LEARN SCALA 3

THE FAST WAY!

Book One

THE ADVENTURE BEGINS!

ALVIN ALEXANDER

Learn Scala 3 The Fast Way!

Learn Scala 3
with small lessons, code examples,
and online exercises

Alvin Alexander

Copyright

Learn Scala 3 The Fast Way! (Book 1: The Adventure Begins)

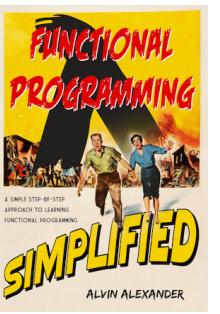
Version 1.0, published December 14, 2022
Version 0.4, published September 18, 2022

Buy the book and find/report issues:
alvinalexander.com/scala/learn-scala-3-the-fast-way-book[2]

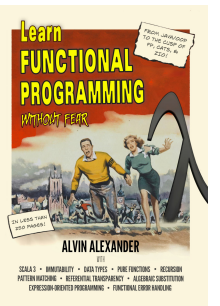
[1] https://alvinalexander.com
[2] https://alvinalexander.com/scala/learn-scala-3-the-fast-way-book

Other books by Alvin Alexander:

Scala Cookbook, 2nd Edition (Amazon.com)[3]



Functional Programming, Simplified
("The Big FP Book," alvinalexander.com)[4]



Learn Functional Programming Without Fear
("The Little FP Book," alvinalexander.com)[5]

[3]https://amzn.to/3du1pMR
[4]https://alvinalexander.com/scala/functional-programming-simplified-book
[5]https://alvinalexander.com//scala/learn-functional-programming-book

Contents

1

About The Author

Hi, my name is Alvin Alexander, and I'm the author of this book. I want to tell you a little about myself so I can share my qualifications, and also let you know why I wrote this book.

In terms of qualifications, I've written the following books on Scala, which have sold tens of thousands of copies:

- *Scala Cookbook, 1st Edition* (700+ pages)
- Scala Cookbook, 2nd Edition (700+ pages, written for Scala 3)[1]
- Functional Programming, Simplified (700+ pages)[2]
- An introductory book for Scala 2, titled *Hello, Scala*
- *Hello, Scala* became the basis for the official *Scala Book*
- When Scala 3 came out, I co-wrote the *Scala 3 Book* for the official Scala website
- Learn Functional Programming Without Fear[3] (a new, smaller book on FP, written for OOP developers)

All of those books are rated 4.5 stars and higher, and *Functional Programming, Simplified* has been one of the highest-rated, best-selling books on functional programming since its release.

In addition to these books I also write about Scala on my website, alvinalexander.com[4], which receives millions of page views every year, and I occasionally post small Scala tips on my Twitter account[5].

I like to think that my niche in the writing world is in making complicated topics easier to understand. That's always been my goal, and I'm glad to say that's what people usually tell me when they send me a "Thanks!" message.

[1] https://amzn.to/3du1pMR
[2] https://alvinalexander.com/scala/functional-programming-simplified-book
[3] https://alvinalexander.gumroad.com/l/learnfp
[4] https://alvinalexander.com
[5] https://twitter.com/alvinalexander

Why I wrote this book

Having written all those books, you might wonder why I'm writing another book about Scala. The first part of my answer is that I want to:

- Write a book about *Scala 3* for people who are new to Scala.
- Write it as a series of "one topic, short lessons" that are as simple as they can be.
- Keep it under 250 pages. (Having written three books over 700 pages long, I know those are hard to write, and can be intimidating to read.)

And finally, the biggest reason:

- *To help you retain what you read!*

To do that I've created:

- A Github repository[6] that includes most of the code shown, along with a few small projects that are good for beginners.
- Online exercises for the lessons.[7]

You'll learn more about all of this as you go through the book, so for now I'll leave it at that.

Thanks for reading, and I hope this is helpful!

All the best,
Alvin Alexander

Longmont, Colorado
September 18, 2022

[6] https://github.com/alvinj/LearnScala3TheFastWayBook1
[7] https://alvinalexander.com/book-exercises/LearnScala3Fast/

2

Welcome to Scala 3!

Now that you know about my background, let me welcome you to this book, which is an introductory book about the *Scala 3* programming language — the most modern, expressive, consistent, interesting, programming language I know.

Scala 3!

In this book I will often say "Scala 3" and not just "Scala." That's because Scala 3 — which was released in the summer of 2021 — has some significant changes from Scala 2, and I generally won't be writing about those differences. Scala 3 is the future of Scala, so that's all this book focuses on.

Who this book is for

I want to be really clear that this book is for *developers who are new to Scala*, and want to learn the latest version of Scala. It is for *beginners*, people who are new to Scala 3.

While I expect that you are new to Scala 3, I do assume that you have a little programming background. My assumptions are:

- You have used another programming language, such as Java, C#, Python, C, or another language
- This means that you have seen how to write at least a little bit of code, and also compile that code (if necessary) and run it
- You may have used Scala 2, but you're still beginning with it
- You are familiar enough with object-oriented programming (OOP) that you know what a class is
- You're comfortable working at your operating system command line

Why this book is unique

I think the most unique thing about this book is that it's for people who want to remember what they learn. When I came across the book, A Smarter Way to Learn JavaScript[1], I was really impressed with the online, interactive component of that book, so I've based a lot of my approach on that book, along with my additional research about how human beings learn and remember new things.

So helping you remember what you learn is THE primary goal of this book. To help achieve that goal, this book has a Github source code repository[2] you can download and experiment with, and a companion website with exercises for each lesson[3].

Many — many! — studies show that humans don't learn by simply reading. We have to do other things like work through exercises and write code to *retain* what we learn, so this emphasis is a HUGE thing that makes this book different and unique!

My own experience from my college days was that the only possible way I could pass a thermodynamics class was to work all the exercises. Initially I tried what I had always done — which was to read the book and then pass the tests — but that failed miserably (literally). That's when I came across this quote:

"One learns by doing the thing."

When I read that quote — which was in the thermodynamics book itself! — I realized I wasn't really putting in the work that was necessary to pass this class, and I clearly wasn't learning. This was one of those "lightbulb going on over your head" moments in life, and it became clear that the only way I was going to pass this class was to work all the exercises in the book.

[1] https://amzn.to/3CpnJ6t
[2] https://github.com/alvinj/LearnScala3TheFastWayBook1
[3] https://alvinalexander.com/book-exercises/LearnScala3Fast/

Exercises and code examples

When this book is complete I plan to provide a link directly to each lesson's exercises. But in this early release I'll just share this one link which takes you to the index of all the lessons:

- alvinalexander.com/book-exercises/LearnScala3Fast[4]

The exercises won't take long, and most importantly, they often present the material in a way that's different from the book to test your understanding.

> Initially I was a little hesitant about the value of this approach, but after I started writing the exercises and showing them to other people, I got really excited about how much these lessons can help people learn Scala faster. It's amazing what happens when you put a book down and then try to answer questions about what you just read! I expect that all of my books in the future will use this approach.

You'll also find that these exercises are a great way to test your knowledge in the future. For instance, imagine that you get away from using what you learned for a week or two. In that situation you can come back to the exercises (rather than re-reading the book) to see what you *really* remember.

In addition to the online exercises, the book also has a corresponding Github repository of example code that you can use and modify:

- The examples on Github[5]

This is a small Scala 3 book

One additional note I want to add is that writing a *small* book on Scala is hard. If you look at the landscape of Scala books you'll see that many of them are 600 pages or more. That's because of two things:

[4] https://alvinalexander.com/book-exercises/LearnScala3Fast/
[5] https://github.com/alvinj/LearnScala3TheFastWayBook1

- Scala has a lot of terrific features, and we want to write about them all
- Scala is an object-oriented programming (OOP) language as well as a functional programming (FP) language, and in some cases — such as covering the Scala collections classes — that means there are OOP and FP versions of each class

I've learned that if you want to write a book about *all* of Scala, it's just going to take a lot of pages.

This also means that if one of your goals is to write an introductory book in 250 pages or less, you can't write about everything. Therefore, I've had to make some tough decisions about what to include in this book and what to leave out.

So that's how this became a book for Scala 3 *beginners*. I decided to forget about writing about *everything* in one book, and just focus on the basics.

So my goal is to get you started on the basics of Scala 3, BUT, I do cover *many* features in this book, and once you understand them, I believe it will be much easier to learn the rest of them. I may follow this book up with another book to take your knowledge to the next level, or you can learn those features in the Scala Cookbook (2nd Edition)[6], which was also written for Scala 3.

> Editor's Note: The book is now larger than 250 pages, in large part because I reformatted it to be easier to read.

[6] https://amzn.to/3du1pMR

3

Why Learn Scala 3?

I hope you already have some idea of what Scala 3 is good for, but if you don't, let me give you my completely biased opinion! :)

Scala offers a fusion of FP and OOP

Way back in 2010, technologies like Google Maps, Gmail, Facebook, and Twitter were all relatively new, and pretty much the only FP language anyone had heard of was Haskell. While I was wandering around Alaska, this was when I first learned about Scala, and I learned that a distinguishing feature of it is that from its origin it has been a fusion of FP and OOP.

Martin Odersky[1] is the creator of Scala, and if you don't know him, he studied under Niklaus Wirth, who created several programming languages, including Pascal, which was often used as a teaching language in colleges in the 1990s. Mr. Odersky originally became known to me (and many others) as the person who brought generics to Java in Java 5.

After that he created a research language named *Pizza*, and that work led him to create Scala. When he created it he strongly believed that a fusion of FP and OOP was possible, and has stated it like this:

> The essence of Scala is a fusion of FP and OOP in a typed setting, with functions for the logic, and objects for the modularity.

As this book progresses you'll see examples of what this means. But for now, just know that this Fusion of FP and OOP is a hallmark of the Scala language.

Scala is expressive

Scala is a concise language, but more importantly, it's *expressive*. This means that you can get a lot of meaning across in a small amount of characters, but

[1] https://en.wikipedia.org/wiki/Martin_Odersky

you can also come back to your code in the future and still read it.

For example, even though you may not have seen any Scala code before, I think you can look at these examples and see that there are no unnecessary characters in these examples, but they're all very readable:

```
val a = 1
val b = "two"

// a "for loop"
for i <- 1 to 3 do println(i)

// a scala method
def min(a: Int, b: Int): Int =
    if a < b then a else b
```

Sometimes when a language is advertised as concise it really means *terse*, which means that it will be hard to read later. But as you'll continue to see, Scala is *expressive*, not terse.

Scala is consistent

On the official Scala website I wrote the pages that compare Scala to other programming languages, and that work led me to realize that Scala is more *consistent* than other programming languages. This is important, because it means that you don't need to learn a lot of new concepts or weird variations of syntax for different conditions. For instance, this is the way you create some common data structures in Scala:

```
val a = List(1, 2, 3)
val b = ArrayBuffer(1, 2, 3)
val c = Set(1, 2, 3)
val d = Map(1 -> "a", 2 -> b)
```

Notice that each type of data structure is created the same way. This may seem like a small point, but in other languages data structures are created with `()`, `[]`, and `{}` symbols, while in Scala they are all just classes. You'll find this same consistency throughout the Scala language.

A terrific JVM language

Simply put, IMHO, Scala is the best programming language available on the Java Virtual Machine (JVM). The name "Scala" comes from the word *scalable*, and true to that name, it's used to power some of the busiest websites in the world. Scala is used by Apple, Disney, GM, Starbucks, Tesla, The Guardian, Twitter, and in many, many more products and companies.

Using Scala you can create server-side applications using frameworks like the Play Framework[2], and many others. The Akka[3] library has the best "actors" library in the world, and they also offer a serverless computing solution.

There are many other Scala libraries and frameworks, and these — along with other FP libraries listed below — power some of the most high-performance websites in the known universe!

The most modern FP libraries

Scala is also the home of two of the world's best and most modern functional programming libraries in Cats[4] and ZIO. It's amazing to think about it, but these truly are *World-Class, Best On Planet Earth* FP libraries. At the time of this writing *ZIO 2* is just being released, and it doesn't get any more modern than that.

A JavaScript replacement

Scala can also be compiled to JavaScript using the Scala.js[5] library. This means that instead of writing client-side applications using JavaScript, you can use Scala! For instance, the "exercises" website that accompanies this book is written with Scala.js. (I share lessons on how to get started with Scala.js in the Scala Cookbook, 2nd Edition[6].)

---

[2] https://www.playframework.com
[3] https://akka.io
[4] https://typelevel.org/cats
[5] https://www.scala-js.org
[6] https://amzn.to/3du1pMR

Native executables

Scala can also be compiled to *native executable* applications with the Scala Native and GraalVM tools. With these tools, Scala is a terrific language for writing command-line applications and microservices.

A great scripting language

Thanks to a tool named Scala-CLI[7] — which you'll learn about shortly — Scala is also a terrific scripting language. Personally, I love being able to write server-side applications, client-side applications, native executables, and scripts with the same programming language and its huge ecosystem of libraries.

Lots of libraries

In addition to FP libraries, the Scala ecosystem has *many* other libraries. You can use your favorite search engine to find libraries for specific tasks, but you can also use these two tools when you're looking for libraries for a specific task:

- Awesome Scala list[8]
- The Scaladex index[9]

The Awesome Scala list is a long list of projects, organized by categories, and shows the number of people who have "starred" projects, and what the projects' commit activity looks like. Scaladex is more of a search engine for Scala projects that shows even more data in its search results.

Scala will change how you think

As someone said many years ago, a great thing about Scala is that it will change how you think about programming. For instance, when I first used Java in the late 1990s I wrote *many, many* custom `for` loops because that's

[7] https://scala-cli.virtuslab.org
[8] https://github.com/lauris/awesome-scala
[9] https://index.scala-lang.org

just the way things were done.

But thanks to the Scala collections classes you'll see that almost all of those custom `for` loops fall into certain categories, like this:

- Filtering
- Transformational
- Informational

While in 2010 some people thought the methods on Scala's collections classes were unusual or overwhelming, these built-in methods are now basically industry standards. These methods mean that instead of writing code like this:

```
// old way to create a new list from an old list
val newList = LinkedList[Int]()
for
    i <- oldList
    if i > 5
do
    val j = i * 2
    newList ++ j
```

you do this:

```
val newList = oldList.filter(_ > 5)
                     .map(_ * 2)
```

Given the fact that `filter` and `map` are both de facto industry-standard methods, which code would you rather read?

I hope you'll agree that the second example is better, simpler, and still very readable. And if you've never seen anything like this before — fear not — you'll know how to read and write it by the end of this book!

> If you want to work on "big data" applications, it may also help to know that this is how you write code with Apache Spark[10].

[10] https://spark.apache.org

4

Your Setup For This Book

As we get close to the first programming lesson, let's start getting your computer environment set up.

For most of the examples that are included in the book, all you need is a browser. I show this as "Setup Option #2" below because I don't think it's the *best* approach, but it's still a good approach.

The reason it's not the best approach is because I think the easiest way to run *all* of the source code examples in the book's Github repository is to have Scala installed on your local computer. Therefore I present that as Option 1.

Setup Option 1: Install Scala CLI

I work at the command line all the time, so for the purposes of this book I recommend installing a tool named Scala CLI[1]. By installing just this one tool, you'll be able to run *all* of the examples I show in the book on your computer.

Some benefits of Scala CLI:

- It runs on your computer, so it's generally faster
- This *one tool* downloads everything you need to run Scala
- It can be used to run all of my Github source code snippets
- It can also be used to run all of my Github scripts (which are small but complete Scala applications)
- It lets you easily include third-party libraries in your code, such as libraries for HTTP, JSON, databases, testing, etc.
- If/when your project gets to be large, Scala CLI has an option to export your settings to a Scala build tool like *sbt* or *Mill*

[1] https://Scala%20CLI.virtuslab.org

The main drawback of Scala CLI is that it will download things like Scala 3 and a version of Java, so it may require a few GB of storage space on your computer. (And yes, I understand that can be a big consideration.)

If you want to install and use Scala CLI[2], just use the Installation link on their website.

> If you're curious about how it works, jump to the *Scala CLI* lesson near the end of this book and check out its examples.

Setup Option 2: Scastie

A second option is that if you don't want to install anything on your computer, you can also run *most* of the book's examples online at the Scastie[3] website.

Scastie is a tool that's created and maintained by the creators of Scala 3, and it lets you run Scala code in your browser. All you have to do is type in your code, press Run, and see the output.

If you're not sure

If you're not sure what you want, go ahead and start with Scastie, because it doesn't require anything to be installed on your computer. But if/when you get to a point where you want to start running things on your local computer, come back here and install Scala CLI.

Note

A third option — which in 2021 was the primary option for working at the command line — is to install the Scala SDK either by (a) manually downloading it, or (b) installing it with another tool like SDKMAN. However, I believe that Scala CLI is a significant improvement over this approach, so I recommend it instead.

[2] https://Scala%20CLI.virtuslab.org
[3] https://scastie.scala-lang.org

> At the time of this writing there is a proposal that Scala CLI should become the future of Scala at the command line, that's how good it is.

Exercises and examples

As I mentioned in the *Welcome to Scala 3!* chapter, the main page for the online exercises can be found at this URL:

- The online exercises[4]

And the Github examples are at this URL:

- The examples on Github[5]

[4] https://alvinalexander.com/book-exercises/LearnScala3Fast/
[5] https://github.com/alvinj/LearnScala3TheFastWayBook1

5

Note 1: Significant Indentation Syntax

A BIG change from Scala 2 to Scala 3 is the introduction of something known as *significant indentation syntax*. This means that instead of writing code like this in Scala 2:

```
for (i <- 1 to 10) {
    println(i)
}
```

we now write that code without the curly braces, like this:

```
for i <- 1 to 10 do
    println
```

In short, Scala 3 gets rid of most curly braces and *indentation* is now important and significant. This new approach is consistent with languages like Python and Haskell, and more importantly, it's cleaner and easier to read. And because programmers spend about 10 times as much time *reading* code as we do *writing* code, readability is huge. (I won't be surprised if all major programming languages adopt this new style in the next 10 years.)

Indenting with four spaces

With this new indentation style I prefer to indent my code with four spaces, and that's what this book uses. Many Scala programmers use two spaces, but I think four spaces makes the code easier to read, so that's why I use it.

I also find that using four spaces makes it more obvious when your functions are getting too long. When you're indenting your code so much that it starts going off the right side of the screen, that's a great hint that you should probably break your functions down into smaller functions.

About those curly braces

I need to mention that technically you can still use curly braces if you want, but pretty much every book and learning resource for Scala 3 — including those created by the Scala Center and the creator of Scala, Martin Odersky — uses the significant indentation syntax. So you can still use curly braces, but it's not the recommended approach.

6

Note 2: Comments

We'll start writing code in the next lesson, but before doing that I also need to note that Scala uses the same comment style that's used by Java and other C-style programming languages. So you can write comments in either of these three ways:

```
// a one-line comment

/*
 * a multi-line comment.
 * more comment stuff here.
 */

/**
 * also a multi-line comment
 * with more comment stuff here.
 */
```

You can also create a one-line comment using this style, but we rarely do because you can just use `//`:

```
/* can also do this, but we rarely do */
```

I'll use comments with many code examples, so I needed to mention this before we start. For example, one thing I often do is to show the result of a computation after a `//` comment, such as this:

```
val a = 1
val b = 2
val c = a + b    // c is 3
```

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

7

Note 3: Something You'll See in the Github Examples

As I created the Github repository of examples[1], I realized that there's one thing I need to mention here in the book, and it goes as follows.

In Scala you can create a *block of code* using curly braces, like this:

```
val a = {
    println("Hello")
    42
}
```

When you do this, the value of the last expression inside the code block is what's assigned to the variable `a`. So in this example the `println` statement executes, and then the last expression inside the block is the integer value `42`. If you copy and paste that code into the REPL, you'll see that `a` has the type `Int` and the value `42`:

```
scala> val a = {
     |     println("Hello")
     |     42
     | }
     |
Hello
val a: Int = 42
```

You can use this technique anywhere you want to in your Scala code, and if there's a second book in this series I'll explain it more there.

> This "block of code" technique is used all the time in Scala, so I'm glad that the issue I ran into forced me to mention it here.

[1] https://github.com/alvinj/LearnScala3TheFastWayBook1

The scope of variables inside a block

But for the purposes of this book's examples, I use this technique for another reason: Any variables that are declared inside a block can only be seen within that block.

In the book's Github repository I use curly braces any time I use the same variable name more than once in a file. For example, if I use the variable name `smallInts` in the `filter` lesson, like this:

```
val smallInts = ints.filter(_ < 3)
println(smallInts)
```

and then use that name again in the same file, I can enclose one or both instances in the Github examples inside curly braces:

```
// 1st example
{
val smallInts = ints.filter(_ < 3)
println(smallInts)
}

// 2nd example
{
val smallInts = ints.filter(i => i < 3)
println(smallInts)
}
```

This technique lets me use the same variable names in the examples that I use in the book's lessons, and I hope this helps you follow the examples more easily.

I don't use this second technique to control scope in my normal Scala code, but when I started creating the Github repository I thought this was the best way to keep the same variable names there that I use in the book.

> If this doesn't make sense yet — fear not! — I believe it will make sense after you see the lessons on variable names, and then the lessons about the `filter` method.

8

Beginning: The Scala REPL

Okay, let's start writing some code!

Assuming you have Scala-CLI installed on your computer, this lesson shows how to start something known as a REPL so you can start writing code. Remember that if you don't have Scala installed on your computer, you can also use the Scastie[1] website.

The REPL

The acronym *REPL* stands for "read/evaluate/print/loop," and it's an interactive tool that lets you write Scala code. I often refer to it as a playground or laboratory, because it's a place where you can run experiments on Scala code to make sure it works like you expect it to. Or if you're not familiar with a language feature or library, the REPL is a place where you can experiment with it.

If you're using Scala-CLI, start the Scala REPL like this from your operating system command line:

```
$ scala-cli repl
```

Or, if you have the Scala 3 SDK[2] installed, start the REPL like this:

```
$ scala
```

In either case you should see a result that looks like this:

```
Welcome to Scala 3.1.1
Type in expressions for evaluation. Or try :help.

scala> _
```

[1] https://scastie.scala-lang.org
[2] https://www.scala-lang.org/download

The `scala>` prompt indicates that you're now inside an interactive REPL session. In here you can experiment with writing Scala code:

```
scala> val x = 1
x: Int = 1

scala> val y = 2
y: Int = 2

scala> x + y
res0: Int = 3
```

As shown in these examples:

- You generally create new variables in Scala with the `val` keyword. (You'll see more on this in the lessons that follow.)
- `x` and `y` are the names of two variables that I created.
- After you enter your code and press the [Enter] key, the REPL output shows the result of your expression, including the variable name you gave it, its data type (such as `Int`), and its value.
- If you don't assign a variable name, as in the third example, the REPL creates its own variable, beginning with the name `res0`, then `res1`, etc. You can then use these variable names just as though you had created them yourself:

```
scala> res0 * 3
res1: Int = 9
```

This is how the REPL works: type your expressions, and see their results. This is why I refer to this as a playground, or a place to experiment.

Tab completion

One "trick" in the REPL is that you can type a value or the name of a variable, then type a decimal, and then press the [Tab] key. The REPL responds by showing all of the methods that are available on your value, such as when you follow those steps on an integer like the number `1`:

```
scala> 1.[Tab]
!=                          finalize                    round
##                          floatValue                  self
```

```
%                           floor                       shortValue
&                           formatted                   sign
*                           getClass                    signum
many more methods listed here ...
```

What happens here is that `1` is an instance of the Scala type `Int`, which is an integer, and all of these methods are available on any integer. For instance, you can continue to type `abs` after the decimal to get the absolute value of an integer:

```
scala> 1.abs
val res1: Int = 1
```

Two REPL tips

At this point there are two other things to know about the REPL. First, you reset the REPL environment with its `:reset` command:

```
scala> :reset
Resetting REPL state.
```

This tells the REPL to forget everything you previously typed in, and to restore itself to its initial state.

Second, you quit a REPL session with the `:quit` command, or by typing the `[Control][d]` keystroke. Either of these ends your REPL session and returns you to your operating system command line.

Scastie, an online REPL

Remember that if you haven't installed Scala-CLI or the Scala 3 SDK on your system, you can also use the Scastie[3] website as an online REPL.

Scastie is a tool that's created and maintained by the creators of Scala 3, and it lets you run Scala code in your browser. You just enter your code, press Run, and see the output.

[3] https://scastie.scala-lang.org

I've worked at command line prompts for many years, so I prefer the REPL, but Scastie is also nice. Until we start writing scripts later in the book, either tool is fine.

Start using the REPL

If you haven't used a REPL environment before, I highly recommend experimenting with it now. Even experienced Scala developers often have a REPL session open while they're coding.

For example, type these expressions into the REPL to see their results:

```
val a = 2
val b = 4
val c = a * b
val d = c / 2
val e = d - 1
d == 3
d == 7
```

Those are all examples of how to work with integers, which have the type `Int` in Scala. Similarly, this example shows how to create a `String` and then get its length:

```
val s = "Hello, world"
s.length
```

Let the experimenting begin!

Exercises

The exercises for this lesson are available here[4].

[4] https://alvinalexander.com/book-exercises/LearnScala3Fast/

9

Beginning: Printing With println

The next important thing to know is how to print output to the command line. This lets you see the output of your calculations, and in Scala we do this with the `println` function:

```
println("Hello, world")
```

In that code, this text is a string — an instance of the Scala `String` class:

```
"Hello, world"
```

We call it a string because it's a *string of characters*.

As shown in that code, strings are enclosed in double-quotes. When you run that code in the REPL, it prints the string `Hello, world` to the command line.

> NOTE: Technically what it really does is print your string, followed by a *newline* character.

As with other programming languages you can concatenate two strings together with the + operator, like this:

```
println("Hello," + " world")
```

Both of those `println` statements print the same output.

> NOTE: As shown, you can use the + symbol to concatenate two strings, but there's a better way to do this, and you'll see that better approach shortly.

print

As mentioned, `println` prints your string, followed by a newline character. When you want to print a string that is *not* followed by a newline character, use `print` instead:

```
print("Hello, world")
```

There's no easy way for you to confirm yet that what I just wrote is true, but you can adjust some scripts later to use `print` instead of `println` so you can see the difference.

STDOUT and STDERR

Technically, the `println` function prints a string to "standard output," which is also known in the computer world as "STDOUT." In a script or command-line application this means that the string is printed to the command line. If instead you want to print a string to standard error (STDERR) — typically for error messages — use this function instead:

```
System.err.println("An error message")
```

In the Unix world you can redirect STDOUT and STDERR to different locations[1], so it's important to note this distinction.

