your Windows filesystem available under the `/mnt/c/` folder:

```
$ cd /mnt/c

$ ls
'Documents and Settings'    PerfLogs        'Program Files (x86)'    Recovery
'Program Files'             ProgramData      Recovery.txt            Users
...                                                                </> 2.2.bash
```

The files in `/mnt/c/` are shared between your Windows environment and your Linux environment:

- You can edit your code on Windows, and run it through the terminal on Linux.

- You can generate files on disk on Linux, and view them in the Windows Explorer

Many of the chapters in this book assume you are running your code in WSL2's Ubuntu/Linux environment, while graphical editors like IntelliJ or VSCode will need to be running on your Windows environment, and WSL2 allows you to swap between Linux and Windows seamlessly. While the Scala language can also be developed directly on Windows, using WSL2 will allow you to avoid compatibility issues and other distractions as you work through this book.

2.2 Installing Java

Scala is a language that runs on the Java Virtual Machine (JVM), and needs Java pre-installed in order to run. To check if you have Java installed, open up your command line (The `Terminal` app on Mac OS-X, WSL2/Ubuntu on Windows) and type in the `java -version` command. If you see the following output (or something similar) it means you already have Java installed:

```
$ java -version
openjdk version "11.0.7" 2020-04-14
OpenJDK Runtime Environment AdoptOpenJDK (build 11.0.7+9)
OpenJDK 64-Bit Server VM AdoptOpenJDK (build 11.0.7+9, mixed mode)
```
</> 2.3.bash

If you already have Java, you can skip forward to Installing Ammonite (*2.3*). On the other hand, if you see something like the following, it means you do not have Java installed yet:

```
$ java -version
-bash: java: command not found
```
</> 2.4.bash

You can download and install a version of the JVM (we will be using version 11 in our examples) via one of the following websites:

- https://adoptopenjdk.net/?variant=openjdk11&jvmVariant=hotspot
- https://docs.aws.amazon.com/corretto/latest/corretto-11-ug/downloads-list.html

The installation instructions vary per operating system, but there are instructions provided for Windows, Mac OS-X, and different flavors of Linux (`.deb` and `.rpm` bundles). Once you have downloaded and installed Java, go back to your command line and make sure that running the `java -version` command correctly produces the above output.

If you are installing Java through the terminal, e.g. on a WSL Ubuntu distribution or on a headless server, you can do so through your standard package manager. e.g. on Ubuntu 18.04 that would mean the following commands:

```
$ sudo apt update

$ sudo apt install default-jdk

$ java -version
openjdk version "11.0.6" 2020-01-14
OpenJDK Runtime Environment (build 11.0.6+10-post-Ubuntu1ubuntu118.04.1)
OpenJDK 64-Bit Server VM (build 11.0.6+10-post-Ubuntu-1ubuntu118.04.1, ...)
```
</> 2.5.bash

Java versions have a high degree of compatibility, so as long as you have some version of Java installed that should be enough to make the examples in this book work regardless of which specific version it is.

2.3 Installing Ammonite

On Mac OS-X, Linux, and Windows WSL2, we can install Ammonite via the following command line commands:

```bash
$ sudo curl -L https://github.com/lihaoyi/Ammonite/releases/download/2.2.0/2.13-2.2.0 \
  -o /usr/local/bin/amm

$ sudo chmod +x /usr/local/bin/amm

$ amm
```
</> 2.6.bash

This should open up the following Ammonite Scala REPL:

```scala
Loading...
Welcome to the Ammonite Repl 2.2.0 (Scala 2.13.2 Java 11.0.7)
@
```
</> 2.7.scala

Once you see this output, it means you are ready to go. You can exit Ammonite using `Ctrl-D`.

On Max OS-X, Ammonite is also available through the Homebrew package manager via `brew install ammonite-repl`

2.3.1 The Scala REPL

The Ammonite REPL is an interactive Scala command-line in which you can enter code expressions and have their result printed:

```scala
@ 1 + 1
res0: Int = 2

@ "i am cow".substring(2, 4)
res1: String = "am"
```
</> 2.8.scala

Invalid code prints an error:

```scala
@ "i am cow".substing(2, 3)
cmd0.sc:1: value substing is not a member of String
did you mean substring?
val res0 = "i am cow".substing(2, 3)
                      ^
Compilation Failed
```
</> 2.9.scala

You can use tab-completion after a `.` to display the available methods on a particular object, a partial method name to filter that listing, or a complete method name to display the method signatures:

```
@ "i am cow".<tab>
...
exists                  maxOption               stripSuffix             //
filter                  min                     stripTrailing
filterNot               minBy                   subSequence
find                    minByOption             substring


@ "i am cow".sub<tab>
subSequence    substring


@ "i am cow".substring<tab>
def substring(x$1: Int): String
def substring(x$1: Int, x$2: Int): String                    </> 2.10.scala
```

If a REPL command is taking too long to run, you can kill it via `Ctrl-C`:

```
@ while (true) { Thread.sleep(1000); println(1 + 1) } // loop forever
2
2
2
2
2
<Ctrl-C>
Interrupted! (`repl.lastException.printStackTrace` for details)

@                                                            </> 2.11.scala
```

2.3.2 Scala Scripts

In addition to providing a REPL, Ammonite can run Scala Script files. A Scala Script is any file containing Scala code, ending in `.sc`. Scala Scripts are a lightweight way of running Scala code that is more convenient, though less configurable, than using a fully-featured build tool like Mill.

For example, we can create the following file `myScript.sc`, using any text editor of your choice (Vim, Sublime Text, VSCode, etc.):

```
                                                             myScript.sc
println(1 + 1) // 2

println("hello" + " " + "world") // hello world

println(List("I", "am", "cow")) // List(I,am,cow)          </> 2.12.scala
```

Note that in scripts, you need to `println` each expression since scripts do not echo out their values. After that, you can then run the script via `amm myScript.sc`:

```bash
$ amm myScript.sc
Compiling /Users/lihaoyi/myScript.sc
2
hello world
List(I, am, cow)                                                </> 2.13.bash
```

The first time you run the script file, it will take a moment to compile the script to an executable. Subsequent runs will be faster since the script is already compiled.

2.3.2.1 Watching Scripts

If you are working on a single script, you can use the `amm -w` or `amm --watch` command to watch a script and re-run it when things change:

```bash
$ amm -w myScript.sc
2
hello world
am
Watching for changes to 2 files... (Ctrl-C to exit)          </> 2.14.bash
```

Now whenever you make changes to the script file, it will automatically get re-compiled and re-run. This is much faster than running it over and over manually, and is convenient when you are actively working on a single script to try and get it right.

You can edit your Scala scripts with whatever editor you feel comfortable with: IntelliJ (*2.5.1*), VSCode (*2.5.3*), or any other text editor.

2.3.3 Using Scripts from the REPL

You can open up a REPL with access to the functions in a Scala Script by running `amm` with the `--predef` flag. For example, given the following script:

```scala
def hello(n: Int) = {                                          myScript.sc
  "hello world" + "!" * n
}                                                              </> 2.15.scala
```

You can then open a REPL with access to it as follows:

```bash
$ amm --predef myScript.sc
Loading...
Welcome to the Ammonite Repl 2.2.0 (Scala 2.13.2 Java 11.0.7)
@ hello(12)
res0: String = "hello world!!!!!!!!!!!!"                       </> 2.16.bash
```

This is convenient when your code snippet is large enough that you want to save it to a file and edit it in a proper editor, but you still want to use the Scala REPL to interactively test its behavior.

Note that if you make changes to the script file, you need to exit the REPL using `Ctrl-D` and re-open it to make use of the modified script. You can also combine `--predef` with `--watch`/`-w`, in which case when you exit with `Ctrl-D` it will automatically restart the REPL if the script file has changed.

2.4 Installing Mill

Mill is a build tool for Scala projects, designed for working with larger Scala projects. While the Ammonite REPL and scripts are great for small pieces of code, they lack support for things like running unit tests, packaging your code, deployment, and other such tasks. For larger projects that require such things, you need to use a build tool like Mill.

2.4.1 Mill Projects

The easiest way to get started with Mill is to download the example project:

```bash
$ curl -L https://github.com/lihaoyi/mill/releases/download/0.8.0/0.8.0-example-3.zip \
   -o example-3.zip

$ unzip example-3.zip

$ cd example-3

$ find . -type f
./build.sc
./mill
./foo/test/src/ExampleTests.scala
./foo/src/Example.scala
```
</> 2.17.bash

You can see that the example project has 4 files. A `build.sc` file that contains the project definition, defining a module `foo` with a test module `test` inside:

```scala
import mill._, scalalib._                                    build.sc

object foo extends ScalaModule {
  def scalaVersion = "2.13.2"
  object test extends Tests {
    def ivyDeps = Agg(ivy"com.lihaoyi::utest:0.7.4")
    def testFrameworks = Seq("utest.runner.Framework")
  }
}
```
</> 2.18.scala

The `test` module definition above comes with a dependency on one third party library: `ivy"com.lihaoyi::utest:0.7.4"`. We will see other libraries as we progress through the book and how to use them in our Mill projects.

The Scala code for the `foo` module lives inside the `foo/src/` folder:

```scala
package foo                                        foo/src/Example.scala
object Example {
  def main(args: Array[String]): Unit = {
    println(hello())
  }
  def hello(): String = "Hello World"
}                                                              </> 2.19.scala
```

While the Scala code for the `foo.test` test module lives inside the `foo/test/src/` folder:

```scala
package foo                                   foo/test/src/ExampleTests.scala
import utest._
object ExampleTests extends TestSuite {
  def tests = Tests {
    test("hello") {
      val result = Example.hello()
      assert(result == "Hello World")
      result
    }
  }
}                                                              </> 2.20.scala
```

Lastly, the example project contains a `mill` file. You can use the `mill` file to compile and run the project, via `./mill ...`:

```bash
$ ./mill foo.compile
Compiling /Users/lihaoyi/test2/example-1/build.sc
...
7 warnings found
[info] Compiling 1 Scala source to /Users/lihaoyi/test2/example-1/out/foo/compile/dest...
[info] Done compiling.

$ ./mill foo.run
Hello World                                                   </> 2.21.bash
```

Note that the first time you run `./mill`, it will take a few seconds to download the correct version of Mill for you to use. While above we run both `./mill foo.compile` and `./mill foo.run`, if you want to run your code you can always just run `./mill foo.run`. Mill will automatically re-compile your code if necessary before running it.

To use Mill in any other project, or to start a brand-new project using Mill, it is enough to copy over the `mill` script file to that project's root directory. You can also download the startup script via:

```
$ curl -L https://github.com/lihaoyi/mill/releases/download/0.8.0/0.8.0 -o mill

$ chmod +x mill
```
</> 2.22.bash

2.4.2 Running Unit Tests

To get started with testing in Mill, you can run `./mill foo.test`:

```
$ ./mill foo.test
------------------------------ Running Tests ------------------------------
+ foo.ExampleTests.hello 10ms  Hello World
```
</> 2.23.bash

This shows the successful result of the one test that comes built in to the example repository.

2.4.3 Creating a Stand-Alone Executable

So far, we have only been running code within the Mill build tool. But what if we want to prepare our code to run without Mill, e.g. to deploy it to production systems? To do so, you can run `./mill foo.assembly`:

```
$ ./mill foo.assembly
```

This creates an `out.jar` file that can be distributed, deployed and run without the Mill build tool in place. By default, Mill creates the output for the `foo.assembly` task in `out/foo/assembly/dest`, but you can use `./mill show` to print out the full path:

```
$ ./mill show foo.assembly
"ref:18e58778:/Users/lihaoyi/test/example-3/out/foo/assembly/dest/out.jar"
```
</> 2.24.scala

You can run the executable assembly to verify that it does what you expect:

```
$ out/foo/assembly/dest/out.jar
Hello World
```
</> 2.25.bash

Now your code is ready to be deployed!

In general, running Scala code in a Mill project requires a bit more setup than running it interactively in the Ammonite Scala REPL or Scala Scripts, but the ability to easily test and package your code is crucial for any production software.

2.5 IDE Support

The most common editors used for working with Scala programs is IntelliJ or VSCode. This section will walk you through installing both of them, but you only really need to install whichever one you prefer to make your way through this book.

2.5.1 Installing IntelliJ for Scala

You can install IntelliJ from the following website. The free Community edition will be enough.

- https://www.jetbrains.com/idea/download

Next, we need to install the IntelliJ Scala plugin, either via the loading screen:
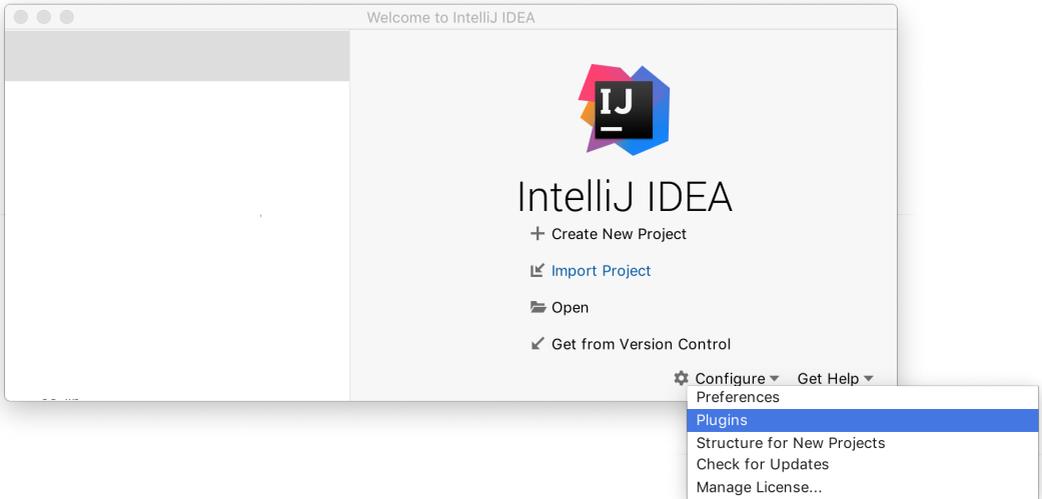
or via the menu bar

From there, go to the `Plugins` page:
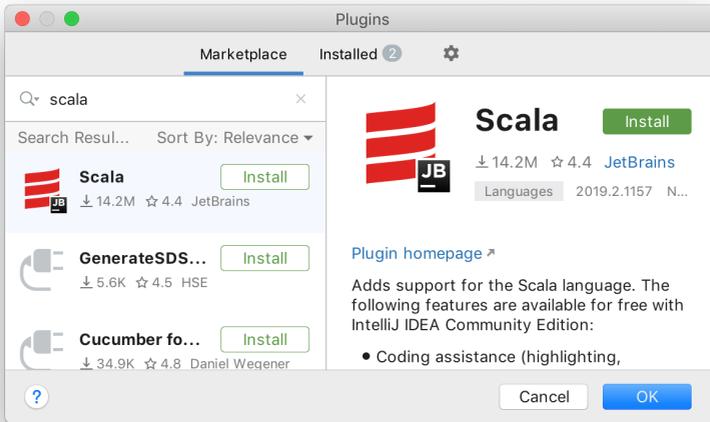
Search for `Scala` and click `Install`. You will then need to re-start the editor.

2.5.2 Integrating IntelliJ with Mill

Once you have IntelliJ installed on your machine, you can load your Mill project via the following terminal command:

```
$ ./mill mill.scalalib.GenIdea/idea
```

Next, use IntelliJ's `File` / `Open` menu item and select the folder your `build.sc` file is in. That will open up the Mill project, with IntelliJ providing code assistance when editing all the code:
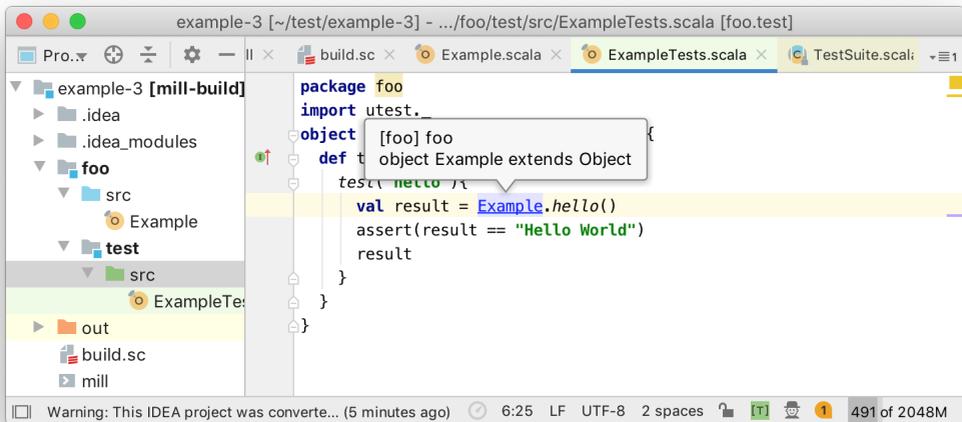
You may see a `Project JDK is not defined: Setup JDK` prompt: in that case, click the `Setup JDK` link and select the version of Java that you installed earlier (*2.2*). Note that every time you make changes to add dependencies or add new modules to your `build.sc`, you need to re-run the `./mill mill.scalalib.GenIdea/idea` command, and restart IntelliJ to have it pick up the changes.

2.5.3 Visual Studio Code Support

Scala also comes with support for the Visual Studio Code text editor, via the Metals plugin:

- https://code.visualstudio.com/
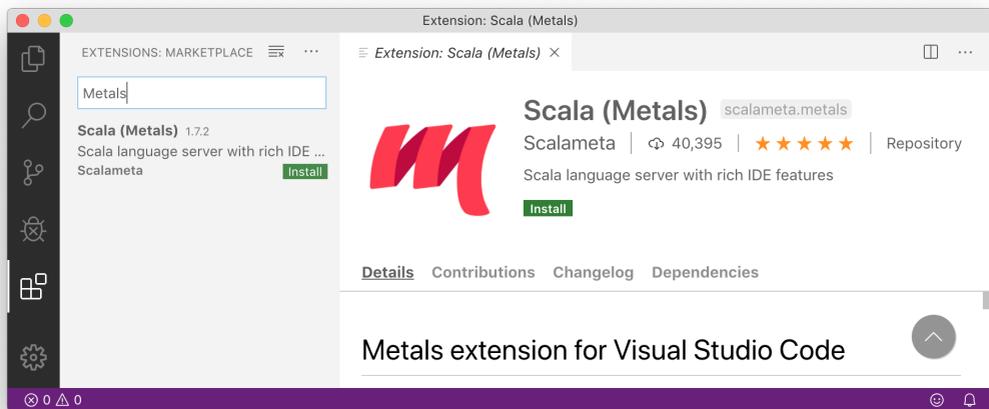
To use VSCode within a Mill build, we need to specify the Mill version in a `.mill-version` file. Then you can open the folder that your Mill build is in and select `Import build`:

```
$ echo "0.8.0" > .mill-version
```
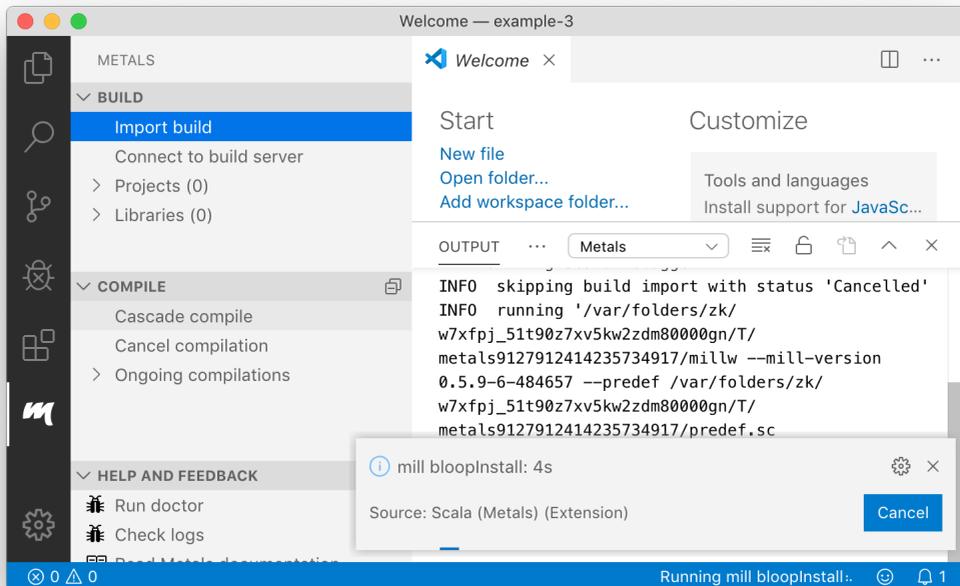

Sometimes the import may time out if your computer is slow or under load, and may need to be retried. If you have trouble getting the import to work even after retrying, you can try using a more recent version of the Metals plugin. Use `F1`, search for `Open Settings (UI)`, and update `Extensions > Metals > Server Version` e.g. to the latest SNAPSHOT version on the Metals/VSCode website:

- https://scalameta.org/metals/docs/editors/vscode.html#using-latest-metals-snapshot

You can also try enabling VSCode Remote Development, by using `F1` and searching for `Remote-WSL: Reopen in WSL`. This will run the VSCode code analysis logic inside the Ubuntu virtual machine, which may be more reliable than doing so directly on the Windows host environment:

- https://code.visualstudio.com/docs/remote/wsl

Once complete, you should be able to show method signature by mousing over them with `Cmd` or `Ctrl`.

```scala
package foo
import utest._

object ExampleTests extends TestSuite{
  def tests = Tests{
    test("hello"){
      val result = Example.hello()
      assert(result == "Hello World")
      result
    }
  }
}
```

Metals also supports other editors such as Vim, Sublime Text, Atom, and others. For more details, refer to their documentation for how to install the relevant editor plugin:

- https://scalameta.org/metals/docs/editors/overview.html

2.6 Conclusion

By now, you should have three main things set up:

- Your Ammonite Scala REPL and Script Runner, that you can run via `amm` or `amm myScript.sc`
- A Mill example project, which you can run via `./mill foo.run` or test via `./mill foo.test`
- Either IntelliJ or VSCode support for your Mill example projects

We will be using Mill as the primary build tool throughout this book, as it is the easiest to get started with. You may also encounter alternate build tools in the wild:

- SBT: https://www.scala-sbt.org/
- Gradle: https://docs.gradle.org/current/userguide/scala_plugin.html
- Maven: https://docs.scala-lang.org/tutorials/scala-with-maven.html

Before you move on to the following chapters, take some time to experiment with these tools: write some code in the Ammonite REPL, create some more scripts, add code and tests to the Mill example projects and run them. These are the main tools that we will use throughout this book.

> **Discuss Chapter 2 online at https://www.handsonscala.com/discuss/2**

3

Basic Scala

```scala
for (i <- Range.inclusive(1, 100)) {
  println(
    if (i % 3 == 0 && i % 5 == 0) "FizzBuzz"
    else if (i % 3 == 0) "Fizz"
    else if (i % 5 == 0) "Buzz"
    else i
  )
}                                              </> 3.1.scala
```

Snippet 3.1: the popular "FizzBuzz" programming challenge, implemented in Scala

This chapter is a quick tour of the Scala language. For now we will focus on the basics of Scala that are similar to what you might find in any mainstream programming language.

The goal of this chapter is to get familiar you enough that you can take the same sort of code you are used to writing in some other language and write it in Scala without difficulty. This chapter will not cover more Scala-specific programming styles or language features: those will be left for **Chapter 5: Notable Scala Features**.

For this chapter, we will write our code in the Ammonite Scala REPL:

```bash
$ amm
Loading...
Welcome to the Ammonite Repl 2.2.0 (Scala 2.13.2 Java 11.0.7)
@
```
<div align="right"></> 3.2.bash</div>

3.1 Values

3.1.1 Primitives

Scala has the following sets of primitive types:

Type	Values
Byte	-128 to 127
Short	-32,768 to 32,767
Int	-2,147,483,648 to 2,147,483,647
Long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Type	Values
Boolean	true, false
Char	'a', '0', 'Z', '包', ...
Float	32-bit Floating point
Double	64-bit Floating point

These types are identical to the primitive types in Java, and would be similar to those in C#, C++, or any other statically typed programming language. Each type supports the typical operations, e.g. booleans support boolean logic `||` `&&`, numbers support arithmetic `+` `-` `*` `/` and bitwise operations `|` `&`, and so on. All values support `==` to check for equality and `!=` to check for inequality.

Numbers default to 32-bit `Int`s. Precedence for arithmetic operations follows other programming languages: `*` and `/` have higher precedence than `+` or `-`, and parentheses can be used for grouping.

```scala
@ 1 + 2 * 3
res0: Int = 7

@ (1 + 2) * 3
res1: Int = 9
```
<div align="right"></> 3.3.scala</div>

`Int`s are signed and wrap-around on overflow, while 64-bit `Long`s suffixed with `L` have a bigger range and do not overflow as easily:

```scala
@ 2147483647
res2: Int = 2147483647
```
```scala
@ 2147483647L
res4: Long = 2147483647L
```
```scala
@ 2147483647 + 1
res3: Int = -2147483648                    </> 3.4.scala
```
```scala
@ 2147483647L + 1L
res5: Long = 2147483648L                    </> 3.5.scala
```

Apart from the basic operators, there are a lot of useful methods on `java.lang.Integer` and `java.lang.Long`:

```scala
@ java.lang.Integer.<tab>
BYTES                   decode                  numberOfTrailingZeros
signum                  MAX_VALUE               divideUnsigned
getInteger              parseUnsignedInt        toBinaryString
...

@ java.lang.Integer.toBinaryString(123)
res6: String = "1111011"

@ java.lang.Integer.numberOfTrailingZeros(24)
res7: Int = 3                                               </> 3.6.scala
```

64-bit `Double`s are specified using the `1.0` syntax, and have a similar set of arithmetic operations. You can also use the `1.0F` syntax to ask for 32-bit `Float`s:

```scala
@ 1.0 / 3.0
res8: Double = 0.3333333333333333
```

```scala
@ 1.0F / 3.0F
res9: Float = 0.33333334F                                  </> 3.7.scala
```

32-bit `Float`s take up half as much memory as 64-bit `Double`s, but are more prone to rounding errors during arithmetic operations. `java.lang.Float` and `java.lang.Double` have a similar set of useful operations you can perform on `Float`s and `Double`s.

3.1.2 Strings

`String`s in Scala are arrays of 16-bit `Char`s:

```scala
@ "hello world"
res10: String = "hello world"                              </> 3.8.scala
```

`String`s can be sliced with `.substring`, constructed via concatenation using `+`, or via string interpolation by prefixing the literal with `s"..."` and interpolating the values with `$` or `${...}`:

```
@ "hello world".substring(0, 5)
res11: String = "hello"

@ "hello world".substring(5, 10)
res12: String = " worl"
```
                                          </> 3.9.scala

```
@ "hello" + 1 + " " + "world" + 2
res13: String = "hello1 world2"

@ val x = 1; val y = 2

@ s"Hello $x World $y"
res15: String = "Hello 1 World 2"

@ s"Hello ${x + y} World ${x - y}"
res16: String = "Hello 3 World -1"
```
                                         </> 3.10.scala

3.1.3 Local Values and Variables

You can name local values with the `val` keyword:

```
@ val x = 1

@ x + 2
res18: Int = 3
```
                                                              </> 3.11.scala

Note that `val`s are immutable: you cannot re-assign the `val` x to a different value after the fact. If you want a local variable that can be re-assigned, you must use the `var` keyword.

```
@ x = 3
cmd41.sc:1: reassignment to val
val res26 = x = 3
              ^
Compilation Failed
```
                                         </> 3.12.scala

```
@ var y = 1

@ y + 2
res20: Int = 3

@ y = 3

@ y + 2
res22: Int = 5
```
                                         </> 3.13.scala

In general, you should try to use `val` where possible: most named values in your program likely do not need to be re-assigned, and using `val` helps prevent mistakes where you re-assign something accidentally. Use `var` only if you are sure you will need to re-assign something later.

Both `val`s and `var`s can be annotated with explicit types. These can serve as documentation for people reading your code, as well as a way to catch errors if you accidentally assign the wrong type of value to a variable

Chapter 3 Basic Scala   42

```
@ val x: Int = 1
```

```
@ var s: String = "Hello"
s: String = "Hello"
```

```
@ s = "World"
```

</> 3.14.scala

```
@ val z: Int = "Hello"
cmd33.sc:1: type mismatch;
 found    : String("Hello")
 required: Int
val z: Int = "Hello"
                 ^
Compilation Failed
```

</> 3.15.scala

3.1.4 Tuples

Tuples are fixed-length collections of values, which may be of different types:

```
@ val t = (1, true, "hello")
t: (Int, Boolean, String) = (1, true, "hello")

@ t._1
res27: Int = 1

@ t._2
res28: Boolean = true

@ t._3
res29: String = "hello"
```

</> 3.16.scala

Above, we are storing a tuple into the local value `t` using the `(a, b, c)` syntax, and then using `._1`, `._2` and `._3` to extract the values out of it. The fields in a tuple are immutable.

The type of the local value `t` can be annotated as a tuple type:

```
@ val t: (Int, Boolean, String) = (1, true, "hello")
```

You can also use the `val (a, b, c) = t` syntax to extract all the values at once, and assign them to meaningful names:

```
@ val (a, b, c) = t
a: Int = 1
b: Boolean = true
c: String = "hello"
```

</> 3.17.scala

```
@ a
res31: Int = 1

@ b
res32: Boolean = true

@ c
res33: String = "hello"
```

</> 3.18.scala

Tuples come in any size from 1 to 22 items long:

```scala
@ val t = (1, true, "hello", 'c', 0.2, 0.5f, 12345678912345L)
t: (Int, Boolean, String, Char, Double, Float, Long) = (
  1,
  true,
  "hello",
  'c',
  0.2,
  0.5F,
  12345678912345L
)
```
<div align="right"></> 3.19.scala</div>

Most tuples should be relatively small. Large tuples can easily get confusing: while working with `._1` `._2` and `._3` is probably fine, when you end up working with `._11` `._13` it becomes easy to mix up the different fields. If you find yourself working with large tuples, consider defining a Class (*3.4*) or Case Class that we will see in **Chapter 5: Notable Scala Features**.

3.1.5 Arrays

Arrays are instantiated using the `Array[T](a, b, c)` syntax, and entries within each array are retrieved using `a(n)`:

```scala
@ val a = Array[Int](1, 2, 3, 4)

@ a(0) // first entry, array indices start from 0
res36: Int = 1

@ a(3) // last entry
res37: Int = 4

@ val a2 = Array[String]("one", "two", "three", "four")
a2: Array[String] = Array("one", "two", "three", "four")

@ a2(1) // second entry
res39: String = "two"
```
<div align="right"></> 3.20.scala</div>

The type parameter inside the square brackets `[Int]` or `[String]` determines the type of the array, while the parameters inside the parenthesis `(1, 2, 3, 4)` determine its initial contents. Note that looking up an Array by index is done via parentheses `a(3)` rather than square brackets `a[3]` as is common in many other programming languages.

You can omit the explicit type parameter and let the compiler infer the Array's type, or create an empty array of a specified type using `new Array[T](length)`, and assign values to each index later:

```scala
@ val a = Array(1, 2, 3, 4)
a: Array[Int] = Array(1, 2, 3, 4)


@ val a2 = Array(
    "one", "two",
    "three", "four"
  )
a2: Array[String] = Array(
  "one", "two",
  "three", "four"
)                                    </> 3.21.scala
```

```scala
@ val a = new Array[Int](4)
a: Array[Int] = Array(0, 0, 0, 0)


@ a(0) = 1


@ a(2) = 100


@ a
res45: Array[Int] = Array(1, 0, 100, 0)


                                     </> 3.22.scala
```

For `Array`s created using `new Array`, all entries start off with the value `0` for numeric arrays, `false` for `Boolean` arrays, and `null` for `String`s and other types. `Array`s are mutable but fixed-length: you can change the value of each entry but cannot change the number of entries by adding or removing values. We will see how to create variable-length collections later in **Chapter 4: Scala Collections**.

Multi-dimensional arrays, or arrays-of-arrays, are also supported:

```scala
@ val multi = Array(Array(1, 2), Array(3, 4))
multi: Array[Array[Int]] = Array(Array(1, 2), Array(3, 4))


@ multi(0)(0)
res47: Int = 1


@ multi(0)(1)
res48: Int = 2


@ multi(1)(0)
res49: Int = 3


@ multi(1)(1)
res50: Int = 4                                              </> 3.23.scala
```

Multi-dimensional arrays can be useful to represent grids, matrices, and similar values.

3.1.6 Options

Scala's `Option[T]` type allows you to represent a value that may or may not exist. An `Option[T]` can either be `Some(v: T)` indicating that a value is present, or `None` indicating that it is absent:

```scala
@ def hello(title: String, firstName: String, lastNameOpt: Option[String]) = {
    lastNameOpt match {
      case Some(lastName) => println(s"Hello $title. $lastName")
      case None => println(s"Hello $firstName")
    }
  }

@ hello("Mr", "Haoyi", None)
Hello Haoyi

@ hello("Mr", "Haoyi", Some("Li"))
Hello Mr. Li
```
</> 3.24.scala

The above example shows you how to construct `Option`s using `Some` and `None`, as well as `match`ing on them in the same way. Many APIs in Scala rely on `Option`s rather than `null`s for values that may or may not exist. In general, `Option`s force you to handle both cases of present/absent, whereas when using `null`s it is easy to forget whether or not a value is null-able, resulting in confusing `NullPointerException`s at runtime. We will go deeper into pattern matching in **Chapter 5: Notable Scala Features**.

`Option`s contain some helper methods that make it easy to work with the optional value, such as `getOrElse`, which substitutes an alternate value if the `Option` is `None`:

```scala
@ Some("Li").getOrElse("<unknown>")
res54: String = "Li"

@ None.getOrElse("<unknown>")
res55: String = "<unknown>"
```
</> 3.25.scala

`Option`s are very similar to a collection whose size is `0` or `1`. You can loop over them like normal collections, or transform them with standard collection operations like `.map`.

```scala
@ def hello2(name: Option[String]) = {
    for (s <- name) println(s"Hello $s")
  }

@ hello2(None) // does nothing

@ hello2(Some("Haoyi"))
Hello Haoyi
```
</> 3.26.scala

```scala
@ def nameLength(name: Option[String]) = {
    name.map(_.length).getOrElse(-1)
  }

@ nameLength(Some("Haoyi"))
res60: Int = 5

@ nameLength(None)
res61: Int = -1
```
</> 3.27.scala

Above, we combine `.map` and `.getOrElse` to print out the length of the name if present, and otherwise print `-1`. We will learn more about collection operations in **Chapter 4: Scala Collections**.

> See example 3.1 - Values

3.2 Loops, Conditionals, Comprehensions

3.2.1 For-Loops

For-loops in Scala are similar to "foreach" loops in other languages: they directly loop over the elements in a collection, without needing to explicitly maintain and increment an index. If you want to loop over a range of indices, you can loop over a `Range` such as `Range(0, 5)`:

```scala
@ var total = 0

@ val items = Array(1, 10, 100, 1000)

@ for (item <- items) total += item

@ total
res65: Int = 1111
```
</> 3.28.scala

```scala
@ var total = 0

@ for (i <- Range(0, 5)) {
    println("Looping " + i)
    total = total + i
  }
Looping 0
Looping 1
Looping 2
Looping 3
Looping 4

@ total
res68: Int = 10
```
</> 3.29.scala

You can loop over nested `Array`s by placing multiple `<-`s in the header of the loop:

```
@ val multi = Array(Array(1, 2, 3), Array(4, 5, 6))

@ for (arr <- multi; i <- arr) println(i)
1
2
3
4
5
6
```
<div align="right">`</>` *3.30.scala*</div>

Loops can have guards using an `if` syntax:

```
@ for (arr <- multi; i <- arr; if i % 2 == 0) println(i)
2
4
6
```
<div align="right">`</>` *3.31.scala*</div>

3.2.2 If-Else

`if-else` conditionals are similar to those in any other programming language. One thing to note is that in Scala `if-else` can also be used as an expression, similar to the `a ? b : c` ternary expressions in other languages. Scala does not have a separate ternary expression syntax, and so the `if-else` can be directly used as the right-hand-side of the `total +=` below.

```
@ var total = 0

@ for (i <- Range(0, 10)) {
    if (i % 2 == 0) total += i
    else total += 2
  }

@ total
res74: Int = 30
```
<div align="right">`</>` *3.32.scala*</div>

```
@ var total = 0

@ for (i <- Range(0, 10)) {
    total += (if (i % 2 == 0) i else 2)
  }

@ total
res77: Int = 30
```
<div align="right">`</>` *3.33.scala*</div>

3.2.3 Fizzbuzz

Now that we know the basics of Scala syntax, let's consider the common "Fizzbuzz" programming challenge:

Write a short program that prints each number from 1 to 100 on a new line.

For each multiple of 3, print "Fizz" instead of the number.

For each multiple of 5, print "Buzz" instead of the number.

For numbers which are multiples of both 3 and 5, print "FizzBuzz" instead of the number.