Exercises

The exercises for this lesson are available here[2].

[1] https://alvinalexander.com/linux-unix/redirect-stdout-stderr-output-same-file-location
[2] https://alvinalexander.com/book-exercises/LearnScala3Fast/

10
Variables: val Fields

In every programming language you create *variables*, and in Scala you do this with the `val` keyword. For example, this is how you create a `String` variable:

```
val firstName = "Alvin"
```

In the REPL you can print the *value* in the variable `firstName` like this:

```
scala> println(firstName)
Alvin
```

Taking this a little further, given these two variables:

```
val firstName = "Alvin"
val lastName = "Alexander"
```

you can use those variables to create a new variable named `fullName` like this:

```
val fullName = firstName + " " + lastName
```

This is what you see when you print the `fullName` variable in the REPL:

```
scala> println(fullName)
Alvin Alexander
```

As you can infer from the examples so far, the general syntax for creating a new variable looks like this:

```
val theVariableName = theVariableValue
```

> TIP: As shown in these examples, in Scala the standard is to create variable names using camel case, like `firstName`, `lastName`, etc. (Conversely, we do *not* name them `first_name` and `last_name`.)

You can't modify val fields

An important thing about `val` fields is that they are *immutable*, meaning that they can't be changed. So if you create a `val` variable like this:

```
val x = 1
```

you can't update it to a new value later. If you try to give `x` a new value in the REPL you'll see an error message like this:

```
x = 2    // ERROR: Reassignment to val x
```

A `val` field in Scala is similar to a `final` field in Java, and like a `const` field in JavaScript.

Variable as in algebra

If it seems unusual that a *variable* can't vary, it's important to know that a `val` field is a "variable" in the algebraic meaning: just like in algebra, once you assign a value to a variable, it can't be changed.

This may seem like a limitation, but I ended up learning a *huge amount* about programming by following this one simple rule:

> Make every variable in Scala a `val` field, unless you have a good reason not to.

When you truly need a variable whose value can be modified (mutated), see the next lesson.

REPL experiments

I always recommend experimenting with things, and to help you get started, here are some experiments you can try in the REPL:

```
val a = 1
val b = (a + 2) * 2
b = 7   // this is an intentional error
val c = "hello"
```

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

11
Variables: var Fields

When you need to create a variable whose value *can* change over time, define the field as a `var` instead of `val`:

```
// assign a value to 'name'
var name = "Reginald Kenneth Dwight"
```

Because `name` is a `var`, you can later change its contents:

```
// some time later, give 'name' a new value
name = "Elton John"
```

As mentioned in the previous lesson, if you try to do this with a `val` field you'll generate an error, but with `var` fields this is perfectly legal.

The new value must be the same type

As you'll see in the upcoming lessons, Scala is a strongly-typed language, and one thing this means is that the variable `name` must always hold a `String` value. So this reassignment works:

```
name = "Fred"
```

but this fails:

```
name = 1   // compiler error
```

As you'll soon see, this is because `"Fred"` is an instance of a class named `String`, so `name` is created as a `String` variable. The attempt to reassign a `1` to `name` fails because `1` is an instance of a different class named `Int`.

Keeping track of data types is one way that Scala is a strongly-typed and type-safe language. Because it's type-safe, the Scala compiler will catch these errors at compile time. Or, if you use an IDE, it will catch them as you type.

But always start with val

Using `val` and `var` are the two ways to create variables in Scala. And now that you've seen both approaches, it's important to reiterate this point:

1. Always declare variables as `val`, unless
2. The variable really does need to vary over time, in which case you should create it as a `var`

If you follow this one simple rule, you'll find that you really don't need to use `var` fields that often. That makes your code safer because you don't have to worry about the variable being unexpectedly changed somewhere else in your code.

> TIP: This was something I never even thought about with Java. Maybe because I didn't take computer science classes in college it never occurred to me to specifically declare my intent like this, i.e., to mark 80-90% (or more) of my Java variables as `final`. This is just one of the ways Scala will change the way you think about programming, and help make you a better programmer.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

12

Variables: Explicitly Declaring The Data Type

In the previous examples I created string and integer variables like this:

```
val name = "Fred"     // String
val count = 1         // Int (an integer)
```

On the human side this syntax is nice because it's concise, and any programmer that has a little experience can tell that the `name` field contains a string and `count` contains an integer.

On the computer side, what happens here is that Scala is smart enough to *implicitly* know those data types, i.e., that `"Fred"` has the type `String` and `1` has the type `Int`. Back in the old days we had to manually declare these things, but there's no reason to do this any more.

Expressiveness

This syntax is also great because there are no extra characters to read! In other languages you have to type something like this to say the same thing:

```
final int count = 1;    // other languages
```

Instead, I would much rather read this Scala code:

```
val count = 1
```

As you'll see throughout this book, Scala is a "concise but readable" language, meaning that there are no wasted characters. There are just enough characters, but no more than that. Because of this, we call Scala *expressive*.

> Because programmers spend roughly ten times as much time *reading* code as we do *writing* code, an expressive language is a very good thing.

Explicitly declaring the data type

That being said, there are also times when it will be helpful to explicitly specify the *data type* of a variable. In those cases you specify the type after the variable name, like this:

```
val name: String = "John Doe"
```

In this specific example there's no reason to do this, but once I show more data types in the coming lessons, you'll see situations where this can be helpful. For now, all you need to know is that when you want to declare the variable's data type, this is the syntax you use:

```
val name:     String     = "John Doe"
    ----      ------       ----------
    the       the          the
    variable  variable     variable
    name      type         value
```

As another example, these are the two ways you create an `Int` (integer) variable in Scala:

```
val answer = 42          // implicit format
val answer: Int = 42     // explicit format
```

As you can see, there is no need to explicitly declare the variable type in these examples. Adding the type just makes your code more verbose, and in general, being a verbose programming language is bad — *verbose* is harder to read.

> When you want to explicitly declare the type when using a `var`, use the same syntax.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

13

Strings: Common Methods

Because `String` is a class, it has methods on it that you can use. This is the way classes work in OOP languages.

These examples demonstrate some of the common methods that you'll use on a `String`, with the results of each method shown after the comment:

```
val a = "hello, world"

a.length             // 12
a.capitalize         // "Hello, world"
a.toUpperCase        // "HELLO, WORLD"
a.indexOf("h")       // 0
a.indexOf("e")       // 1
a.substring(0, 2)    // "he"
a.substring(0, 3)    // "hel"
a.substring(1, 3)    // "el"
```

A `String` is an immutable data type, meaning that once it's created, it can never be changed. So when you call any of these methods, you always have to assign the result to a new variable:

```
val b = a.capitalize     // b: "Hello, world"
val c = a.toUpperCase    // c: "HELLO, WORLD"
```

Many more methods

There are actually *many* more methods available to a `String` instance. For instance, when I type a string in the REPL, then add a period, and then press the `[Tab]` key, the REPL tells me that there are 248 methods, to be precise:

```
scala> "yo".
JLine: do you wish to see all 248 possibilities (50 lines)?
```

But don't be intimidated by that because you'll probably only use 10 to 20 methods on a regular basis. After that it's nice to know that all these other

methods are there to help you solve your problems, but the most common 10-20 methods will get you through most days.

In the lessons that follow, you'll see many of these common methods.

Seq[Char]

It's worth noting that a Scala `String` can be treated as a `Seq[Char]` — sequence of characters — whenever you want to work with it this way. For example, you can access the elements in a `String` using the usual sequence syntax (specifying the index inside parentheses) and doing that returns the element as a `Char` value:

```
scala> val s = "hello"
val s: String = hello

scala> s(0)
val res0: Char = h
```

A `for` loop is another good example of this:

```
scala> for c <- s do println(c)
h
e
l
l
o
```

14

Strings: Interpolators

Before we move on to other data types, it will be helpful to demonstrate two more things about Scala strings:

- String interpolators
- Multiline strings

In this lesson we'll look at string interpolators, and in the next lesson we'll look at multiline strings.

String interpolators

Scala has the notion of "string interpolators." For example, given these two variables:

```
val firstName = "Alvin"
val lastName = "Alexander"
```

You *can* print the full name like this:

```
println(firstName + " " + lastName)   // prints "Alvin Alexander"
```

That's how we used to do things 20 years ago in other languages, but with Scala the preferred way to print that same result is like this:

```
println(s"$firstName $lastName")      // prints "Alvin Alexander"
```

Notice that the string is prepended with the character `s`:

```
println(s"$firstName $lastName")
        ^
```

This is Scala's way of letting you declare that the following string should be *interpolated*, meaning that the string contains variables that should be interpreted before they are put into the string.

I'll explain the reason for the `s` character in just a moment, but before doing that, it's important to note that when you use an *expression* inside an interpolated string, you need to use curly braces around the expression:

```
println(s"Two plus two equals ${2 + 2}")
println(s"Two times two equals ${2 * 2}")
```

So far I've shown this technique with `println` statements, but you can use it anywhere you use a `String`:

```
val x = s"Two plus two equals ${2 + 2}"
```

Other interpolators

Now I can explain the reason for the letter `s` that precedes the string: `s` is a method, and it provides just one way that a string can be interpolated. For instance, there's another built-in interpolator named `f` that lets you format strings just like the C programming language `printf` syntax.

For example, I once wrote a little logging library for Scala, and in it I use the `f` interpolator like this:

```
bw.write(f"$time | $logLevel%-5s | $classname | $msg\n")
```

This pads the second field (`logLevel`) to be five characters wide and make it left-justified, so its output looks like this:

```
04:52:51:541 | INFO  | Bar | this is an info message from class Bar
04:52:51:541 | WARN  | Bar | this is a warn message from class Bar
04:52:51:541 | DEBUG | Bar | this is an error message from class Bar
```

In addition to the `s` and `f` interpolators, there are other interpolators, and a real key is that you can write your own interpolators! For example, Scala SQL libraries typically offer a `sql` interpolator that looks like this:

```
val result = sql"select * from $table where $name = $value"
```

In more complex examples that `sql` interpolator can work with multiple column names and multiple values, and its return type can be something other than a `String`. For instance, depending on the library, it can return a `SqlStatement` or something similar (i.e., some fully-formatted and ready to run SQL).

If you're like me, when you first saw that `s` before the string it may have struck you as unusual, but now that you see the logic behind it, it turns out to be a terrific benefit. Library writers often take advantage of this.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast

15
Strings: Multiline Strings

The second extra thing to know about the Scala `String` type is that you can create *multiline* strings by using """ instead of " when creating the string:

```
val address = """
    Alvin Alexander
    123 Main Street
    Talkeetna, AK 99676
"""
```

This works fine, but it's important to note that this creates a multiline string with leading spaces in it:

```
    Alvin Alexander
    123 Main Street
    Talkeetna, AK 99676
```

A technique you can use to remove those leading spaces is to begin each line with the `|` symbol, and then add the `stripMargin` method at the end of the string:

```
val address = """
    |Alvin Alexander
    |123 Main Street
    |Talkeetna, AK 99676
""".stripMargin
```

This left-justifies the string, so when you print it, it now looks like this:

```
Alvin Alexander
123 Main Street
Talkeetna, AK 99676
```

In that code, `stripMargin` is a *method* that's available on instances of the Scala `String` class. It was created for this situation, and by default it expects the `|` symbol to be used to begin each line. You can also specify a different character, if you prefer:

```
val address = """
    #Alvin Alexander
    #123 Main Street
    #Talkeetna, AK 99676
""".stripMargin('#')
```

In that example I use the # character to begin each line, and then specify that character by calling `stripMargin('#')`.

Multiline strings and interpolators

Multiline strings are just strings — they are of the `String` data type — so they can also be used with interpolators:

```
val address = s"""
    |$name
    |$street
    |$city, $state $zip
""".stripMargin
```

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

16
Numeric Data Types

Now that you've seen strings and integers, a next good thing to know is that Scala comes with the following built-in numeric data types:

- `Byte`
- `Short`
- `Int`
- `Long`
- `Float`
- `Double`

And when you're working with extremely large numbers you can also use:

- `BigInt`
- `BigDecimal`

The numeric data types

As a practical matter, until numbers get very large, most programmers generally use integers (`Int`) and double values (`Double`). Because of that, these are the defaults in Scala, as you can see in the REPL:

```
scala> val x = 42
val x: Int = 42

scala> val y = 42.0
val y: Double = 42.0
```

When I create `x`, Scala is smart enough to implicitly know that `x` is an `Int`, and when I create `y`, Scala also implicitly infers that it is a `Double`.

On the rare occasions that you might need to use the other numeric data types, declare them when you create your variables, like this:

```
val a: Byte = 1
val b: Long = 1
val c: Short = 1
val d: Float = 1.0

// you can declare Int and Double this way (though it isn't necessary):
val e: Int = 1
val f: Double = 1.0
```

Scala 3 also lets you declare numbers using underscore characters to make them more readable:

```
val a = 1_000_000    // Int
val b = 1_000_000L   // Long (created with the 'L' after the number)
val c = 1_234.56     // Double
```

This is a really useful feature when you're working with large numbers.

BigInt and BigDecimal

When you need to create really, really large numbers, use the `BigInt` and `BigDecimal` data types:

```
val a = BigInt(1_234_567_890_123_456L)
val b = BigDecimal(123_456_789.012)
```

Use `BigInt` when you need integer-type numbers that are larger than `Long`, and `BigDecimal` when using very large floating-point numbers. Some developers also use `BigDecimal` for working with currency.

Data type sizes

For your reference, this table provides details about Scala's data types:

```
DATA TYPE  DEFINITION

Boolean    true or false

Byte       8-bit signed two's complement integer
           (-2^7 to 2^7-1, inclusive)
           -128 to 127
```

```
Short      16-bit signed two's complement integer
           (-2^15 to 2^15-1, inclusive)
           -32,768 to 32,767

Int        32-bit two's complement integer
           (-2^31 to 2^31-1, inclusive)
           -2,147,483,648 to 2,147,483,647

Long       64-bit two's complement integer
           (-2^63 to 2^63-1, inclusive)
           -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807

Float      32-bit IEEE 754 single-precision float
           1.40129846432481707e-45 to 3.40282346638528860e+38
           (positive or negative)

Double     64-bit IEEE 754 double-precision float
           4.94065645841246544e-324 to 1.79769313486231570e+308
           (positive or negative)

Char       16-bit unsigned Unicode character
           (0 to 2^16-1, inclusive)
           0 to 65,535

String     a sequence of Char values
```

For more details on `BigInt` and `BigDecimal`, see their Scaladoc pages:

- BigInt[1]
- BigDecimal[2]

Exercises

The exercises for this lesson are available here[3].

[1] https://www.scala-lang.org/api/current/scala/math/BigInt.html
[2] https://www.scala-lang.org/api/current/scala/math/BigDecimal.html
[3] https://alvinalexander.com/book-exercises/LearnScala3Fast/

17

Constructs: Mathematical Expressions

Now that you've seen Scala's data types, we can look at mathematical expressions. In this area, Scala is very much like other programming languages. These examples show the math operations on `Int` values:

```
val a = 1
val b = a + 10    // 11
val c = b * 2     // 22
val d = c - 2     // 20
val e = d / 2     // 10
val f = e % 3     // 1 (modulus operator)
```

As shown, those are the symbols you use for addition, multiplication, subtraction, division, and the modulus operation.

In that example I use different names for each variable because, as mentioned, `val` fields are like algebraic variables and cannot vary. If you prefer to use just one variable in situations like this, this is a place where you use a `var`, which *can* be reassigned to hold new values:

```
var a = 1     // note that 'a' is now a 'var'

a = a + 10    // 11
a = a * 2     // 22
a = a - 2     // 20
a = a / 2     // 10
a = a % 3     // 1 (modulus operator)
```

Scala does not have the `++` and `--` operators that you see with some other programming languages, but when you need to increment or decrement a number you do it with the `+=` and `-=` operators, like this:

```
var a = 1
a += 1    // a == 2
a -= 1    // a == 1
```

While this might seem a little less convenient, a nice thing about this is that the same approach works with `Double` values and other numeric data types:

```
var a = 10.0    // a Double
a += 1.5        // 11.5
a -= 3.0        // 8.5
```

You'll see this sort of thoughtful consistency throughout the Scala language.

A note about these "operators"

For the sake of simplicity I refer to these mathematical symbols as *operators*, but they're actually *methods*. That is, symbols like +, -, etc., that look like operators are really methods on the numeric data type classes. If you're not familiar with OOP and classes this will make more sense later in this book, but for now, just know that this is evidence of Scala being a true object-oriented programming language.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

18

Constructs: `if`/`then`/`else`

Now that we've covered that background, we can start looking at Scala's control structures. We'll start with the Scala 3 `if`/`then` syntax.

Given two variables `a` and `b`, this is how you write a one-line `if` statement in Scala 3:

```
if a == b then println(a)
```

Similarly, this is how you put multiple lines of code after an `if`:

```
if a == b then
    println("a equals b, as you can see:")
    println(a)
```

When you need an `else` clause, add it like this:

```
if a == b then
    println("a equals b, as you can see:")
    println(a)
else
    println("a did not equal b")
```

And when you need "else if" clauses, add them like this:

```
if a == b then
    println("a equals b, as you can see:")
    println(a)
else if a == c then
    println("a equals c:")
    println(a)
else if a == d then
    println("a equals d:")
    println(a)
else
    println("hmm, something else ...")
```

It can be easier to read `if` statements with an `end if` at the end of them, so you can always add that, if you prefer:

```
if a == b then
    println("a equals b")
else if a == c then
    println("a equals c:")
else if a == d then
    println("a equals d:")
else
    println("hmm, something else ...")
end if
```

In this lesson I demonstrated `if` *statements*, meaning that the `if/then` construct is used for a *side effect*, which in this case was printing output. In the next lesson you'll see how to use `if` *expressions*, which are `if/then` constructs that return a result.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

19

Expression-Oriented Programming (EOP)

`if` statements give us our first opportunity to introduce something known as *Expression-Oriented Programming*, or EOP.

Statements vs expressions

One of the terrific things about Scala is that every line of code can be an expression, and not just a statement.

In programming, a *statement* is a block of code that does not return a result, and is used solely for its side effect. For instance, this line of code is a statement because it does not return a result:

```
if a == b then println(a)
```

Lines of code like that are used solely for side effects, and in this case the side effect is printing to STDOUT.

Conversely, an *expression* is a block of code that does return a result, and typically has no side effects. In this lesson I show how to use the `if/then` construct as an expression.

> Technically it's more accurate to say that the previous `if/then` example does not return a *useful* result. It returns something — a type called `Unit`, which is like `void` in other languages — but we generally don't care about it.

Using 'if' as an expression

A great thing about Scala is that each of its constructs can be used as an expression. This includes the `if/then` construct.

What this means is that the `if/then` construct returns a result, and you can use that result, such as assigning it to a variable, like this:

```
val c = if a < b then a else b
```

For some people that code can be easier to read if it's on multiple lines, so you can also write it like this:

```
val c =
    if a < b then a else b
```

or this:

```
val c =
    if a < b then
        a
    else
        b
```

All of those examples return the same result, and can be read as, "If `a` is less than `b`, assign the value of `a` to `c`. If not, assign the value of `b` to `c`."

> If you're familiar with the *ternary operator* syntax in Java[1] and other languages, you can see that there is no need for a special syntax in Scala: you just write a normal `if/then` expression.

Preview: Using if/then as the body of a method

As a quick peek into the future, as you'll see in future lessons, because the `if/then` construct is an expression, you can also use it as the body of a Scala function, like this:

```
def min(a: Int, b: Int): Int =
    if a < b then a else b
```

That code defines a function named `min` that returns the minimum value of the two integer parameters that are passed into it, `a` and `b`.

For the purposes of this lesson, the important thing is that `if/then` is an expression and returns a value, and because of this it can be used as the entire body of a function. This is one of the beautiful things about EOP, and you'll see much more of this in the lessons that follow.

[1] https://alvinalexander.com/java/edu/pj/pj010018

Finally, even though I haven't introduced functions yet, if you have experience with other programming languages, I suspect that you may understand how that function works. You can see it in action in these examples:

```
println(min(1, 2))      // prints "1"

val x = min(1, 1_000)   // x is assigned the value 1
```

As you'll see throughout this book, EOP is another feature that makes Scala concise and readable, i.e., *expressive*.

Exercises

The exercises for this lesson are available here[2].

[2] https://alvinalexander.com/book-exercises/LearnScala3Fast/

20
Tuples

Before we get into the following lessons on sequence classes, it will help if we take a first look at tuples.

A *tuple* is a heterogeneous collection of elements, which means that a tuple can contain different types of elements. For instance, this is a tuple that contains an `Int` and a `String`:

```
val t = (1, "yo")
```

Similarly, this is a tuple that contains an `Int`, `String`, `Char`, and `Double`:

```
val t = (1, "1", '1', 1.1)
```

> I didn't mention it previously, but you create a `Char` (character) by putting it inside single-quotes.

As shown, you create a tuple by putting parentheses around the elements you want inside it. Like the sequence classes you're about to see, a tuple can hold as many elements as you need, though I typically use it for small collections like these.

A tuple is also an *immutable* data structure, meaning that its elements can't be changed and its size can't be changed.

Accessing tuple elements

A tuple works like a sequence in that the elements are stored in the order you place them in. In Scala 3 you access tuple elements by their index number. For instance, given this tuple:

```
val t = (42, "fish")
```

you access its elements as `t(0)` and `t(1)`:

```
t(0)     // 42
t(1)     // "fish"
```

You can also determine how many elements are in a tuple like this:

```
t.size   // 2
```

We won't be using this functionality just yet, but ...

The importance of tuples

In terms of these lessons, the important part about tuples is that we need to see them now because they're used in the following lessons on sequences.

In the longer term, tuples are important because they can be used in other ways!

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

21

Collections: The List Class

A *sequence* in Scala is an ordered list of values, meaning that the values are returned in the order you put them in. Scala has a few different types of sequences that you can use for different needs, including `List`, `Vector`, and `ArrayBuffer`. But for our purposes, a good one to start with is `List`.

Scala's `List` class is immutable, meaning that its elements can't be changed and the list can't be resized. It's implemented as a linked-list, so it's good for small lists, but if you want to have fast access to its one-millionth or one-billionth element, you'll want to use a `Vector` instead. (I discuss this more as we go along.)

> NOTE: For reasons of efficiency and performance, the different sequence classes *store* your elements in different ways, but the important part is that they are *returned* to you in the proper order.

Using List

These examples show how to create new lists of different types:

```
val ints = List(1, 2, 3)
val doubles = List(1.1, 2.2, 3.3)
val names = List("Aleka", "Christina", "Alvin")
```

As shown, you usually don't need to explicitly declare the type of the `List`, but when you want or need to do that, you can:

```
val ints: List[Int]       = List(1, 2, 3)
val doubles: List[Double] = List(1.1, 2.2, 3.3)
val names: List[String]   = List("Aleka", "Christina", "Alvin")
```

In that code, `List[Int]` can be read as "A list of integers," `List[Double]` can be read as "A list of double values," and so on.

Typically you'll only explicitly show the data type when it isn't 100% obvious what's contained in the `List`. That won't be a big problem now, but when

your code gets more complicated there can be situations where you'll want to explicitly declare the type like this.

Accessing List elements

When you need to access the elements stored in a `List`, you access them by their index number, like this:

```
list(0)    // the first element
list(1)    // the second element
```

Like most other programming languages, Scala is 0-based when it comes to working with indexes on sequences, so the first element is referred to as "element 0," the second is "element 1," etc. Here's a complete example using a `List[String]` — a list of strings — that shows how this works:

```
val list = List("a", "b", "c")

println(list(0))    // prints a
println(list(1))    // prints b
println(list(2))    // prints c
```

As you'll see throughout this book, using parentheses to access elements in a collection is used consistently in Scala.

Lists are immutable

As mentioned, the `List` class is an immutable data structure, meaning that you can't add, update, or remove elements, and you can't resize an existing list. For example, given this `List[Int]`:

```
val ints = List(1, 2, 3)
```

these attempts to add or update elements will fail to compile:

```
ints += 4       // error
ints(0) = 10    // error
```

The next lesson begins to show the correct ways to update lists.

Discussion

If the `List` class being immutable seems like a big restriction, fear not! There are a few things that will ease your concerns.

First, if you're only interested in OOP, Scala has other types of sequence classes that are *mutable*, meaning that you *can* modify their elements. I'll show those shortly.

Second, if you have at least a mild interest in FP, you'll find that it's surprising how often you *don't* need a mutable sequence class. Because of this, I often reach for the `List` class first, and then change it to a mutable sequence if I *really* need one.

Third, as I mentioned earlier, another approach is to create `a` as a `var` variable, in which case you can then assign the new result back to `a`:

```
var a = List(1, 2, 3)
a = a ++ List(4, 5)      // a: List(1, 2, 3, 4, 5)
```

Some OOP developers I have talked to really like this approach of using (a) an immutable sequence with (b) a mutable variable.

A performance note

When you have a small sequence that contains just a few elements, it doesn't matter too much what sequence class you use. But when you have large lists, you may want to use a `Vector` or `ArrayBuffer` instead of a `List`. The process for choosing a sequence type is explained in the lessons that follow, but for now just note that `Vector` is immutable just like `List`, but it works *much* faster with large lists when you need to access an element directly, like `list(10_000_000)`. And `ArrayBuffer` is like a *mutable* version of `Vector`, so you'll use it when you have a sequence that you'll constantly be modifying.

Those things being said, I'll continue to use `List` in the lessons that follow because it's a nice class for small, immutable sequences.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

22

Collections: Updating List Elements

Because `List` is immutable, the way you add, remove, and update elements is to (a) use its add/update/delete methods while (b) assigning the result to a new variable. The following examples begin to demonstrate this process. More examples are then shown in the *Sequences: More Methods* lesson that follows.

Appending and prepending elements

First, to *append* one element to a `List`, use its `:+` method, and to add multiple elements, use its `++` method:

```
val a = List(1, 2, 3)
val b = a :+ 4             // add one element
val c = b ++ List(5, 6)    // add multiple elements
```

> TIP: The `:+` and `++` methods are used with both `List` and `Vector`.

Because `List` is a linked-list, the preferred way to work with it is to *prepend* elements to it:

```
val a = List(2, 3)     // List(2, 3)
val b = 1 :: a         // List(1, 2, 3)
val c = 0 :: b         // List(0, 1, 2, 3)
```

With the `List` class, prepending is actually a faster operation than appending, but when your lists are small, this isn't a huge concern.