We can accomplish this as follows:

```scala
@ for (i <- Range.inclusive(1, 100)) {
    if (i % 3 == 0 && i % 5 == 0) println("FizzBuzz")
    else if (i % 3 == 0) println("Fizz")
    else if (i % 5 == 0) println("Buzz")
    else println(i)
  }
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
...                                                          </> 3.34.scala
```

Since `if-else` is an expression, we can also write it as:

```scala
@ for (i <- Range.inclusive(1, 100)) {
    println(
      if (i % 3 == 0 && i % 5 == 0) "FizzBuzz"
      else if (i % 3 == 0) "Fizz"
      else if (i % 5 == 0) "Buzz"
      else i
    )
  }                                                          </> 3.35.scala
```

3.2.4 Comprehensions

Apart from using `for` to define loops that perform some action, you can also use `for` together with `yield` to transform a collection into a new collection:

```scala
@ val a = Array(1, 2, 3, 4)

@ val a2 = for (i <- a) yield i * i
a2: Array[Int] = Array(1, 4, 9, 16)

@ val a3 = for (i <- a) yield "hello " + i
a3: Array[String] = Array("hello 1", "hello 2", "hello 3", "hello 4")
```
</> 3.36.scala

Similar to loops, you can filter which items end up in the final collection using an `if` guard inside the parentheses:

```scala
@ val a4 = for (i <- a if i % 2 == 0) yield "hello " + i
a4: Array[String] = Array("hello 2", "hello 4")
```
</> 3.37.scala

Comprehensions can also take multiple input arrays, `a` and `b` below. This flattens them out into one final output `Array`, similar to using a nested for-loop:

```scala
@ val a = Array(1, 2); val b = Array("hello", "world")

@ val flattened = for (i <- a; s <- b) yield s + i
flattened: Array[String] = Array("hello1", "world1", "hello2", "world2")
```
</> 3.38.scala

You can also replace the parentheses `()` with curly brackets `{}` if you wish to spread out the nested loops over multiple lines, for easier reading. Note that the order of `<-`s within the nested comprehension matters, just like how the order of nested loops affects the order in which the loop actions will take place:

```scala
@ val flattened = for{
    i <- a
    s <- b
  } yield s + i
flattened: Array[String] = Array("hello1", "world1", "hello2", "world2")

@ val flattened2 = for{
    s <- b
    i <- a
  } yield s + i
flattened2: Array[String] = Array("hello1", "hello2", "world1", "world2")
```
</> 3.39.scala

We can use comprehensions to write a version of FizzBuzz that doesn't print its results immediately to the console, but returns them as a `Seq` (short for "sequence"):

```scala
@ val fizzbuzz = for (i <- Range.inclusive(1, 100)) yield {
    if (i % 3 == 0 && i % 5 == 0) "FizzBuzz"
    else if (i % 3 == 0) "Fizz"
    else if (i % 5 == 0) "Buzz"
    else i.toString
  }
fizzbuzz: IndexedSeq[String] = Vector(
  "1",
  "2",
  "Fizz",
  "4",
  "Buzz",
...                                                          </> 3.40.scala
```

We can then use the `fizzbuzz` collection however we like: storing it in a variable, passing it into methods, or processing it in other ways. We will cover what you can do with these collections later, in **_Chapter 4: Scala Collections_**.

> See example **3.2 - LoopsConditionals**

3.3 Methods and Functions

3.3.1 Methods

You can define methods using the `def` keyword:

```scala
@ def printHello(times: Int) = {
    println("hello " + times)
  }

@ printHello(1)
hello 1

@ printHello(times = 2) // argument name provided explicitly
hello 2                                                      </> 3.41.scala
```

Passing in the wrong type of argument, or missing required arguments, is a compiler error. However, if the argument has a default value, then passing it is optional.

```
@ printHello("1") // wrong type of argument
cmd128.sc:1: type mismatch;
 found   : String("1")
 required: Int
val res128 = printHello("1")
                        ^
Compilation Failed
```

</> 3.42.scala

```
@ def printHello2(times: Int = 0) = {
    println("hello " + times)
  }


@ printHello2(1)
hello 1


@ printHello2()
hello 0
```

</> 3.43.scala

3.3.1.1 Returning Values from Methods

Apart from performing actions like printing, methods can also return values. The last expression within the curly brace {} block is treated as the return value of a Scala method.

```
@ def hello(i: Int = 0) = {
    "hello " + i
  }


@ hello(1)
res96: String = "hello 1"
```

</> 3.44.scala

You can call the method and print out or perform other computation on the returned value:

```
@ println(hello())
hello 0


@ val helloHello = hello(123) + " " + hello(456)
helloHello: String = "hello 123 hello 456"


@ helloHello.reverse
res99: String = "654 olleh 321 olleh"
```

</> 3.45.scala

3.3.2 Function Values

You can define function values using the `=>` syntax. Functions values are similar to methods, in that you call them with arguments and they can perform some action or return some value. Unlike methods, functions themselves are values: you can pass them around, store them in variables, and call them later.

```scala
@ var g: Int => Int = i => i + 1

@ g(10)
res101: Int = 11

@ g = i => i * 2

@ g(10)
res103: Int = 20
```
<div align="right">`</> 3.46.scala`</div>

Note that unlike methods, function values cannot have optional arguments (i.e. with default values) and cannot take type parameters via the `[T]` syntax. When a method is converted into a function value, any optional arguments must be explicitly included, and type parameters fixed to concrete types. Function values are also anonymous, which makes stack traces involving them less convenient to read than those using methods.

In general, you should prefer using methods unless you really need the flexibility to pass as parameters or store them in variables. But if you need that flexibility, function values are a great tool to have.

3.3.2.1 Methods taking Functions

One common use case of function values is to pass them into methods that take function parameters. Such methods are often called "higher order methods". Below, we have a class `Box` with a method `printMsg` that prints its contents (an `Int`), and a separate method `update` that takes a function of type `Int => Int` that can be used to update `x`. You can then pass a function literal into `update` in order to change the value of `x`:

```scala
@ class Box(var x: Int) {
    def update(f: Int => Int) = x = f(x)
    def printMsg(msg: String) = {
      println(msg + x)
    }
  }
```
<div align="right">`</> 3.47.scala`</div>

```scala
@ val b = new Box(1)

@ b.printMsg("Hello")
Hello1

@ b.update(i => i + 5)

@ b.printMsg("Hello")
Hello6
```
<div align="right">`</> 3.48.scala`</div>

Simple functions literals like `i => i + 5` can also be written via the shorthand `_ + 5`, with the underscore `_` standing in for the function parameter.

```
@ b.update(_ + 5)

@ b.printMsg("Hello")
Hello11                                                                        </> 3.49.scala
```

This placeholder syntax for function literals also works for multi-argument functions, e.g. `(x, y) => x + y` can be written as `_ + _`.

Any method that takes a function as an argument can also be given a method reference, as long as the method's signature matches that of the function type, here `Int => Int`:

```
@ def increment(i: Int) = i + 1

@ val b = new Box(123)

@ b.update(increment) // Providing a method reference

@ b.update(x => increment(x)) // Explicitly writing out the function literal

@ b.update{x => increment(x)} // Methods taking a single function can be called with {}s

@ b.update(increment(_)) // You can also use the `_` placeholder syntax

@ b.printMsg("result: ")
result: 127                                                                    </> 3.50.scala
```

3.3.2.2 Multiple Parameter Lists

Methods can be defined to take multiple parameter lists. This is useful for writing higher-order methods that can be used like control structures, such as the `myLoop` method below:

```
@ def myLoop(start: Int, end: Int)
            (callback: Int => Unit) = {
    for (i <- Range(start, end)) {
      callback(i)
    }
  }

                              </> 3.51.scala
```

```
@ myLoop(start = 5, end = 10) { i =>
    println(s"i has value ${i}")
  }
i has value 5
i has value 6
i has value 7
i has value 8
i has value 9                       </> 3.52.scala
```

The ability to pass function literals to methods is used to great effect in the standard library, to concisely perform transformations on collections. We will see more of that in **Chapter 4: Scala Collections**.

3.4 Classes and Traits

You can define classes using the `class` keyword, and instantiate them using `new`. By default, all arguments passed into the class constructor are available in all of the class' methods: the `(x: Int)` above defines both the private fields as well as the class' constructor. `x` is thus accessible in the `printMsg` function, but cannot be accessed outside the class:

```scala
@ class Foo(x: Int) {
    def printMsg(msg: String) = {
      println(msg + x)
    }
  }
```
</> 3.53.scala

```scala
@ val f = new Foo(1)

@ f.printMsg("hello")
hello1

@ f.x
cmd120.sc:1: value x is not a member of Foo
Compilation Failed
```
</> 3.54.scala

To make x publicly accessible you can make it a `val`, and to make it mutable you can make it a `var`:

```scala
@ class Bar(val x: Int) {
    def printMsg(msg: String) = {
      println(msg + x)
    }
  }
```
</> 3.55.scala

```scala
@ val b = new Bar(1)

@ b.x
res122: Int = 1
```
</> 3.56.scala

```scala
@ class Qux(var x: Int) {
    def printMsg(msg: String) = {
      x += 1
      println(msg + x)
    }
  }
```
</> 3.57.scala

```scala
@ val q = new Qux(1)

@ q.printMsg("hello")
hello2

@ q.printMsg("hello")
hello3
```
</> 3.58.scala

You can also use `val`s or `var`s in the body of a class to store data. These get computed once when the class is instantiated:

```scala
@ class Baz(x: Int) {
    val bangs = "!" * x
    def printMsg(msg: String) = {
      println(msg + bangs)
    }
  }                          </> 3.59.scala
```

```scala
@ val z = new Baz(3)

@ z.printMsg("hello")
hello!!!
                           </> 3.60.scala
```

3.4.1 Traits

traits are similar to interfaces in traditional object-oriented languages: a set of methods that multiple classes can inherit. Instances of these classes can then be used interchangeably.

```scala
@ trait Point{ def hypotenuse: Double }

@ class Point2D(x: Double, y: Double) extends Point{
    def hypotenuse = math.sqrt(x * x + y * y)
  }

@ class Point3D(x: Double, y: Double, z: Double) extends Point{
    def hypotenuse = math.sqrt(x * x + y * y + z * z)
  }

@ val points: Array[Point] = Array(new Point2D(1, 2), new Point3D(4, 5, 6))

@ for (p <- points) println(p.hypotenuse)
2.23606797749979
8.774964387392123
                                                          </> 3.61.scala
```

Above, we have defined a Point trait with a single method def hypotenuse: Double. The subclasses Point2D and Point3D both have different sets of parameters, but they both implement def hypotenuse. Thus we can put both Point2Ds and Point3Ds into our points: Array[Point] and treat them all uniformly as objects with a def hypotenuse method, regardless of what their actual class is.

> **See example 3.4 - ClassesTraits**

3.5 Conclusion

In this chapter, we have gone through a lightning tour of the core Scala language. While the exact syntax may be new to you, the concepts should be mostly familiar: primitives, arrays, loops, conditionals, methods, and classes are part of almost every programming language. Next we will look at the core of the Scala standard library: the Scala Collections.

> **Exercise:** Define a `def flexibleFizzBuzz` method that takes a `String => Unit` callback function as its argument, and allows the caller to decide what they want to do with the output. The caller can choose to ignore the output, `println` the output directly, or store the output in a previously-allocated array they already have handy.
>
> ```scala
> @ flexibleFizzBuzz(s => {} /* do nothing
> */)
>
> @ flexibleFizzBuzz(s => println(s))
> 1
> 2
> Fizz
> 4
> Buzz
> ...
> ```
> </> 3.62.scala
>
> ```scala
> @ var i = 0
>
> @ val output = new Array[String](100)
>
> @ flexibleFizzBuzz{s =>
>     output(i) = s
>     i += 1
>   }
>
> @ output
> res125: Array[String] = Array(
>   "1",
>   "2",
>   "Fizz",
>   "4",
>   "Buzz",
> ...
> ```
> </> 3.63.scala
>
> **See example 3.5 - FlexibleFizzBuzz**

Exercise: Write a recursive method `printMessages` that can receive an array of `Msg` class instances, each with an optional `parent` ID, and use it to print out a threaded fashion. That means that child messages are print out indented underneath their parents, and the nesting can be arbitrarily deep.

```scala
class Msg(val id: Int, val parent: Option[Int], val txt: String)
def printMessages(messages: Array[Msg]): Unit = ...                    </> 3.64.scala
```

```scala
printMessages(Array(   TestPrintMessages.sc
  new Msg(0, None, "Hello"),
  new Msg(1, Some(0), "World"),
  new Msg(2, None, "I am Cow"),
  new Msg(3, Some(2), "Hear me moo"),
  new Msg(4, Some(2), "Here I stand"),
  new Msg(5, Some(2), "I am Cow"),
  new Msg(6, Some(5), "Here me moo, moo")
))                              </> 3.65.scala
```

```
#0 Hello                        expected.txt
    #1 World
#2 I am Cow
    #3 Hear me moo
    #4 Here I stand
    #5 I am Cow
        #6 Here me moo, moo

                        </> 3.66.output
```

See example **3.6 - PrintMessages**

Exercise: Define a pair of methods `withFileWriter` and `withFileReader` that can be called as shown below. Each method should take the name of a file, and a function value that is called with a `java.io.BufferedReader` or `java.io.BufferedWriter` that it can use to read or write data. Opening and closing of the reader/writer should be automatic, such that a caller cannot forget to close the file. This is similar to Python "context managers" or Java "try-with-resource" syntax.

```scala
withFileWriter("File.txt") { writer =>     TestContextManagers.sc
  writer.write("Hello\n"); writer.write("World!")
}
val result = withFileReader("File.txt") { reader =>
  reader.readLine() + "\n" + reader.readLine()
}
assert(result == "Hello\nWorld!")
                                      </> 3.67.scala
```

You can use the Java standard library APIs `java.nio.file.Files.newBufferedWriter` and `newBufferedReader` for working with file readers and writers. We will get more familiar with working with files and the filesystem in *Chapter 7: Files and Subprocesses*.

See example **3.7 - ContextManagers**

Discuss Chapter 3 online at **https://www.handsonscala.com/discuss/3**

4

Scala Collections

```scala
@ def stdDev(a: Array[Double]): Double = {
    val mean = a.sum / a.length
    val squareErrors = a.map(x => x - mean).map(x => x * x)
    math.sqrt(squareErrors.sum / a.length)
  }                                                        </> 4.1.scala
```

Snippet 4.1: calculating the standard deviation of an array using Scala Collection operations

The core of the Scala standard library is its *collections*: a common set of containers and data structures that are shared by all Scala programs. Scala's collections make it easy for you to manipulate arrays, linked lists, sets, maps and other data structures in convenient ways, providing built-in many of the data structures needed for implementing a typical application.

This chapter will walk through the common operations that apply to all collection types, before discussing the individual data structures and when you might use each of them in practice.

4.1 Operations

Scala collections provide many common operations for constructing them, querying them, or transforming them. These operations are present on the `Array`s we saw in **Chapter 3: Basic Scala**, but they also apply to all the collections we will cover in this chapter: Vectors (*4.2.1*), Sets (*4.2.3*), Maps (*4.2.4*), etc.

4.1.1 Builders

```scala
@ val b = Array.newBuilder[Int]
b: mutable.ArrayBuilder[Int] = ArrayBuilder.ofInt

@ b += 1

@ b += 2

@ b.result()
res3: Array[Int] = Array(1, 2)
```
</> 4.2.scala

Builders let you efficiently construct a collection of unknown length, "freezing" it into the collection you want at the end. This is most useful for constructing `Array`s or immutable collections where you cannot add or remove elements once the collection has been constructed.

4.1.2 Factory Methods

```scala
@ Array.fill(5)("hello") // Array with "hello" repeated 5 times
res4: Array[String] = Array("hello", "hello", "hello", "hello", "hello")

@ Array.tabulate(5)(n => s"hello $n") // Array with 5 items, each computed from its index
res5: Array[String] = Array("hello 0", "hello 1", "hello 2", "hello 3", "hello 4")

@ Array(1, 2, 3) ++ Array(4, 5, 6) // Concatenating two Arrays into a larger one
res6: Array[Int] = Array(1, 2, 3, 4, 5, 6)
```
</> 4.3.scala

Factory methods provide another way to instantiate collections: with every element the same, with each element constructed depending on the index, or from multiple smaller collections. This can be more convenient than using Builders (*4.1.1*) in many common use cases.

> **See example 4.1 - BuildersFactories**

4.1.3 Transforms

```scala
@ Array(1, 2, 3, 4, 5).map(i => i * 2) // Multiply every element by 2
res7: Array[Int] = Array(2, 4, 6, 8, 10)

@ Array(1, 2, 3, 4, 5).filter(i => i % 2 == 1) // Keep only elements not divisible by 2
res8: Array[Int] = Array(1, 3, 5)

@ Array(1, 2, 3, 4, 5).take(2) // Keep first two elements
res9: Array[Int] = Array(1, 2)

@ Array(1, 2, 3, 4, 5).drop(2) // Discard first two elements
res10: Array[Int] = Array(3, 4, 5)

@ Array(1, 2, 3, 4, 5).slice(1, 4) // Keep elements from index 1-4
res11: Array[Int] = Array(2, 3, 4)

@ Array(1, 2, 3, 4, 5, 4, 3, 2, 1, 2, 3, 4, 5, 6, 7, 8).distinct // Removes all duplicates
res12: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8)
```
</> 4.4.scala

Transforms take an existing collection and create a new collection modified in some way. Note that these transformations create copies of the collection, and leave the original unchanged. That means if you are still using the original array, its contents will not be modified by the transform:

```scala
@ val a = Array(1, 2, 3, 4, 5)
a: Array[Int] = Array(1, 2, 3, 4, 5)

@ val a2 = a.map(x => x + 10)
a2: Array[Int] = Array(11, 12, 13, 14, 15)

@ a(0) // Note that `a` is unchanged!
res15: Int = 1

@ a2(0)
res16: Int = 11
```
</> 4.5.scala

The copying involved in these collection transformations does have some overhead, but in most cases that should not cause issues. If a piece of code does turn out to be a bottleneck that is slowing down your program, you can always convert your `.map`/`.filter`/etc. transformation code into mutating operations over raw `Arrays` or In-Place Operations (*4.3.4*) over Mutable Collections (*4.3*) to optimize for performance.

> **See example 4.2 - Transforms**

4.1.4 Queries

```
@ Array(1, 2, 3, 4, 5, 6, 7).find(i => i % 2 == 0 && i > 4)
res17: Option[Int] = Some(6)


@ Array(1, 2, 3, 4, 5, 6, 7).find(i => i % 2 == 0 && i > 10)
res18: Option[Int] = None


@ Array(1, 2, 3, 4, 5, 6, 7).exists(x => x > 1) // are any elements greater than 1?
res19: Boolean = true


@ Array(1, 2, 3, 4, 5, 6, 7).exists(_ < 0) // same as a.exists(x => x < 0)
res20: Boolean = false
```

Queries let you search for elements without your collection, returning either a `Boolean` indicating if a matching element exists, or an `Option` containing the element that was found. This can make it convenient to find things inside your collections without the verbosity of writing for-loops to inspect the elements one by one.