Removing elements

There are many ways to remove elements from a `List`, and I cover those in the *Sequences: More Methods* lesson. As a quick preview of that lesson, a

common approach is to use the `List` class `filter` method, like this:

```
val a = List(1, 2, 3, 4, 5)
val b = a.filter(_ > 3)        // b: List(4, 5)
```

I don't want to duplicate that lesson too much, but for this lesson I just want to give you a preview of one way to "remove" elements from a `List` (remembering to assign the result to a new `List`).

Updating elements

There are also many ways to update a `List`, so I'll just share one approach here that truly lets you update an element according to its position in the list:

```
val a = List(1, 2, 3)
val b = a.updated(0, 100)    // b: List(100, 2, 3)
val c = b.updated(1, 200)    // c: List(100, 200, 3)
```

The first call to the `updated` method updates the value at index `0` in the list, giving it a new value of `100`. The second call then updates the value at index `1` in the list, giving it a new value of `200`.

For more examples of the methods you can use, see the *Sequences: More Methods* lesson. Or, if you're really curious and want to see *many* more examples, see my blog post, 100+ Scala List class examples[1].

Exercises

The exercises for this lesson are available here[2].

[1] https://alvinalexander.com/scala/list-class-methods-examples-syntax/
[2] https://alvinalexander.com/book-exercises/LearnScala3Fast/

23

Collections: Other Sequence Classes

Before we get into more lessons on how to use sequence classes, it's important for me to be clear about the `List` class and other sequence classes in Scala.

As mentioned, `List` is an immutable, linked-list, sequence class:

- *Immutable* means that the elements in a `List` cannot be changed, and the size of a `List` cannot be changed.
- *Linked-list* means that one element in a list is daisy-chained to the next element in the list. If you took programming classes in college, you probably wrote a linked-list in one of your classes. (This type of sequence is also known as a *linear* sequence.)
- *Sequence* means that the class contains an ordered sequence of elements. The elements are always returned to you in the order that you put them into the sequence.

Scala has other sequence classes

At this point there are two important things to know about Scala's sequence classes:

- Scala has other sequence classes, and each is intended for a specific purpose
- Because Scala is an object-oriented language, the sequence classes are created in a specific hierarchy

Of the other sequence classes, the most important ones to know are:

- `Vector` is an immutable, *indexed* sequence
- `ArrayBuffer` is a *mutable*, *indexed* sequence

We'll look at those in a few moments, but first we need to look at some other things.

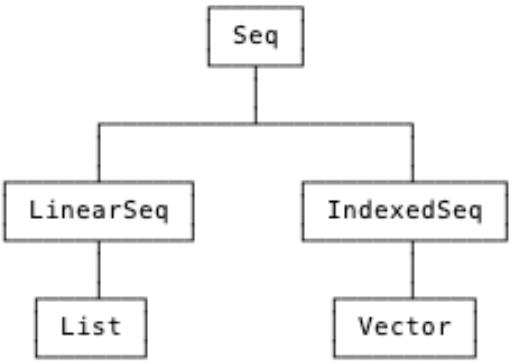


Figure 23.1: A subset of Scala's immutable sequence classes

Indexed

Indexed means that any element in a sequence can be accessed very rapidly. For instance, to access the one-millionth element in a `List`, you have to start at the first element in the list and follow the linked-list daisy-chain until you get to the one-millionth element, and that's a slow process, requiring one million operations. But with a `Vector` or `ArrayBuffer` — because they are created with a tree-like data structure — accessing the one-millionth element requires just a few hops.

`Vector` and `ArrayBuffer` are *much* faster for this purpose, and their structure also makes other operations, such as *appending* elements, much faster than `List`. To be clear, I only use `List` for small sequences.

The sequence class hierarchy

Because Scala is an OOP language, the `List`, `Vector`, and `ArrayBuffer` classes extend other data types. For example, this figure shows part of the Scala class hierarchy for immutable sequence classes:

As shown, both `List` and `Vector` extend the base class `Seq`. This gives them many common methods that are implemented in `Seq` (and other classes above `Seq` that I don't show.) But after `Seq` they diverge, and `List` extends `LinearSeq`, and `Vector` extends `IndexedSeq`, which gives them the performance attributes I just described.

Choosing a sequence

Because of these attributes, you generally use these sequence classes at the following times:

- Use `List` or `Vector` when you want an immutable sequence
- Prefer `Vector` over `List` when (a) you need to randomly access elements in the sequence, (b) the size gets large, or (c) when you'll be constantly appending elements to the sequence
- Use `ArrayBuffer` when you want a mutable sequence class (for instance, when you know that you will constantly add, remove, and update elements)

Also because of these attributes:

- `Vector` and `ArrayBuffer` are typically your "go to" classes
- `List` and `Vector` are used in an FP style, and in OOP when you know you're sequence won't be mutated
- `ArrayBuffer` is used in an OOP style

Performance considerations

When you get into advanced use cases, the Scaladoc for these classes can also help you with additional information, such as performance characteristics. For instance, the `ArrayBuffer` Scaladoc page[1] states, "Append, update and random access take constant time (amortized time). Prepends and removes are linear in the buffer size." The *constant time* portion of the description is one reason that `ArrayBuffer` is the preferred mutable sequence.

Similarly, the `Vector` Scaladoc page[2] states, "It provides random access and updates in `O(log n)` time, as well as very fast append/prepend/tail/init (amortized `O(1)`, worst case `O(log n)`). Because vectors strike a good balance between fast random selections and fast random functional updates, they are currently the default implementation of immutable indexed sequences."

[1] https://www.scala-lang.org/api/current/scala/collection/mutable/ArrayBuffer.html
[2] https://www.scala-lang.org/api/current/scala/collection/immutable/Vector.html

I also include performance details in the Scala Cookbook[3].

List

All of that being said, the `List` class is a nice class to use when you're working with small sequences. I tend to use it a lot, and it's used a lot in the Scala library code as well.

> The Scala library creators once ran a test where they replaced every instance of `List` in the libraries with `Vector`, and they actually saw a slight slowdown in the code, likely because most of their sequences are small, and `List` is more efficient with small sequences than `Vector`.

Therefore, I'll show the `List` class in the following lessons, but remember that you can also use the `Vector` classes in these examples, because both classes are immutable sequences.

Exercises

The exercises for this lesson are available here[4].

[3] https://amzn.to/3du1pMR
[4] https://alvinalexander.com/book-exercises/LearnScala3Fast/

24

Constructs: for Loops

Now that we've seen the `List` class we can begin looking at `for` loops, which let us iterate over elements in a sequence. For example, given this list:

```
val ints = List(1, 2, 3)
```

This REPL example shows how to iterate over every element in the list to print them with the `println` function:

```
scala> for i <- ints do println(i)
1
2
3
```

You can infer from that example that the general syntax of a `for` loop is:

```
for element <- listOfElements do somethingToDoWith(element)
```

When the "something to do" part of your code requires multiple lines, use this syntax:

```
for
    i <- ints
do
    // this doesn't really require multiple lines,
    // but imagine that it does
    val j = i * 10
    println(j)
```

Here's another example that shows another way the `for`/`do` loop can be formatted:

```
val names = List("adam", "alex", "bob")

// again imagine that this requires multiple lines:
for name <- names do
    val capName = name.capitalize
    println(capName)
```

Either of these indentation styles will work, and that example results in this output:

```
Adam
Alex
Bob
```

I keep noting that you should imagine that this code requires multiple lines, and that's because it really doesn't; it can all be on one line, like this:

```
for name <- names do println(name.capitalize)
```

Personally, I often find myself writing multiple lines of code, and then realizing, "Wait, I can condense this, and this, and then that," and I end up with just one or two lines of code. Having concise — but still readable! — code often happens in Scala.

Remembering EOP

Earlier I mentioned EOP — Expression-Oriented Programming. Remember that in EOP we program using *expressions* and not *statements*.

However, `for` *loops* are a construct that don't really return anything. Because of this they are effectively statements, and are only used for their *side effects*. This is something I never thought about in Java and other OOP languages, but once you're exposed to EOP, you realize that something now feels different about `for` loops. It starts to occur to you, "Hmm, I see now that this is a statement, not an expression." It's not that statements are necessarily bad — you certainly need to be able to print — but they begin to stand out to you.

Two notes

As a first note, I mentioned earlier that technically the `for` loop shown does have a return type. That type is named `Unit`, and it's like `void` or `Void` in

some other languages. This just means that the return type is empty and essentially useless. This leads to another important point: every programming *statement* will have a `Unit` return type, because statements are always used for their side effects, and have no useful return type.

The `println` function is a great example of this. We know that it's only used for its side effect of printing to STDOUT, and this is its actual type signature:

```
def println(x: Any): Unit = ???
```

Even though I haven't discussed functions yet, if you've worked with other programming languages you can probably tell that this is a function that takes a parameter named `x` whose type is `Any`, and it returns the `Unit` data type. So again, when you see that some block of code returns `Unit`, your first thought should be, "There's a side effect here."

The second note is to remember that wherever I demonstrate the `List` class, you can generally also use the `Vector` class, because both are immutable.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

25

Constructs: for Expressions

As I got better with Scala I found that I used `for` *loops* much less often, and started to use a similar construct: `for` *expressions*.

A `for` expression is similar to a `for` loop, except that it really is an *expression* — it returns something of value. You create a `for` expression by replacing the `do` keyword with the `yield` keyword. For example, given this list of integers:

```
val xs = List(1, 2, 3)
```

you can create a *new* list of integers from that list, where each element in the new list is twice the value of the elements in `xs` like this:

```
val ys = for x <- xs yield x * 2
```

If you place that code in the Scala REPL, you'll see that `ys` has the type `List[Int]`, and its contents are `List(2, 4, 6)`:

```
scala> val ys = for x <- xs yield x * 2
val ys: List[Int] = List(2, 4, 6)
```

You can think of the `for`/`yield` expression working like this:

> "For every element in the list `xs`, double each element, and then put that new element in a temporary new list. When you have finished doubling every element, return the entire new list."

Here's another example of a `for` expression, this time using a list of strings:

```
val names = Seq("luke", "leia")   // a sequence of names
val capNames =
    for
        name <- names
    yield
        // put as many lines of code as are necessary
        // for your algorithm
        name.capitalize
```

After that code runs, `capNames` contains `List("Luke", "Leia")`. This is because the `for`/`yield` expression works like this:

- Get the first element from the list (`"luke"`)
- Capitalize it (`"Luke"`)
- Add that to the new list
- Get the next element (`"leia"`)
- Capitalize it (`"Leia"`)
- Add that to the new list
- All elements have been processed, so return the new list, and assign its value to `capNames`

It really is an expression

I try not to repeat myself too many times, but since we're just getting into this topic: To be clear, the `for`/`yield` combination truly is an expression; it returns a value, the value calculated after the `yield` keyword. That's why this is called a "`for` expression." In other programming languages you may also see this concept referred to as a "`for` comprehension."

Transforming the data type

Before we move on, here's another example of a `for` expression:

```
val xs = List("a", "bb", "ccc")
val ys = for x <- xs yield x.length  // ys: List[Int] = List(1, 2, 3)
```

As shown in the comment, after the code is run, `ys` has those contents. So a cool thing about a `for` expression is that you can use it to create a new type of collection: Here I start with `xs` as a `List[String]` and convert that into `ys`, which is a `List[Int]`.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

26

Constructs: for Expressions With Multiple Generators

Backtracking for a moment, here's another `for` *loop* in the REPL:

```
scala> for i <- 1 to 3 do println(i)
1
2
3
```

In that example, this portion of the code is known as a *generator*, because it generates the numbers `1` through `3`:

```
i <- 1 to 3
```

Similarly, in this code:

```
val ints = List(1, 2, 3)
for i <- ints do println(i)
```

this portion of the code is also a generator:

```
i <- ints
```

I mention this for two reasons. First, it's a piece of technical goo you'll need to know when talking to other Scala developers. Second, `for` loops and expressions can have multiple generators:

```
val ints = List(1, 2)
val chars = List('a', 'b')

for
    i <- ints
    c <- chars
do
    println(s"i = $i, c = $c")
```

That `for` loop results in this output:

```
i = 1, c = a
i = 1, c = b
i = 2, c = a
i = 2, c = b
```

As with other programming languages it's possible to nest one `for` loop inside another, but in Scala this isn't necessary, and as you'll see in the lessons that follow, there are other benefits to this "generator" approach.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

27

Constructs: Adding ‘if’ Clauses to ‘for’ Expressions

You can add `if` statements to `for` expressions, and in Scala they’re implemented in a really smart way. For example, given this list:

```
val fruits = List("apple", "banana", "cherry", "date")
```

In Scala, you *could* write an old-school `for` loop like this:

```
// don’t do this, there’s a better way
for
    f <- fruits
do
    if f.length > 5 then println(f)
```

That code prints this output, as desired:

```
banana
cherry
```

However, in Scala, this is the *correct* approach:

```
for
    f <- fruits
    if f.length > 5
do
    println(f)
```

In my opinion this is a *much* better approach, because it separates our code into two distinct sections:

```
for
    f <- fruits         // generator
    if f.length > 5     // guard
do
    println(f)          // business logic
```

As shown, `if` expressions inside the generator area are called *guards* (which you can also think of as filters). A great thing about this approach is that you can put all of your generators and guards after the `for` keyword, and your business logic after the `do` (or `yield` in `for` expressions). It turns out that this "separation of concerns" makes our code much easier to read.

> TIP: If you're familiar with SQL, this is a bit like a SQL `UPDATE` statement, which has distinct `UPDATE`, `SET`, and `WHERE` clauses.

Here's a similar example that uses a `for` *expression* instead of a `for` *loop*:

```
val capFruits =
    for
        f <- fruits
        // add as many guards as you need
        if f.length > 5
        if f.length < 10
        if f.startsWith("c")
    yield
        // add as much business logic as
        // you need here
        f.capitalize
```

As shown, you can add as many guards as you need, and as much business logic as you need for the current problem — and having them in separate code blocks makes your code easier to read.

If you run that code in the Scala REPL you'll see that `capFruits` has this data type and content:

```
capFruits: List[String] = List(Cherry)
```

As a summary, you've now seen both *generators* and *guards* in both `for` loops and `for` expressions.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

28

Constructs: try/catch/finally (Part 1)

Scala has a `try/catch/finally` construct that helps you handle the exceptions that can occur in your code, such as when you attempt to access a website and its down, or when you try to read a file that doesn't exist. The basic idea is that you try to access some resource, and if it fails, you catch the resulting exception and deal with it.

For example, when you attempt to read a file you can run into two possible exceptions:

- `FileNotFoundException`
- `IOException`

When you need to work with those exceptions, you'll write code that looks like this:

```
// somewhere earlier in your code:
var content = ""

// then later in your code you try to read a file's content
// into that variable:
try
    content = readFile(filename)
catch
    case e: FileNotFoundException =>
        System.err.println("Couldn't find that file.")
    case e: IOException =>
     System.err.println("Had an IOException trying to read that file")
```

Notice that you write your normal "success path" code inside the `try` block. You do whatever you want to do there, and then catch its potential exceptions in the `catch` block. If everything works, your `content` variable is updated with text from the file, but if it fails, you print those error statements.

If you've used Java, the concept is the same, but the `case` syntax here is consistent with the `case` syntax that's used in `match` expressions, which you'll see later in this book. You can also use a `finally` clause, which I'll show next.

Discussion

Showing the `finally` clause takes a little bit more work, and it will also help if we cover a few more topics before I address it. For those details, see the second `try/catch/finally` lesson, which you'll find just after the lessons on functions.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

29

Constructs: The while Loop

Scala also has a `while` loop, but because Scala puts an emphasis on functional programming — and because the collections classes have dozens of built-in methods — I rarely use it. But if you need it, this is what it looks like:

```
var i = 0

while i < 3 do
    println(i)
    i += 1
```

If you prefer, you can add `end while` to the end of the loop:

```
while i < 3 do
    println(i)
    i += 1
end while
```

The `while` loop is generally used for a side-effect, such as printing to the console with `println` or updating a mutable variable, and when you write code in an FP style you avoid side effects as much as possible, so you don't use `while` loops. Frankly, I have written maybe one `while` loop every year or two, so I don't want to put too much emphasis on it here.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

More information

For more information about how FP avoids side effects, see my book, Functional Programming, Simplified[2].

[2]https://alvinalexander.com/scala/functional-programming-simplified-book

30

Collections: The foreach Method and Anonymous Functions

The `List` class gave us a nice way to demonstrate `for` loops and expressions, so I took a little time to show those. As you'll see in the next few lessons, another reason I showed those examples is because they give me a way to explain some of the most commonly-used methods that are available on sequence classes like `List`.

As you'll see, most of the methods I'm about to demonstrate are replacements for custom `for` expressions. They all loop over your collection, and do something different with that collection, based on the algorithm you supply. You often supply your algorithm as a small snippet of code that's known as an anonymous function. Everything about this approach makes your code easier to read and maintain.

> Back when I first started learning Scala in 2010, these methods were definitely not *standard*, but since then they have been adopted by other languages like Java, Swift, Kotlin, etc.

foreach

The `foreach` method is available on all Scala sequence classes. It basically has a built-in `for` loop that lets you specify what you want to do with each element. `foreach` does not return anything useful, so it's always used for some sort of side effect, such as printing:

```
val ints = List(1, 2, 3)
ints.foreach(println)      // prints 1, 2, and 3 on separate lines
```

A first look at anonymous functions

That second line of code uses something known as an *anonymous function*. Let's look at how it works. First, given this list:

```
val ints = List(1, 2, 3)
```

the *long version* of how you could write the previous code looks like this:

```
ints.foreach(i => println(i))
```

This can be read as, "For every element `i` in the list `ints`, print that element using the `println` function." You can see where the variable `i` is used in the code here:

```
ints.foreach(i => println(i))
             ^            ^
```

In this code you can think of the `=>` symbol as being a *transformer*. It transforms the thing on the left — the variable `i` — into the thing on the right, which in this case is a `println` function:

```
// the variable 'i' is transformed into a println statement
i => println(i)
```

In this specific example it turns out that there's something interesting — and also common — going on here:

- Only one variable comes out of the list at a time
- That variable is referenced only one time on the right side of the `=>` symbol

For most methods on sequences — such as `foreach` — the first condition is usually true. When the second condition is also true, Scala lets you take a few additional shortcuts. The first shortcut is that Scala lets you use the `_` character to refer to that single element:

```
ints.foreach(println(_))
```

This shortens your code, but still leaves it being readable. However, since that `_` really doesn't add any value, Scala lets you omit it completely:

```
ints.foreach(println)
```

In summary, you *could* write the code like this:

```
ints.foreach(i => println(i))
```

but the Scala way is to write it like this:

```
ints.foreach(println)
```

foreach is like a 'for' loop

One more note: The `foreach` method is often similar to writing a `for` loop. For instance, given this `List[String]`:

```
val strings = List("a", "bb", "ccc")
```

these two approaches produce the same result:

```
for s <- strings do println(s.length)
strings.foreach(s => println(s.length))
```

What I found in my own experience — being mostly familiar with Java at the time — was that when I started working with Scala I kept writing `for` loops, but these days I always reach for methods like `foreach`, `filter`, and `map`, in combination with small anonymous functions.

> I like to think of anonymous functions as being small snippets of code that are easy to read.

Higher-Order Functions

The `foreach` method and the two other sequence methods I'm about to show — `filter`, and `map` — are technically known as *Higher-Order Functions*, or HOFs.

If that term sounds scary, fear not, it just means that a method like `foreach`, which is an HOF, can take a function as an input parameter. As you'll see in the next several lessons, this lets us pass little snippets of code into `foreach`, `filter`, and `map`. Those HOFs are each written in slightly different ways, so they apply your custom code snippet to the sequence to achieve a desired effect.

In the book, Learn You A Haskell For Great Good![1], the author makes this statement:

[1] http://amzn.to/1POaUCv

> "Haskell functions can take functions as parameters and return functions as return values. A function that does either of those is called a higher order function. Higher order functions aren't just a part of the Haskell experience, they pretty much are the Haskell experience."

Similarly, they are pretty much the Scala experience as well. Once you get used to them, you'll be writing expressive code: concise, and still readable.

(In Scala, functions can also return functions, and I show how to do that in the Scala Cookbook[2] and Functional Programming, Simplified[3].)

Exercises

The exercises for this lesson are available here[4].

[2] https://amzn.to/3du1pMR
[3] https://alvinalexander.com/scala/functional-programming-simplified-book
[4] https://alvinalexander.com/book-exercises/LearnScala3Fast/

31

Collections: Using The map Method

In the previous lesson you saw that given this `List[String]`:

```
val strings = List("a", "bb", "ccc")
```

this `for` *loop* and `foreach` method call are equivalent:

```
for s <- strings do println(s.length)    // print each element in 'strings'
strings.foreach(s => println(s.length))  // print each element in 'strings'
```

This might make you wonder, what is the equivalent of this `for` *expression* (i.e., the combination of `for` and `yield`):

```
val lengths = for s <- strings yield s.length
```

It turns out that Scala sequence classes have a method for exactly this situation, and for mathematical reasons that method is called `map`:

```
val lengths = strings.map(_.length)
```

When either of those last two examples are run, the variable `lengths` will be a `List[Int]`, and it will contain `List(1, 2, 3)`.

Before I move on to an explanation of how this works, here's another example in the Scala REPL that shows how to use the `map` method on a `List`:

```
scala> val ucStrings = strings.map(_.toUpperCase)
val ucStrings: List[String] = List(A, BB, CCC)
```

As shown, that creates a new variable named `ucStrings` that is a `List[String]`, and it contains the value, `List("A", "BB", "CCC")`.

About the name "map"

As for the name `map`, it comes from mathematical concepts. It turns out that when you start with a list of elements like this:

```
"a"
"bb"
"ccc"
```

and you apply a function like "length" to those elements, you generate a new list like this:

```
1
2
3
```

Mathematicians call this process "mapping": You're applying a function to each value in the first list to create a corresponding value in the new list. Then you can say that the initial value "maps" to that new corresponding value. Using Scala's `=>` (transformation) symbol, that mapping process looks like this:

```
INITIAL          NEW
 VALUE          VALUE
-------         -----
"a"       =>      1
"bb"      =>      2
"ccc"     =>      3
```

So for this specific example, when you apply a `length` function to each initial string, you find that those strings map to corresponding lengths like this:

- `"a"` maps to `1`
- `"bb"` maps to `2`
- `"ccc"` maps to `3`

Similarly, if you apply an "upper case" function to the original list, you get these transformations:

```
"a"     =>   "A"
"bb"    =>   "BB"
"ccc"   =>   "CCC"
```

It may help to think of "map" as "transform"

If the name `map` is new to you — as it once was to me — it may help to think of it being named "transform" instead. This becomes a little more apparent when we write this code using the long format that includes Scala's `=>` *transformer* symbol:

```
val lengths = strings.map(s => s.length)
```

Any time you see the `=>` symbol in Scala, think *transform*, or, "the thing on the left side of the `=>` symbol is being transformed into the thing on the right side of the symbol."

> Bill Venners, the creator of ScalaTest and many other things, once suggested that the `=>` symbol should be called the "rocket symbol." Although that never took off, it may also help to convey the idea of transforming something (in this case from one place to another).

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

32

Collections: Using The filter Method

In the previous lesson you saw how to *transform* elements in one list to create a new list. In this exercise you'll see how to filter the elements from one list to create a new list. As you'll see, the process is very similar, and uses another method that's available on sequence classes. And to demonstrate that the `Vector` class works just like `List`, this lesson will use `Vector`.

To demonstrate the filtering process, start with this `Vector[Int]`:

```
val ints = Vector(1, 2, 3, 4, 5)
```

Now you can call the `filter` method on `ints` to create a new sequence:

```
val smallInts = ints.filter(_ < 3)
```

If you put that code in the Scala REPL you'll see that `smallInts` has the type `Vector[Int]`, and its value is `Vector(1, 2)`, i.e., a sequence that contains the values `1` and `2`.

Note that when you need to use the long anonymous function form — such as when your algorithms get larger — that syntax looks like this:

```
val smallInts = ints.filter(i => i < 3)
```

Discussion

The `filter` method is just like writing this `for` expression:

```
val smallInts =
    for
        i <- ints
        if i < 3
    yield
        i
```

But while that code produces an identical result, it's much more verbose than the `filter` method. So we use `filter` instead.

Way back in my Java/OOP programming days I *always* wrote custom `for` loops, but once I learned Scala, I realized that methods like `foreach`, `map`, and `filter` eliminate the need for those custom loops.

So a great thing about these methods is that (a) you rarely need to write custom `for` loops or `for` expressions, and (b) you won't have to read other people's custom `for` loops and expressions. :)

More seriously, because `filter` is still easy to read and it's significantly smaller than that custom `for` loop, we use `filter`.

Another way to think about this is that the creators of the Scala collections classes have already written these custom `for` loops and expressions, put them in standard methods, and unit-tested them, so we don't have to worry about that portion of our code.

Helping to remember how filter works

Sometimes when I get away from Scala for a while and then come back to it, I can't remember the way `filter` works — i.e., whether it retains the elements in the anonymous function or discards them. But in short, it means *retain*.

I haven't come up with a *terrific* way to remember this yet, but I found one approach that helps, so I'll share it here: If you think of `filter` as being spelled as `filtre` instead (the British spelling), you can think of the letters as standing for "*F*ind *I*n *L*ist *T*hen *R*etain *E*ach." So again, by applying a function to a sequence with `filter`, you retain the desired elements.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

33

Collections: Combining map and filter

As you saw in the previous lessons, methods like `map` and `filter` return a new sequence from an existing sequence. A great thing about this approach is that it means that you can chain these method calls together to create a solution for your current data transformation needs. For instance, given this list:

```
val colors = List("red", "blue", "yellow", "green", "purple")
```

you can apply `filter` and then apply `map` like this:

```
val result = colors.filter(_.length < 5)
                   .map(_.toUpperCase)
```

When that is run, `result` contains `List("RED", "BLUE")`, because (a) `filter` only retains the strings whose length is less than `5` characters, and then (b) `map` transforms each string in that list to uppercase. That code is just like writing code like this, but it's less verbose:

```
val result1 = colors.filter(_.length < 5)
val result2 = result1.map(_.toUpperCase)
```

If you prefer to see the data types

When I first started with Scala I think it helped me to see the data types of each expression. So if you're feeling that way, you can also write that code like this:

```
val colors: List[String] = List(
    "red", "blue", "yellow", "green", "purple"
)
```

```
// short solution
val result: List[String] = colors.filter(_.length < 5)
                                 .map(_.toUpperCase)

// longer solution
val result1: List[String] = colors.filter(_.length < 5)
val result2: List[String] = result1.map(_.toUpperCase)
```

You can also see these data types in a modern IDE like IntelliJ IDEA or VS Code, so most developers don't show the types like this, since you can usually just hover over a variable with your mouse cursor to see its data type.