4.1.5 Aggregations

4.1.5.1 mkString

Stringifies the elements in a collection and combines them into one long string, with the given separator. Optionally can take a start and end delimiter:

```
@ Array(1, 2, 3, 4, 5, 6, 7).mkString(",")
res21: String = "1,2,3,4,5,6,7"


@ Array(1, 2, 3, 4, 5, 6, 7).mkString("[", ",", "]")
res22: String = "[1,2,3,4,5,6,7]"
```

4.1.5.2 foldLeft

Takes a starting value and a function that it uses to combine each element of your collection with the starting value, to produce a final result:

```
@ Array(1, 2, 3, 4, 5, 6, 7).foldLeft(0)((x, y) => x + y) // sum of all elements
res23: Int = 28


@ Array(1, 2, 3, 4, 5, 6, 7).foldLeft(1)((x, y) => x * y) // product of all elements
res24: Int = 5040


@ Array(1, 2, 3, 4, 5, 6, 7).foldLeft(1)(_ * _) // same as above, shorthand syntax
res25: Int = 5040
```

In general, `foldLeft` is similar to a `for`-loop and accumulator `var`, and the above sum-of-all-elements `foldLeft` call can equivalently be written as:

```
@ {
  var total = 0
  for (i <- Array(1, 2, 3, 4, 5, 6, 7)) total += i
  total
  }
total: Int = 28
```

4.1.5.3 groupBy

Groups your collection into a `Map` of smaller collections depending on a key:

```
@ val grouped = Array(1, 2, 3, 4, 5, 6, 7).groupBy(_ % 2)
grouped: Map[Int, Array[Int]] = Map(0 -> Array(2, 4, 6), 1 -> Array(1, 3, 5, 7))

@ grouped(0)
res26: Array[Int] = Array(2, 4, 6)

@ grouped(1)
res27: Array[Int] = Array(1, 3, 5, 7)
```

> **See example 4.3 - QueriesAggregations**

4.1.6 Combining Operations

It is common to chain more than one operation together to achieve what you want. For example, here is a function that computes the standard deviation of an array of numbers:

```
@ def stdDev(a: Array[Double]): Double = {
    val mean = a.foldLeft(0.0)(_ + _) / a.length
    val squareErrors = a.map(_ - mean).map(x => x * x)
    math.sqrt(squareErrors.foldLeft(0.0)(_ + _) / a.length)
  }

@ stdDev(Array(1, 2, 3, 4, 5))
res29: Double = 1.4142135623730951

@ stdDev(Array(3, 3, 3))
res30: Double = 0.0
```

Scala collections provide a convenient helper method `.sum` that is equivalent to `.foldLeft(0.0)(_ + _)`, so the above code can be simplified to:

```scala
@ def stdDev(a: Array[Double]): Double = {
    val mean = a.sum / a.length
    val squareErrors = a.map(_ - mean).map(x => x * x)
    math.sqrt(squareErrors.sum / a.length)
  }
```
<div align="right"></> 4.12.scala</div>

As another example, here is a function that uses `.exists`, `.map` and `.distinct` to check if an incoming grid of numbers is a valid Sudoku grid:

```scala
@ def isValidSudoku(grid: Array[Array[Int]]): Boolean = {
    !Range(0, 9).exists{i =>
      val row = Range(0, 9).map(grid(i)(_))
      val col = Range(0, 9).map(grid(_)(i))
      val square = Range(0, 9).map(j => grid((i % 3) * 3 + j % 3)((i / 3) * 3 + j / 3))
      row.distinct.length != row.length ||
      col.distinct.length != col.length ||
      square.distinct.length != square.length
    }
  }
```
<div align="right"></> 4.13.scala</div>

This implementation receives a Sudoku grid, represented as a 2-dimensional `Array[Array[Int]]`. For each `i` from `0` to `9`, we pick out a single row, column, and 3x3 square. It then checks that each such row/column/square has 9 unique numbers by calling `.distinct` to remove any duplicates, and then checking if the `.length` has changed as a result of that removal.

We can test this on some example grids to verify that it works:

```
@ isValidSudoku(Array(                        @ isValidSudoku(Array(
    Array(5, 3, 4,   6, 7, 8,   9, 1, 2),        Array(5, 3, 4,   6, 7, 8,   9, 1, 2),
    Array(6, 7, 2,   1, 9, 5,   3, 4, 8),        Array(6, 7, 2,   1, 9, 5,   3, 4, 8),
    Array(1, 9, 8,   3, 4, 2,   5, 6, 7),        Array(1, 9, 8,   3, 4, 2,   5, 6, 7),

    Array(8, 5, 9,   7, 6, 1,   4, 2, 3),        Array(8, 5, 9,   7, 6, 1,   4, 2, 3),
    Array(4, 2, 6,   8, 5, 3,   7, 9, 1),        Array(4, 2, 6,   8, 5, 3,   7, 9, 1),
    Array(7, 1, 3,   9, 2, 4,   8, 5, 6),        Array(7, 1, 3,   9, 2, 4,   8, 5, 6),

    Array(9, 6, 1,   5, 3, 7,   2, 8, 4),        Array(9, 6, 1,   5, 3, 7,   2, 8, 4),
    Array(2, 8, 7,   4, 1, 9,   6, 3, 5),        Array(2, 8, 7,   4, 1, 9,   6, 3, 5),
    Array(3, 4, 5,   2, 8, 6,   1, 7, 9)         Array(3, 4, 5,   2, 8, 6,   1, 7, 8)
  ))                                           )) // bottom right cell should be 9
res33: Boolean = true      </> 4.14.scala    res34: Boolean = false      </> 4.15.scala
```

Chaining collection transformations in this manner will always have some overhead, but for most use cases the overhead is worth the convenience and simplicity that these transforms give you. If collection transforms do become a bottleneck, you can optimize the code using Views (*4.1.8*), In-Place Operations (*4.3.4*), or finally by looping over the raw `Array`s yourself.

> **See example 4.4 - Combining**

4.1.7 Converters

You can convert among `Array`s and other collections like Vector (*4.2.1*)s and Set (*4.2.3*) using the `.to` method:

```
@ Array(1, 2, 3).to(Vector)
res35: Vector[Int] = Vector(1, 2, 3)


@ Vector(1, 2, 3).to(Array)
res36: Array[Int] = Array(1, 2, 3)


@ Array(1, 1, 2, 2, 3, 4).to(Set)
res37: Set[Int] = Set(1, 2, 3, 4)                              </> 4.16.scala
```

4.1.8 Views

When you chain multiple transformations on a collection, we are creating many intermediate collections that are immediately thrown away. For example, in the following snippet:

```scala
@ val myArray = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)

@ val myNewArray = myArray.map(x => x + 1).filter(x => x % 2 == 0).slice(1, 3)
myNewArray: Array[Int] = Array(4, 6)                                      </> 4.17.scala
```

The chain of `.map` `.filter` `.slice` operations ends up traversing the collection three times, creating three new collections, but only the last collection ends up being stored in `myNewArray` and the others are discarded.
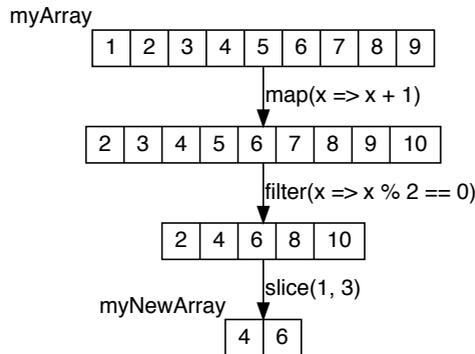
This creation and traversal of intermediate collections is wasteful. In cases where you have long chains of collection transformations that are becoming a performance bottleneck, you can use the `.view` method together with `.to` to "fuse" the operations together:

```scala
@ val myNewArray = myArray.view.map(_ + 1).filter(_ % 2 == 0).slice(1, 3).to(Array)
myNewArray: Array[Int] = Array(4, 6)                                      </> 4.18.scala
```

Using `.view` before the `map/filter/slice` transformation operations defers the actual traversal and creation of a new collection until later, when we call `.to` to convert it back into a concrete collection type:
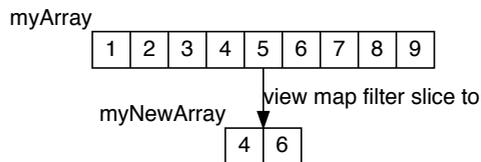
This allows us to perform this chain of `map/filter/slice` transformations with only a single traversal, and only creating a single output collection. This reduces the amount of unnecessary processing and memory allocations.

4.2 Immutable Collections

While `Array`s are the low-level primitive, most Scala applications are built upon its mutable and immutable collections: `Vector`s, `List`s, `Set`s, and `Map`s. Of these, immutable collections are by far the most common.

Immutable collections rule out an entire class of bugs due to unexpected modifications, and are especially useful in multi-threaded scenarios where you can safely pass immutable collections between threads without worrying about thread-safety issues. Most immutable collections use Structural Sharing (*4.2.2*) to make creating updated copies cheap, allowing you to use them in all but the most performance critical code.

4.2.1 Immutable Vectors

`Vector`s are fixed-size, immutable linear sequences. They are a good general-purpose sequence data structure, and provide efficient *O(log n)* performance for most operations.

```scala
@ val v = Vector(1, 2, 3, 4, 5)
v: Vector[Int] = Vector(1, 2, 3, 4, 5)


@ v(0)
res42: Int = 1


@ val v2 = v.updated(2, 10)
v2: Vector[Int] = Vector(1, 2, 10, 4, 5)


@ v2
res44: Vector[Int] = Vector(1, 2, 10, 4, 5)


@ v // note that `v` did not change!
res45: Vector[Int] = Vector(1, 2, 3, 4, 5)
                                    </> 4.19.scala
```

```scala
@ val v = Vector[Int]()
v: Vector[Int] = Vector()


@ val v1 = v :+ 1
v1: Vector[Int] = Vector(1)


@ val v2 = 4 +: v1
v2: Vector[Int] = Vector(4, 1)


@ val v3 = v2.tail
v3: Vector[Int] = Vector(1)



                                    </> 4.20.scala
```

Unlike `Array`s where `a(...) = ...` mutates it in place, a `Vector`'s `.updated` method returns a new `Vector` with the modification while leaving the old `Vector` unchanged. Due to Structural Sharing (*4.2.2*), this is a reasonably-efficient *O(log n)* operation. Similarly, using `:+` and `+:` to create a new `Vector` with additional elements on either side, or using `.tail` to create a new `Vector` with one element removed, are all *O(log n)* as well:

`Vector`s support the same set of Operations (*4.1*) that `Array`s and other collections do: builders (*4.1.1*), factory methods (*4.1.2*), transforms (*4.1.3*), etc.

In general, using `Vector`s is handy when you have a sequence you know will not change, but need flexibility in how you work with it. Their tree structure makes most operations reasonably efficient, although they will never be quite as fast as `Array`s for in-place updates or Immutable Lists (*4.2.5*) for adding and removing elements at the front.

4.2.2 Structural Sharing

`Vector`s implement their *O(log n)* copy-and-update operations by re-using portions of their tree structure. This avoids copying the whole tree, resulting in a "new" `Vector` that shares much of the old tree structure with only minor modifications.

Consider a large `Vector`, `v1`:

```
@ val v1 = Vector(1, 2, 0, 9,   7, 2, 9, 6,   ...,   3, 2, 5, 5,   4, 8, 4, 6)
```

This is represented in-memory as a tree structure, whose breadth and depth depend on the size of the `Vector`:
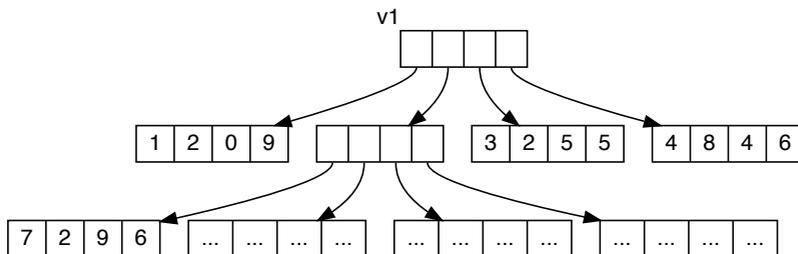
This example is somewhat simplified - a `Vector` in Scala has 32 elements per tree node rather than the 4 shown above - but it will serve us well enough to illustrate how the `Vector` data structure works.

Let us consider what happens if we want to perform an update, e.g. replacing the fifth value `7` in the above `Vector` with the value `8`:

```
@ val v2 = v1.updated(4, 8)

@ v2
res50: Vector[Int] = Vector(1, 2, 0, 9, 8, 2, 9, 6, ..., 3, 2, 5, 5, 4, 8, 4, 6)
                                                                    </> 4.21.scala
```

This is done by making updated copies of the nodes in the tree that are in the direct path down to the value we wish to update, but re-using all other nodes unchanged:
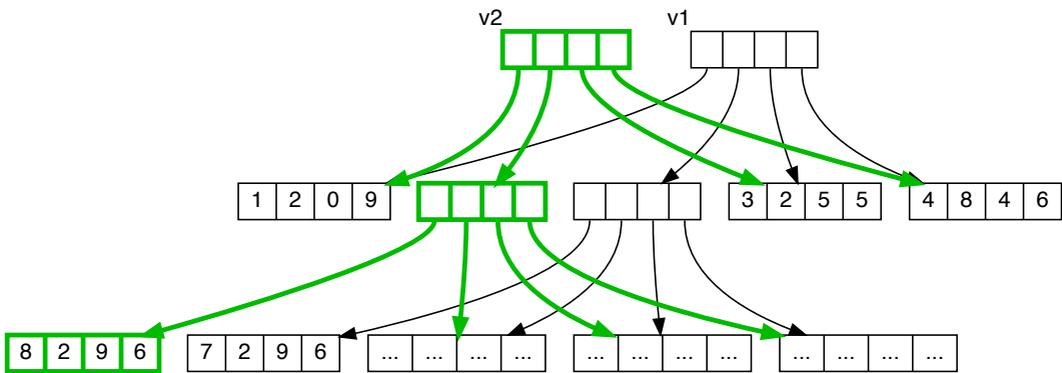
In this example `Vector` with 9 nodes, only 3 of the nodes end up needing to be copied. In a large `Vector`, the number of nodes that need to be copied is proportional to the height of the tree, while other nodes can be re-used: this structural sharing is what allows updated copies of the `Vector` to be created in only *O(log n)* time. This is much less than the *O(n)* time it takes to make a full copy of a mutable `Array` or other data structure.

Nevertheless, updating a `Vector` does always involve a certain amount of copying, and will never be as fast as updating mutable data structures in-place. In some cases where performance is important and you are updating a collection very frequently, you might consider using a mutable ArrayDeque (*4.3.1*) which has faster *O(1)* update/append/prepend operations, or raw `Arrays` if you know the size of your collection in advance.

A similar tree-shaped data structure is also used to implement Immutable Sets (*4.2.3*) and Immutable Maps (*4.2.4*).

> **See example 4.6 - ImmutableVectors**

4.2.3 Immutable Sets

Scala's immutable `Set`s are unordered collections of elements without duplicates, and provide an efficient *O(log n)* `.contains` method. `Set`s can be constructed via `+` and elements removed by `-`, or combined via `++`. Note that duplicates elements are discarded:

```
@ val s = Set(1, 2, 3)
s: Set[Int] = Set(1, 2, 3)

@ s.contains(2)
res51: Boolean = true

@ s.contains(4)
res52: Boolean = false
                              </> 4.22.scala
```

```
@ Set(1, 2, 3) + 4 + 5
res53: Set[Int] = HashSet(5, 1, 2, 3, 4)

@ Set(1, 2, 3) - 2
res54: Set[Int] = Set(1, 3)

@ Set(1, 2, 3) ++ Set(2, 3, 4)
res55: Set[Int] = Set(1, 2, 3, 4)
                              </> 4.23.scala
```

The uniqueness of items within a `Set` is also sometimes useful when you want to ensure that a collection does not contain any duplicates.

You can iterate over `Set`s using for-loops, but the order of items is undefined and should not be relied upon:

```scala
@ for (i <- Set(1, 2, 3, 4, 5)) println(i)
5
1
2
3
4
```
</> 4.24.scala

Most immutable `Set` operations take time $O(log\ n)$ in the size of the `Set`. This is fast enough for most purposes, but in cases where it isn't you can always fall back to Mutable Sets (*4.3.2*) for better performance. `Set`s also support the standard set of operations common to all collections.

> **See example 4.7 - ImmutableSets**

4.2.4 Immutable Maps

Immutable maps are unordered collections of keys and values, allowing efficient lookup by key:

```scala
@ val m = Map("one" -> 1, "two" -> 2, "three" -> 3)
m: Map[String, Int] = Map("one" -> 1, "two" -> 2, "three" -> 3)

@ m.contains("two")
res58: Boolean = true

@ m("two")
res59: Int = 2
```
</> 4.25.scala

You can also use `.get` if you're not sure whether a map contains a key or not. This returns `Some(v)` if the key is present, `None` if not:

```scala
@ m.get("one")
res60: Option[Int] = Some(1)

@ m.get("four")
res61: Option[Int] = None
```
</> 4.26.scala

While `Map`s support the same set of operations as other collections, they are treated as collections of tuples representing each key-value pair. Conversions via `.to` requires a collection of tuples to convert from, `+` adds tuples to the `Map` as key-value pairs, and `for` loops iterate over tuples:

```
@ Vector(("one", 1), ("two", 2), ("three", 3)).to(Map)
res62: Map[String, Int] = Map("one" -> 1, "two" -> 2, "three" -> 3)

@ Map[String, Int]() + ("one" -> 1) + ("three" -> 3)
res63: Map[String, Int] = Map("one" -> 1, "three" -> 3)

@ for ((k, v) <- m) println(k + " " + v)
one 1
two 2
three 3
```
<div align="right">`</> 4.27.scala`</div>

Like `Set`s, the order of items when iterating over a `Map` is undefined and should not be relied upon, and most immutable `Map` operations take time $O(log\ n)$ in the size of the `Map`.

> **See example 4.8 - ImmutableMaps**

4.2.5 Immutable Lists

```
@ val myList = List(1, 2, 3, 4, 5)
myList: List[Int] = List(1, 2, 3, 4, 5)

@ myList.head
res66: Int = 1

@ val myTail = myList.tail
myTail: List[Int] = List(2, 3, 4, 5)

@ val myOtherList = 0 :: myList
myOtherList: List[Int] = List(0, 1, 2, 3, 4, 5)

@ val myThirdList = -1 :: myList
myThirdList: List[Int] = List(-1, 1, 2, 3, 4, 5)
```
<div align="right">`</> 4.28.scala`</div>

Scala's immutable `List`s are a singly-linked list data structure. Each node in the list has a value and pointer to the next node, terminating in a `Nil` node. `List`s have a fast $O(1)$ `.head` method to look up the first item in the list, a fast $O(1)$ `.tail` method to create a list without the first element, and a fast $O(1)$ `::` operator to create a new `List` with one more element in front.

`.tail` and `::` are efficient because they can share much of the existing `List`: `.tail` returns a reference to the next node in the singly linked structure, while `::` adds a new node in front. The fact that multiple lists can share nodes means that in the above example, `myList`, `myTail`, `myOtherList` and `myThirdList` are actually mostly the same data structure:
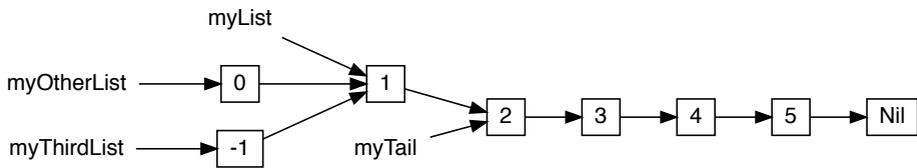
This can result in significant memory savings if you have a large number of collections that have identical elements on one side, e.g. paths on a filesystem which all share the same prefix. Rather than creating an updated copy of an `Array` in $O(n)$ time, or an updated copy of a `Vector` in $O(log\ n)$ time, pre-pending an item to a `List` is a fast $O(1)$ operation.

The downside of `List`s is that indexed lookup via `myList(i)` is a slow $O(n)$ operation, since you need to traverse the list starting from the left to find the element you want. Appending/removing elements on the right hand side of the list is also a slow $O(n)$, since it needs to make a copy of the entire list. For use cases where you want fast indexed lookup or fast appends/removes on the right, you should consider using Vectors (*4.2.1*) or mutable ArrayDeques (*4.3.1*) instead.

> **See example 4.9 - ImmutableLists**

4.3 Mutable Collections

Mutable collections are in general faster than their immutable counterparts when used for in-place operations. However, mutability comes at a cost: you need to be much more careful sharing them between different parts of your program. It is easy to create bugs where a shared mutable collection is updated unexpectedly, forcing you to hunt down which line in a large codebase is performing the unwanted update.

A common approach is to use mutable collections locally within a function or private to a class where there is a performance bottleneck, but to use immutable collections elsewhere where speed is less of a concern. That gives you the high performance of mutable collections where it matters most, while not sacrificing the safety that immutable collections give you throughout the bulk of your application logic.

4.3.1 Mutable ArrayDeques

`ArrayDeque`s are general-purpose mutable, linear collections that provide efficient *O(1)* indexed lookups, *O(1)* indexed updates, and *O(1)* insertion and removal at both left and right ends:

```scala
@ val myArrayDeque = collection.mutable.ArrayDeque(1, 2, 3, 4, 5)
myArrayDeque: collection.mutable.ArrayDeque[Int] = ArrayDeque(1, 2, 3, 4, 5)

@ myArrayDeque.removeHead()
res71: Int = 1

@ myArrayDeque.append(6)
res72: collection.mutable.ArrayDeque[Int] = ArrayDeque(2, 3, 4, 5, 6)

@ myArrayDeque.removeHead()
res73: Int = 2

@ myArrayDeque
res74: collection.mutable.ArrayDeque[Int] = ArrayDeque(3, 4, 5, 6)
```

</> 4.29.scala

`ArrayDeque`s are implemented as a circular buffer, with pointers to the logical start and end of the collection within the buffer. The operations above can be visualized as follows, from left to right:
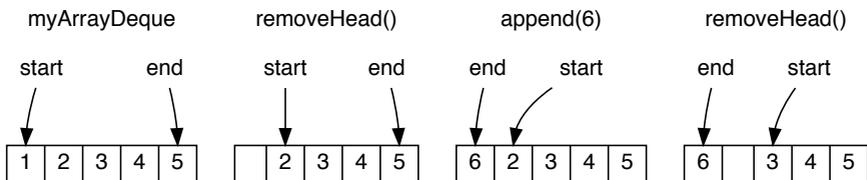
An `ArrayDeque` tries to re-use the same underlying `Array` as much as possible, only moving the `start` and `end` pointers around as elements get added or removed from either end. Only if the total number of elements grows beyond the current capacity does the underlying `Array` get re-allocated, and the size is increased by a fix multiple to keep the amortized cost of this re-allocation small.

As a result, operations on an `ArrayDeque` are much faster than the equivalent operations on an immutable `Vector`, which has to allocate *O(log n)* new tree nodes for every operation you perform.

`ArrayDeque`s have the standard suite of Operations (*4.1*). They can serve many roles:

- An `Array` that can grow: an `Array.newBuilder` does not allow indexed lookup or modification while the array is being built, and an `Array` does not allow adding more elements. An `ArrayDeque` allows both

- A faster, mutable alternative to immutable `Vector`s, if you find adding/removing items from either end using `:+`/`+:` or `.tail`/`.init` is a bottleneck in your code. Appending and prepending to `ArrayDeque`s is much faster than the equivalent `Vector` operations

- A first-in-first-out Queue, by inserting items to the right via `.append`, and removing items via `.removeHead`

- A first-in-last-out Stack, by inserting items to the right via `.append`, and removing items via `.removeLast`

If you want to "freeze" a mutable `ArrayDeque` into an immutable `Vector`, you can use `.to(Vector)`:

```
@ myArrayDeque.to(Vector)
res75: Vector[Int] = Vector(3, 4, 5, 6)                              </> 4.30.scala
```

Note that this makes a copy of the entire collection.

`ArrayDeque`s implement the abstract `collection.mutable.Buffer` interface, and can also be constructed via the `collection.mutable.Buffer(...)` syntax.

> **See example 4.10 - MutableArrayDeques**

4.3.2 Mutable Sets

The Scala standard library provides mutable `Set`s as a counterpart to the immutable `Set`s we saw earlier. Mutable sets also provide efficient `.contains` checks ($O(1)$), but instead of constructing new copies of the `Set` via `+` and `-`, you instead add and remove elements from the Set via `.add` and `.remove`:

```
@ val s = collection.mutable.Set(1, 2, 3)
s: mutable.Set[Int] = HashSet(1, 2, 3)

@ s.contains(2)
res77: Boolean = true

@ s.contains(4)
res78: Boolean = false                        </> 4.31.scala
```

```
@ s.add(4)

@ s.remove(1)

@ s
res81: mutable.Set[Int] = HashSet(2, 3, 4)

                                    </> 4.32.scala
```

You can "freeze" a mutable `Set` into an immutable `Set` by using `.to(Set)`, which makes a copy you cannot mutate using `.add` or `.remove`, and convert it back to a mutable `Set` the same way. Note that each such conversion makes a copy of the entire set.

> **See example 4.11 - MutableSets**

4.3.3 Mutable Maps

Mutable `Map`s are again just like immutable `Map`s, but allow you to mutate the `Map` by adding or removing key-value pairs:

```scala
@ val m = collection.mutable.Map("one" -> 1, "two" -> 2, "three" -> 3)
m: mutable.Map[String, Int] = HashMap("two" -> 2, "three" -> 3, "one" -> 1)

@ m.remove("two")
res83: Option[Int] = Some(2)

@ m("five") = 5

@ m
res85: mutable.Map[String, Int] = HashMap("five" -> 5, "three" -> 3, "one" -> 1)
```
`</> 4.33.scala`

Mutable `Map`s have a convenient `getOrElseUpdate` function, that allows you to look up a value by key, and compute/store the value if there isn't one already present:

```scala
@ val m = collection.mutable.Map("one" -> 1, "two" -> 2, "three" -> 3)

@ m.getOrElseUpdate("three", -1) // already present, returns existing value
res87: Int = 3

@ m // `m` is unchanged
res88: mutable.Map[String, Int] = HashMap("two" -> 2, "three" -> 3, "one" -> 1)

@ m.getOrElseUpdate("four", -1) // not present, stores new value in map and returns it
res89: Int = -1

@ m // `m` now contains "four" -> -1
res90: mutable.Map[String, Int] = HashMap(
  "two" -> 2,
  "three" -> 3,
  "four" -> -1,
  "one" -> 1
)
```
`</> 4.34.scala`

`.getOrElseUpdate` makes it convenient to use a mutable `Map` as a cache: the second parameter to `.getOrElseUpdate` is a lazy "by-name" parameter, and is only evaluated when the key is not found in the `Map`. This provides the common "check if key present, if so return value, otherwise insert new value and return that" workflow built in. We will go into more detail how by-name parameters work in **Chapter 5: Notable Scala Features**.

Chapter 4 Scala Collections   75

Mutable `Map`s are implemented as hash-tables, with `m(...)` lookups and `m(...) = ...` updates being efficient *O(1)* operations.

> See example **4.12 - MutableMaps**

4.3.4 In-Place Operations

All mutable collections, including `Array`s, have in-place versions of many common collection operations. These allow you to perform the operation on the mutable collection without having to make a transformed copy:

```scala
@ val a = collection.mutable.ArrayDeque(1, 2, 3, 4)
a: mutable.ArrayDeque[Int] = ArrayDeque(1, 2, 3, 4)

@ a.mapInPlace(_ + 1)
res92: mutable.ArrayDeque[Int] = ArrayDeque(2, 3, 4, 5)

@ a.filterInPlace(_ % 2 == 0)
res93: mutable.ArrayDeque[Int] = ArrayDeque(2, 4)

@ a // `a` was modified in place
res94: mutable.ArrayDeque[Int] = ArrayDeque(2, 4)
```

</> 4.35.scala

Apart from those shown above, there is also `dropInPlace`, `sliceInPlace`, `sortInPlace`, etc. Using in-place operations rather than normal transformations avoids the cost of allocating new transformed collections, and can help in performance-critical scenarios.

> See example **4.13 - InPlaceOperations**

4.4 Common Interfaces

In many cases, a piece of code does not care exactly what collection it is working on. For example, code that just needs something that can be iterated over in order can take a `Seq[T]`:

```scala
@ def iterateOverSomething[T](items: Seq[T]) = {
    for (i <- items) println(i)
  }

@ iterateOverSomething(Vector(1, 2, 3))
1
2
3

@ iterateOverSomething(List(("one", 1), ("two", 2), ("three", 3)))
(one,1)
(two,2)
(three,3)
```
</> 4.36.scala

Code that needs something which provides efficient indexed lookup doesn't care if it's an `Array` or `Vector`, but cannot work with a `List`. In that case, your code can take an `IndexedSeq[T]`:

```scala
@ def getIndexTwoAndFour[T](items: IndexedSeq[T]) = (items(2), items(4))

@ getIndexTwoAndFour(Vector(1, 2, 3, 4, 5))
res99: (Int, Int) = (3, 5)

@ getIndexTwoAndFour(Array(2, 4, 6, 8, 10))
res100: (Int, Int) = (6, 10)
```
</> 4.37.scala

The hierarchy of data types we have seen so far is as follows:

Iterable

collection.Set   collection.Seq   collection.Map

Set   mutable.Set   Seq   mutable.IndexedSeq   Map   mutable.Map

List   IndexedSeq   mutable.Buffer   Array

Vector   mutable.ArrayDeque
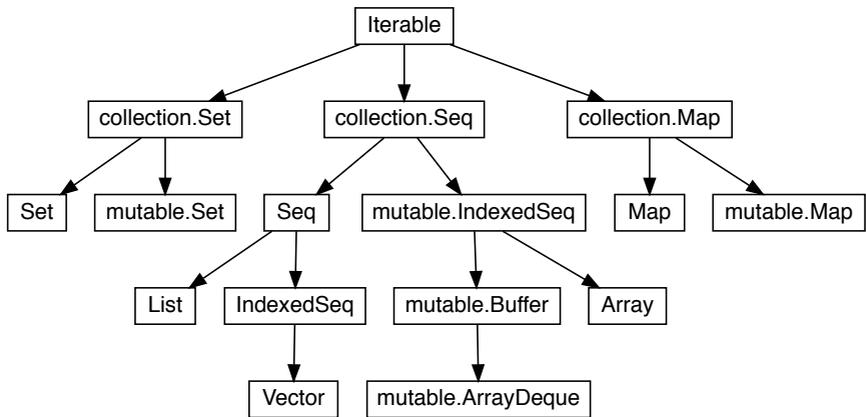
Depending on what you want your code to be able to accept, you can pick the relevant type in the hierarchy: `Iterable`, `IndexedSeq`, `Seq`, `collection.Seq`, etc. In general, most code defaults to using immutable `Seq`s, `Set`s and `Map`s. Mutable collections under the `collection.mutable` package are only used where necessary, and it is best to keep them local within a function or private to a class. `collection.{Seq,Set,Map}` serve as common interfaces to both mutable and immutable collections.

See example **4.14 - CommonInterfaces**

4.5 Conclusion

In this chapter, we have gone through the basic collections that underlie the Scala standard library: `Array`, immutable `Vector/Set/Map/List`, and mutable `ArrayDeque/Set/Map`. We have seen how to construct collections, query them, convert one to another, and write functions that work with multiple possible collection types.

This chapter should have given you a foundation for competently working with Scala's collections library, which is widely used throughout every Scala program. We will now go through some of the more unique features of the Scala language, to round off your introduction to Scala.

Exercise: Modify the `def` `isValidSudoku` method we defined in this chapter to allow testing the validity of partially-filled Sudoku grids, with un-filled cells marked by the value `0`.

```scala
@ isValidSudoku(Array(
    Array(3, 1, 6,    5, 7, 8,    4, 9, 2),
    Array(5, 2, 9,    1, 3, 4,    7, 6, 8),
    Array(4, 8, 7,    6, 2, 9,    5, 3, 1),

    Array(2, 6, 3,    0, 1, 0,    0, 8, 0),
    Array(9, 7, 4,    8, 6, 3,    0, 0, 5),
    Array(8, 5, 1,    0, 9, 0,    6, 0, 0),

    Array(1, 3, 0,    0, 0, 0,    2, 5, 0),
    Array(0, 0, 0,    0, 0, 0,    0, 7, 4),
    Array(0, 0, 5,    2, 0, 6,    3, 0, 0)
  ))
res101: Boolean = true          </> 4.38.scala
```

```scala
@ isValidSudoku(Array(
    Array(3, 1, 6,    5, 7, 8,    4, 9, 3),
    Array(5, 2, 9,    1, 3, 4,    7, 6, 8),
    Array(4, 8, 7,    6, 2, 9,    5, 3, 1),

    Array(2, 6, 3,    0, 1, 0,    0, 8, 0),
    Array(9, 7, 4,    8, 6, 3,    0, 0, 5),
    Array(8, 5, 1,    0, 9, 0,    6, 0, 0),

    Array(1, 3, 0,    0, 0, 0,    2, 5, 0),
    Array(0, 0, 0,    0, 0, 0,    0, 7, 4),
    Array(0, 0, 5,    2, 0, 6,    3, 0, 0)
  )) // top right cell should be 2
res102: Boolean = false         </> 4.39.scala
```

See example 4.15 - PartialValidSudoku

Exercise: Write a `def renderSudoku` method that can be used to pretty-print a Sudoku grid as shown below: with the zeroes representing unfilled cells left out, and each 3x3 square surrounded by horizontal and vertical lines.

```scala
@ renderSudoku(Array(
    Array(3, 1, 6,   5, 7, 8,   4, 9, 2),
    Array(5, 2, 9,   1, 3, 4,   7, 6, 8),
    Array(4, 8, 7,   6, 2, 9,   5, 3, 1),

    Array(2, 6, 3,   0, 1, 0,   0, 8, 0),
    Array(9, 7, 4,   8, 6, 3,   0, 0, 5),
    Array(8, 5, 1,   0, 9, 0,   6, 0, 0),

    Array(1, 3, 0,   0, 0, 0,   2, 5, 0),
    Array(0, 0, 0,   0, 0, 0,   0, 7, 4),
    Array(0, 0, 5,   2, 0, 6,   3, 0, 0)
  ))
```

`</> 4.40.scala`

```scala
res103: String = """
+-------+-------+-------+
| 3 1 6 | 5 7 8 | 4 9 2 |
| 5 2 9 | 1 3 4 | 7 6 8 |
| 4 8 7 | 6 2 9 | 5 3 1 |
+-------+-------+-------+
| 2 6 3 |   1   |   8   |
| 9 7 4 | 8 6 3 |     5 |
| 8 5 1 |   9   | 6     |
+-------+-------+-------+
| 1 3   |       | 2 5   |
|       |       |     7 4 |
|     5 | 2   6 | 3     |
+-------+-------+-------+
"""
```

`</> 4.41.output-scala`

You might find the `Array.grouped` operator useful for this, though you can also do without it:

```scala
@ Array(3, 1, 6,   5, 7, 8,   4, 9, 2).grouped(3).toArray
res104: Array[Array[Int]] = Array(Array(3, 1, 6), Array(5, 7, 8), Array(4, 9, 2))
```

`</> 4.42.scala`

See example 4.16 - RenderSudoku

Discuss Chapter 4 online at https://www.handsonscala.com/discuss/4

5

Notable Scala Features

```scala
@ def getDayMonthYear(s: String) = s match {
    case s"$day-$month-$year" => println(s"found day: $day, month: $month, year: $year")
    case _ => println("not a date")
  }

@ getDayMonthYear("9-8-1965")
found day: 9, month: 8, year: 1965

@ getDayMonthYear("9-8")
not a date                                                    </> 5.1.scala
```

Snippet 5.1: using Scala's pattern matching feature to parse simple string patterns

This chapter will cover some of the more interesting and unusual features of Scala. For each such feature, we will cover both what the feature does as well as some common use cases to give you an intuition for what it is useful for.

Not every feature in this chapter will be something you use day-to-day. Nevertheless, even these less-commonly-used features are used often enough that it is valuable to have a high-level understanding for when you eventually encounter them in the wild.

5.1 Case Classes and Sealed Traits

5.1.1 Case Classes

`case class`es are like normal `class`es, but meant to represent `class`es which are "just data": where all the data is immutable and public, without any mutable state or encapsulation. Their use case is similar to "structs" in C/C++, "POJOs" in Java or "Data Classes" in Python or Kotlin. Their name comes from the fact that they support pattern matching (*5.2*) via the `case` keyword.

Case classes are defined with the `case` keyword, and can be instantiated without `new`. All of their constructor parameters are public fields by default.

```
@ case class Point(x: Int, y: Int)

@ val p = Point(1, 2)
p: Point = Point(1, 2)
```
</> 5.2.scala

```
@ p.x
res2: Int = 1

@ p.y
res3: Int = 2
```
</> 5.3.scala

`case class`s give you a few things for free:

- A `.toString` implemented to show you the constructor parameter values

- A `==` implemented to check if the constructor parameter values are equal

- A `.copy` method to conveniently create modified copies of the case class instance

```
@ p.toString
res4: String = "Point(1,2)"

@ val p2 = Point(1, 2)

@ p == p2
res6: Boolean = true
```
</> 5.4.scala

```
@ val p = Point(1, 2)

@ val p3 = p.copy(y = 10)
p3: Point = Point(1, 10)

@ val p4 = p3.copy(x = 20)
p4: Point = Point(20, 10)
```
</> 5.5.scala

Like normal classes, you can define instance methods or properties in the body of the `case class`:

```
@ case class Point(x: Int, y: Int) {
    def z = x + y
  }
```
</> 5.6.scala

```
@ val p = Point(1, 2)

@ p.z
res12: Int = 3
```
</> 5.7.scala

`case class`es are a good replacement for large tuples, since instead of extracting their values via `._1` `._2` `._7` you can extract the values via their names like `.x` and `.y`. That is much easier than trying to remember exactly what field `._7` in a large tuple represents!

5.1.2 Sealed Traits

`trait`s can also be defined `sealed`, and only extended by a fixed set of `case` `class`es. In the following example, we define a `sealed trait` `Point` extended by two `case` `class`es: `Point2D` and `Point3D`:

```
@ {
  sealed trait Point
  case class Point2D(x: Double, y: Double) extends Point
  case class Point3D(x: Double, y: Double, z: Double) extends Point
  }

@ def hypotenuse(p: Point) = p match {
    case Point2D(x, y) => math.sqrt(x * x + y * y)
    case Point3D(x, y, z) => math.sqrt(x * x + y * y + z * z)
  }

@ val points: Array[Point] = Array(Point2D(1, 2), Point3D(4, 5, 6))

@ for (p <- points) println(hypotenuse(p))
2.23606797749979
8.774964387392123
```
</> 5.8.scala

The core difference between normal `trait`s and `sealed trait`s can be summarized as follows:

- Normal `trait`s are *open*, so any number of classes can inherit from the trait as long as they provide all the required methods, and instances of those classes can be used interchangeably via the `trait`'s required methods.

- `sealed trait`s are *closed*: they only allow a fixed set of classes to inherit from them, and all inheriting classes must be defined together with the trait itself in the same file or REPL command (hence the curlies `{}` surrounding the `Point`/`Point2D`/`Point3D` definitions above).

Because there are only a fixed number of classes inheriting from `sealed trait` `Point`, we can use pattern matching in the `def hypotenuse` function above to define how each kind of `Point` should be handled.

5.1.3 Use Cases for Normal v.s. Sealed Traits

Both normal `trait`s and `sealed trait`s are common in Scala applications: normal `trait`s for interfaces which may have any number of subclasses, and `sealed trait`s where the number of subclasses is fixed.

Normal `trait`s and `sealed trait`s make different things easy:

- A normal `trait` hierarchy makes it easy to add additional sub-classes: just define your class and implement the necessary methods. However, it makes it difficult to add new methods: a new method needs to be added to all existing subclasses, of which there may be many.

- A `sealed trait` hierarchy is the opposite: it is easy to add new methods, since a new method can simply pattern match on each sub-class and decide what it wants to do for each. However, adding new sub-classes is difficult, as you need to go to all existing pattern matches and add the `case` to handle your new sub-class

In general, `sealed trait`s are good for modelling hierarchies where you expect the number of sub-classes to change very little or not-at-all. A good example of something that can be modeled using `sealed trait` is JSON:

```scala
@ {
  sealed trait Json
  case class Null() extends Json
  case class Bool(value: Boolean) extends Json
  case class Str(value: String) extends Json
  case class Num(value: Double) extends Json
  case class Arr(value: Seq[Json]) extends Json
  case class Dict(value: Map[String, Json]) extends Json
}                                                      </> 5.9.scala
```

- A JSON value can only be JSON null, boolean, number, string, array, or dictionary.

- JSON has not changed in 20 years, so it is unlikely that anyone will need to extend our JSON `trait` with additional subclasses.

- While the set of sub-classes is fixed, the range of operations we may want to do on a JSON blob is unbounded: parse it, serialize it, pretty-print it, minify it, sanitize it, etc.

Thus it makes sense to model a JSON data structure as a closed `sealed trait` hierarchy rather than a normal open `trait` hierarchy.

> **See example 5.2 - SealedTrait**

5.2 Pattern Matching

5.2.1 Match

Scala allows pattern matching on values using the `match` keyword. This is similar to the `switch` statement found in other programming languages, but more flexible: apart from `match`ing on primitive integers and strings, you can also use `match` to extract values from ("destructure") composite data types like tuples and `case class`es. Note that in many examples below, there is a `case _ =>` clause which defines the default case if none of the earlier cases matched.

5.2.1.1 Matching on `Int`s

```scala
@ def dayOfWeek(x: Int) = x match {
    case 1 => "Mon"; case 2 => "Tue"
    case 3 => "Wed"; case 4 => "Thu"
    case 5 => "Fri"; case 6 => "Sat"
    case 7 => "Sun"; case _ => "Unknown"
  }

@ dayOfWeek(5)
res19: String = "Fri"

@ dayOfWeek(-1)
res20: String = "Unknown"
```
</> 5.10.scala

5.2.1.2 Matching on `String`s

```scala
@ def indexOfDay(d: String) = d match {
    case "Mon" => 1; case "Tue" => 2
    case "Wed" => 3; case "Thu" => 4
    case "Fri" => 5; case "Sat" => 6
    case "Sun" => 7; case _ => -1
  }

@ indexOfDay("Fri")
res22: Int = 5

@ indexOfDay("???")
res23: Int = -1
```
</> 5.11.scala

5.2.1.3 Matching on tuple `(Int, Int)`

```scala
@ for (i <- Range.inclusive(1, 100)) {
    val s =  (i % 3, i % 5) match {
      case (0, 0) => "FizzBuzz"
      case (0, _) => "Fizz"
      case (_, 0) => "Buzz"
      case _ => i
    }
    println(s)
  }
1
2
Fizz
4
Buzz
...
```
</> 5.12.scala

5.2.1.4 Matching on tuple `(Boolean, Boolean)`

```scala
@ for (i <- Range.inclusive(1, 100)) {
    val s = (i % 3 == 0, i % 5 == 0) match {
      case (true, true) => "FizzBuzz"
      case (true, false) => "Fizz"
      case (false, true) => "Buzz"
      case (false, false) => i
    }
    println(s)
  }
1
2
Fizz
4
Buzz
...
```
</> 5.13.scala

5.2.1.5 Matching on Case Classes:

```scala
@ case class Point(x: Int, y: Int)

@ def direction(p: Point) = p match {
    case Point(0, 0) => "origin"
    case Point(_, 0) => "horizontal"
    case Point(0, _) => "vertical"
    case _ => "diagonal"
  }

@ direction(Point(0, 0))
res28: String = "origin"

@ direction(Point(1, 1))
res29: String = "diagonal"

@ direction(Point(10, 0))
res30: String = "horizontal"                </> 5.14.scala
```

5.2.1.6 Matching on String Patterns:

```scala
@ def splitDate(s: String) = s match {
    case s"$day-$month-$year" =>
      s"day: $day, mon: $month, yr: $year"
    case _ => "not a date"
  }

@ splitDate("9-8-1965")
res32: String = "day: 9, mon: 8, yr: 1965"

@ splitDate("9-8")
res33: String = "not a date"                </> 5.15.scala
```

(Note that pattern matching on string patterns only supports simple glob-like patterns, and doesn't support richer patterns like Regular Expressions. For those, you can use the functionality of the `scala.util.matching.Regex` *class)*

5.2.2 Nested Matches

Patterns can also be nested, e.g. this example matches a string pattern within a `case class` pattern:

```scala
@ case class Person(name: String, title: String)

@ def greet(p: Person) = p match {
    case Person(s"$firstName $lastName", title) => println(s"Hello $title $lastName")
    case Person(name, title) => println(s"Hello $title $name")
  }

@ greet(Person("Haoyi Li", "Mr"))
Hello Mr Li

@ greet(Person("Who?", "Dr"))
Hello Dr Who?                               </> 5.16.scala
```

Patterns can be nested arbitrarily deeply. The following example matches string patterns, inside a `case class`, inside a tuple:

```scala
@ def greet2(husband: Person, wife: Person) = (husband, wife) match {
    case (Person(s"$first1 $last1", _), Person(s"$first2 $last2", _)) if last1 == last2 =>
      println(s"Hello Mr and Ms $last1")

    case (Person(name1, _), Person(name2, _)) => println(s"Hello $name1 and $name2")
  }

@ greet2(Person("James Bond", "Mr"), Person("Jane Bond", "Ms"))
Hello Mr and Ms Bond

@ greet2(Person("James Bond", "Mr"), Person("Jane", "Ms"))
Hello James Bond and Jane
```
<div align="right">*</> 5.17.scala*</div>

5.2.3 Loops and Vals

The last two places you an use pattern matching are inside `for`-loops and `val` definitions. Pattern matching in `for`-loops is useful when you need to iterate over collections of tuples:

```scala
@ val a = Array[(Int, String)]((1, "one"), (2, "two"), (3, "three"))

@ for ((i, s) <- a) println(s + i)
one1
two2
three3
```
<div align="right">*</> 5.18.scala*</div>

Pattern matching in `val` statements is useful when you are sure the value will match the given pattern, and all you want to do is extract the parts you want. If the value doesn't match, this fails with an exception:

```scala
@ case class Point(x: Int, y: Int)

@ val p = Point(123, 456)

@ val Point(x, y) = p
x: Int = 123
y: Int = 456
```
<div align="right">*</> 5.19.scala*</div>

```scala
@ val s"$first $second" = "Hello World"
first: String = "Hello"
second: String = "World"

@ val flipped = s"$second $first"
flipped: String = "World Hello"

@ val s"$first $second" = "Hello"
scala.MatchError: Hello
```
<div align="right">*</> 5.20.scala*</div>

5.2.4 Pattern Matching on Sealed Traits and Case Classes

Pattern matching lets you elegantly work with structured data comprising case classes and sealed traits. For example, let's consider a simple sealed trait that represents arithmetic expressions:

```scala
@ {
  sealed trait Expr
  case class BinOp(left: Expr, op: String, right: Expr) extends Expr
  case class Literal(value: Int) extends Expr
  case class Variable(name: String) extends Expr
}                                                                </> 5.21.scala
```

Where `BinOp` stands for "Binary Operation". This can represent the arithmetic expressions, such as the following

`x + 1`	`BinOp(Variable("x"), "+", Literal(1))`

`x * (y - 1)`	`BinOp(`
	`  Variable("x"),`
	`  "*",`
	`  BinOp(Variable("y"), "-", Literal(1))`
	`)` </> 5.22.scala

`(x + 1) * (y - 1)`	`BinOp(`
	`  BinOp(Variable("x"), "+", Literal(1)),`
	`  "*",`
	`  BinOp(Variable("y"), "-", Literal(1))`
	`)` </> 5.23.scala

For now, we will ignore the parsing process that turns the string on the left into the structured `case class` structure on the right: we will cover that in *Chapter 19: Parsing Structured Text*. Let us instead consider two things you may want to do once you have parsed such an arithmetic expression to the `case class`es we see above: we may want to print it to a human-friendly string, or we may want to evaluate it given some variable values.

5.2.4.1 Stringifying Our Expressions

Converting the expressions to a string can be done using the following approach:

- If an `Expr` is a `Literal`, the string is the value of the literal
- If an `Expr` is a `Variable`, the string is the name of the variable
- If an `Expr` is a `BinOp`, the string is the stringified left expression, followed by the operation, followed by the stringified right expression

Converted to pattern matching code, this can be written as follows:

```scala
@ def stringify(expr: Expr): String = expr match {
    case BinOp(left, op, right) => s"(${stringify(left)} $op ${stringify(right)})"
    case Literal(value) => value.toString
    case Variable(name) => name
  }                                                      </> 5.24.scala
```

We can construct some of `Expr`s we saw earlier and feed them into our `stringify` function to see the output:

```scala
@ val smallExpr = BinOp(
    Variable("x"),
    "+",
    Literal(1)
  )

@ stringify(smallExpr)
res52: String = "(x + 1)"
                              </> 5.25.scala
```

```scala
@ val largeExpr = BinOp(
    BinOp(Variable("x"), "+", Literal(1)),
    "*",
    BinOp(Variable("y"), "-", Literal(1))
  )

@ stringify(largeExpr)
res54: String = "((x + 1) * (y - 1))"
                              </> 5.26.scala
```

5.2.4.2 Evaluating Our Expressions

Evaluation is a bit more complex than stringifying the expressions, but only slightly. We need to pass in a `values` map that holds the numeric value of every variable, and we need to treat `+`, `-`, and `*` operations differently:

```scala
@ def evaluate(expr: Expr, values: Map[String, Int]): Int = expr match {
    case BinOp(left, "+", right) => evaluate(left, values) + evaluate(right, values)
    case BinOp(left, "-", right) => evaluate(left, values) - evaluate(right, values)
    case BinOp(left, "*", right) => evaluate(left, values) * evaluate(right, values)
    case Literal(value) => value
    case Variable(name) => values(name)
  }

@ evaluate(smallExpr, Map("x" -> 10))
res56: Int = 11

@ evaluate(largeExpr, Map("x" -> 10, "y" -> 20))
res57: Int = 209
                                                       </> 5.27.scala
```

Overall, this looks relatively similar to the `stringify` function we wrote earlier: a recursive function that pattern matches on the `expr: Expr` parameter to handle each `case class` that implements `Expr`. The cases handling child-free `Literal` and `Variable` are trivial, while in the `BinOp` case we recurse on both left and right children before combining the result. This is a common way of working with recursive data structures in any language, and Scala's `sealed trait`s, `case class`es and pattern matching make it concise and easy.

This `Expr` structure and the printer and evaluator we have written are intentionally simplistic, just to give us a chance to see how pattern matching can be used to easily work with structured data modeled as `case class`es and `sealed trait`s. We will be exploring these techniques much more deeply in **Chapter 20: Implementing a Programming Language**.

> See example 5.3 - PatternMatching

5.3 By-Name Parameters

```scala
@ def func(arg: => String) = ...
```

Scala also supports "by-name" method parameters using a `: => T` syntax, which are evaluated each time they are referenced in the method body. This has three primary use cases:

1. Avoiding evaluation if the argument does not end up being used
2. Wrapping evaluation to run setup and teardown code before and after the argument evaluates
3. Repeating evaluation of the argument more than once

5.3.1 Avoiding Evaluation

The following `log` method uses a by-name parameter to avoid evaluating the `msg: => String` unless it is actually going to get printed. This can help avoid spending CPU time constructing log messages (here via `"Hello " + 123 + " World"`) even when logging is disabled:

```scala
@ var logLevel = 1

@ def log(level: Int, msg: => String) = {
    if (level > logLevel) println(msg)
  }
```
</> 5.28.scala

```scala
@ log(2, "Hello " + 123 + " World")
Hello 123 World

@ logLevel = 3

@ log(2, "Hello " + 123 + " World")
```
<no output>
</> 5.29.scala

Often a method does not end up using all of its arguments all the time. In the above example, by not computing log messages when they are not needed, we can save a significant amount of CPU time and object allocations which may make a difference in performance-sensitive applications.

The `getOrElse` and `getOrElseUpdate` methods we saw in **Chapter 4: Scala Collections** are similar: these methods do not use the argument representing the default value if the value we are looking for is already present. By making the default value a by-name parameter, we do not have to evaluate it in the case where it does not get used.

5.3.2 Wrapping Evaluation

Using by-name parameters to "wrap" the evaluation of your method in some setup/teardown code is another common pattern. The following `measureTime` function defers evaluation of `f: => Unit`, allowing us to run `System.currentTimeMillis()` before and after the argument is evaluated and thus print out the time taken:

```scala
@ def measureTime(f: => Unit) = {
    val start = System.currentTimeMillis()
    f
    val end = System.currentTimeMillis()
    println("Evaluation took " + (end - start) + " milliseconds")
  }

@ measureTime(new Array[String](10 * 1000 * 1000).hashCode())
Evaluation took 24 milliseconds

@ measureTime { // methods taking a single arg can also be called with curly brackets
    new Array[String](100 * 1000 * 1000).hashCode()
  }
Evaluation took 287 milliseconds
```
</> 5.30.scala

There are many other use cases for such wrapping:

- Setting some thread-local context while the argument is being evaluated
- Evaluating the argument inside a `try-catch` block so we can handle exceptions
- Evaluating the argument in a `Future` so the logic runs asynchronously on another thread

These are all cases where using by-name parameter can help.

5.3.3 Repeating Evaluation

The last use case we will cover for by-name parameters is repeating evaluation of the method argument. The following snippet defines a generic `retry` method: this method takes in an argument, evaluates it within a `try`-`catch` block, and re-executes it on failure with a maximum number of attempts. We test this by using it to wrap a call which may fail, and seeing the `retrying` messages get printed to the console.

```scala
@ def retry[T](max: Int)(f: => T): T = {
    var tries = 0
    var result: Option[T] = None
    while (result == None) {
      try { result = Some(f) }
      catch {case e: Throwable =>
        tries += 1
        if (tries > max) throw e
        else {
          println(s"failed, retry #$tries")
        }
      }
    }
    result.get
}
                                    </> 5.31.scala
```

```scala
@ val httpbin = "https://httpbin.org"

@ retry(max = 5) {
    // Only succeeds with a 200 response
    // code 1/3 of the time
    requests.get(
      s"$httpbin/status/200,400,500"
    )
}
call failed, retry #1
call failed, retry #2
res68: requests.Response = Response(
  "https://httpbin.org/status/200,400,500",
  200,
...                                 </> 5.32.scala
```

Above we define `retry` as a generic function taking a type parameter `[T]`, taking a by-name parameter that computes a value of type `T`, and returning a `T` once the code block is successful. We can then use `retry` to wrap a code block of any type, and it will retry that block and return the first `T` it successfully computes.

Making `retry` take a by-name parameter is what allows it to repeat evaluation of the `requests.get` block where necessary. Other use cases for repetition include running performance benchmarks or performing load tests. In general, by-name parameters aren't something you use very often, but when necessary they let you write code that manipulates the evaluation of a method argument in a variety of useful ways: instrumenting it, retrying it, eliding it, etc.

We will learn more about the `requests` library that we used in the above snippet in **Chapter 12: Working with HTTP APIs**.

> See example **5.4 - ByName**

5.4 Implicit Parameters

An *implicit parameter* is a parameter that is automatically filled in for you when calling a function. For example, consider the following class `Foo` and the function `bar` that takes an `implicit foo: Foo` parameter:

```scala
@ class Foo(val value: Int)
```

```scala
@ def bar(implicit foo: Foo) = foo.value + 10
```
</> 5.33.scala

If you try to call `bar` without an implicit `Foo` in scope, you get a compilation error. To call `bar`, you need to define an implicit value of the type `Foo`, such that the call to `bar` can automatically resolve it from the enclosing scope:

```scala
@ bar
cmd4.sc:1: could not find implicit
          value for parameter foo: Foo
val res4 = bar
          ^
Compilation Failed
```
</> 5.34.scala

```scala
@ implicit val foo: Foo = new Foo(1)
foo: Foo = ammonite.$sess.cmd1$Foo@451882b2

@ bar // `foo` is resolved implicitly
res72: Int = 11

@ bar(foo) // passing in `foo` explicitly
res73: Int = 11
```
</> 5.35.scala

Implicit parameters are similar to the *default values* we saw in **Chapter 3: Basic Scala**. Both of them allow you to pass in a value explicitly or fall back to some default. The main difference is that while default values are "hard coded" at the definition site of the method, implicit parameters take their default value from whatever `implicit` is in scope at the call-site.

We'll now look into a more concrete example where using implicit parameters can help keep your code clean and readable, before going into a more advanced use case of the feature for Typeclass Inference (*5.5*).

5.4.1 Passing ExecutionContext to Futures

As an example, code using `Future` needs an `ExecutionContext` value in order to work. As a result, we end up passing this `ExecutionContext` everywhere, which is tedious and verbose:

```scala
def getEmployee(ec: ExecutionContext, id: Int): Future[Employee] = ...
def getRole(ec: ExecutionContext, employee: Employee): Future[Role] = ...

val executionContext: ExecutionContext = ...

val bigEmployee: Future[EmployeeWithRole] = {
  getEmployee(executionContext, 100).flatMap(
    executionContext,
    e =>
      getRole(executionContext, e)
        .map(executionContext, r => EmployeeWithRole(e, r))
  )
}
```
</> 5.36.scala

`getEmployee` and `getRole` perform asynchronous actions, which we then `map` and `flatMap` to do further work. Exactly how the `Future`s work is beyond the scope of this section: for now, what is notable is how every operation needs to be passed the `executionContext` to do their work. We will will revisit these APIs in **Chapter 13: Fork-Join Parallelism with Futures**.

Without implicit parameters, we have the following options:

- Passing `executionContext` explicitly is verbose and can make your code harder to read: the logic we care about is drowned in a sea of boilerplate `executionContext` passing

- Making `executionContext` global would be concise, but would lose the flexibility of passing different values in different parts of your program

- Putting `executionContext` into a thread-local variable would maintain flexibility and conciseness, but it is error-prone and easy to forget to set the thread-local before running code that needs it

All of these options have tradeoffs, forcing us to either sacrifice conciseness, flexibility, or safety. Scala's implicit parameters provide a fourth option: passing `executionContext` implicitly, which gives us the conciseness, flexibility, and safety that the above options are unable to give us.

5.4.2 Dependency Injection via Implicits

To resolve these issues, we can make all these functions take the `executionContext` as an implicit parameter. This is already the case for standard library operations like `flatMap` and `map` on `Future`s, and we can modify our `getEmployee` and `getRole` functions to follow suit. By defining `executionContext` as an `implicit`, it will automatically get picked up by all the method calls below.

```scala
def getEmployee(id: Int)(implicit ec: ExecutionContext): Future[Employee] = ...
def getRole(employee: Employee)(implicit ec: ExecutionContext): Future[Role] = ...

implicit val executionContext: ExecutionContext = ...

val bigEmployee: Future[EmployeeWithRole] = {
  getEmployee(100).flatMap(e =>
    getRole(e).map(r =>
      EmployeeWithRole(e, r)
    )
  )
}
```
</> 5.37.scala

Using implicit parameters can help clean up code where we pass the same shared context or configuration object throughout your entire application:

- By making the "uninteresting" parameter passing implicit, it can focus the reader's attention on the core logic of your application.

- Since implicit parameters can be passed explicitly, they preserve the flexibility for the developer in case they want to manually specify or override the implicit parameter being passed.

- The fact that missing implicits are a compile time error makes their usage much less error-prone than thread-locals. A missing implicit will be caught early on before code is compiled and deployed to production.

> **See example 5.5 - ImplicitParameters**

5.5 Typeclass Inference

A second way that implicit parameters are useful is by using them to associate values to types. This is often called a *typeclass*, the term originating from the Haskell programming language, although it has nothing to do with types and `class`es in Scala. While typeclasses are a technique built on the same `implicit` language feature described earlier, they are an interesting and important enough technique to deserve their own section in this chapter.

5.5.1 Problem Statement: Parsing Command Line Arguments

Let us consider the task of parsing command-line arguments, given as `String`s, into Scala values of various types: `Int`s, `Boolean`s, `Double`s, etc. This is a common task that almost every program has to deal with, either directly or by using a library.

A first sketch may be writing a generic method to parse the values. The signature might look something like this:

```scala
def parseFromString[T](s: String): T = ...

val args = Seq("123", "true", "7.5")
val myInt = parseFromString[Int](args(0))
val myBoolean = parseFromString[Boolean](args(1))
val myDouble = parseFromString[Double](args(2))                          </> 5.38.scala
```

On the surface this seems impossible to implement:

- How does the `parseCliArgument` know how to convert the given `String` into an arbitrary `T`?

- How does it know what types `T` a command-line argument can be parsed into, and which it cannot? For example, we should not be able to parse a `java.net.DatagramSocket` from an input string.

5.5.2 Separate Parser Objects

A second sketch at a solution may be to define separate parser objects, one for each type we need to be able to parse. For example:

```scala
trait StrParser[T]{ def parse(s: String): T }
object ParseInt extends StrParser[Int]{ def parse(s: String) = s.toInt }
object ParseBoolean extends StrParser[Boolean]{ def parse(s: String) = s.toBoolean }
object ParseDouble extends StrParser[Double]{ def parse(s: String) = s.toDouble }
                                                                        </> 5.39.scala
```

We can then call these as follows:

```scala
val args = Seq("123", "true", "7.5")
val myInt = ParseInt.parse(args(0))
val myBoolean = ParseBoolean.parse(args(1))
val myDouble = ParseDouble.parse(args(2))                      </> 5.40.scala
```

This works. However, it then leads to another problem: if we wanted to write a method that didn't parse a `String` directly, but parsed a value from the console, how would we do that? We have two options.

5.5.2.1 Re-Using Our StrParsers

The first option is writing a whole new set of `object`s dedicated to parsing from the console:

```scala
trait ConsoleParser[T]{ def parse(): T }
object ConsoleParseInt extends ConsoleParser[Int]{
  def parse() = scala.Console.in.readLine().toInt
}
object ConsoleParseBoolean extends ConsoleParser[Boolean]{
  def parse() = scala.Console.in.readLine().toBoolean
}
object ConsoleParseDouble extends ConsoleParser[Double]{
  def parse() = scala.Console.in.readLine().toDouble
}

val myInt = ConsoleParseInt.parse()
val myBoolean = ConsoleParseBoolean.parse()
val myDouble = ConsoleParseDouble.parse()                     </> 5.41.scala
```

The second option is defining a helper method that receives a `StrParser[T]` as an argument, which we would need to pass in to tell it how to parse the type `T`

```scala
def parseFromConsole[T](parser: StrParser[T]) = parser.parse(scala.Console.in.readLine())

val myInt = parseFromConsole[Int](ParseInt)
val myBoolean = parseFromConsole[Boolean](ParseBoolean)
val myDouble = parseFromConsole[Double](ParseDouble)          </> 5.42.scala
```

Both of these solutions are clunky:

1. The first because we need to duplicate all the `Int`/`Boolean`/`Double`/etc. parsers. What if we need to parse input from the network? From files? We would need to duplicate every parser for each case.

2. The second because we need to pass these `ParseFoo` objects everywhere. Often there is only a single `StrParser[Int]` we can pass to `parseFromConsole[Int]`. Why can't the compiler infer it for us?

5.5.3 Solution: Implicit StrParser

The solution to the problems above is to make the instances of `StrParser` `implicit`:

```scala
trait StrParser[T]{ def parse(s: String): T }
object StrParser{
  implicit object ParseInt extends StrParser[Int]{
    def parse(s: String) = s.toInt
  }
  implicit object ParseBoolean extends StrParser[Boolean]{
    def parse(s: String) = s.toBoolean
  }
  implicit object ParseDouble extends StrParser[Double]{
    def parse(s: String) = s.toDouble
  }
}
```
</> 5.43.scala

We put the `implicit object` `ParseInt`, `ParseBoolean`, etc. in an `object` `StrParser` with the same name as the `trait` `StrParser` next to it. An `object` with the same name as a `class` that it is defined next to is called a *companion object*. Companion objects are often used to group together implicits, static methods, factory methods, and other functionality that is related to a `trait` or `class` but does not belong to any specific instance. Implicits in the companion object are also treated specially, and do not need to be imported into scope in order to be used as an implicit parameter.

Note that if you are entering this into the Ammonite Scala REPL, you need to surround both declarations with an extra pair of curly brackets `{...}` so that both the `trait` and `object` are defined in the same REPL command.

Now, while we can still explicitly call `ParseInt.parse(args(0))` to parse literal strings as before, we can now write a generic function that automatically uses the correct instance of `StrParser` depending on what type we asked it to parse:

```scala
def parseFromString[T](s: String)(implicit parser: StrParser[T]) = {
  parser.parse(s)
}

val args = Seq("123", "true", "7.5")
val myInt = parseFromString[Int](args(0))
val myBoolean = parseFromString[Boolean](args(1))
val myDouble = parseFromString[Double](args(2))
```
</> 5.44.scala

This looks similar to our initial sketch, except by taking an `(implicit parser: StrParser[T])` parameter the function can now automatically infer the correct `StrParser` for each type it is trying to parse.

5.5.3.1 Re-Using Our Implicit StrParsers

Making our `StrParser[T]`s implicit means we can re-use them without duplicating our parsers or passing them around manually. For example, we can write a function that parses strings from the console:

```scala
def parseFromConsole[T](implicit parser: StrParser[T]) = {
  parser.parse(scala.Console.in.readLine())
}


val myInt = parseFromConsole[Int]
```
</> 5.45.scala

The call to `parseFromConsole[Int]` automatically infers the `StrParser.ParseInt` implicit in the `StrParser` companion object, without needing to duplicate it or tediously pass it around. That makes it very easy to write code that works with a generic type `T` as long as `T` has a suitable `StrParser`.

5.5.3.2 Context-Bound Syntax

This technique of taking an implicit parameter with a generic type is common enough that the Scala language provides dedicated syntax for it. The following method signature:

```scala
def parseFromString[T](s: String)(implicit parser: StrParser[T]) = ...
```

Can be written more concisely as:

```scala
def parseFromString[T: StrParser](s: String) = ...
```

This syntax is referred to as a *context bound*, and it is semantically equivalent to the `(implicit parser: StrParser[T])` syntax above. When using the context bound syntax, the implicit parameter isn't given a name, and so we cannot call `parser.parse` like we did earlier. Instead, we can resolve the implicit values via the `implicitly` function, e.g. `implicitly[StrParser[T]].parse`.

5.5.3.3 Compile-Time Implicit Safety

As Typeclass Inference uses the the same `implicit` language feature we saw earlier, mistakes such as attempting to call `parseFromConsole` with an invalid type produce a compile error:

```scala
@ val myDatagramSocket = parseFromConsole[java.net.DatagramSocket]
cmd19.sc:1: could not find implicit value for parameter parser:
            ammonite.$sess.cmd11.StrParser[java.net.DatagramSocket]
val myDatagramSocket = parseFromConsole[java.net.DatagramSocket]
                                       ^
Compilation Failed
```
</> 5.46.scala

Similarly, if you try to call a method taking an `(implicit parser: StrParser[T])` from another method that does not have such an implicit available, the compiler will also raise an error:

```scala
@ def genericMethodWithoutImplicit[T](s: String) = parseFromString[T](s)
cmd2.sc:1: could not find implicit value for parameter parser:
         ammonite.$sess.cmd0.StrParser[T]
def genericMethodWithoutImplicit[T](s: String) = parseFromString[T](s)
                                                                  ^
Compilation Failed
```
<div align="right">*</> 5.47.scala*</div>

Most of the things we have done with Typeclass Inference could also be achieved using runtime reflection. However, relying on runtime reflection is fragile, and it is very easy for mistakes, bugs, or mis-configurations to make it to production before failing catastrophically. In contrast, Scala's `implicit` feature lets you achieve the same outcome but in a safe fashion: mistakes are caught early at compile-time, and you can fix them at your leisure rather than under the pressure of a ongoing production outage.

5.5.4 Recursive Typeclass Inference

We have already seen how we can use the typeclass technique to automatically pick which `StrParser` to use based on the type we want to parse to. This can also work for more complex types, where we tell the compiler we want a `Seq[Int]`, `(Int, Boolean)`, or even nested types like `Seq[(Int, Boolean)]`, and the compiler will automatically assemble the logic necessary to parse the type we want.

5.5.4.1 Parsing Sequences

For example, the following `ParseSeq` function provides a `StrParser[Seq[T]]` for any `T` which itself has an implicit `StrParser[T]` in scope:

```scala
implicit def ParseSeq[T](implicit p: StrParser[T]) = new StrParser[Seq[T]]{
  def parse(s: String) = s.split(',').toSeq.map(p.parse)
}
```
<div align="right">*</> 5.48.scala*</div>

Note that unlike the `implicit object`s we defined earlier which are singletons, here we have an `implicit def`. Depending on the type `T`, we would need a different `StrParser[T]`, and thus need a different `StrParser[Seq[T]]`. `implicit def ParseSeq` would thus return a different `StrParser` each time it is called with a different type `T`.

From this one defintion, we can now parse `Seq[Boolean]`s, `Seq[Int]`s, etc.

```scala
@ parseFromString[Seq[Boolean]]("true,false,true")
res99: Seq[Boolean] = ArraySeq(true, false, true)


@ parseFromString[Seq[Int]]("1,2,3,4")
res100: Seq[Int] = ArraySeq(1, 2, 3, 4)
```
<div align="right">*</> 5.49.scala*</div>

What we are effectively doing is teaching the compiler how to produce a `StrParser[Seq[T]]` for any type `T` as long as it has an implicit `StrParser[T]` available. Since we already have `StrParser[Int]`, `StrParser[Boolean]`, and `StrParser[Double]` available, the `ParseSeq` method gives `StrParser[Seq[Int]]`, `StrParser[Seq[Boolean]]`, and `StrParser[Seq[Double]]` for free.

The `StrParser[Seq[T]]` we are instantiating has a parse method that receives a parameter `s: String` and returns a `Seq[T]`. We just needed to implement the logic necessary to do that transformation, which we have done in the code snippet above.

5.5.4.2 Parsing Tuples

Similar to how we defined an `implicit def` to parse `Seq[T]`s, we could do the same to parse tuples. We do so below by assuming that tuples are represented by `key=value` pairs in the input string:

```scala
implicit def ParseTuple[T, V](implicit p1: StrParser[T], p2: StrParser[V]) =
  new StrParser[(T, V)]{
    def parse(s: String) = {
      val Array(left, right) = s.split('=')
      (p1.parse(left), p2.parse(right))
    }
  }
```
<div align="right">*</> 5.50.scala*</div>

This definition produces a `StrParser[(T, V)]`, but only for a type `T` and type `V` for which there are `StrParser`s available. Now we can parse tuples, as `=`-separated pairs:

```scala
@ parseFromString[(Int, Boolean)]("123=true")
res102: (Int, Boolean) = (123, true)

@ parseFromString[(Boolean, Double)]("true=1.5")
res103: (Boolean, Double) = (true, 1.5)
```
<div align="right">*</> 5.51.scala*</div>

5.5.4.3 Parsing Nested Structures

The two definitions above, `implicit def ParseSeq` and `implicit def ParseTuple`, are enough to let us also parse sequences of tuples, or tuples of sequences:

```scala
@ parseFromString[Seq[(Int, Boolean)]]("1=true,2=false,3=true,4=false")
res104: Seq[(Int, Boolean)] = ArraySeq((1, true), (2, false), (3, true), (4, false))

@ parseFromString[(Seq[Int], Seq[Boolean])]("1,2,3,4,5=true,false,true")
res105: (Seq[Int], Seq[Boolean]) = (ArraySeq(1, 2, 3, 4, 5), ArraySeq(true, false, true))
```
<div align="right">*</> 5.52.scala*</div>

Note that in this case we cannot handle nested `Seq[Seq[T]]`s or nested tuples due to how we're naively splitting the input string. A more structured parser handles such cases without issues, allowing us to specify an arbitrarily complex output type and automatically inferring the necessary parser. We will use a serialization library that uses this technique in ***Chapter 8: JSON and Binary Data Serialization***.

Most statically typed programming languages can infer types to some degree: even if not every expression is annotated with an explicit type, the compiler can still figure out the types based on the program structure. Typeclass derivation is effectively the reverse: by providing an explicit type, the compiler can infer the program structure necessary to provide a value of the type we are looking for.

In the example above, we just need to define how to handle the basic types - how to produce a `StrParser[Boolean]`, `StrParser[Int]`, `StrParser[Seq[T]]`, `StrParser[(T, V)]` - and the compiler is able to figure out how to produce a `StrParser[Seq[(Int, Boolean)]]` when we need it.

> See example **5.6 - TypeclassInference**

5.6 Conclusion

In this chapter, we have explored some of the more unique features of Scala. Case Classes or Pattern Matching you will use on a daily basis, while By-Name Parameters, Implicit Parameters, or Typeclass Inference are more advanced tools that you might only use when dictated by a framework or library. Nevertheless, these are the features that make the Scala language what it is, providing a way to tackle difficult problems more elegantly than most mainstream languages allow.

We have walked through the basic motivation and use cases for these features in this chapter. You will get familiar with more use cases as we see the features in action throughout the rest of this book.

This chapter will be the last in which we discuss the Scala programming language in isolation: subsequent chapters will introduce you to much more complex topics like working with your operating system, remote services, and third-party libraries. The Scala language fundamentals you have learned so far will serve you well as you broaden your horizons, from learning about the Scala language itself to using the Scala language to solve real-world problems.

Exercise: Define a function that uses pattern matching on the `Expr`s we saw earlier to perform simple algebraic simplifications:

`(1 + 1)`	`2`
`((1 + 1) * x)`	`(2 * x)`
`((2 - 1) * x)`	`x`
`(((1 + 1) * y) + ((1 - 1) * x))`	`(2 * y)`

See example **5.7 - Simplify**

Exercise: Modify the `def retry` function earlier that takes a by-name parameter and make it perform an exponential backoff, sleeping between retries, with a configurable initial `delay` in milliseconds:

```scala
retry(max = 50, delay = 100 /*milliseconds*/) {
  requests.get(s"$httpbin/status/200,400,500")
}                                                </> 5.53.scala
```

See example **5.8 - Backoff**

Exercise: Modify the typeclass-based `parseFromString` method we saw earlier to take a JSON-like format, where lists are demarcated by square brackets with comma-separated elements. This should allow it to parse and construct arbitrarily deep nested data structures automatically via typeclass inference:

```scala
@ parseFromString[Seq[Boolean]]("[true,false,true]") // 1 layer of nesting
res1: Seq[Boolean] = List(true, false, true)

@ parseFromString[Seq[(Seq[Int], Seq[Boolean])]]( // 3 layers of nesting
    "[[[1],[true]],[[2,3],[false,true]],[[4,5,6],[false,true,false]]]"
  )
res2: Seq[(Seq[Int], Seq[Boolean])] = List(
  (List(1), List(true)),
  (List(2, 3), List(false, true)),
  (List(4, 5, 6), List(false, true, false))
)

@ parseFromString[Seq[(Seq[Int], Seq[(Boolean, Double)])]]( // 4 layers of nesting
    "[[[1],[[true,0.5]]],[[2,3],[[false,1.5],[true,2.5]]]]"
  )
res3: Seq[(Seq[Int], Seq[(Boolean, Double)])] = List(
  (List(1), List((true, 0.5))),
  (List(2, 3), List((false, 1.5), (true, 2.5)))
)                                                          </> 5.54.scala
```

A production-ready version of this `parseFromString` method exists in `upickle.default.read`, which we will see in *Chapter 8: JSON and Binary Data Serialization*.

See example 5.9 - Deserialize

Exercise: How about using typeclasses to generate JSON, rather than parse it? Write a `writeToString` method that uses a `StrWriter` typeclass to take nested values parsed by `parseFromString`, and serialize them to the same strings they were parsed from.

```scala
@ writeToString[Seq[Boolean]](Seq(true, false, true))
res1: String = "[true,false,true]"


@ writeToString(Seq(true, false, true)) // type can be inferred
res2: String = "[true,false,true]"


@ writeToString[Seq[(Seq[Int], Seq[Boolean])]](
    Seq(
      (Seq(1), Seq(true)),
      (Seq(2, 3), Seq(false, true)),
      (Seq(4, 5, 6), Seq(false, true, false))
    )
  )
res3: String = "[[[1],[true]],[[2,3],[false,true]],[[4,5,6],[false,true,false]]]"


@ writeToString(
    Seq(
      (Seq(1), Seq((true, 0.5))),
      (Seq(2, 3), Seq((false, 1.5), (true, 2.5)))
    )
  )
res4: String = "[[[1],[[true,0.5]]],[[2,3],[[false,1.5],[true,2.5]]]]"</> 5.55.scala
```

See example 5.10 - Serialize

Discuss Chapter 5 online at https://www.handsonscala.com/discuss/5

Part II: Local Development

The second part of this book explores the core tools and techniques necessary for writing Scala applications that run on a single computer. We will cover algorithms, files and subprocess management, data serialization, scripts and build pipelines. This chapter builds towards a capstone project where we write an efficient incremental static site generator using the Scala language.

6

Implementing Algorithms in Scala

```scala
def breadthFirstSearch[T](start: T, graph: Map[T, Seq[T]]): Set[T] = {
  val seen = collection.mutable.Set(start)
  val queue = collection.mutable.ArrayDeque(start)
  while (queue.nonEmpty) {
    val current = queue.removeHead()
    for (next <- graph(current) if !seen.contains(next)) {
      seen.add(next)
      queue.append(next)
    }
  }
  seen.toSet
}                                                                  </> 6.1.scala
```

Snippet 6.1: a simple breadth-first-search algorithm we will implement using Scala in this chapter

In this chapter, we will walk you through the implementation of a number of common algorithms using the Scala programming language. These algorithms are commonly taught in schools and tested at professional job interviews, so you have likely seen them before.

By implementing them in Scala, we aim to get you more familiar with using the Scala programming language to solve small problems in isolation. We will also see how some of the unique language features we saw in **Chapter 5: Notable Scala Features** can be applied to simplify the implementation of these well-known algorithms. This will prepare us for subsequent chapters which will expand in scope to include many different kinds of systems, APIs, tools and techniques.

7

Files and Subprocesses

```scala
@ os.walk(os.pwd).filter(os.isFile).map(p => (os.size(p), p)).sortBy(-_._1).take(5)
res60: IndexedSeq[(Long, os.Path)] = ArrayBuffer(
  (6340270L, /Users/lihaoyi/test/post/Reimagining/GithubHistory.gif),
  (6008395L, /Users/lihaoyi/test/post/SmartNation/routes.json),
  (5499949L, /Users/lihaoyi/test/post/slides/Why-You-Might-Like-Scala.js.pdf),
  (5461595L, /Users/lihaoyi/test/post/slides/Cross-Platform-Development-in-Scala.js.pdf),
  (4576936L, /Users/lihaoyi/test/post/Reimagining/FluentSearch.gif)
)
                                                                    </> 7.1.scala
```

Snippet 7.1: a short Scala code snippet to find the five largest files in a directory tree

Working with files and subprocesses is one of the most common things you do in programming: from the Bash shell, to Python or Ruby scripts, to large applications written in a compiled language. At some point everyone will have to write to a file or talk to a subprocess. This chapter will walk you through how to perform basic file and subprocess operations in Scala.

This chapter finishes with two small projects: building a simple file synchronizer, and building a streaming subprocess pipeline. These projects will form the basis for **Chapter 17: Multi-Process Applications** and **Chapter 18: Building a Real-time File Synchronizer**

8

JSON and Binary Data Serialization

```scala
@ val output = ujson.Arr(
    ujson.Obj("hello" -> "world", "answer" -> 42),
    true
  )

@ output(0)("hello") = "goodbye"

@ output(0)("tags") = ujson.Arr("awesome", "yay", "wonderful")

@ println(output)
[{"hello":"goodbye","answer":42,"tags":["awesome","yay","wonderful"]},true] </> 8.1.scala
```

Snippet 8.1: manipulating a JSON tree structure in the Scala REPL

Data serialization is an important tool in any programmer's toolbox. While variables and classes are enough to store data within a process, most data tends to outlive a single program process: whether saved to disk, exchanged between processes, or sent over the network. This chapter will cover how to serialize your Scala data structures to two common data formats - textual JSON and binary MessagePack - and how you can interact with the structured data in a variety of useful ways.

The JSON workflows we learn in this chapter will be used later in *Chapter 12: Working with HTTP APIs* and *Chapter 14: Simple Web and API Servers*, while the binary serialization techniques we learn here will be used later in *Chapter 17: Multi-Process Applications*.

9

Self-Contained Scala Scripts

```scala
os.write(
  os.pwd / "out" / "index.html",
  doctype("html")(
    html(
      body(
        h1("Blog"),
        for ((_, suffix, _) <- postInfo)
        yield h2(a(href := ("post/" + mdNameToHtml(suffix)))(suffix))
      )
    )
  )
)
                                                           </> 9.1.scala
```

Snippet 9.1: rendering a HTML page using the third-party Scalatags HTML library

Scala Scripts are a great way to write small programs. Each script is self-contained and can download its own dependencies when necessary, and make use of both Java and Scala libraries. This lets you write and distribute scripts without spending time fiddling with build configuration or library installation.

In this chapter, we will write a static site generator script that uses third-party libraries to process Markdown input files and generate a set of HTML output files, ready for deployment on any static file hosting service. This will form the foundation for *Chapter 10: Static Build Pipelines*, where we will turn the static site generator into an efficient incremental build pipeline by using the Mill build tool.

10

Static Build Pipelines

```scala
import mill._

def srcs = T.source(millSourcePath / "src")

def concat = T{
  os.write(T.dest / "concat.txt",  os.list(srcs().path).map(os.read(_)))
  PathRef(T.dest / "concat.txt")
}
```
</> 10.1.scala

Snippet 10.1: the definition of a simple Mill build pipeline

Build pipelines are a common pattern, where you have files and assets you want to process but want to do so efficiently, incrementally, and in parallel. This usually means only re-processing files when they change, and re-using the already processed assets as much as possible. Whether you are compiling Scala, minifying Javascript, or compressing tarballs, many of these file-processing workflows can be slow. Parallelizing these workflows and avoiding unnecessary work can greatly speed up your development cycle.

This chapter will walk through how to use the Mill build tool to set up these build pipelines, and demonstrate the advantages of a build pipeline over a naive build script. We will take the the simple static site generator we wrote in *Chapter 9: Self-Contained Scala Scripts* and convert it into an efficient build pipeline that can incrementally update the static site as you make changes to the sources. We will be using the Mill build tool in several of the projects later in the book, starting with *Chapter 14: Simple Web and API Servers*.
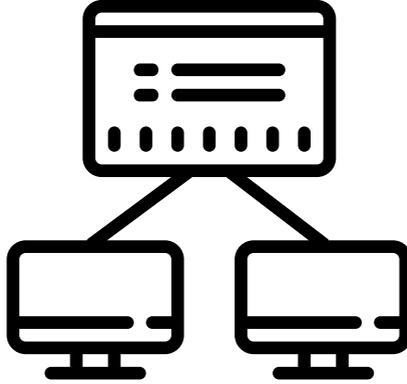
Part III: Web Services

The third part of this book covers using Scala in a world of servers and clients, systems and services. We will explore using Scala both as a client and as a server, exchanging HTML and JSON over HTTP or Websockets. This part builds towards two capstone projects: a parallel web crawler and an interactive chat website, each representing common use cases you are likely to encounter using Scala in a networked, distributed environment.

11

Scraping Websites

```scala
@ val doc = Jsoup.connect("http://en.wikipedia.org/").get()

@ doc.title()
res2: String = "Wikipedia, the free encyclopedia"

@ val headlines = doc.select("#mp-itn b a")
headlines: select.Elements =
<a href="/wiki/Bek_Air_Flight_2100" title="Bek Air Flight 2100">Bek Air Flight 2100</a>
<a href="/wiki/Assassination_of_..." title="Assassination of ...">2018 killing</a>
<a href="/wiki/State_of_the_..." title="State of the...">upholds a ruling</a>
...
                                                                    </> 11.1.scala
```

Snippet 11.1: scraping Wikipedia's front-page links using the Jsoup third-party library in the Scala REPL

The user-facing interface of most networked systems is a website. In fact, often that is the *only* interface! This chapter will walk you through using the Jsoup library from Scala to scrape human-readable HTML pages, unlocking the ability to extract data from websites that do not provide access via an API.

Apart from third-party scraping websites, Jsoup is also a useful tool for testing the HTML user interfaces that we will encounter in **Chapter 14: Simple Web and API Servers**. This chapter is also a chance to get more familiar with using Java libraries from Scala, a necessary skill to take advantage of the broad and deep Java ecosystem. Lastly, it is an exercise in doing non-trivial interactive development in the Scala REPL, which is a great place to prototype and try out pieces of code that are not ready to be saved in a script or project.

12

Working with HTTP APIs

```scala
@ requests.post(
    "https://api.github.com/repos/lihaoyi/test/issues",
    data = ujson.Obj("title" -> "hello"),
    headers = Map("Authorization" -> s"token $token")
  )
res1: requests.Response = Response(
  "https://api.github.com/repos/lihaoyi/test/issues",
  201,
  "Created",
...                                                            </> 12.1.scala
```

Snippet 12.1: interacting with Github's HTTP API from the Scala REPL

HTTP APIs have become the standard for any organization that wants to let external developers integrate with their systems. This chapter will walk you through how to access HTTP APIs in Scala, building up to a simple use case: migrating Github issues from one repository to another using Github's public API.

We will build upon techniques learned in this chapter in *Chapter 13: Fork-Join Parallelism with Futures*, where we will be writing a parallel web crawler using the Wikipedia JSON API to walk the graph of articles and the links between them.

13

Fork-Join Parallelism with Futures

```scala
def fetchAllLinksParallel(startTitle: String, depth: Int): Set[String] = {
  var seen = Set(startTitle)
  var current = Set(startTitle)
  for (i <- Range(0, depth)) {
    val futures = for (title <- current) yield Future{ fetchLinks(title) }
    val nextTitleLists = futures.map(Await.result(_, Inf))
    current = nextTitleLists.flatten.filter(!seen.contains(_))
    seen = seen ++ current
  }
  seen
}                                                            </> 13.1.scala
```

Snippet 13.1: a simple parallel web-crawler implemented using Scala Futures

The Scala programming language comes with a Futures API. Futures make parallel and asynchronous programming much easier to handle than working with traditional techniques of threads, locks, and callbacks.

This chapter dives into Scala's Futures: how to use them, how they work, and how you can use them to parallelize data processing workflows. It culminates in using Futures together with the techniques we learned in **Chapter 12: Working with HTTP APIs** to write a high-performance concurrent web crawler in a straightforward and intuitive way.

14

Simple Web and API Servers

```scala
object MinimalApplication extends cask.MainRoutes {
  @cask.get("/")
  def hello() = {
    "Hello World!"
  }

  @cask.post("/do-thing")
  def doThing(request: cask.Request) = {
    request.text().reverse
  }

  initialize()
}
```
</> 14.1.scala

Snippet 14.1: a minimal Scala web application, using the Cask web framework

Web and API servers are the backbone of internet systems. While in the last few chapters we learned to access these systems from a *client's* perspective, this chapter will teach you how to provide such APIs and Websites from the *server's* perspective. We will walk through a complete example of building a simple real-time chat website serving both HTML web pages and JSON API endpoints. We will re-visit this website in **Chapter 15: Querying SQL Databases**, where we will convert its simple in-memory datastore into a proper SQL database.

15

Querying SQL Databases

```scala
@ ctx.run(query[City].filter(_.population > 5000000).filter(_.countryCode == "CHN"))
res16: List[City] = List(
  City(1890, "Shanghai", "CHN", "Shanghai", 9696300),
  City(1891, "Peking", "CHN", "Peking", 7472000),
  City(1892, "Chongqing", "CHN", "Chongqing", 6351600),
  City(1893, "Tianjin", "CHN", "Tianjin", 5286800)
)
```
</> 15.1.scala

Snippet 15.1: using the Quill database query library from the Scala REPL

Most modern systems are backed by relational databases. This chapter will walk you through the basics of using a relational database from Scala, using the Quill query library. We will work through small self-contained examples of how to store and query data within a Postgres database, and then convert the interactive chat website we implemented in **Chapter 14: Simple Web and API Servers** to use a Postgres database for data storage.
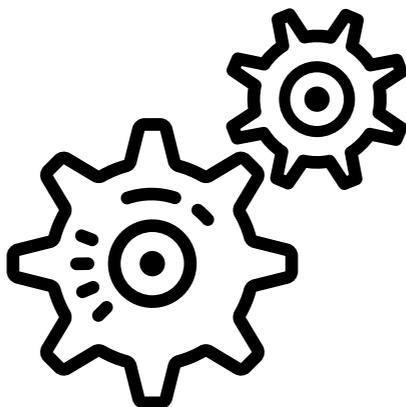
Part IV: Program Design

The fourth and last part of this book explores different ways of structuring your Scala application to tackle real-world problems. This chapter builds towards another two capstone projects: building a real-time file synchronizer and building a programming-language interpreter. These projects will give you a glimpse of the very different ways the Scala language can be used to implement challenging applications in an elegant and intuitive manner.

16

Message-based Parallelism with Actors

```scala
class SimpleUploadActor()(implicit cc: castor.Context) extends castor.SimpleActor[String]{
  def run(msg: String) = {
    val res = requests.post("https://httpbin.org/post", data = msg)
    println("response " + res.statusCode)
  }
}                                                                      </> 16.1.scala
```

Snippet 16.1: a simple actor implemented in Scala using the Castor library

Message-based parallelism is a technique that involves splitting your application logic into multiple "actors", each of which can run concurrently, and only interacts with other actors by exchanging asynchronous messages. This style of programming was popularized by the Erlang programming language and the Akka Scala actor library, but the approach is broadly useful and not limited to any particular language or library.

This chapter will introduce the fundamental concepts of message-based parallelism with actors, and how to use them to achieve parallelism in scenarios where the techniques we covered in *Chapter 13: Fork-Join Parallelism with Futures* cannot be applied. We will first discuss the basic actor APIs, see how they can be used in a standalone use case, and then see how they can be used in more involved multi-actor pipelines. The techniques in this chapter will come in useful later in *Chapter 18: Building a Real-time File Synchronizer*.

17

Multi-Process Applications

```scala
def send[T: Writer](out: DataOutputStream, msg: T): Unit = {
  val bytes = upickle.default.writeBinary(msg)
  out.writeInt(bytes.length)
  out.write(bytes)
  out.flush()
}
def receive[T: Reader](in: DataInputStream) = {
  val buf = new Array[Byte](in.readInt())
  in.readFully(buf)
  upickle.default.readBinary[T](buf)
}
                                                                    </> 17.1.scala
```

Snippet 17.1: RPC send and receive methods for sending data over an operating system pipe or network

While all our programs so far have run within a single process, in real world scenarios you will be working as part of a larger system, and the application itself may need to be split into multiple processes. This chapter will walk you through how to do so: configuring your build tool to support multiple Scala processes, sharing code and exchanging serialized messages. These are the building blocks that form the foundation of any distributed system.

As this chapter's project, we will be building a simple multi-process file synchronizer that can work over a network. This chapter builds upon the simple single-process file synchronizer in **Chapter 7: Files and Subprocesses**, and will form the basis for **Chapter 18: Building a Real-time File Synchronizer**.

18

Building a Real-time File Synchronizer

```scala
object SyncActor extends castor.SimpleActor[Msg]{
  def run(msg: Msg): Unit = msg match {
    case ChangedPath(value) => Shared.send(agent.stdin.data, Rpc.StatPath(value))
    case AgentResponse(Rpc.StatInfo(p, remoteHash)) =>
      val localHash = Shared.hashPath(src / p)
      if (localHash != remoteHash && localHash.isDefined) {
        Shared.send(agent.stdin.data, Rpc.WriteOver(os.read.bytes(src / p), p))
      }
  }
}
                                                            </> 18.1.scala
```

Snippet 18.1: an actor used as part of our real-time file synchronizer

In this chapter, we will write a file synchronizer that can keep the destination folder up to date even as the source folder changes over time. This chapter serves as a capstone project, tying together concepts from *Chapter 17: Multi-Process Applications* and *Chapter 16: Message-based Parallelism with Actors*.

The techniques in this chapter form the basis for "event driven" architectures, which are common in many distributed systems. Real-time file synchronization is a difficult problem, and we will see how we can use the Scala language and libraries to approach it in an elegant and understandable way.

19

Parsing Structured Text

```scala
@ def parser[_: P] =
    P( ("hello" | "goodbye").! ~ " ".rep(1) ~ ("world" | "seattle").! ~ End )


@ fastparse.parse("hello seattle", parser(_))
res41: Parsed[(String, String)] = Success(("hello", "seattle"), 13)


@ fastparse.parse("hello    world", parser(_))
res42: Parsed[(String, String)] = Success(("hello", "world"), 15)
```

</> 19.1.scala

Snippet 19.1: parsing simple text formats using the FastParse library

One common programming task is parsing structured text. This chapter will introduce how to parse text in Scala using the FastParse library, before diving into an example where we write a simple arithmetic parser in Scala. This will allow you to work competently with unusual data formats, query languages, or source code for which you do not already have an existing parser at hand.

We will build upon the parsing techniques learned in this chapter as part of **Chapter 20: Implementing a Programming Language**.

20

Implementing a Programming Language

```scala
def evaluate(expr: Expr, scope: Map[String, Value]): Value = expr match {
  case Expr.Str(s) => Value.Str(s)
  case Expr.Dict(kvs) => Value.Dict(kvs.map{case (k, v) => (k, evaluate(v, scope))})
  case Expr.Plus(left, right) =>
    val Value.Str(leftStr) = evaluate(left, scope)
    val Value.Str(rightStr) = evaluate(right, scope)
    Value.Str(leftStr + rightStr)
}                                                                 </> 20.1.scala
```

Snippet 20.1: evaluating a syntax tree using pattern matching

This chapter builds upon the simple parsers we learned in ***Chapter 19: Parsing Structured Text***, and walks you through the process of implementing a simple programming language in Scala.

Working with programming language source code is a strength of Scala: parsing, analyzing, compiling, or interpreting it. This chapter should will you how easy it is to write a simple interpreter to parse and evaluate program source code in Scala. Even if your goal is not to implement an entirely new programming language, these techniques are still useful: for writing linters, program analyzers, query engines, and other such tools.
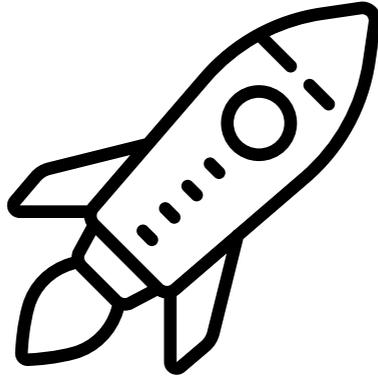
Conclusion

This is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.

Winston Churchill

If you have made it this far through *Hands-on Scala*, you should by now be comfortable using the Scala programming language in a wide range of scenarios. You've implemented algorithms, API clients, web servers, file synchronizers, and programming languages. You've dealt with concurrency and parallelism. You've worked with the filesystem, databases, data serialization, and many other cross-cutting concerns that you would find in any real-world software system.

This book only walks you through a narrow slice of the Scala ecosystem: there is a wealth of libraries and frameworks that people use writing Scala in production, and it is impossible to cover them all in one book. Nevertheless, the core concepts you learned here apply regardless of which specific toolset you end up using. Breadth-first search is breadth-first search, and a HTTP request is a HTTP request, regardless of how the exact method calls are spelled.

Scala is a flexible, broadly useful programming language. By now you have seen how Scala can be used to tackle even difficult, complex problems in an elegant and straightforward manner. While this book is not the final word in learning Scala, it should be enough for you to take off on your own, and get started solving real problems and delivering real value using the Scala language.