Combine as many as you need

You can combine as many `filter` and `map` calls as you need to achieve a desired solution:

```
val result = colors.filter(_.length < 5)
                   .filter(_.startsWith("r"))
                   .map(_.toUpperCase)
```

When this code is run, `result` contains `List("RED")`.

Combining functional methods together like this is exactly how you write "big data" applications using Apache Spark and other frameworks.

> We refer to these methods as "functional methods" because they are *functional* in the FP sense: `List` and `Vector` are immutable, so the only way the methods can possibly work is to return a new sequence.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

34

Collections: More Sequence Methods

As I mentioned, the `List` class and other sequence classes have *dozens* of methods which will keep you from having to write custom `for` loops and `for` expressions. I won't attempt to demonstrate them all, but in this lesson and the next one, I'll provide examples of the most commonly-used methods.

To demonstrate these methods, I'll use this sample `List[Int]`:

```
val a = List(10, 20, 30, 40, 10)
```

In the examples that follow, imagine that you're writing code like this, assigning the result of a method to a new variable:

```
val b = a.filter(_ < 25)
```

However, in my examples I won't show the new variable `b`. Instead of writing code like that, I'll just show the results after each comment.

Given that introduction, here are the first few examples:

```
a.filter(_ < 25)                    // List(10, 20, 10)
a.filter(_ > 100)                   // List()
a.filterNot(_ < 25)                 // List(30, 40)
```

These are two methods for filtering data in a list, and this is a favorite tip on how I remember what "filter" means:

> I think of `filter` as being spelled "filtre," and then I remember it as "Find In List Then Retain Each." So `filter` means *retain* or *keep*.

Next, here are methods related to *dropping* and *taking* elements:

```
a.drop(2)                          // List(30, 40, 10)
a.dropRight(2)                     // List(10, 20, 30)
a.dropWhile(_ < 25)                // List(30, 40, 10)

a.take(3)                          // List(10, 20, 30)
a.takeRight(2)                     // List(40, 10)
a.takeWhile(_ < 30)                // List(10, 20)
```

In these examples, `drop` means "drop from the resulting list," and `take` means "keep in the resulting list."

Here are a few more methods:

```
a.distinct                         // List(10, 20, 30, 40)
a.intersect(List(19,20,21))        // List(20)
a.slice(2, 4)                      // List(30, 40)
```

In these examples:

- `distinct` returns the distinct (unique) elements in the list
- `intersect` returns the elements that are in both lists
- `slice` returns a slice (or segment) of the list

These methods are a good thing!

When I first started with Scala I was intimidated by how many methods there are on sequence classes, but, the reality is that these are all methods that we used to write custom `for` loops for previously. And there are at least two bad things about writing custom `for` loops when you don't have to:

- It's verbose (and harder to read)
- It's error-prone

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

35

Collections: Even More Sequence Methods

To demonstrate even more methods that are available on `List`, `Vector`, and other `Seq` classes, first create a new variable for us to work with:

```
val firstTen = (1 to 10).toList
```

You haven't seen this code before, but it populates `firstTen` so that it's a `List` that contains the integers `1` through `10`:

```
List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Now that we have this sample data, let's look at some more methods that are available on a `List`:

```
firstTen.contains(2)                    // true
firstTen.containsSlice(List(3,4,5))     // true
firstTen.count(_ % 2 == 0)              // 5
firstTen.endsWith(List(9,10))           // true
firstTen.exists(_ > 10)                 // false
firstTen.forall(_ < 20)                 // true
firstTen.indexWhere(_ == 3)             // 2

firstTen.max                            // 10
firstTen.min                            // 1
firstTen.sum                            // 55
firstTen.product                        // 3628800

firstTen.startsWith(List(1,2))          // true
```

I like to think of all of these methods as "informational" methods — they tell you something about the contents of your list. Most of their names are self-explanatory, so I'll only add notes for a few of them:

- The `contains` example asks, "Is there a number `2` *anywhere* within this list?"

- `count` returns a count of all the elements where your test condition is `true`
- `exists` lets you supply an anonymous function so you can ask whether the condition you supply is `true` or not
- `forall` gives you a way to ask if *all* elements in the list return `true` for the condition you supply

And as a bit more explanation on the `count` example:

```
firstTen.count(_ % 2 == 0)                  // 5
```

it returns `5` because there are five elements in the list for which "the modulus of `2`" is equal to `0`: `2`, `4`, `6`, `8`, `10`.

Again, there are more methods than this on Scala sequence classes, but I believe these are the most commonly-used methods.

In that code this anonymous function is how you ask if the modulus of `2` is equal to `0`:

```
_ % 2 == 0
```

As shown in previous lessons, you can also write that in the long format like this:

```
firstTen.count(i => i % 2 == 0)
```

> NOTE: Several more methods return the `Option` type, and those are discussed later in this book.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

36

Importing Code With import Statements

Before we get into the next lesson, we need to take a quick look at using `import` statements, which let us import other code into the scope of our code.

Some things are automatically available

In Scala some things are made available to you automatically. For instance, you can automatically use `String`, `Int`, and some other data types:

```
val name = "Alvin"    // String
val age = 33          // Int
val weight = 200.0    // Double
val boo = true        // Boolean
```

You can also use some functions like `println` right away:

```
println("Hello, world")
```

Other classes like `List` and `Vector` are also available automatically. You don't have to do anything special, you just use them. These two classes in particular are available to you automatically because Scala emphasizes using functional (immutable) classes like these.

Other things are NOT automatically available

But to use many other classes and libraries — including third-party libraries you will integrate into your applications — you'll need to "import" them into the current scope before you can use them. For instance, if you want to use the Scala `Source` class to read a file, you must first import it:

```
import scala.io.Source
```

After you've imported it, you can then use it. For example, this code that uses `Source` reads and prints every line from the `/etc/passwd` file on a Unix-

based computer:

```
for
    line <- Source.fromFile("/etc/passwd").getLines
do
    println(line)
```

That works when you import `scala.io.Source` as shown, but if you forget to import it, you'll see this "Not found" error when you try to use that code:

```
-- Error:
1 |for line <- Source.fromFile("/etc/passwd").getLines do println(line)
  |            ^^^^^^
  |            Not found: Source
```

So that's the purpose of `import` statements: they import code into your current scope so you can use it.

Importing multiple things

You may also need to import more than one thing from another library. For instance, if you try to read a file using the `java.io` library, that process can throw an exception. This means that you not only need to import the `File` class, but also its related *exception* classes.

In Scala you can do this by importing one class per line:

```
import java.io.File
import java.io.IOException
import java.io.FileNotFoundException
```

Scala also lets you use this syntax to declare all those classes in the `java.io` package on one line:

```
import java.io.{File, IOException, FileNotFoundException}
```

That line says, "Import the `File`, `IOException`, and `FileNotFoundException` classes from the `java.io` package."

> We'll look at *packages* later in this book, but for now you can just think of them as being a place to put related code, such as the file-reading code in this example. (Or, if you're familiar with

Java, Scala packages are almost identical to Java packages.)

As a rule of thumb, when you import up to three classes from a package, use that `{...}` syntax. When you import more than three classes, use this syntax:

```
import java.io.*
```

That tells Scala to import *everything* from the `java.io` package. This makes everything in this package available to you.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

37

Collections: ArrayBuffer

As mentioned earlier, `ArrayBuffer` is a sequence class with these characteristics:

- It's *mutable*, meaning that it can grow and shrink, and its elements can be updated.
- It's *indexed*, so you can access all of its elements — like the one-millionth element — very quickly.

Because of these characteristics, `ArrayBuffer` is Scala's preferred general-purpose, mutable, indexed sequence.

How to create an ArrayBuffer

To create an `ArrayBuffer`, first import its definition into the scope of your application:

```
import scala.collection.mutable.ArrayBuffer
```

Then create an empty `ArrayBuffer` like this:

```
val strings = ArrayBuffer[String]()
val ints = ArrayBuffer[Int]()
```

If you happen to know how many elements the `ArrayBuffer` will store, you can also give it an initial size like this:

```
val ints = new ArrayBuffer[Int](100_000_000)
```

When you do this the `ArrayBuffer` will be created a little more slowly because all of those spaces are created right now, but when you later add the elements to it that process will be faster, because those slots already exist.

If you have some initial elements you want to put into it, you create the `ArrayBuffer` like this:

```
val names = ArrayBuffer("Bert", "Ernie", "Grover")
```

As with `List` and `Vector`, you can also explicitly declare the variable's data type — as shown here — but again, this is generally not necessary:

```
val names: ArrayBuffer[String] = ArrayBuffer("Bert", "Ernie", "Grover")
         ---------------------
```

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

38

Collections: Accessing and Updating ArrayBuffer Elements

You access `ArrayBuffer` elements using the 0-based index syntax, just like with the `List` and `Vector` classes. This example shows the technique:

```
import scala.collection.mutable.ArrayBuffer

val names = ArrayBuffer("Bert", "Ernie", "Grover")
println(names(0))    // prints "Bert"
println(names(1))    // prints "Ernie"
println(names(2))    // prints "Grover"
```

As with `List` and `Vector`, `ArrayBuffer` elements can be accessed in `for` loops:

```
for name <- names do println(name)
```

Updating ArrayBuffer elements

Because `ArrayBuffer` is a mutable sequence, you can also update its individual elements using the index-based approach:

```
val names = ArrayBuffer("Bert", "Ernie", "Grover")

names(0) = "BERT"
names(1) = "ERNIE"
names(2) = "GROVER"

names: ArrayBuffer("BERT", "ERNIE", "GROVER")
```

In addition to this way of updating elements, you can also use built-in collections methods like `map` to update the elements. I'll demonstrate that approach shortly.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

39

Collections: How To Initially Populate An ArrayBuffer

For the lessons that follow this one, it will help if we take a few moments to look at other ways to populate `ArrayBuffer` with initial values. These techniques can be helpful in your normal code, and are particularly useful in tests and experiments.

One of the most common approaches to populate an `ArrayBuffer` is to use its `range` method:

```
import scala.collection.mutable.ArrayBuffer

ArrayBuffer.range(1, 3)             // ArrayBuffer(1, 2)
ArrayBuffer.range('a', 'c')         // ArrayBuffer(a, b)
```

As shown in that output, the `range` method does not include the last element you provide, such as `3` and `c` in those examples. As you'll see, this is because Scala provides `to` and `until` filler methods, and `range` works like `until`.

You can also provide a third parameter to the `range` method which lets you define the number of values to skip when the data is generated:

```
ArrayBuffer.range(1, 6, 2)          // ArrayBuffer(1, 3, 5)
```

That third value is known as a *step*, or increment.

While this lesson is about how to initially populate an `ArrayBuffer`, it's important to mention that the `range` method is also available on other sequence classes:

```
List.range(1, 3)                    // List(1, 2)
Vector.range(1, 3)                  // Vector(1, 2)
```

Using 'to' and 'until'

Next, here are the `to` and `until` approaches to creating a new `ArrayBuffer`:

```
(1 to 3).toBuffer                  // ArrayBuffer(1, 2, 3)
(1 until 3).toBuffer               // ArrayBuffer(1, 2)

('a' to 'c').toBuffer              // ArrayBuffer('a', 'b', 'c')
('a' until 'c').toBuffer           // ArrayBuffer('a', 'b')
```

Notice the difference in the result between using `to` and `until`: `to` includes the final value you specify (it's *inclusive*), but `until` does not. (I suspect that I use `to` about 99% of the time.) Also notice that the `toBuffer` method creates an `ArrayBuffer`.

Similarly, you can also add the `by` approach to `to` and `until`, telling the algorithm the number of elements to skip in the generation process:

```
(1 to 5 by 2).toBuffer             // ArrayBuffer(1, 3, 5)
(1 until 5 by 2).toBuffer          // ArrayBuffer(1, 3)

(1 to 5).by(2).toBuffer            // ArrayBuffer(1, 3, 5)
(1 until 5).by(2).toBuffer         // ArrayBuffer(1, 3)
```

All of those examples also work with the `Char` type:

```
('a' to 'f').by(2).toBuffer        // ArrayBuffer('a', 'c', 'e')
```

'to' and 'until' are just methods

Bear in mind that with all of these examples, `to` and `until` are methods that are available on instances of the `Int` class. So these examples:

```
(1 to 5).toBuffer
(1 to 5 by 2).toBuffer
```

can also be written like this instead:

```
1.to(5).toBuffer
1.to(5).by(2).toBuffer
```

For this particular purpose most people think the first version of the code is easier to read, so we use it instead of the second approach.

In certain situations Scala lets you use spaces instead of decimals to invoke methods. This lets you write more readable code, and in advanced uses also lets you create your own Domain Specific Language (DSL).

For instance, some testing libraries and build tools take advantage of this technique. Here's one example from the sbt build tool[1] documentation:

```
ThisBuild / scalaVersion := "2.13.6"
ThisBuild / organization := "com.example"

lazy val hello = (project in file("."))
    .settings(
        name := "Hello"
    )
```

Exercises

The exercises for this lesson are available here[2].

[1] https://www.scala-sbt.org/index.html
[2] https://alvinalexander.com/book-exercises/LearnScala3Fast/

40

Collections: How to Add Elements to ArrayBuffer

You typically *add* elements to an `ArrayBuffer` using these methods:

- Use += to add one element to an `ArrayBuffer`
- Use ++= to add multiple elements to an `ArrayBuffer`

For example, given this initial `ArrayBuffer`:

```
import scala.collection.mutable.ArrayBuffer
val ints = ArrayBuffer(1, 2, 3)   // ArrayBuffer(1, 2, 3)
```

Add one element to it with the += method:

```
ints += 4                         // ArrayBuffer(1, 2, 3, 4)
```

And add multiple elements to it using another sequence and the ++= method:

```
ints ++= List(5, 6)               // ArrayBuffer(1, 2, 3, 4, 5, 6)
ints ++= Vector(7, 8)          // ArrayBuffer(1, 2, 3, 4, 5, 6, 7, 8)
```

More methods you can use

While those two methods are the most commonly used, you can also use these methods to add elements to an existing `ArrayBuffer`:

- `append`, `appendAll`
- `insert`, `insertAll`
- `prepend`, `prependAll`

Additionally, as you'll see below, the `clear` method removes all elements from the `ArrayBuffer`.

These methods are demonstrated in the following examples. First, here are the *append* methods:

```
val a = ArrayBuffer(1, 2, 3)      // ArrayBuffer(1, 2, 3)
a.append(4)                       // ArrayBuffer(1, 2, 3, 4)
a.appendAll(List(5, 6))           // ArrayBuffer(1, 2, 3, 4, 5, 6)
```

Next, after first clearing the values in `a`, these methods show the *prepend* and *insert* methods:

```
a.clear                           // ArrayBuffer()

a += 9                            // ArrayBuffer(9)
a.append(10)                      // ArrayBuffer(9, 10)
a.prepend(7)                      // ArrayBuffer(7, 9, 10)

a.insert(0, 6)                // ArrayBuffer(6, 7, 9, 10)
a.insert(2, 8)                // ArrayBuffer(6, 7, 8, 9, 10)
a.insertAll(0, Vector(4, 5))  // ArrayBuffer(4, 5, 6, 7, 8, 9, 10)
a.prependAll(List(2, 3))    // ArrayBuffer(2, 3, 4, 5, 6, 7, 8, 9, 10)
```

As shown, insert or prepend one element at a time with `insert` and `prepend`, and insert/prepend multiple elements with `insertAll` and `prependAll` while giving those methods a new sequence.

A note about 'for' loops

Lastly, note that you *can* add elements to an `ArrayBuffer` in a `for` loop:

```
import scala.collection.mutable.ArrayBuffer
val ints = ArrayBuffer[Int]()
for i <- 1 to 5 do ints += i   // ints: ArrayBuffer(1, 2, 3, 4, 5)
```

However, because of all of the methods I just showed, I've rarely needed to do this. Just like `foreach`, `map`, and `filter` are replacements for `for` loops and expressions, the methods just shown are replacements for this manual approach.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

41

Collections: How to Remove Elements from ArrayBuffer

Because `ArrayBuffer` is mutable, you can remove elements from it. These are the most commonly-used methods:

- `-=` removes a single element
- `--=` removes multiple elements that are declared in another sequence
- `clear` removes all the elements from an `ArrayBuffer`

These methods are demonstrated in the following examples. First, given this `ArrayBuffer[Char]`:

```
import scala.collection.mutable.ArrayBuffer

val a = ArrayBuffer.range('a', 'g')   // ArrayBuffer(a, b, c, d, e, f)
```

Remove a single element using `-=`:

```
a -= 'a'                  // ArrayBuffer(b, c, d, e, f)
```

Remove multiple elements using `--=` and specifying another sequence of elements to remove:

```
a --= List('b', 'c')      // ArrayBuffer(d, e, f)
a --= Vector('e', 'f')    // ArrayBuffer(d)
```

Other methods

There are other methods you can use to remove elements, including `clear` and `remove`. As its name implies, `clear` completely clears out the `ArrayBuffer`:

```
val a = ArrayBuffer.range('a', 'h')  // ArrayBuffer(a, b, c, d, e, f, g)
a.clear                           // ArrayBuffer[Char] = ArrayBuffer()
```

`remove` lets you remove values by specifying the index to remove:

```
val a = ArrayBuffer.range('a', 'h')  // ArrayBuffer(a, b, c, d, e, f, g)
a.remove(0)                          // ArrayBuffer(b, c, d, e, f, g)
```

You can also specify an index followed by the number of elements to remove from that point on:

```
a.remove(2, 3)                         // ArrayBuffer(b, c, g)
```

However, you need to be careful with `remove` because it can throw exceptions:

```
a.clear              // clears 'a' so it contain no elements
a.remove(0)          // ERROR: java.lang.IndexOutOfBoundsException
a.remove(100, 10)    // ERROR: java.lang.IndexOutOfBoundsException
```

Therefore, if you do use `remove` you'll want to check the `ArrayBuffer` size before using it, and account for possible exceptions.

> TIP: Rather than having to worry about exceptions with `remove`, it can be safer to use a method like `filterInPlace`, which is shown in the next lesson.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

42

Collections: Other ArrayBuffer Methods

Like `List` and `Vector`, the `ArrayBuffer` class has dozens of methods you can use that cover almost every conceivable situation. That being said, you normally only use somewhere around 10 to 20 methods on a regular basis, and I'll show many of those here.

map

As with `List` and `Vector`, one of the most important methods is `map`. As with `List` and `Vector`, `map` doesn't modify the `ArrayBuffer`, but instead returns a new `ArrayBuffer` with the modified elements, so you need to assign the result to a new variable:

```
import scala.collection.mutable.ArrayBuffer

val a = ArrayBuffer(1, 2, 3)
val b = a.map(_ * 2)    // b: ArrayBuffer(2, 4, 6)
```

Or, instead of making `a` a `val` field, define it as a `var` so you can assign the modified values back to `a`:

```
var a = ArrayBuffer(1, 2, 3)
a = a.map(_ * 2)        // a: ArrayBuffer(2, 4, 6)
```

On a personal note, because I tend to program in a functional style I don't use `ArrayBuffer` very often, but because it is mutable it makes sense to (a) define your variable as a `val` and then (b) use `mapInPlace` to mutate its contents. (However, when you're working with large datasets, always be sure to test the performance of different options like these.)

mapInPlace

However, because `ArrayBuffer` is mutable, you may want to use `mapInPlace` instead of `map`. As its name implies, it modifies the `ArrayBuffer` you call it on, so you can still define your variable as a `val` field:

```
val a = ArrayBuffer(1, 2, 3)
a.mapInPlace(_ * 2)    // ArrayBuffer(2, 4, 6)
```

filter and filterInPlace

Similarly, `ArrayBuffer` also offers `filter` and `filterInPlace` methods that work in the same way:

```
// filter
val a = ArrayBuffer(1, 2, 3, 4, 5)
val b = a.filter(_ > 2)   // ArrayBuffer(3, 4, 5)

// filterInPlace
val a = ArrayBuffer(1, 2, 3, 4, 5)
a.filterInPlace(_ > 2)    // ArrayBuffer(3, 4, 5)
```

Other functional methods

`ArrayBuffer` has the same functional methods as `List` and `Vector`. A *functional method* is a method that doesn't modify the current sequence, but instead returns a new, modified sequence. Here are some examples of the most common methods:

```
val a = ArrayBuffer.range(0, 5)       // ArrayBuffer(0, 1, 2, 3, 4)
val b = a.drop(2)                     // b: ArrayBuffer(2, 3, 4)
val b = a.dropWhile(_ < 3)            // b: ArrayBuffer(3, 4)
val b = a.take(2)                     // b: ArrayBuffer(0, 1)
val b = a.takeWhile(_ < 3)            // b: ArrayBuffer(0, 1, 2)
```

You can also convert `ArrayBuffer` into other sequences:

```
val a = ArrayBuffer.range(0, 4)       // ArrayBuffer(0, 1, 2, 3)
val b = a.toList                      // b: List(0, 1, 2, 3)
val b = a.toVector                    // b: Vector(0, 1, 2, 3)
```

You can use this technique when you first need a sequence that is going to change a lot, so you start with an `ArrayBuffer`. And then when the changes are finished, you can convert the `ArrayBuffer` to `List` or `Vector` so it will be immutable, and you're guaranteed that it can't be changed. (Sharing an immutable data structure is always safer.)

trimStart and trimEnd

If you need to change the size of an `ArrayBuffer` you can use these methods to modify the `ArrayBuffer` in place:

```
val a = ArrayBuffer.range('a', 'h')  // ArrayBuffer(a, b, c, d, e, f, g)
a.trimStart(2)                        // ArrayBuffer(c, d, e, f, g)
a.trimEnd(2)                          // ArrayBuffer(c, d, e)
```

Exercises

The exercises for this lesson are available here[1].

More information

Again, `ArrayBuffer` has dozens of methods, so see these resources for more details on those:

- The ArrayBuffer Scaladoc[2]
- My ArrayBuffer examples page[3]

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/
[2] https://www.scala-lang.org/api/current/scala/collection/mutable/ArrayBuffer.html
[3] https://alvinalexander.com/scala/arraybuffer-class-methods-syntax-examples-reference

43

Collections: What About Array?

If you're familiar with Java and the JVM, you know that the *array* type in Java is commonly used, and you might be wondering why I don't write about it.

The short story is that Scala *does* have an `Array` class that wraps the Java array, but I don't recommend it because it doesn't neatly fall into the mutable or immutable types. What I mean by that is:

- The `Array` is *immutable* in that it can't be resized
- But it's *mutable* in that its elements can be changed

This code demonstrates what I mean by that second comment:

```
val a = Array(1, 2, 3)
a(0) = 100
a    // Array(100, 2, 3)
```

So, because `Array` is neither mutable or immutable, I don't use it. I think that once you use Scala you appreciate knowing that you're working with an immutable or mutable data structure, and because `Array` doesn't neatly fall into either camp, I don't use it.

But ArrayBuffer wraps array

Also remember that `ArrayBuffer` is your go-to class for those times when you want a mutable, indexed sequence. And as it turns out, it uses an array behind the scenes, but it hides that implementation detail from us. As the `ArrayBuffer` Scaladoc[1] states, "An implementation of the `Buffer` class using an *array* to represent the assembled sequence internally. Append, update and random access take constant time (amortized time). Prepends and removes are linear in the buffer size."

[1] https://www.scala-lang.org/api/current/scala/collection/mutable/ArrayBuffer.html

But you will see Array

One reason I added this lesson is because sometimes you will see `Array` in the Scala API. For instance, because the Scala `String` is heavily based on the Java `String`, when you call the `split` method on a `String`, you'll get an `Array[String]` as the result, as shown in this REPL example:

```
scala> "foo bar".split(" ")
val res0: Array[String] = Array(foo, bar)
```

What I usually do when an `Array` makes an appearance like this is to quickly convert it to what I want, i.e., a `List`, `Vector`, or `ArrayBuffer`:

```
scala> "foo bar".split(" ").toList
val res1: List[String] = List(foo, bar)

scala> "foo bar".split(" ").toVector
val res2: Vector[String] = Vector(foo, bar)

scala> "foo bar".split(" ").toBuffer
val res3: scala.collection.mutable.Buffer[String] =
          ArrayBuffer(foo, bar)
```

44

Collections: Map

A `Map` is a data structure that lets you store data as key/value pairs. It's similar to a Java `Map` or Python dictionary.

Scala lets you create both immutable and mutable `Map` types, and the *immutable* class is the default that you get if you write code like this:

```
// you get an immutable Map by default
val peeps = Map(
    "first_name" -> "Alvin",
    "last_name" -> "Alexander"
)
```

This is because Scala makes the immutable collections classes available to you by default, to help emphasize a functional style of programming.

But because this is a book for beginners, these `Map` lessons will focus on the *mutable* `Map` class, which you'll need to import to use.

Creating a mutable Map

The mutable `Map` class can be modified, meaning that elements can directly be added, deleted, and updated. The first thing you usually do with mutable classes is to import them, like this:

```
import scala.collection.mutable.Map
```

However, in this particular case — because both the immutable class and mutable class are both named `Map` — I find this to be a wee bit confusing when I come back to my code some time later. Therefore, rather than importing the mutable `Map`, I specify the full path to the mutable class, like this:

```
val states = scala.collection.mutable.Map(
    "AL" -> "Alabama",
    "AK" -> "Alaska"
)
```

This creates a mutable `Map` with type `[String, String]`, meaning that both the *key* and *value* have the type `String`. For the first element the string `"AL"` is the key, and `"Alabama"` is the value.

Accessing elements

Just like Scala's sequence classes, you access mutable `Map` values by specifying their key as the index:

```
println(states("AL"))   // prints "Alabama"
println(states("AK"))   // prints "Alaska"
```

Creating an empty Map

If you ever need to create a mutable `Map` that is initially empty, use this syntax, specifying the data types for the map's key and value:

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

Then you can add elements to it as shown in the following lessons.

Also, note that this is consistent with how you create an empty `ArrayBuffer`:

```
val strings = scala.collection.mutable.ArrayBuffer[String]()
```

I like to point out Scala's consistency, because I think it helps to reduce the load on your brain.

If Maps are difficult …

If you find that thinking about Maps are difficult, here's a tip that I learned many years ago that may help.

What I learned is that a list is a lot like a map. The only difference is that with a list, the *key* is always an integer. So with a list like this:

```
val letters = List("a", "b", "c")
```

you supply the implicit integer keys to access each value:

```
letters(0)    // "a"
letters(1)    // "b"
letters(2)    // "c"
```

I say that the keys are *implicit* because those integer keys are created for you automatically. But for many practical purposes, that `List` is the same as this `Map`, where you provide the keys *explicitly*:

```
val letters = scala.collection.mutable.Map(
    0 -> "a",
    1 -> "b",
    2 -> "c",
)
```

Notice that you again access the values by supplying the integer keys:

```
letters(0)    // "a"
letters(1)    // "b"
letters(2)    // "c"
```

So, in this example, the `Map` is extremely similar to a `List`, and the main difference is that you supply the integer keys explicitly when you create the `Map`.

So maps work like this, except that the key can be *any data type*, not just integers. To help emphasize the data type aspect of maps, here's another example that has the type `Map[Double, String]`, meaning that the keys have the type `Double` and the values have the type `String`:

```
val letters = scala.collection.mutable.Map(
    0.0 -> "a",
    1.1 -> "b",
    2.2 -> "c",
)
```

If you struggle with maps as I once did, I hope this is helpful.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

45

Collections: Map, Adding Elements

A mutable `Map` can be updated, so to add elements you:

- Specify the key/value pairs to be added
- Use the += or ++= methods to add those pairs

For example, we start again with a `Map` of the type `Map[Int, String]`, meaning that the keys are integers and the values are strings:

```
// a new mutable Map
val a = scala.collection.mutable.Map(1 -> "a")
```

Given that `Map`, you can add elements to it like this:

```
// adding elements
a += (2 -> "b")                  // add using a tuple
a ++= Map(3 -> "c", 4 -> "d")    // add with another Map
a ++= List(5 -> "e", 6 -> "f")  // you can even add new pairs with a sequence
```

If you put those examples in the REPL, you'll see that the variable `a` now looks like this:

```
scala.collection.mutable.Map[Int, String] =
    HashMap(1 -> a, 2 -> b, 3 -> c, 4 -> d, 5 -> e, 6 -> f)
```

This tells us:

- `a` is an instance of the `scala.collection.mutable.Map` class
- Its keys have the type `Int`
- Its values are of the type `String`
- The data is stored in a specific type of `Map` known as a `HashMap`

A `HashMap` is just a `Map` that is implemented with a hashtable. If you've taken computer science classes you probably learned about hashtables, and if not,

don't worry, it's just one way to implement a `Map` that makes it very fast.

+= and ++= are used consistently

As you saw in the last several lessons, `ArrayBuffer` and this `Map` class are both mutable, and they both use the `+=` and `++=` methods to add new elements to them.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

46

Collections: Map, Updating Elements

To update existing elements in a mutable `Map`:

- Specify an existing key
- Specify the new value you want it to have using the = assignment operator

For example, given this mutable `Map` that we had at the end of the last lesson:

```
scala.collection.mutable.Map[Int, String] =
    HashMap(1 -> a, 2 -> b, 3 -> c, 4 -> d, 5 -> e, 6 -> f)
```

this example shows how to update the value for an existing key. In this example the key I'm working with is `1`, and the new desired value is `"AA"`:

```
// first, note that for the key 1, the current value is "a":
println(a(1))    // "a"

// now provide a new value for the key 1
a(1) = "AA"

// now this is the result for the key 1:
println(a(1))    // "AA"

// here's the new Map:
// a: HashMap(1 -> AA, 2 -> b, 3 -> c, 4 -> d, 5 -> e, 6 -> f)
```

Similarly, this is how you replace the value where the key is `2`:

```
a(2) = "BB"
```

And after those two updates, here's the final `Map`:

```
// a: HashMap(1 -> AA, 2 -> BB, 3 -> c, 4 -> d, 5 -> e, 6 -> f)
```

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

47

Collections: Map, Deleting Elements

To remove elements from a mutable `Map`:

- Specify the keys to remove from the `Map`
- Use those keys with the `-=` and `--=` methods

For example, given this `Map` that we had at the end of the last lesson:

```
a: HashMap(1 -> AA, 2 -> BB, 3 -> c, 4 -> d, 5 -> e, 6 -> f)
```

these examples show how the deletion process works:

```
// remove elements by specifying the keys to remove
a -= 6             // remove one element, where the key is 6
a --= List(4, 5)   // remove multiple elements using a sequence

// final result:
// HashMap(1 -> AA, 2 -> BB, 3 -> c)
```

As I note in the comments, you specify the *key* values you want to remove from the `Map` when using these methods.

-= and –= are used consistently

As you just saw in the last several lessons, `ArrayBuffer` and this `Map` class are both mutable, and they both use the `-=` and `--=` methods to add new elements to them.

Exercises

The exercises for this lesson are available here[1].

More information

For more information, here's a tutorial I wrote about working with mutable maps:

- How to add, update, and remove elements with mutable Maps[2]

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/
[2] https://alvinalexander.com/scala/how-to-add-update-remove-mutable-map-elements-scala-cookbook/

48

Collections: Map, How To Loop Over Maps

You can manually loop over a `Map` using a `for` loop. For instance, given this mutable `Map`:

```
val shipmates = scala.collection.mutable.Map(
    "Captain" -> "Kirk",
    "1st Officer" -> "Spock",
    "Doctor" -> "McCoy"
)
```

you can loop (iterate) over the map's keys and values and print them like this:

```
for (k, v) <- shipmates do println(s"key: $k, value: $v")
```

That `for` loop results in this output:

```
key: Captain, value: Kirk
key: 1st Officer, value: Spock
key: Doctor, value: McCoy
```

A key to this solution is knowing that when you loop over a `Map`, it returns each pair as a tuple. That's why I use this `(k, v)` syntax at the beginning of the loop:

```
for (k, v) ...
```

In the `do` portion of the loop you receive each value as a two-element tuple. I give the tuple variables the names `k` and `v`, but you can call them anything you want:

```
for (currentKey, currentValue) ...
```

Updating a mutable Map in a loop

You can update the values of a mutable `Map` inside a `for` loop. For example, given this same mutable map:

```
val shipmates = scala.collection.mutable.Map(
    "Captain" -> "Kirk",
    "1st Officer" -> "Spock",
    "Doctor" -> "McCoy"
)
```

this code shows how to convert each value to uppercase:

```
// note: this code can go on one line, if you prefer
for
    (k, v) <- shipmates
do
    shipmates(k) = v.toUpperCase
```

After that code runs, the REPL shows that `shipmates` has these values:

```
HashMap(Doctor -> MCCOY, Captain -> KIRK, 1st Officer -> SPOCK)
```

However, note that this is one of those custom `for` loops I warned you about earlier in the book. The next lesson shows the built-in `Map` methods that will keep you from having to write custom `for` loops like this.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

49

Collections: Map, Common Map Methods

Because I'm showing the mutable `Map` class, in this lesson I'll demonstrate methods that operate directly on your `Map` variable.

A unique thing about these methods compared to `ArrayBuffer` and the other sequence classes is that your anonymous functions typically receive the key/value pairs as a two-element tuple, so the important thing to know is how to handle this situation. I'll demonstrate this in the following examples.

filterInPlace

The `filterInPlace` method lets you do what its name implies, filter the `Map` elements. So, given this `Map`:

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

Use `filterInPlace` like this:

```
map.filterInPlace((k,v) => k > 1)

// 'map' now has these contents: Map(2 -> "two", 3 -> "three")
```

As shown, that filters out the `Map` element whose key is `1`.

An important part of that solution is knowing that your anonymous function receives a two-element tuple, so you handle that as shown in this underlined code:

```
map.filterInPlace((k,v) => k > 1)
                  -----
```

This anonymous function can be read as, "I expect to receive a two-element tuple, and I named those variables `k` and `v` (short for *key* and *value*). Then I filter the `Map` so I only retain elements where the key is greater than `1`." The custom algorithm in this example is underlined here:

```
map.filterInPlace((k,v) => k > 1)
                           -----
```

mapValuesInPlace

The mutable `Map` class doesn't have a `mapInPlace` method like `ArrayBuffer`, possibly because that name feels a little ambiguous about what it might do. Therefore it is named `mapValuesInPlace` instead, and works like this:

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

map.mapValuesInPlace((k,v) => v.toUpperCase)

// 'map' now contains these values:
// Map(1 -> "ONE", 2 -> "TWO", 3 -> "THREE")
```

Where the previous example filtered elements out of the `Map`, this method is used to update the map's values.

Other methods

Here's a small set of other `Map` methods that are available to you:

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

map.contains(1)           // true (tests your value against the keys)
map.count((k,v) => k > 1)     // 2
map.exists((k,v) => k == 1)   // true
```

```
map.isEmpty                 // false
map.keys                    // Set(1, 2, 3)
map.size                    // 3
```

Additionally, `getOrElse` is an interesting method that returns the value for the key you supply, but if the key doesn't exist, it uses the default value you supply as the second parameter:

```
map.getOrElse(50, "fifty")
```

clear

To clear all elements from a mutable `Map`, call its `clear` method:

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

map.clear    // empty, and has the type: Map[Int, String]
```

Exercises

The exercises for this lesson are available here[1].

More information

See the following links for more information:

- The mutable Map class Scaladoc[2]
- Scala has many different types of `Map` classes, and I write about them in How to choose a Map implementation in Scala (Map, ListMap, LinkedHashMap, SortedMap, TreeMap)[3]

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/
[2] https://www.scala-lang.org/api/current/scala/collection/mutable/Map.html
[3] https://alvinalexander.com/scala/how-to-choose-map-implementation-class-sorted-scala-cookbook

50
Collections: Set

A *set* is an iterable type of collection that contains only unique elements. The immutable `Set` class is available to you by default, so you create one like this:

```
val set = Set(1, 1, 1, 2, 2, 3)
```

Because a `Set` only contains unique elements, the result of that code is that `set` contains these values:

```
Set(1, 2, 3)
```

The Iterable type, and Set methods

Many of the methods you've seen on the previous Scala sequence classes are defined in the Iterable data type[1]. This includes `filter`, `map`, the "drop" and "take" methods, and many more. Because `Set`, `List`, and `Vector` all descend from `Iterable`, using `Set` is similar to using these other sequences, although `Set` is not technically a sequence.

Because `Set` is an iterable, you can use the methods on it that I've already demonstrated with `List` and `Vector`:

```
set.foreach(println)         // prints "1 2 3" on separate lines
for s <- set do println(s)   // same result as 'foreach'

// add one element while creating a new set:
val a = set + 4              // Set(1, 2, 3, 4)

// add multiple elements from another sequence:
val b = set ++ List(4, 5)    // Set(1, 2, 3, 4, 5)
```

> Note again that the + and ++ operators are used consistently with immutable collections.

[1] https://www.scala-lang.org/api/current/scala/collection/immutable/Iterable.html

Don't count on the elements being in order

Once a set gets to a certain size, you'll begin to see in the REPL that it has a specific type of `HashSet`:

```
scala> val b = set + (4, 5)
val b: Set[Int] = HashSet(5, 1, 2, 3, 4)
```

This is just a technical detail, and it means that the set is implemented using a hashtable, which gives you rapid access to its elements. However, the REPL output shows a key point about sets: you can't rely on their elements being returned in a certain order. If you run `b.foreach(println)` you'll see that `5` is the first element printed.

> Summary: A set is useful anytime you need a data type that mostly works like a sequence, but only contains unique elements. As shown, when you try to add duplicate elements to a set, the duplicates are discarded.

Exercises

The exercises for this lesson are available here[2].

[2] https://alvinalexander.com/book-exercises/LearnScala3Fast/

51
Collections: Ranges

As its name implies, `Range` is a class that lets you create a range of values. You typically create ranges of `Int` values, but can also create ranges of `Char` values.

You actually saw the `Range` in action earlier in this book in examples like this:

```
for i <- 1 to 3 do println(i)
1
2
3
```

The `Range` is created in this underlined portion of code:

```
for i <- 1 to 3 do println(i)
         ------
```

Both the `Int` and `Char` classes have `to` or `until` methods that let you create ranges with the desired elements. The `until` method works the same way, but excludes the last value in the range:

```
for i <- 1 until 3 do println(i)

// 'until' results in:
1
2
```

As shown in the following examples, if you want a range of some other data type, you begin with these types and then create what you want by converting the result with the `map` method.

> TIP: As shown in the next lesson, to populate sequences you can also use the `range` method of the desired sequence class (`List`, `Vector`, `ArrayBuffer`, etc.).

Background: A little more about Scala

This is a good time to mention that these two lines of code are the same thing:

```
1 to 3
1.to(3)
```

They both create a `Range[Int]` that contain the same values. The way this code works is:

- `to` is a method that is available on the Scala `Int` type
- When a method for a type takes one parameter, that method can be used in both ways shown

The first approach is known as *infix notation*, which just means that the method can be used in between the two values using spaces. Generally speaking, we don't often use the infix notation in normal code, but with the `to` and `until` methods it makes your code more readable, so we use it here.

> NOTE: I say *normal* code because this is one of the techniques available in Scala that let us write what we call *Domain Specific Languages*, or DSLs. You'll see DSLs used in places like the sbt build tool[1], and in testing libraries like ScalaTest[2].

Creating a range with 'to'

Getting back to ranges, here are some examples of creating ranges with the `to` method, with the result shown after each comment:

```
1 to 5         // will contain Range(1, 2, 3, 4, 5)
1 to 10 by 2   // will contain Range(1, 3, 5, 7, 9)
1 to 10 by 3   // will contain Range(1, 4, 7, 10)
```

But ranges are lazy

The results I show after the comments are actually a bit of a half-truth, because a range is created *lazily*, meaning that *the elements are only evaluated*

[1] https://www.scala-sbt.org
[2] https://www.scalatest.org

when they're needed. For instance, if you put the code from the second example in the Scala 3 REPL, you'll see this output:

```
scala> 1 to 10 by 2
val res0: Range = inexact Range 1 to 10 by 2
```

This is a way of telling you that the range hasn't been evaluated yet — it doesn't contain any elements right now, but it knows how to create them. To force it to be evaluated and contain elements, you have to do something to it, such as converting the elements to a `List`:

```
scala> (1 to 10 by 2).toList
val res1: List[Int] = List(1, 3, 5, 7, 9)
```

Laziness

If you haven't seen this before, this is how things work in Big Data tools like Apache Spark, which is used to analyze enormous datasets. *Lazy* collections — in this case `Range` — are lazily evaluated. The way this works is that `Range` has two types of methods:

- *Transformation* methods that are *lazy* or *non-strict*
- *Action* methods that are *strict*, and behave the way you're used to in most languages, i.e., forcing the action to occur immediately

For instance, given this range:

```
val r = 1 to 10    // result: Range 1 to 10
```

here are examples of lazy/non-strict methods, with their actual REPL output shown after each comment:

```
r.by(2)            // Range 1 to 10 by 2
r.drop(5)          // Range 6 to 10
r.distinct         // Range 1 to 10
r.tail             // Range 2 to 10
```

Because these method calls are lazy, they don't do much of anything. They will only be executed when they are forced to run by an action method. Conversely, there are examples of action methods (that quite literally force some action to occur):

```
r.filter(_ > 5)    // Vector(6, 7, 8, 9, 10)
r.min              // 1
r.max              // 10
r.size             // 10
r.toList           // List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

One way to demonstrate the laziness of ranges is to type this code into the REPL:

```
val a = 1 to Int.MaxValue
```

When you do that you should see that the REPL returns with its output almost immediately. That's because this process is lazy, and not much happens yet. But if you try to do the same thing *and* also convert that to a `List`, that process will take *much* longer, and may also fail with a `java.lang.OutOfMemoryError`:

```
// takes much longer and will probably fail:
val a = (1 to Int.MaxValue).toList
```

This demonstrates the difference between the lazy code in the first example, followed by the strict/action code in the second example.

While it's good to know about lazy and strict methods, don't fret about this too much. I used ranges for many months before I knew about these details.

Creating a range with 'until'

Now that you've seen the `to` method, it's helpful to know that ranges can also be created with the `until` method, which excludes (skips) the last value you supply. Here's a comparison of the two:

```
(1 to 3).toVector          // Vector(1, 2, 3)
(1 until 3).toVector       // Vector(1, 2)

(1 to 10 by 3).toList      // List(1, 4, 7, 10)
(1 until 10 by 3).toList   // List(1, 4, 7)
```

As shown, `to` keeps the last element you specify, while `until` does not. Other than that, they behave the same. (Technically we say that `to` is *inclusive* and `until` is *exclusive*.)

Char ranges

You can also use all of the previous approaches with `Char` values:

```
'a' to 'e'                  // NumericRange a to e
'a' until 'e'               // NumericRange a until e

// 'to' is inclusive, 'until' is not
('a' to 'e').toList         // List(a, b, c, d, e)
('a' until 'e').toList      // List(a, b, c, d)

// you can also use a step with Char
('a' to 'e' by 2).toList    // List(a, c, e)
('a' to 'e').by(2).toList   // List(a, c, e)
```

Exercises

The exercises for this lesson are available here[3].

[3] https://alvinalexander.com/book-exercises/LearnScala3Fast/

52

Collections: Creating Collections from Ranges

A great thing about ranges is that you can create other collections from them. For instance, given this range:

```
val r = 1 to 5    // result: Range 1 to 5
```

These are a few ways you can create a new collection from an existing `Range`:

```
r.toList                        // List(1, 2, 3, 4, 5)
r.toVector                      // Vector(1, 2, 3, 4, 5)

r.filter(_ > 3)                 // Vector(4, 5)
r.filter(_ > 3).toList          // List(4, 5)
r.filter(_ > 3).map(_ * 2.0)    // Vector(8.0, 10.0)
```

You can even create a `Map` from a `Range` like this:

```
r.filter(_ > 3)
 .map(i => (i, i * 2.0))
 .toMap
```

In this last example, because I transform the given value `i` into the two-element tuple `(i, i * 2.0)` with the `map` method, this approach creates a type that can be transformed into a `Map`. The REPL shows that `map` creates this data type:

```
IndexedSeq[(Int, Double)] = Vector((4,8.0), (5,10.0))
```

After that, the `toMap` method transforms that structure into a `Map[Int, Double]` that contains these values:

```
Map(4 -> 8.0, 5 -> 10.0)
```

> TIP: In the output that shows `IndexedSeq`, you can think of `IndexedSeq` as being a more general (abstract) version of `Vector`, i.e., it is a parent class of `Vector`.

Char ranges

Similarly, here are some examples with character ranges:

```
val letters = ('a' to 'c').toList         // List(a, b, c)
val letters = ('a' to 'f').by(2).toList   // List(a, c, e)
```

In both of those examples, `letters` has the type, `List[Char]`.

Populating sequences with the 'range' method

Because I'm talking about ranges, I should also mention that the sequence classes also have a `range` method:

```
Vector.range(1, 3)        // Vector(1, 2)
List.range(1, 6, 2)       // List(1, 3, 5)

import collection.mutable.ArrayBuffer
ArrayBuffer.range(1, 3)   // ArrayBuffer(1, 2)
```

As shown in those examples, the second parameter is not included in the result, so it works like the `until` method. The third parameter is the step size, which defaults to `1`, but can be changed:

```
List.range(1, 10)      // List(1, 2, 3, 4, 5, 6, 7, 8, 9)
List.range(1, 10, 2)   // List(1, 3, 5, 7, 9)
List.range(1, 10, 3)   // List(1, 4, 7)
List.range(1, 10, 4)   // List(1, 5, 9)
```

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

More information

For more information on how to populate sequences with data, see this tutorial:

- How to fill/populate a Scala sequence[2]

And for more advanced sequence-populating needs, see these:

- How to generate random numbers, characters, and sequences in Scala[3]
- Different ways to create random strings in Scala[4]
- How to create a list of alpha or alphanumeric characters[5]

Those examples should be helpful when you need to create random sequences, such as for testing.

[2] https://alvinalexander.com/source-code/scala-how-fill-populate-list-same-different-elements
[3] https://alvinalexander.com/scala/how-to-generate-random-numbers-characters-sequences-in-scala/
[4] https://alvinalexander.com/scala/creating-random-strings-in-scala/
[5] https://alvinalexander.com/scala/create-list-alpha-alphanumeric-characters-in-scala/

53

Functions and Methods

NOTE: Before we get started on the following lessons I need to note that it may appear that I use the terms *function* and *method* interchangeably throughout the remainder of this book. Technically everything that's defined with the `def` keyword is a *method*, but my personal approach is to call it a *function* if it's defined outside of a class, and a *method* when it's defined in a class. This is not a Scala standard, it's just the way I've come to think about things. No doubt this is influenced by thinking of Scala as a *functional* programming language (i.e., based on functions).

Functions

As with other programming languages, Scala functions are a place where you put reusable algorithms. For example, this is a function that multiplies a given `Int` by `2`. I'll put it on two lines so it's easier to read:

```
def double(i: Int): Int =
    2 * i
```

Once you get used to Scala, you'll generally put code like this on one line:

```
def double(i: Int): Int = 2 * i
```

In either case, this is how you use a function after you've created it:

```
val x = double(2)     // x is an Int with the value 4
println(double(3))    // prints the number 6
```

Looking at that function again

When you define a function it will have these components:

- The `def` keyword starts the definition of a function
- The function name
- The function input parameter(s)

- The function return type
- The body of the function

Given that background, here's that `double` function again:

```
def double(i: Int): Int = 2 * i
```

`double` can be read as, "This is a function named `double` that takes an integer parameter `i`. Its body multiplies `i` by `2`, and then returns that value as an integer (`Int`)."

Multiple input parameters

Notice that this example has only one input parameter. When your function requires multiple parameters, just add them inside the parentheses, separated by commas:

```
def add(i: Int, j: Int): Int = i + j
def add(i: Int, j: Int, k: Int): Int = i + j + k

def printIntWithMessage(msg: String, int: Int): Unit =
    println(s"$msg $int")
```

The last function doesn't return anything, so you put `Unit` as its return type.

> TIP: Remember that any time you see a function with a `Unit` return type you should think, "Whatever it does, it's being used for its side effect."

Multiline functions

With Scala's expressiveness a surprising number of functions can be reduced to one line, but of course you'll also need functions whose body has multiple lines. Here's an example of how to declare a multiline function that uses the *if/then expression* demonstrated earlier in the book:

```
def removeTrailingS(s: String): String =
    if s.length == 0 then
        s
    else if s.last == 's' then
        s.dropRight(1)
    else
        s
```

Two notes about this code:

- I never use `null` values in my code, so in my functions I never check for `null` values
- The `last` method on a `String` will throw an exception on an empty string, so that's why I first check for a zero-length string

Other than that, as its name implies, if a `String` passed into that function contains a letter `s` as its last character, this function removes that `s`:

```
removeTrailingS("")          // ""
removeTrailingS("Big")       // "Big"
removeTrailingS("Belly")     // "Belly"
removeTrailingS("Burgers")   // "Burger"  (the 's' is removed)
```

This is a first look at how to write functions in Scala.

An "Advanced Scala" note

In this book I don't normally talk about advanced Scala features, but in this lesson I want to at least give you a hint about an advanced feature, because you may see it in the Scaladoc or internet examples and wonder how it works.

The short story is that Scala functions can have *multiple parameter groups*, so you can write an `add` function like this:

```
def add(i: Int)(j: Int): Int = i + j
```

Notice that `i` and `j` are in two different parameter groups — two sets of parentheses — in that function definition:

```
def add(i: Int)(j: Int): Int = i + j
        ------  ------
```

With that definition you can call `add` like this:

```
val a = add(1)(2)    // 'a' will be 3
```

This technique isn't important for this book, but it becomes an incredibly useful feature in advanced Scala programming. Again, I only mention it now because it's something that may come up in your initial adventures, so I thought I should mention it.

A note about seeing ??? with functions

Possibly because Martin Odersky uses Scala as a teaching language, and also because Scala is such a powerful FP language, he decided to let us "sketch" functions like this:

```
def add(i: Int, j: Int): Int = ???
```

In Scala this is perfectly legal code, so if you put it in the REPL you'll see that it compiles. It will throw a `scala.NotImplementedError` if you try to *call* `add`, but it does compile.

I may use this technique in this book whenever I want to indicate that the function body doesn't matter. And I definitely use it in Functional Programming, Simplified[1], because when you write *pure functions*, generally all you have to do is look at function signatures to know what a function can possibly do. For instance, if you know that this is a pure function — meaning that it doesn't have any side effects, doesn't mutate anything, doesn't throw exceptions, and `s` is it's only input — you can ask yourself, what can this function *possibly* do:

```
def mysteryFunction(s: String): Int = ???
```

Well, it takes a `String` and returns an `Int`, so the only things I can think of are that `mysteryFunction` returns:

- The string length
- A numerical hash of the string, such as an MD5 hash

[1] https://alvinalexander.com/scala/functional-programming-simplified-book

If the function is poorly written you can argue that it may also attempt to convert the `String` to an `Int`, such as attempting to convert `"1"` or `"zen"` to an `Int`. But as you'll see in the following lessons on functions, you won't write a function like this in Scala. We use other techniques, and that would be reflected in the function's return type. (You'll learn about this in the following lessons!)

Exercises

The exercises for this lesson are available here[2].

[2] https://alvinalexander.com/book-exercises/LearnScala3Fast/

54

Functions: Creating A Main Method

In this lesson we'll begin writing complete programs that you can run from your operating system command line. If you're using Scastie[1] you may be able to keep using it, but at this point I'm going to assume that you've installed Scala-CLI[2] or the Scala SDK.

main methods

In Scala 3 you add an annotation to a function to make it a special type of function (or method) known as a *main method*. A `main` method is the method that starts your application. It's also referred to as the *entry point* of your application.

For example, when you put this code in a file named `Hello.scala`:

```
def double(i: Int): Int = i * 2

@main
def hello() =
    println("Hello, world")
    val x = double(2)
    println(s"2 * 2 = $x")
```

This creates a complete application that you can run with `scala-cli`. This shows how you run it, and its output:

```
$ scala-cli Hello.scala
Hello, world
2 * 2 = 4
```

[1] https://scastie.scala-lang.org
[2] https://scala-cli.virtuslab.org

As shown, when it runs it first prints the string in the `println` method. Then it calls the `double` function to double the number `2`, saving that value in the variable `x`. Then it prints the final output.

scalac and scala

Note that if you are not using `scala-cli` and are using the `scalac` and `scala` commands, you first compile the code with `scalac`:

```
$ scalac Hello.scala
```

This creates several output files, which you can see here:

```
$ ls -1 Hello*
Hello$package$.class
Hello$package.class
Hello$package.tasty
Hello.scala
```

With it compiled successfully you can now run your `main` method with the `scala` command:

```
$ scala hello
```

That command produces the same output as before:

```
Hello, world
2 * 2 = 4
```

Command-line parameters

When you define a `main` method you can also specify that it expects input parameters. For instance, in this example I declare that the `main` method expects a `String` input parameter which I have named `name`:

```
// i won't use these functions, but i want to show that
// you can define as many as you need here:
def add(i: Int, j: Int): Int = i + j
def double(i: Int): Int = i * 2
```

```
// in Scala 3 you can also create “top level” variables like this:
val x = double(2)

@main
def hello(name: String) =
    println(s"Hello, $name")
    println(s"2 * 2 = $x")
```

When you save this code to `HelloParameters.scala` and run the application, it now expects you to provide a string as an input parameter. With `scala-cli` you run the code like this, providing the command-line argument after the `--` symbol:

```
$ scala-cli HelloParameters.scala -- Alvin
Hello, Alvin
2 * 2 = 4
```

However, note that because I don't do any error-checking, this program will fail if you don't provide an input parameter, so both of these attempts will fail:

```
$ scala-cli HelloParameters.scala --
Illegal command line: more arguments expected

$ scala-cli HelloParameters.scala
Illegal command line: more arguments expected
```

As an exercise, if you want to add error-checking to this code, one approach is to use `String` methods like `isEmpty` or `length`:

```
"foo".isEmpty    // false
"".isEmpty       // true
"foo".length     // 3
```

scala-cli vs “scalac and scala”

This example demonstrates an important difference between running applications with `scala-cli` or the `scala` command. When your application requires command-line arguments:

- With `scala-cli`, pass those arguments after the `--` symbol
- With `scala`, compile and run the command like this:

```
// compile the code
$ scalac HelloParameters.scala

// run the code
$ scala HelloParameters Alvin
```

I've come to prefer `scala-cli` for small applications and scripts, so I want to note that there's a difference in how it works here. You'll see other `scala-cli` command-line techniques in the "Example" lessons at the end of the book.

Exercises

The exercises for this lesson are available here[3].

[3] https://alvinalexander.com/book-exercises/LearnScala3Fast/

55

Functions: Defaults For Function Parameters

Function parameters can also have default values. Here's an example that isn't terribly useful, but it does show the technique:

```
def printHello(name: String = "Alvin"): Unit =
    println(s"Hello, $name")
```

That code can be read as, "If the caller to `printHello` doesn't provide the `name` value, use the string `"Alvin"` as the default. But if they do provide a value, use it instead." Here's how it works:

```
printHello()         // Hello, Alvin
printHello("Joe")    // Hello, Joe
```

A more useful example

As a more useful example, imagine that you're writing a function that needs to make a connection to a URL. Because you're a good programmer, you want to account for both a connection timeout and a read timeout. (This keeps your application from getting stuck when the website you're calling is down or responding slowly.) This code shows how you handle this situation, i.e., by providing default values for those two timeout fields:

```
def getUrlData(
    url: String,
    connectionTimeout: Int = 5_000,
    readTimeout: Int = 5_000
): String =
   println(s"cTimeout = $connectionTimeout, rTimeout = $readTimeout")
    // your real code would be here
    "Here's the data!"
```

Providing default values lets other programmers use your code in these different ways, with the `println` output shown after the comments:

```
getUrlData("https...")
    // result: cTimeout = 5000, rTimeout = 5000

getUrlData("https...", 2_500)
    // result: cTimeout = 2500, rTimeout = 5000

getUrlData("https...", 10_000, 10_000)
    // result: cTimeout = 10000, rTimeout = 10000
```

This is how those examples work:

- In the first case I don't provide either timeout value, so both defaults are used.
- In the second case I provide the `connectionTimeout` but not the `readTimeout`, so my `connectionTimeout` value is used, and the default is used for the `readTimeout`.
- In the third case I provide both values, so both of my values are used.

As those examples also show, it appears that I have three different `getUrlData` methods, but as you saw, there's really only one. This is really nice for users of your function, and a great benefit of default parameter values.

Also note that because default parameters let you skip parameters when you call a function, if you have a mix of some regular parameters and some default parameters when defining a function, the default ones should be listed last in your function definition. (If they're listed first, there won't be any way for callers to skip those fields because the others are required.)

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

56

Functions: Named Parameters

Scala also lets you provide the *names* of function parameters when you call a function. To use this feature you don't have to do anything special when creating your function, just write it as usual:

```
def truncate(string: String, length: Int): String =
    string.take(length)
```

Then when you call it you can provide the parameter names and desired values:

```
val a = truncate(
    string = "freedom",
    length = 4
)

// result: a == "free"
```

As I mention in the Scala Cookbook[1], this functionality is useful when a function has multiple parameters of the same type. For example, this method call may be a little hard to read:

```
engage(true, true, true, false)
```

But it's easier to understand when you can see the parameter names:

```
engage(
    speedIsSet = true,
    directionIsSet = true,
    picardSaidMakeItSo = true,
    turnedOffParkingBrake = false
)
```

This feature isn't used often in Scala, but it's there if you want to use it.

[1] https://amzn.to/3du1pMR

Exercises

The exercises for this lesson are available here[2].

[2] https://alvinalexander.com/book-exercises/LearnScala3Fast/

57

Functions: Handling a Variable Number of Parameters

Functions can be created so that a parameter can be repeated as many times as needed by a caller. This is referred to as a *varargs* parameter.

So far we've written methods that take one instance of a parameter:

```
def printOne(i: Int) = println(i)
```

But to make this function take *zero or more integers*:

1. Add a * after the parameter type, then
2. Handle that parameter like a sequence in the method body

Here's an example:

```
def printZeroOrMore(ints: Int*) = ints.foreach(println)
```

Now all of these examples will print the integers they're given:

```
printZeroOrMore()
printZeroOrMore(1)
printZeroOrMore(1, 2)
printZeroOrMore(1, 2, 3)
```

Note that even the first example, where I don't pass in any parameters, is legal.

varargs rules

There are two rules about varargs parameters:

- A function can only have one varargs parameter
- If the function has multiple parameters, it must be the last parameter

For example, this function takes a `String` parameter and then a variable-number of integers:

```
def printStuff(string: String, ints: Int*) =
    println(s"s = $string")
    ints.foreach(println)
```

It can be called like this:

```
printStuff("yo")
printStuff("yo", 1)
printStuff("yo", 1, 2)
printStuff("yo", 1, 2, 3)   // as many ints as desired
```

To examine why those two rules are required, take a look at these invalid functions and ask yourself why they won't compile:

```
// these functions won't compile
def badFunction1(i: Int*, j: Int) = ???
def badFunction2(i: Int*, j: Int*) = ???
```

If you don't see the problem right away, take a look at this code and try to understand what the compiler will think:

```
badFunction1(1, 2, 3)
badFunction2(1, 2, 3)
```

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

58

Functions: Functional Error Handling, Part 2

These are Scala's three built-in error-handling data types:

- `Option`/`Some`/`None`
- `Try`/`Success`/`Failure`
- `Either`/`Left`/`Right`

`Try`/`Success`/`Failure` and `Either`/`Left`/`Right` are similar to `Option`/`Some`/`None` in that there's a base type (`Option`, `Try`, `Either`), and then those base types have sub-types for the success and failure cases.

Try/Success/Failure

When you use `Try`, its `Failure` type contains a `String` that holds the error message you want to pass to callers when an error occurs. This is how `makeInt` is written with `Try`:

```
// you must first import these types:
import scala.util.{Try, Success, Failure}

// when you have a method call that throws an exception,
// the simplest approach is to call 'Try' with your algorithm that can
// throw the exception. then it returns a 'Success' when things
// succeed, and a 'Failure' when an exception occurs:
def makeInt(s: String): Try[Int] = Try(s.toInt)
```

This is what its results look like when it's called:

```
makeInt("1")     // Success(1)
makeInt("one")   // Failure(java.lang.NumberFormatException...)
```

As shown:

- You get a `Success` back when the conversion works. The `Success` contains the `Int`.
- You get a `Failure` back when the process fails. The `Failure` contains the exception message.

As with `Option`, you often handle the result with a `match` expression:

```
makeInt(aString) match
    case Success(i) => println(s"Success: i = $i")
    case Failure(s) => println(s"Failed: msg = $s")
```

Either/Left/Right

The `Either`/`Left`/`Right` classes give you more flexibility than the two previous approaches. Typically the `Left` contains information about a possible error and `Right` contains the value you want in the success case, but technically you can do anything you want.

That being said, here's the same method using `Either`:

```
def makeInt(s: String): Either[String, Int] =
    try
        Right(s.toInt)
    catch
        // for this example i convert the exception to a string
        case e: NumberFormatException => Left(e.getMessage)
```

In this example I return a `String` in the `Left` position, but I could also return an exception:

```
// can return Exception or Throwable
def makeInt(s: String): Either[Exception, Int] =
    try
        Right(s.toInt)
    catch
        // return the exception
        case e: NumberFormatException => Left(e)
```

In that solution, `Either[Exception, Int]` can read as, "If the computation

succeeds, I'll get an `Int` back that's wrapped in the `Right`. If it fails I'll get the exception information in the `Left`." This demonstrates the flexibility of `Either` as compared to `Option` and `Try`.

Here's how that second function is used:

```
makeInt(aString) match
    case Right(i) => println(s"Success: i = $i")
    case Left(e) => println(s"Failed, exception = $e")
```

`Either` is an interesting data type, and as I mention in the Scala Cookbook[1], using it helps to get you ready for programming in an FP style. The Cats and ZIO libraries both reference multiple data types a lot.

Shorter versions of those functions

I mentioned that these functions can all be written more concisely than what I have shown, and I showed a short form in the `Try` example. Here are short versions of all the functions that use a Scala type known as `allCatch`:

```
import scala.util.{Try,Success,Failure}
import scala.util.control.Exception.*

def makeInt(s: String): Option[Int] =
    allCatch.opt(s.toInt)

def makeInt(s: String): Either[Throwable, Int] =
    allCatch.either(s.toInt)

def makeInt(s: String): Try[Int] =
    allCatch.toTry(Try(s.toInt))

// the Try approach shown earlier
def makeInt(s: String): Try[Int] = Try {
    // use curly braces here to demonstrate that
    // this body could contain multiple lines
    s.toInt
}
```

[1] https://amzn.to/3du1pMR

Exercises

The exercises for this lesson are available here[2].

More information

- Some Scala Exception 'allCatch' examples[3]

[2] https://alvinalexander.com/book-exercises/LearnScala3Fast/
[3] https://alvinalexander.com/scala/scala-exception-allcatch-examples-option-try-either

59

Functions: Using Functions with HOFs

Earlier in this book I noted that collections methods like `foreach`, `filter`, and `map` are *Higher-Order Functions*, or HOFs. This just means that they can accept other functions as input parameters. (HOFs can also *return* functions, but that's well outside the scope of this book.)

In those earlier lessons I show that you can pass *anonymous functions* into those HOFs to achieve a desired effect, such as these examples:

```
val strings = List("jyn", "cassian", "bodhi")

// foreach
strings.foreach(s => println(s.length))

// filter
val a = strings.filter(_.length < 4)

// map
val capNames = strings.map(_.toUpperCase)
```

In addition to taking anonymous functions as input parameters, those methods can also take "regular" functions — the ones you've been learning in the last few lessons. Let's see how this works.

A 'filter' example

To get started, here's a function that returns `true` if the `Int` it's given is an even number, or `false` otherwise:

```
def isEven(i: Int): Boolean =
    i % 2 == 0
```

These examples show how `isEven` works:

```
isEven(1)   // false
isEven(2)   // true
```

A great thing about HOFs is that not only can they take anonymous functions, but they also take regular functions like `isEven`:

```
val ints = List(1, 2, 3, 4, 5)
ints.filter(isEven)    // ints: List(2, 4)
```

This example works because:

- `ints` is a `List[Int]` (a list of integers)
- `filter` passes one element at a time to your function
- `filter` is an HOF that takes your function as an input parameter, and your function must:
  - take an `Int` as input
  - and return a `Boolean`

Because `isEven` matches that description — it takes an `Int` input parameter and returns a `Boolean` — it can be used with `filter`.

> For more information on how this works, see the link I share at the end of this lesson.

A 'map' example

Similarly, imagine that you have this function, which returns a truncated version of the string it's given, so that it's no more than two characters in length:

```
def truncate(s: String): String = s.take(2)
```

Because `truncate` takes a `String` input parameter, it can be used with `map` on a `List[String]`:

```
val a = List("jyn", "cassian", "bodhi")
val b = a.map(truncate)   // b: List("jy", "ca", "bo")
```

Once again this works because:

- `map` passes one element at a time to your function
- In this case `map` is being called on a `List[String]`, so each element it passes has the type `String`
- The `truncate` function takes a single `String` as its input parameter

As you saw in the earlier anonymous function examples, a function that's passed to `map` can return *any* data type. So there is no requirement about its return type.

Notice in this example that `truncate` would be much more useful if we could pass it a desired maximum length, like this:

```
def truncate(s: String, maxLength: Int): String = s.take(maxLength)
```

This leads to the question: Because `truncate` now takes two input parameters, can we still use it with `map`?

The answer is yes, we can:

```
val b = a.map(truncate(_, 3))   // List("jyn", "cas", "bod")
```

Also note that if you prefer, that code can be written like this instead:

```
val b = a.map(s => truncate(s, 3))
```

Summary

These examples are intended to show that in addition to passing anonymous functions into HOFs, you can also pass them regular functions (as well as methods that are defined in classes).

For *many* more details on how HOFs work, see this excerpt from my book, Functional Programming, Simplified[1]:

- How to Write Scala Functions That Take Functions as Input Parameters[2]

Exercises

The exercises for this lesson are available here[3].

[1] https://alvinalexander.com/scala/functional-programming-simplified-book
[2] https://alvinalexander.com/scala/fp-book/how-write-functions-take-function-input-parameters
[3] https://alvinalexander.com/book-exercises/LearnScala3Fast/

60

Constructs: try/catch/finally (Part 2)

Earlier in the book I showed how to use `try` and `catch` with your code, but I didn't show the `finally` clause because we needed to see a few more things first, specifically the functional error handling topic we just covered. Having seen that, we can now write a function that combines `try`/`catch`/`finally` and `Try`.

Writing a file-reading function

Knowing what we know so far, let's write a function to read a text file.

Also knowing only what I've shown so far, I'll need to use a `null` value here, but fear not, it's safely contained inside this function. Here's a heavily-documented version of the function:

```
// [1] import the code we'll be using
import scala.util.{Try, Success, Failure}
import scala.io.{BufferedSource, Source}

// [2] define our function to take a string that is the name of
// the file to be read. as shown in the functional error handling
// lesson, the function returns a Try.
def readTextFile(filename: String): Try[String] =
    // [3] declare this as a var here because we need to reference it
    // in both the 'try' and 'finally' blocks.
    var source: BufferedSource = null
    try
        // [4] do the work here to read the file. in here you assume
        // the code works as desired, and then return the final
        // result inside a Success.
        source = Source.fromFile(filename)
        val lines = for line <- source yield line
      Success(lines.mkString)  // this converts 'lines' to a String
    catch
```

```
      // [5] here you catch any possible errors. i use the Throwable
      // type because it is the parent of all Java exception classes,
        // so by referencing it, i can be sure i catch all possible
        // exceptions:
        case t: Throwable => Failure(t)
    finally
        // [6] here you want to make sure you close the 'source'.
        // it will be null if 'Source.fromFile' threw an exception,
        // but it won't be null if everything worked fine.
        if source != null then source.close()
```

Those comments describe the code pretty well, so I won't add any more discussion here. Here's the same function without all those comments:

```
import scala.util.{Try, Success, Failure}
import scala.io.{BufferedSource, Source}

def readTextFile(filename: String): Try[String] =
    var source: BufferedSource = null
    try
        source = Source.fromFile(filename)
        val lines = for line <- source yield line
        Success(lines.mkString)
    catch
        case t: Throwable => Failure(t)
    finally
        if source != null then source.close
```

Given that function, this code shows how to call it and handle its result with a `match` expression:

```
val maybeContents = readTextFile("/etc/passwd")
maybeContents match
    case Success(contents)  => println(contents)
   case Failure(exception) => System.err.println(exception.getMessage)
```

In that code, because `readTextFile` returns the `Try` type, I handle that result with the `match` expression shown and its two cases. In the `Success` case we got what we wanted — the contents of the file — so we print them out. In the `Failure` case something went wrong and we got an exception, so we print its message with the `getMessage` method that's available on all exceptions.

Discussion

In this lesson I snuck in a `null` value, but as I mentioned, it's safely contained inside this function. As you learn more about FP you'll see that there are ways to work without `null` values, but when you need to reference a variable inside both the `try` and `finally` clauses, this is how you need to do it.

It may also help to know that I originally tried to include this `finally` lesson in the first `try`/`catch`/`finally` lesson, but then I realized how much you'd need to know before using `finally`! As a result I moved it here, which let me write a complete file-reading function, and I hope this was a helpful approach.

Lastly, this code is included in the source code repository for this book, so you can check that code out and experiment with it as desired on your own computer.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

61
Domain Modeling (DM)

Next, it's time to switch gears and talk about domain modeling (DM) in Scala 3.

In both FP and OOP, programmers model the world around us. For instance, we see things like pizza, people, cars, etc., and we model these things, typically with a construct like a *class*. Language creators also found that *interfaces* are a good thing, so most languages have some sort of interface construct.

Because Scala supports both FP and OOP, it takes domain modeling a step further and provides ways to model our world in both domains. The approaches are a little different, so I'll describe each approach separately.

> Speaking of modeling, did you know: The C++ programming language was originally referred to as, "C With Classes."

Tools for OOP domain modeling

Because most programmers are familiar with OOP, I'll start with it first.

When programming in an OOP style, you'll primarily use these Scala constructs:

- Traits
- Enums
- Classes
- Objects

In the OOP style you'll use these constructs in these ways:

- **Traits:** In an OOP style you'll primarily use traits as interfaces. If you've used Java, you can use Scala traits just like interfaces in Java 8 and newer, with both abstract and concrete members. Classes will later be used to implement these traits.
- **Enums:** You'll primarily use enums to create simple sets of constants, like the positions of a display (top, bottom, left, and right).

- **Classes:** In OOP you'll primarily use plain classes, not `case` classes (which are discussed below). You'll also define their constructor parameters as `var` fields so they can be mutated. They'll also contain methods based on those mutable fields. You'll override the default accessor and mutators methods (getters and setters) as needed.
- **Objects:** You'll primarily use the `object` construct as a way to create the equivalent of static methods in Java, like a `StringUtils` object that contains "static" methods that operate on strings.

All of these tools (constructs) are described in the lessons that follow.

Tools for FP domain modeling

When programming in an FP style, you'll primarily use these constructs:

- Traits
- Enums
- Case classes
- Objects

In the FP style you'll use these constructs as follows:

- **Traits:** Traits are used to create small, logically-grouped units of behavior. The behaviors are typically written as `def` methods, and they are also implemented as pure functions. These traits will later be combined into concrete objects.
- **Enums:** Enums are used to create algebraic data types (ADTs). Don't let this term scare you, ADTs are actually a simple concept with a scary name.
- **Case classes:** You'll use `case` classes to create objects with immutable fields (known as *immutable records* in some languages, such as the `record` type in Java 14). Case classes are like plain classes, but have additional, standard methods baked into them to make FP easier.
- **Objects:** In FP you'll typically use objects as a way to make one or more traits "real," in a process that's technically known as *reification*.

Exercises

The exercises for this lesson are available here[1].

More information

In this book I won't discuss some of these FP concepts, such as the last one (reification). However, if you're interested in them, I describe them in detail in these two books:

- Scala Cookbook, 2nd Edition[2]
- Functional Programming, Simplified[3]

The reason I don't discuss them much here is because FP is a big topic, and this is a relatively small, "introduction to Scala" book.

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/
[2] https://amzn.to/3du1pMR
[3] https://alvinalexander.com/scala/functional-programming-simplified-book

62

DM: Classes, Part 1: Constructors

As mentioned, Scala gives you a number of ways to model the world around you, and a basic way that's found in every OOP language is the *class*. As with other OOP languages, a Scala class is a template or blueprint for:

- Creating objects (instances of classes) that have a particular data structure
- Providing initial values for its internal state (in its constructor)
- Providing behaviors through member methods

In this lesson we'll begin looking at the first two points, and then we'll look at the third point in the lessons that follow.

> When you use the book's examples in the Github repository, you'll see that Scala class names do not have to match the name of the file that they are in. (I mention this because in Java, public class names must match the file name.)

The class constructor

When you model things around you, such as a person, you'll find that everything you model has *attributes*. For example, a person has a name, age, height, weight, gender, and other attributes.

When you define a class you typically specify the most important of these attributes as *constructor parameters*. For example, in the physical world a person has at least a first and last name, and these are both required, so we declare those parameters in the constructor:

```
class Person(var firstName: String, var lastName: String)
```

When a class starts having many constructor parameters I write it on multiple lines, like this:

```
class Person(
    var firstName: String,
    var lastName: String
)
```

With either of those styles, in Scala 3 you create a new instance of a `Person` like this:

```
val john = Person("John", "Doe")
val mary = Person("Mary", "Doe")
```

This is what I mean by a blueprint or template: you can literally create trillions of unique *instances* of a class from the blueprint. Each active instance, such as `john` or `mary`, is then considered a running object.

Accessing the constructor parameters

Here's another instance of a `Person`:

```
val p = Person("Reginald", "Dwight")
```

Given that instance, you can access its two constructor parameter fields like this:

```
println(p.firstName)   // prints "Reginald"
println(p.lastName)    // prints "Dwight"
```

Because I declared both of the parameters as `var` fields, you can also modify them:

```
p.firstName = "Elton"
p.lastName = "John"
```

When you print those fields again, you'll see that they have the new values:

```
println(p.firstName)   // prints "Elton"
println(p.lastName)    // prints "John"
```

Defining constructor parameters as `var` fields is common in the OOP style.

var vs val

You can also declare those two fields as `val` parameters instead:

```
class Person(
    val firstName: String,
    val lastName: String
)
```

When you use `val` for the fields they are read-only — meaning that you can access them but not update them — as shown here:

```
// create a new Person
val p = Person("Reginald", "Dwight")

// print the fields
println(p.firstName)    // prints "Reginald"
println(p.lastName)     // prints "Dwight"

// errors
p.firstName = "Elton"   // ERROR: Reassignment to val firstName
p.lastName = "John"     // ERROR: Reassignment to val lastName
```

In the real world the way programmers define class parameters goes like this:

- When programming in an OOP style, fields are typically mutable, so constructor parameters are declared as `var`
- When programming in an FP style, fields are always immutable, so constructor parameters are declared as `val`

Because this book doesn't get deep into FP, we'll define most fields as `var` so they can be mutated (updated).

val, var, and nothing

You'll almost always want to define constructor parameters as either `val` or `var`. You *can* define parameters without `val` or `var`, but I generally don't do that. Leaving those keywords off constructor parameters creates a field that is accessible inside the class, but not by other classes, as you can see in this REPL example:

```
scala> class Dog(name: String)
// defined class Dog

scala> val d = Dog("Rover")
val d: Dog = Dog@2bd4780c

scala> d.name
-- Error: ----------------------------------------------------
1 |d.name
  |^^^^^^
  |value name cannot be accessed as a member of (d : Dog) from
  |module class rs$line$8$.
1 error found
```

and this REPL example, where I create a method inside the class named `getName`:

```
scala> class Dog(name: String):
     |     def getName = name
// defined class Dog

scala> val d = Dog("Rover")
val d: Dog = Dog@5814b4fb

scala> d.getName
val res1: String = Rover
```

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

63

DM: Classes, Part 2: Adding Members And "Getter" Methods

Now that you've seen a class that takes constructor parameters, let's look at an OOP-style class that also has private fields and methods. When they're defined inside a class, fields and methods are called *class members* or just *members*.

First, this class is similar to our last example, but it includes a method named `fullName` that returns the `firstName` and `lastName` fields together in a `String`:

```
class Employee(var firstName: String, var lastName: String):
    def fullName: String = s"$firstName $lastName"
end Employee
```

You create an instance of this `Employee` class just like we did with the `Person` class:

```
val e = Employee("Reginald", "Dwight")

println(e.firstName)   // prints "Reginald"
println(e.lastName)    // prints "Dwight"

e.firstName = "Elton"
e.lastName = "John"
```

The new thing here is that because it also has the `fullName` method, you can invoke it on any instance of the class:

```
println(e.fullName)   // prints "Elton John"
```

This is how *public* methods in a class work. As shown, you don't need to add a "public" keyword to the method; it's public by default. Now you just add as many methods to a class as you need to model the thing you're modeling.

A private field and getter method

To show how things might be done on larger projects, let's add a *private* field to the class that's not in the constructor, and a public method to go along with it:

```
class Employee(var firstName: String, var lastName: String):
    private var _salary = 0d
    def salary: Double = _salary
    def fullName: String = s"$firstName $lastName"
end Employee
```

In this example, `_salary` is a new private mutable field, and because its value is `0d`, the field is implicitly defined as a `Double` with that initial value. You can also explicitly declare the data type for `_salary` like this:

```
private var _salary: Double = 0d
```

but that's not really necessary (or helpful) in this case.

I also defined a public "getter" method named `salary` that returns the employee's salary. In other languages like Java it's recommended that a getter method be named `getSalary`, but Scala doesn't have that standard. Scala believes more in the Uniform Access Principle[1], where in the future, `salary` can be the name of a method or a field. The main idea is that to the end user of your class, that detail shouldn't matter to them.

Here's how this class can be used:

```
val e = Employee("John", "Doe")

// print the constructor parameter fields
println(s"firstName = ${e.firstName}, lastName = ${e.lastName}")

// print the salary field
println(s"Salary = ${e.salary}")
```

[1] https://en.wikipedia.org/wiki/Uniform_access_principle

When you run that code it prints this output:

```
firstName = John, lastName = Doe
Salary = 0.0
```

When you want to create a class that has private fields and public getter methods that correspond to those fields, this is the approach you take. (Though you don't have to use the naming approach I just showed; use whatever works best for you.)

Note: Don't use Double for currency

Before moving on, I should note that in the real world you shouldn't use the `Double` class for storing currency. It's better to use the `BigDecimal` class or a currency library. Some people also use `Long` instead.

The issue here is that `Double` on the JVM is susceptible to rounding errors that don't work well with currency, as you can easily demonstrate in the REPL:

```
scala> 0.10 + 0.20
val res0: Double = 0.30000000000000004
```

Exercises

The exercises for this lesson are available here[2].

[2] https://alvinalexander.com/book-exercises/LearnScala3Fast/

64

DM: Classes, Part 3: Adding "Setter" Methods

Because the process of creating "setter" methods takes a little more explanation, I've broken it out into a separate lesson here.

Picking up where we left off with the previous example, so far you can access the employee's salary, but you can't modify it, so let's fix that problem by adding a "setter" method:

```
class Employee(var firstName: String, var lastName: String):
    private var _salary = 0d
    def salary: Double = _salary
    // the new setter method:
    def salary_=(newSalary: Double): Unit =
        _salary = newSalary
    def fullName: String = s"$firstName $lastName"
end Employee
```

With this code the employee's salary can now be changed as well as accessed:

```
val e = Employee("John", "Doe")

// update the salary field
e.salary = 1_000_000d

println(s"firstName = ${e.firstName}, lastName = ${e.lastName}")
println(s"Salary = ${e.salary}")
```

When run, that code results in this output:

```
firstName = John, lastName = Doe
Salary = 1000000.0
```

The setter syntax

The setter syntax is a little unusual, so let's look at it again:

```
def salary_=(newSalary: Double): Unit =
    _salary = newSalary
```

To me, the quirky thing about this is that the method has the name `salary_=`. While it's a bit unusual, it's a way of saying, "I want to override the = method for the `salary` field, to allow me to write code like this":

```
e.salary = 1_000_000d
```

As I look at that `salary_=` syntax today, it's probably one of the most unusual things about Scala, so if you get comfortable with that, everything else should be much more straightforward.

The complete class

Here's a documented version of this class that explains things a little more:

```
/**
 * Employee is a class that takes two constructor parameters,
 * and both parameters are mutable.
 */
class Employee(var firstName: String, var lastName: String):

    // This is a private, mutable field that has the Double type.
    // Note that the field is named _salary, but it could be named
    // theSalary, or any other meaningful name. Also, because the
    // field is private, it can't be accessed outside the class.
    private var _salary = 0d

    // This is a "getter" method for the _salary field. Users
    // will call this method to get the current salary value.
    def salary: Double = _salary

    // This is the "setter" method for the _salary field. Users
    // will call this method to update the salary. Note that it
    // can be declared on one line, but I use two lines here to
    // make it more clear.
```

```
    def salary_=(newSalary: Double): Unit =
        _salary = newSalary

    // This method returns the full name of the Employee as a
    // String.
    def fullName: String = s"$firstName $lastName"
end Employee
```

Now you can use that class as it's shown in all the previous examples.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

65

DM: Classes: Auxiliary Constructors and Default Parameter Values

In addition to having a primary constructor, a class can have additional auxiliary constructors. This lets you create class instances in multiple ways.

That being said, you'll rarely need this feature! As you'll see in the next lesson, just like functions, constructor parameters can have default values, and this often eliminates the need for auxiliary constructors.

Creating auxiliary constructors

This code shows how to create a primary class constructor as well as two auxiliary constructors:

```
// [0] this is the primary constructor
class Person(var firstName: String, var lastName: String):

    // [1] a one-arg auxiliary constructor
    def this(firstName: String) =
        this(firstName, "<unknown>")

    // [2] a zero-arg auxiliary constructor
    def this() =
        this("<unknown>", "<unknown>")

    override def toString = s"$firstName $lastName"

end Person
```

I define these auxiliary constructors on multiple lines to make them clear, but because they are small, each can be on just one line.

In addition to those auxiliary constructors I also added a `toString` method to the class so you can see the values that are created. The `toString` method is invoked in situations such as printing an object.

With these three constructors you can create a new `Person` instance in the following ways:

```
Person("Alvin", "Alexander")   // Alvin Alexander
Person("Alvin")                // Alvin <unknown>
Person()                       // <unknown> <unknown>
```

You can create as many auxiliary constructors as you need to solve the problem at hand. The main key to know is that an auxiliary constructor must call a constructor that has been defined before it.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

66

DM: Classes: Default Constructor Parameters

While being able to create auxiliary constructors is a useful feature, it turns out that it isn't needed that often in Scala because just as with function parameters, constructor parameters can have default values. If I rewrite the primary constructor from the previous lesson like this:

```
class Person(
    var firstName: String = "<unknown>",
    var lastName: String = "<unknown>"
):
    override def toString = s"$firstName $lastName"
```

there is no need for auxiliary constructors. If you provide the two parameters when creating a new instance of the class, your values are used; otherwise the default values are used, just as with the auxiliary constructors:

```
// same output as before
Person("Alvin", "Alexander")   // Alvin Alexander
Person("Alvin")                // Alvin <unknown>
Person()                       // <unknown> <unknown>
```

Default constructor parameter values often eliminate the need for auxiliary constructors.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

67
DM: Enums

An enumeration (or "enum") is used to create a type that consists of a set of named values. Scala 3 enums are used when you want to model a set of constants in the world, such as directions (north, south, east, west), positions on a display (top, bottom, left, right), toppings on a pizza, and other finite sets of values.

For instance, this is how you define the suits in a deck of cards:

```
enum Suit:
    case Clubs, Diamonds, Hearts, Spades
```

and here's some code that uses this enum:

```
@main def enumTest =

    // import the enum cases
    import Suit.*

    // use those cases in your code like any other data type
    def printEnum(suit: Suit): Unit = suit match
        case Clubs    => println("clubs")
        case Diamonds => println("diamonds")
        case Hearts   => println("hearts")
        case Spades   => println("spades")

    // print the values
    printEnum(Clubs)       // clubs
    printEnum(Diamonds)    // diamonds
    printEnum(Hearts)      // hearts
    printEnum(Spades)      // spades
```

While I generally use enums with `match` expressions like this, if for some reason you ever need to loop over the values in an enum you can use this approach:

```
for s <- Suit.values do println(s)
```

That code results in this output:

```
Clubs
Diamonds
Hearts
Spades
```

A more complicated example

In a more complicated example, here are some enums you might create when modeling a pizza store application:

```
enum CrustSize:
   case Small, Medium, Large

enum CrustType:
   case Thin, Thick, Regular

enum Topping:
   case Cheese, Pepperoni, Mushrooms, GreenPeppers, Olives
```

To use these enums, first import their instances, and then use them in expressions and parameters, just like a class, trait, or other type:

```
import CrustSize.*
import CrustType.*
import Topping.*

// later code uses the enums like this:
if currentCrustSize == Small then ...

// and this:
currentCrustSize match
    case Small => ...
    case Medium => ...
    case Large => ...
```

```
// and this:
class Pizza(
    var crustSize: CrustSize,
    var crustType: CrustType,
    val toppings: ArrayBuffer[Topping]
)
```

Again, enums can be used like other Scala data types.

One thing to note about enums is that they contain a *set* of values, meaning that just like the `Set` class, all of the instances (cases) in an `enum` must be unique.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

68

DM: Enums: More Details

Like classes, enums can also take parameters and have members, such as fields and methods. This example shows how you can model HTTP status codes using an enum that has a parameter named `code`:

```
enum HttpResponse(val code: Int):
    case Ok extends HttpResponse(200)
    case MovedPermanently extends HttpResponse(301)
    case InternalServerError extends HttpResponse(500)
    // more cases here ...
```

As shown, each `case` extends `HttpResponse`, as shown in the underlined code here:

```
enum HttpResponse(val code: Int):

    // the cases call the constructor:
    case Ok extends HttpResponse(200)
            ------------------------
```

Given that code, this is how you access the `code` field on the enum instances:

```
import HttpResponse.*

Ok.code                    // 200
MovedPermanently.code      // 301
InternalServerError.code   // 500
```

Enums can have members

Scala 3 takes the `enum` implementation pretty far, so not only can enums have parameters, they can also have members, such as fields and methods.

The `Planet` example on the Scala 3 enum page[1] demonstrates how to implement `enum` parameters, fields, and methods:

```
enum Planet(mass: Double, radius: Double):
   private final val G = 6.67300E-11
   def surfaceGravity = G * mass / (radius * radius)
   def surfaceWeight(otherMass: Double) = otherMass * surfaceGravity

   case Mercury extends Planet(3.303e+23, 2.4397e6)
   case Earth   extends Planet(5.976e+24, 6.37814e6)
   // more planets here ...
```

When to use enums

Because enums have all of these capabilities, a good question is, "When should I use an `enum` instead of a `class`, `trait`, or `object`?"

The key to remember is that enums are typically used to model a small, *finite* set of possible values, such as the `Planet` example, where there are only eight (or nine) planets in our solar system (depending on who's counting).

Also, the individual cases in an `enum` can't have unique methods. For instance, you might want the `Earth` instance to have a method to measure carbon dioxide levels, but you can't do that for just the `Earth` instance; it would have to be declared like the `surfaceWeight` method, and then be available for all planets.

Exercises

The exercises for this lesson are available here[2].

More information

Enums can also be used to model Algebraic Data Types (ADTs), which I demonstrate in the Scala Cookbook, 2nd Edition[3].

[1] https://https://dotty.epfl.ch/docs/reference/enums/enums.html
[2] https://alvinalexander.com/book-exercises/LearnScala3Fast/
[3] https://amzn.to/3du1pMR

69

DM: Traits: Using As Interfaces

The Scala `trait` is a powerful tool for domain modeling. You can use traits as simple interfaces and give them concrete methods as well. With both approaches you can compose classes by implementing one or more traits.

The simplest way to use a trait is as an interface. An *interface* is a construct that only has abstract members. As an example, imagine that you want to write some code to model any animal that has a tail, like a dog or cat. A first thing you might think is that tails can wag, so you define a trait like this, with two method signatures and no method body:

```
trait HasTail:
    def startTail(): Unit
    def stopTail(): Unit
```

In that trait both methods are *abstract* because they don't have a body. We just declare each method's parameters and its return type. Whoever implements this trait will need to provide behavior for those methods. (In these examples the methods don't take any parameters, but when you need them, just specify them as you do when writing any other function or method.)

Moving on, another thing about a dog is that it can speak, so you would model this ability as another trait:

```
trait CanSpeak:
    def speak(): Unit
```

Given these traits, you can now create a `Dog` class by (a) extending the traits and (b) providing behavior for their abstract methods:

```
class Dog extends HasTail, CanSpeak:
    def startTail() = println("Tail is wagging")
    def stopTail() = println("Tail is stopped")
    def speak() = println("Woof")
```

Given that implementation, now you can create a new instance of a `Dog` and call all of those methods on it:

```
val d = Dog()
d.speak()        // Woof
d.startTail()    // Tail is wagging
d.stopTail()     // Tail is stopped
```

In summary, this is how you use traits as interfaces:

- Define the traits with abstract methods
- Create a class that extends the traits and implements those methods

The benefit of using traits as interfaces

If you haven't used this approach before, a benefit of using interfaces like this is that other code can refer to these abstract interfaces rather than just concrete classes. This makes your code more flexible.

For instance, I *could* write a method that takes a `Dog` variable, but a more powerful approach is to write a method that takes a variable of the type `CanSpeak`:

```
def saySomething(speaker: CanSpeak) = speaker.speak()
```

Now if I also have a `Cat` class that implements `CanSpeak`:

```
class Cat extends CanSpeak:
    def speak() = println("Meow")
```

the `saySomething` method can take a `Dog`, a `Cat`, or anything else that implements `CanSpeak`:

```
saySomething(Dog())    // Woof
saySomething(Cat())    // Meow
```

This is a benefit of using traits as interfaces to compose classes. In another use, methods can also return the `CanSpeak` type:

```
def getSpeaker(s: String): CanSpeak = ???
```

Now any method that calls `getSpeaker` only has to concern themselves with the capabilities of the `CanSpeak` trait.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

70

DM: Traits: Adding Behaviors

Traits can also have methods that have implemented (concrete) behaviors. To demonstrate this, I can rewrite the previous example like this:

```
trait HasTail:
    def startTail() = println("Tail is started")
    def stopTail() = println("Tail is stopped")

trait CanSpeak:
    def speak(): Unit
```

We say that the first trait has *concrete* methods, because the methods have a body. The second trait only has an abstract method, which does not have a body. So in this case I can create a `Dog` class like this:

```
class Dog extends HasTail, CanSpeak:
    def speak() = println("Woof")
```

This class uses the `startTail` and `stopTail` methods from the `HasTail` trait, and implements the `speak` method of the `CanSpeak` trait.

You can override the default methods, so I can also create a `Cat` class like this:

```
class Cat extends HasTail, CanSpeak:
    override def startTail() = println("Yeah, I'm not doing that")
    override def stopTail() = println("Never started")
    def speak() = println("Meow")
```

Notice in this case that I define all three methods in the `Cat` class. The `startTail` and `stopTail` methods override the methods from the `HasTail` trait, and the `speak` method implements the `speak` method of the `CanSpeak` trait.

This is how those classes work:

```
val d = Dog()
d.startTail()    // Tail is started
d.stopTail()     // Tail is stopped
d.speak()        // Woof

val c = Cat()
c.startTail()    // Yeah, I'm not doing that
c.stopTail()     // Never started
c.speak()        // Meow
```

As shown:

- Traits can have abstract methods
- Traits can have concrete methods
- Classes that extend traits can implement abstract methods and either (a) use the concrete methods as they are, or (b) override them

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

71

DM: Objects, Part 1: Singletons

A Scala `object` is a class that has *exactly one instance*. In some languages these are referred to as Singleton objects, or Singletons.

> If you're familiar with Java, the end result is similar to using `static` members in Java classes.

Somewhat similar to the concept of providing package names for your code, objects let you group methods and fields within a namespace. You'll see what this means in a few moments.

Three important things

There are three important things to know about objects, and I'll demonstrate those in this lesson and the two that follow:

- Singleton objects
- Companion objects
- `apply` methods in companion objects

Creating a Singleton object

When you're dealing with the physical world you might create a Singleton object to represent something you want only one instance of, such as a keyboard, mouse, or perhaps if you have a single cash register in a pizza restaurant:

```
object CashRegister:
    def open() = println("opened")
    def close() = println("closed")
```

Because `CashRegister` is defined as an object, there can only ever be one instance of it. Therefore, you don't attempt to create a new instance of it. Instead, you just access its members, like this:

```
@main def objectTest =
    CashRegister.open()
    CashRegister.close()
```

A more complex Singleton

If you wanted to track the number of times the register drawer is opened and closed, you could add additional code to the `object` like this:

```
object CashRegister:

    // two private counters that cannot be accessed directly
    private var numOpens = 0
    private var numCloses = 0

    // two methods related to physically opening and closing
    // the register drawer
    def open() =
        // in the real world, here you would do whatever
        // is needed to open the register drawer
        numOpens += 1
    def close() =
        // assume that you got a signal here that the register
        // drawer was closed
        numCloses += 1

    // "getter" methods for the private variables
    def getNumberOfOpens = numOpens
    def getNumberOfCloses = numCloses
```

This shows how that code works:

```
CashRegister.open()
CashRegister.close()
CashRegister.getNumberOfOpens     // 1
CashRegister.getNumberOfCloses    // 1
```

You don't have two make those two `Int` fields private; I just do that to show a new technique. I also use the "get" naming scheme, though again, in Scala you can name these methods whatever you like.

Utility classes

A first possible use of objects is to create what I call "utility" methods that you want to group together. For example, in every programming language I have ever used I end up creating:

- File utilities
- String utilities
- Network utilities

In Scala these become objects that I name `FileUtils`, `StringUtil`, and `NetworkUtils`. For example, here are a few of my Scala 3 string utilities:

```
object StringUtils:

    // return true if the string is null or empty
    def isNullOrEmpty(s: String): Boolean =
        if s==null || s.trim.equals("") then true else false

    // convert "big belly burger" to "Big Belly Burger"
    def capitalizeAllWords(s: String): String =
        s.split(" ")
         .map(_.capitalize)
         .mkString(" ")
```

Again, because these methods are defined in an `object` (instead of a `class`), they can be called directly on the object:

```
StringUtils.isNullOrEmpty("")
    // result: true

StringUtils.capitalizeAllWords("big belly burger")
    // result: Big Belly Burger
```

When you're using many methods from an object, it can be easier to import all the members from the object, so you don't have to precede each method with the object's name:

```
import StringUtils.*

isNullOrEmpty("")
capitalizeAllWords("big belly burger")
```

With modern IDEs you can easily track down where `isNullOrEmpty` and `capitalizeAllWords` come from, so your code can be shorter and easier to read when you write it like this.

This example shows what I meant by the earlier "namespace" comment: objects give you a wrapper or container in which you can put all these related methods. The code works almost exactly as though all these methods are in a package named `StringUtils`.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

72

DM: Objects, Part 2: Companion Objects

Scala has this cool and unique thing called a *companion object*. While it sounds fancy, it's just an `object` that is declared in the same file as a `class` that has the same name. When this approach is used, the class is also the object's companion. A cool thing about a companion class and object is that they can access the private methods and fields (members) of their companion. (It's like they're married and they share a joint bank account.)

The concept

While this first example isn't terribly useful, when this class and object are in the same file, they're companions of each other:

```
// probably in a file named Person.scala
class Person(var name: String)
object Person
```

A first use of companion objects

One way companion objects are used is as a place to store methods and values that are not specific to instances of the companion class. (Remember that there can be many instances of a `class`, but only one instance of an `object`.)

Here's a more useful example of a companion object:

```
// companion class
class Pizza (var crustType: String):
    override def toString = s"Crust type is $crustType"

// (continued on the next page)
```

```
// companion object
object Pizza:
    // two fields
    val CRUST_TYPE_THIN = "THIN"
    val CRUST_TYPE_THICK = "THICK"

    // one method
    def calculatePrice(p: Pizza): Double =
        // put a fancy pizza-pricing algorithm here
        0.0
```

> NOTE: I don't recommend defining constants as strings, but I do that here to keep this example as simple as possible.

Now when you want to create and use a new instance of a `Pizza` you can do it like this:

```
@main def pizzaTest =
    val p = Pizza(Pizza.CRUST_TYPE_THICK)
    println(p)
    println(Pizza.calculatePrice(p))
```

The output of that code is:

```
Crust type is THICK
0.0
```

Summary

From what you can see so far, the features and benefits of a companion object are:

- You put the instance members in the `class`
- You put the "static" members in the `object`
- The class and object can access each other's `private` fields and methods
- Users of your code can find both of those in the same location (i.e., under the `Pizza` namespace, in this example)

There's also another benefit that I'll show in the next lesson.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

73

DM: Objects, Part 3: apply Methods In Companion Objects

Another cool feature of companion objects is that you can use them as a way to create constructors for your class. Technically they're not constructors, but in the end they work just like constructors. If you've heard of the *Factory Method* in OOP, putting this type of code in an `object` is similar to that.

The process

The way you do this is you create one or more methods named `apply` in the companion object. Each `apply` method creates a different way that instances of your class can be created.

For example, assuming that you want to create constructors for a `Person` class:

1. Define a `Person` class and `Person` object in the same file
2. Make the `Person` class constructor private
3. Define one or more `apply` methods in the `object` to serve as factory methods (which work just like constructors)

Example 1

For the first two steps, create the `class` and `object` in the same file, and make the `Person` class constructor private:

```
// note the 'private' keyword here
class Person private(val name: String):
    // define any instance members you need here

object Person:
    // define any static members you need here
```

Then create one or more `apply` methods in the companion object:

```
class Person private(val name: String):
    override def toString = name

object Person:
    // the "constructor"
    def apply(name: String): Person = new Person(name)
```

Given this code, you can now create new `Person` instances as shown in these examples:

```
val bert = Person("Bert")
val a = List(Person("Bert"), Person("Ernie"))
```

In Scala 2 a benefit of this approach was that it eliminated the need for the `new` keyword when creating new instances of a class. But because `new` isn't needed in most situations in Scala 3, you may want to use this technique because you prefer this factory approach. In fact, I've seen that several experienced developers like to create their class "constructors" this way. Also note that the `new` keyword is required in the `apply` method to create a new `Person` instance.

Discussion

"But how does this work," you might ask. And that's an excellent question!

An `apply` method defined in a companion object is treated specially by the Scala compiler. Essentially, there's a little syntactic sugar baked into the compiler so that when it sees this code:

```
val p = Person("Fred Flintstone")
```

one of the first things it does is to look around and see if it can find an `apply` method in a companion object. If it does find a matching `apply` method in a companion object — i.e., an `apply` method that takes one `String` parameter — the compiler turns that code into this code:

```
val p = Person.apply("Fred Flintstone")
```

You can demonstrate this for yourself by putting this code into the Scala REPL.

Because of this, `apply` is often referred to as a factory method or perhaps a "builder." Technically it's not a constructor, but it works just like one.

Example 2: More factory methods!

When you want to use this technique to provide multiple ways to build a class, define multiple `apply` methods with the desired signatures:

```
class Person private(var name: String, var age: Int):
    override def toString = s"$name is $age years old"

object Person:
    // three ways to build a Person
    def apply(): Person = new Person("", 0)
    def apply(name: String): Person = new Person(name, 0)
   def apply(name: String, age: Int): Person = new Person(name, age)
```

That code creates three ways to build new `Person` instances:

```
println(Person())              //  is 0 years old
println(Person("Bert"))        // Bert is 0 years old
println(Person("Ernie", 22))   // Ernie is 22 years old
```

Because `apply` is just a function, you can implement it however you see fit. For instance, you can construct a `Person` from a two-element tuple, or even construct a `Seq[Person]` from a variable number of two-element tuples:

```
object Person:
    def apply(t: (String, Int)): Person = new Person(t(0), t(1))
    def apply(ts: (String, Int)*): Seq[Person] =
        for t <- ts yield new Person(t(0), t(1))
```

> TIP: Remember that the `*` character after a method parameter makes the field a *varargs* parameter.

Those two `apply` methods are used like this:

```
// create a person from a tuple
val john = Person(("John", 30))
```

```
// create multiple people using a variable number of tuples
val peeps = Person(
    ("Kenny", 33),
    ("Julia", 31)
)
```

As shown, this is a great way to provide "builders" to create new instances of a class.

Lastly, I haven't mentioned it before, but the underlined code here defines a parameter named `t` that is a two-element tuple:

```
def apply(t: (String, Int)): Person = ???
          ----------------
```

Technically this is a two-element tuple of the type `(String, Int)`, meaning that it holds a `String` and an `Int`. As a result, I can pass the `apply` method tuple values like these:

```
("Kenny", 33),
("Julia", 31)
```

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

74

DM: An OOP Domain Modeling Example

This lesson is a little longer than all the others because it demonstrates an OOP domain modeling example. For this example, imagine that you're working on the server side of a web application for a pizza store. As you think about this domain, you'll see that a pizza store typically deals with these concerns:

- Pizza
- Toppings
- Crust
- Breadsticks
- Customers
- Orders

Those are some obvious nouns that we quickly think of when considering a pizza store, and nouns usually become classes (or class-like constructs) in DM. So let's start creating code for them. To keep this relatively short, I'll work through the data types we need in a slightly different order.

Toppings

Because there are a finite number of possible toppings, a good way to model them is with the Scala 3 `enum`:

```
enum Topping:
    case Cheese, Pepperoni, Sausage, Mushrooms, Onions
```

Crust

As it turns out, there are also small, finite numbers of possible crust sizes and thickness, so they make good enums as well:

```
enum CrustSize:
    case Small, Medium, Large

enum CrustType:
    case Regular, Thin, Thick
```

Pizza

If I was teaching a class, I would have started by sketching a `Pizza` class like this:

```
class Pizza( ...
```

At this point I'd ask, "What constructor parameters do you think a `Pizza` should take?" I think you'd probably answer that you'd want to be able to set the crust information as well as any initial toppings, so that's why I show those enums first. Now I can show that I would start to create a `Pizza` class like this:

```
import collection.mutable.ArrayBuffer

class Pizza(
    var crustSize: CrustSize = Medium,
    var crustType: CrustType = Regular,
    val toppings: ArrayBuffer[Topping]
)
```

A few notes about that code:

- Since we're coding in an OOP style, I define `crustSize` and `crustType` as `var` fields.
- Because `toppings` is an `ArrayBuffer`, which is mutable, I define it as a `val` field (because there's no need for the collection type and variable type to both be mutable).

So far the `Pizza` class has no body to it, but we'll come up with some methods for it once we start thinking about `Pizza` class behaviors (i.e., verbs).

Pizza behaviors

As you think about the lifecycle of a pizza in a software application, you'll realize that customers may add and remove toppings, and change the crust size and thickness. So if you imagine the code you'd like to write, it might look like this:

```
import Topping.*
import CrustSize.*
import CrustType.*

// we can already do this
val p = Pizza(Medium, Regular, ArrayBuffer(Cheese))

// and this works, too
p.crustSize = Large
p.crustType = Thin
p.toppings

// but these methods DO NOT exist yet
p.addTopping(Pepperoni)
p.addToppings(Pepperoni, Mushrooms)
p.removeAllToppings()
```

Because you like the way those last three method calls look, you decide to add these methods to the `Pizza` class, so it now looks like this:

```
import collection.mutable.ArrayBuffer

import Topping.*
import CrustSize.*
import CrustType.*

class Pizza(
    var crustSize: CrustSize = Medium,
    var crustType: CrustType = Regular,
    val toppings: ArrayBuffer[Topping]
):
    def addTopping(t: Topping): Unit =
        toppings += t
```

```
    def addToppings(ts: Topping*): Unit =
        toppings.appendAll(ts)

    def removeAllToppings(): Unit =
        toppings.clear()

    override def toString = s"""
        |A $crustSize pizza with a $crustType crust.
        |Toppings: ${p.toppings.mkString(", ")}""".stripMargin

end Pizza
```

Most of those methods use techniques I already described in this book. For instance, the topping-related methods just use methods of the `ArrayBuffer` class.

One thing that's new in this code is that in the `toString` method I use this code inside the interpolated string:

```
p.toppings.mkString(", ")
```

What's going on there is that `toppings` is an `ArrayBuffer`, and `mkString` is a method that's available on sequence classes, and lets you turn the sequence elements into a `String`. In this case I separate each element with the comma character, so if a `Pizza` has `Cheese` and `Pepperoni` toppings, its result looks like this:

```
p.toppings.mkString(", ")    // "Cheese, Pepperoni"
```

You can debate about what an OOP `Pizza` class API should look like, but this example shows how to implement the code I thought I'd like to see.

More data types

At this point I would continue to sketch out all the other data types I need. For instance, I know that a customer has a name, phone number, and address, so I start sketching it like this:

```
class Customer(
    var firstName: String,
    var lastName: String,
    var phone: String,
    var address: Address
)
```

This tells me that I need an `Address` type, which I also create as a class with its attributes:

```
class Address(
    var street1: String,
    var street2: Option[String],
    var city: String,
    var state: String,
    var postalCode: String
)
```

Note that the `street2` field has the `Option` type. This just means that this field is optional, because for some addresses this will have no value. So when that field *isn't* needed you define it as a `None`:

```
val address = Address(
    "123 Main Street",
    None,
    "Talkeetna",
    "AK",
    "99676"
)
```

and when it *is* needed you define it as a `Some[String]`:

```
val address = Address(
    "123 Main Street",
    Some("Unit 101"),
    "Talkeetna",
    "AK",
    "99676"
)
```

If we kept going down this road we'd continue to sketch more classes, such as the concept of an order:

```
class Order(
    val pizzas: ArrayBuffer[Pizza],
    var customer: Customer
)
```

For me this is a pretty typical approach to starting domain modeling. I usually start sketching nouns as classes, and noun-attributes as constructor parameters and class fields.

Adding behaviors

After I sketch out those known nouns and attributes I begin to think about how the classes will interact in my application. These lead me to understand the behaviors I need from them, and those behaviors become methods.

For instance, when you start thinking that you'll need to print a customer's full name on the receipt, you might decide to give the `Customer` class a new `fullName` method:

```
class Customer(
    var firstName: String,
    var lastName: String,
    var phone: String,
    var address: Address
):
    def fullName = s"$firstName $lastName"

end Customer
```

So as you think about a class and its interactions and behaviors, you think about what class should have the responsibility for each behavior. In this case I thought the `Customer` class should have the responsibility for determining the customer's full name, so I put the `fullName` method inside the `Customer` class.

Another example

That was one look at DM with just pizzas, but what happens when we start adding in other products like beverages, breadsticks, and cheesesticks? Because I know that each of these products will have a price, I might start by creating a `Product` trait that will work as an interface:

```
trait Product
    def price: BigDecimal
```

Then I'd start creating classes that extend that trait:

```
class Pizza extends Product
class Beverage extends Product
class Breadsticks extends Product
class Cheesesticks extends Product
```

After that we might have the concept of a line item that appears on a receipt, where each line item is a product (16-ounce beverage) and a quantity:

```
class LineItem(var product: Product, var quantity: Int)
```

And then a complete order might consist of a customer and a list of line items:

```
class Order(
    var customer: Customer,
    val lineItems: ArrayBuffer[LineItem]
)
```

And finally in this new modeling effort I'd begin with a simpler `Customer` class:

```
class Customer(var name: String)
```

Given these new data types, you can now imagine creating an order within the application:

```
// an empty order
val o = Order()

// add a product
o.addItem(Breadsticks)

// add another product
val p = Pizza(Large, Thick, ArrayBuffer(Cheese, Pepperoni))
o.addItem(p)

// how i'd like removing and adding to work
o.removeItem(Breadsticks)
```

```
o.addItem(Cheesesticks)
o.addItem(Beverage)

// add a customer to the order
val c = Customer("Alvin")
o.setCustomer(c)

// print the receipt
println(o.receipt)
```

When we start thinking about the interactions between classes, as in this exercise, it leads us to imagine the methods we'd like to see on those classes. For instance, this exercise tells us that we'd like the `Order` class to have these attributes:

- Its constructor should have no required parameters
- It should have `addItem`, `removeItem`, `setCustomer`, and `receipt` methods

Therefore, I'd go back to the `Order` class and start to sketch out these methods like this:

```
class Order:
    def addItem(p: Product): Unit = ???
    def removeItem(p: Product): Unit = ???
    def setCustomer(c: Customer): Unit = ???
    def receipt: String = ???
end Order
```

So that's the beginning of an `Order` class. At this point I encourage you to write the remainder of what's needed to make this code compile and run.

75

DM: Case Classes

Although this book doesn't get too much into FP, we'll take a few moments to look at the *case class*, which is like a regular Scala class, with additional features. These features are intended for programming in an FP style, but they can also be used in OOP.

As opposed to a "regular" Scala class, a `case` class generates a lot of code for you, with the following benefits:

- An `apply` method is generated in a companion object, and it acts as a factory method for us to create new instances of the class.
- Case class constructor parameters are public `val` fields by default, which means that accessor methods are generated for each constructor parameter.
- An `unapply` method is generated, which makes it easy to use `case` classes in `match` expressions. (This is huge for FP.)
- A `copy` method is generated. This lets you easily (a) copy an object while (b) updating one or more fields in the object. For instance, this lets you update someone's last name when cloning an instance of a `Person`.
- `equals` and `hashCode` methods are generated, which lets you compare objects and easily use them as keys in Maps and Sets.
- A default `toString` method is generated, which is helpful for debugging.

Example

To demonstrate a `case` class, add the word `case` before the usual `class` declaration:

```
case class Person(name: String, relation: String)
```

Also note that the parameters don't need to be declared as `val` or `var`; they are `val` by default.

Now you can create a new instance of a `Person`:

```
val christina = Person("Christina", "niece")
```

You can access the fields in the class, but can't mutate them:

```
christina.name            // "Christina"
christina.name = "Joe"    // error: reassignment to val field
```

Next, because `equals` and `hashCode` methods are generated for you, you can easily compare two instances of a class:

```
val hannah = Person("Hannah", "niece")
christina == hannah    // false
```

This is what the `toString` method does for you:

```
println(christina)     // prints "Person(Christina,niece)"
```

And this is an example of how to use the `copy` method:

```
// a new case class:
case class BaseballTeam(name: String, lastWorldSeriesWin: Int)

// create an instance of it:
val cubs1908 = BaseballTeam("Chicago Cubs", 1908)

// copy the original object while changing one of its values:
val cubs2016 = cubs1908.copy(lastWorldSeriesWin = 2016)
```

As shown, this method lets you clone an object while simultaneously updating the values in any number of its fields (i.e., you update only the fields you need to update).

Discussion

Case classes are *huge* for FP. Along with enums and traits, they are the primary way of domain modeling in FP. This is because:

- In FP nothing is mutable.
- Because the parameters aren't mutable, it's easy to generate `equals` and `hashCode` methods for them.

- In FP, rather than mutate an object, we use a process I call "update as you copy," and the `copy` method gives you what you need (as shown in the `BaseballTeam` example).
- In Scala, we use `match` expressions all the time, and although I'm skipping those details for now, just know that the `unapply` method is what lets `case` classes work so well in those expressions.

Exercises

The exercises for this lesson are available here[1].

More information

For more information on the "update as your copy" approach, see my longer tutorial, Scala/FP Idiom: Update as You Copy, Don't Mutate[2].

And for 700+ pages of additional details on FP, see my book, Functional Programming, Simplified[3].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/
[2] https://alvinalexander.com/scala/fp-book/update-as-you-copy-dont-mutate-case-classes/
[3] https://alvinalexander.com/scala/functional-programming-simplified-book

76

Constructs: match Expressions

At their most basic level, Scala's `match` expressions are like `switch` statements in other languages, but:

- They're way more powerful
- They are expressions, so they return a value
- They can match numbers, strings, traits, classes, case classes, and pretty much any data type in general
- Each `case` in a `match` expression can match multiple things

The following examples demonstrate the basic features of `match` expressions.

Example 1

Here's one example of how to use a `match` expression. First, given this enum:

```
enum Suit:
    case Clubs, Diamonds, Hearts, Spades
```

In the rest of your code you first import that enum:

```
import Suit.*
```

Now you can use those `enum` types in a `match` expression. For instance, assume that somewhere in your code you create a variable named `suit` that is bound to one of those `enum` values, like this:

```
val suit = Hearts
```

Now somewhere else in your code you can "match" whatever happens to be in the `suit` variable like this:

```
suit match
    case Clubs    => println("clubs")
    case Diamonds => println("diamonds")
    case Hearts   => println("hearts")
    case Spades   => println("spades")
```

In this example the code prints `"hearts"`, but in other instances the other cases will also be matched. Note that in this example I ignore the return value of the `match` expression, and just use it to print a string for each possible enum value, so I really use it as a statement. (More on this in a moment.)

An important point with this code is that both you and the compiler know that all possible cases are accounted for in the `match` expression, because the `enum` has only those four possible values. This is a terrific benefit of a strongly-typed language.

Example 2

When you might try to match *many* different possible values, such as every integer value in the universe, you may also want to add a wildcard case to the end of your `match` expression:

```
// assume that 'number' is an Int that's set earlier in the code:
val number = 1

// and then some time later in the code:
number match
    case 1 => println("One")
    case 2 => println("Two")
    case _ => println("Some other value")
```

In this example, the first `case` statement matches the number `1`, the second `case` statement matches the number `2`, and then the third `case` statement uses the `_` wildcard character, which means that it matches every other possible `Int` value.

Example 3

When you want to access the value of the default case on the right side of the `=>` symbol, you can't do that with the `_` character. But all you have to do is give that case a name, such as `default`:

```
// assume that 'number' is an Int that's set earlier in the code:
val number = 1

number match
    case 1 => println("One")
    case 2 => println("Two")
    case default => println(s"You gave me $default")
```

Using the name `default` in the `case` statement — instead of the _ character — lets you access it on the right side of the `=>` symbol:

```
case default => println(s"You gave me $default")
     -------                           --------
```

This name can be almost anything you want, but I often use `default` because it makes the most sense.

Example 4

In those examples I use the `match` expression for a side effect, but it's very often used to return a value. For example, given the same `Suit` enum as before, I'll create an example of how to use it with a `match` expression. First, given that same `enum` setup:

```
enum Suit:
    case Clubs, Diamonds, Hearts, Spades

import Suit.*
```

you can use `Suit` in a `match` expression to create a function:

```
def suitToString(suit: Suit): String = suit match
    case Clubs    => "clubs"
    case Diamonds => "diamonds"
    case Hearts   => "hearts"
    case Spades   => "spades"
```

That function matches all of the known `Suit` values and returns a `String` that corresponds to the `Suit` value that's passed in. Now you can call that method like this:

```
suitToString(Clubs)      // "clubs"
suitToString(Diamonds)   // "diamonds"
suitToString(Hearts)     // "hearts"
suitToString(Spades)     // "spades"
```

Discussion

These are just a few examples of how to use the `match` expression, and the lessons that follow show several more. If you're coming to Scala from a language that doesn't have a construct like this, get ready, you'll find that you use it all the time in Scala!

> TIP: If you ever hear people talking about *pattern matching* in Scala, they're talking about `match` expressions.

Exercises

The exercises for this lesson are available here[1].

[1] https://alvinalexander.com/book-exercises/LearnScala3Fast/

77

Constructs: match Expressions, More Details

`match` expressions can be used in *many* more ways. In this lesson I'll show a few more examples of different approaches so you can get a flavor of them.

Matching multiple patterns per line

One thing you can do is to match multiple patterns in each `case` statement. Given an integer `i`, this shows that technique:

```
i match
    case 1 | 3 | 5 | 7 | 9 => println("odd")
    case 2 | 4 | 6 | 8 | 10 => println("even")
    case _ => println("something else")
```

The way this code works is that if `i` is an odd number that's less than `10` the first line is matched, and if it's an even number up to `10` the second line is matched. The `|` symbol is used as an "or" symbol, just like we use in `if/then` expressions, so the first line is "1 or 3 or 5 or …"

You can do the same thing with strings and any other data type. In this example `cmd` is a `String`:

```
cmd match
    case "start" | "go" =>
        println("starting")
    case "stop" | "quit" | "exit" =>
        println("stopping")
    case _ =>
        println("doing nothing")
```

Adding 'if' statements to cases

Another thing you can do is add `if` expressions in `case` statements. Again given an integer `i`, this example matches ranges of values:

```
i match
    case a if 0 to 9 contains a =>
        println("0-9 range: " + a)
    case b if 10 to 19 contains b =>
        println("10-19 range: " + b)
    case c if 20 to 29 contains c =>
        println("20-29 range: " + c)
    case _ =>
        println("Hmmm...")
```

As indicated by the `println` statements, the first `case` statement matches the `Int` values `0` to `9`, the second one matches `10` to `19`, etc. The final `case` statement is a catch-all (default) case that catches all other values that are outside of the ranges I show.

> If you can't remember this syntax, fear not — I can't remember it either! That's why I created *many* blog posts about `match` expressions on my website[1].

Exercises

The exercises for this lesson are available here[2].

[1] https://alvinalexander.com
[2] https://alvinalexander.com/book-exercises/LearnScala3Fast/

78
Scala CLI

Until somewhere in the year 2021 I would have written this book using the Scala SDK and its `scalac` and `scala` commands to compile and run your code, respectively. But since that time the Scala CLI command[1] project has come along, and it greatly simplifies the "getting started with Scala" experience, so I'm using it in this book. My main reasons are:

- It's much easier to use, especially when you're first getting started with Scala
- It lets you easily compile and run both (a) small applications and (b) Scala scripts
- In those applications it lets you easily include third-party libraries, such as HTTP, JSON, and HTML libraries for working on the internet, and testing libraries so you can include tests in your code
- Until your project gets large, you can use it instead of a more-complicated build tool

At the time of this writing there's also an effort to make Scala CLI the official Scala command (`scala`). I don't know if that will ever come to complete fruition, but if it does, you'll be in great shape.

Because I'm using Scala CLI, I want to share a little "cheat sheet" of how to use it in this lesson.

> NOTE: At the time of this writing, Scala CLI is at version 0.1.12, so be aware that some of the following features may change over time.

[1] https://scala-cli.virtuslab.org

Starting the REPL

To start the Scala REPL, use this command:

```
$ scala-cli repl
```

Compiling and running applications

If you have an application with an `@main` method named `hello` in a file named `Hello.scala`, and you want to run that application, this is all you have to do with Scala CLI:

```
$ scala-cli Hello.scala
```

That command both *compiles* and *runs* your application in one step. If instead you want to do those in separate steps, do this:

```
$ scala-cli compile Hello.scala
$ scala-cli run hello
```

Compiling and running scripts

Scala CLI currently makes a distinction between Scala *applications*, which are in `.scala` files, and Scala *scripts*, which are in files that end with the `.sc` extension. A main difference between the two is that a script doesn't have to have an `@main` method; it can just consist of a series of statements and expressions.

To demonstrate how to run a script, here's a script named `MyScript.sc` that consists of just one line:

```
args.foreach(println)
```

With Scala CLI you run the script and give it a couple of command-line arguments like this:

```
$ scala-cli MyScript.sc -- hello world
hello
world
```

Notice that the `--` characters are required before your command-line arguments.

A self-enclosed script

If you're working on a Unix-based system, such as macOS or Linux, you can also make a script self-enclosed by starting it with the first line shown here:

```
#!/usr/bin/env -S scala-cli shebang
println("Hello, world")
```

The first line of code is the Unix and Scala CLI way to say that you want this script to execute the `scala-cli` command with the `shebang` command-line option. `scala-cli` is then given the rest of the script to interpret.

> If you haven't seen *anything* like this before, please know that Unix shell scripts start with the `#!` characters, followed by something like `/bin/sh` or `/usr/bin/env`. This is the Unix way to say, "Run this script using the interpreter that follows the `#!` characters."

Assuming that code is in a file named `HelloWorld.sc`, and you have Scala CLI installed, just make the script executable:

```
$ chmod +x HelloWorld.sc
```

and then run it like this:

```
$ ./HelloWorld.sc
```

If you want a script to handle command-line arguments, access those arguments with the special (implicitly available) variable named `args`:

```
#!/usr/bin/env -S scala-cli shebang
args.foreach(println)
```

If that script is named `PrintAll.sc`, you can make it executable and then run it like this:

```
# make it executable
$ chmod +x PrintAll.sc
```

```
# run it with command-line arguments
$ ./PrintAll.sc 1 2 3
1
2
3
```

> Notice that when you use the "shebang" line to start your script, you don't need to use the `--` symbol before your command-line arguments, but you do need to use `--` when you run a script with `scala-cli`.

Using an alias for scala-cli

If you get tired of typing `scala-cli`, you can create an alias for it on Unix systems like this:

```
$ alias sc=scala-cli
```

Now you can just type `sc` instead of `scala-cli`:

```
$ sc Hello.scala
```

'using' directives

Scala CLI has a concept of "`using` directives." These directives give you a way to control how your code will be run, including specifying what Scala version is used, and what third-party libraries your application uses.

You add these directives with this special syntax, which must be put in your script or application before any other Scala code:

```
//> using scala "3"
```

With this directive at the top of your file, when you run your application with `scala-cli` it will see this line and then interpret the rest of your code using the latest version of *Scala 3* (such as Scala 3.1.1, for example).

In addition to specifying which Scala version should be used, you can also specify which dependencies your application requires. For example, I use these two directives at the beginning of my HTTP GET and POST examples:

```
//> using scala "3"
//> using lib "com.softwaremill.sttp.client3::core::3.7.2"

// the rest of the Scala code follows ...
```

There are many more things you can do with `using` directives. See the Scala CLI website[2] for more details.

Even more

On top of these basics there's *much* more that Scala CLI can do for you. In addition to what I've shown, you can:

- Include multiple dependencies
- Declare specific Scala versions to use (such as 2.12.15 or 3.1.1)
- Run tests
- Put your code into "watch" mode, so your code is recompiled automatically whenever you make a change to it
- Create small projects
- Integrate with your IDE
- Package your applications
- More!

For all those details, see the Scala CLI website[3].

[2] https://scala-cli.virtuslab.org/
[3] https://scala-cli.virtuslab.org

79

Example: Command-Line I/O

In the scripts that follow this lesson we'll deal with input-to and output-from your code, so in this lesson I'll first show how to work with command-line I/O.

As a brief overview:

- When you want to print output to the command line (STDOUT), use these two functions:
  - `println` prints a string, followed by a newline character.
  - `print` prints a string, but does not follow it with a newline character.
- When you want to print to STDERR (such as for error messages), use `System.err.println`.
- When you want to read input from the command line, use the `scala.io.StdIn.readLine` function.

An I/O example

A basic print-and-read interaction from a Scala command-line script or application looks like this:

```
// use this 'import' statement to make the 'readLine' function
// available to your code:
import scala.io.StdIn.readLine

// i use 'print' here because i want the command-line cursor
// to stay on the same line after this string is printed:
print("What's your name? ")

// read the user's input, and then print it back to them:
val name = readLine()
println(s"You said your name is $name")
```

Here's that same code without the comments:

```
import scala.io.StdIn.readLine
print("What's your name? ")
val name = readLine()
println(s"You said your name is $name")
```

I don't do any error-checking in that code, but assuming that you name the script `CmdLine.sc`, an interaction with it starts like this:

```
$ scala-cli CmdLine.sc
What's your name? _
```

Then once I give it my name it ends up like this:

```
$ scala-cli CmdLine.sc
What's your name? Alvin
You said your name is Alvin
```

I encourage you to modify this code. For instance, if you want to validate the user input, you can check its length with `name.length`.

Prompting for name and age

There are different ways to read input from the command line, but I generally prefer to use the `readLine` function and then convert the input as desired. For instance, without doing any error-checking to make sure `age` is a legal integer, this is one way to prompt for a person's name and age:

```
import scala.io.StdIn.readLine

print("What's your name? ")
val name = readLine()

print("What's your age (in years)? ")
val ageString = readLine()
val age = ageString.toInt

println(s"Your name is $name and your age is $age")
```

If you name this file `NameAndAge.sc` you can run it like this:

```
$ scala-cli NameAndAge.sc
What's your name? Al
What's your age (in years)? 11
Your name is Al and your age is 11
```

Note that this code will blow up with an exception if `age` is not a valid integer, so for a robust script you'll want to perform some error-checking there.

If you prefer to use different methods to read in the different data types you expect, you can find those methods in the `scala.io.StdIn` object[1]. For instance, when you expect to read in an integer you can use `readInt` instead:

```
val age = scala.io.StdIn.readInt()
```

However, note that this will also throw an exception if the user enters something other than an integer value. That's one reason I normally just stick with `readLine`, so I can handle that conversion manually myself, and then handle errors at that time.

[1] https://www.scala-lang.org/api/current/scala/io/StdIn$.html

80

Example: A Command-Line Timer

Some days when I first start working I struggle to get my mind into my work, and when this happens I use the *Pomodoro technique* of setting a timer for about 20 minutes. I then work as hard as I can for that time, then the timer goes off and I take a short break. After the break I do another 20-minute sprint, etc.

To help with this process, I wrote a "timer script" in Scala. I keep the following code in a file named `timer.sc`. Here's a heavily-commented version of my `timer` script:

```
#!/usr/bin/env -S scala-cli shebang
//> using scala "3"
// those first two lines are the Scala CLI way of saying (a) run this
// as a script, and (b) use the latest version of Scala 3 to run it.

// my assumptions:
//   * this code is in a file named 'timer.sc'
//   * the file is made executable with a command like:
//     'chmod +x timer.sc'
//   * you have 'scala-cli' installed
//
// usage:
//     timer.sc minutes-before-alarm <gain-control>
//     timer.sc 10
//     timer.sc 20 -10
//     ('gain-control' should be something like -10 or -20)

// import what we need from the Java Sound library
// -----------------------------------------------------------
import javax.sound.sampled.*

// if you don't get the right number of command-line args, quit
// -----------------------------------------------------------
if args.length < 1 then showUsageAndExit()
```

```
// initialize the values from the user input. note that these
// can fail because i don't verify that they are Int values.
// ------------------------------------------------------------
val minutesToWait = args(0).toInt
val gainControl = if args.length == 2 then args(1).toInt else -30


// and the action begins ...
// ------------------------------------------------------------
println(s"Timer started. Waiting $minutesToWait minutes.\n")


// wait the desired time, sleeping one minute in between checks
// ------------------------------------------------------------
for i <- 1 to minutesToWait do
    Thread.sleep(60_000)
    println(s"time remaining: ${minutesToWait-i} ...")


// the 'for' loop ended, so play the sound twice. my sound lasts
// about 7 seconds, so i sleep that long in between plays. note
// that you will need to find your own sound file to play.
// ------------------------------------------------------------
for i <- 1 to 2 do
    playSoundfile("./gong.wav")
    Thread.sleep(7_000)


// my two functions are shown below here
// ------------------------------------------------------------

// note: i wrap several lines here to fit the book's width
def showUsageAndExit() =
    Console.err.println(
      "Usage: timer.sc minutes-before-alarm <gain-control>")
    Console.err.println("Ex:    timer.sc 10")
    Console.err.println("Ex:    timer.sc 10 -20")
    Console.err.println(
      "       'gain-control' should be something like -10 or -20")
    System.exit(1)
```

```
/**
 * This is some "Java Audio" specific code. Note that I don't
 * do any error-checking, so if the sound file doesn't exist,
 * this code throws an exception.
 */
@throws(classOf[java.io.FileNotFoundException])
def playSoundfile(f: String): Unit =
    val audioInputStream =
        AudioSystem.getAudioInputStream(java.io.File(f))
    val clip = AudioSystem.getClip
    clip.open(audioInputStream)
    val floatGainControl = clip
        .getControl(FloatControl.Type.MASTER_GAIN)
        .asInstanceOf[FloatControl]
    //reduce volume by x decibels (like -10f or -20f):
    floatGainControl.setValue(gainControl)
    clip.start
```

Discussion

Here's a little discussion of some portions of the code that I haven't mentioned previously in this book.

First, I use this code to process the command-line arguments:

```
val minutesToWait = args(0).toInt
val gainControl = if args.length == 2 then args(1).toInt else -30
```

Scala CLI makes the `args` variable available to you implicitly, and it's a normal sequence that contains those arguments (where each argument is separated by a space on the command line). As I note in the comments, this code can throw an exception because I don't check to see if the values are integers. But I'm okay with that because until now I have only used this personally, so it doesn't matter to me if the code throws an exception or if I get a well-formatted error message.

Next, this code handles the time to wait before the sound is played:

```
for i <- 1 to minutesToWait do
    Thread.sleep(60_000)
    println(s"time remaining: ${minutesToWait-i} ...")
```

The `Thread.sleep` portion of the code is a Java method call, and as shown, it takes time values in milliseconds, so `60_000` means "go to sleep for 60 seconds."

Lastly, the `playSoundfile` function uses the Java Sound library[1] to play my audio file. I keep the audio file in the same directory as this `timer.sc` script, so in the code I refer to that file as `"./gong.wav"`, where the `"./"` portion of that means "it should be in the current directory."

This code also shows how to properly declare that a function throws an exception:

```
@throws(classOf[java.io.FileNotFoundException])
```

Instead of throwing functions like this you should use the `Option`, `Try`, and `Either` classes, as shown earlier in this book, but again, this is a script that I have just used personally, so I left that code as-is.

If you ever wanted a command-line timer, I hope this is helpful.

[1] https://docs.oracle.com/javase/8/docs/technotes/guides/sound/index.html

81

Example: HTTP GET and POST Requests

One of the things you do quite a bit as a programmer is to access remote websites using the HTTP protocol. To help test HTTP code, someone was kind enough to create the *httpbin.org* website for testing purposes, and the code in this lesson uses that website. Also, I assume that you have made HTTP requests before, so I don't get into some details here, such as the difference between GET and POST requests.

Before we jump into the code it's important to say that there are Java classes and methods we *could* use to build your own HTTP library, but as a practical matter, most professionals use a well-tested library that's built by others for these types of things. Therefore, in the examples that follow I use the 'sttp' Client[1] library from a company named Software Mill.

Making an HTTP GET request

An HTTP GET request makes a call to a URL, and expects to receive some content in return. To do this with sttp, put this code in a file named `HttpGet.scala`, and then run it as shown in the comments:

```
//> using scala "3"
//> using lib "com.softwaremill.sttp.client3::core::3.7.2"
// the two previous lines are the Scala CLI way to say that
// you want to use (a) the latest version of Scala 3, and
// (b) the 'sttp' client library.

// run me with 'scala-cli HttpGet.scala'

import sttp.client3.*
```

[1] https://sttp.softwaremill.com/en/latest/

```
@main def doGet =
    val backend = HttpURLConnectionBackend()
    val response = basicRequest
                      .get(uri"http://httpbin.org/get")
                      .send(backend)
    println(response)
```

Notice that the code uses the `get` method of the `basicRequest` type, which corresponds to an HTTP `GET` request. When it runs you'll see that it returns some content. (Or, if httpbin.org is down or slow, you may see an error result.)

In this code it's important to know that sttp can use different "backend" software implementations to connect to websites, and per their documentation, the `HttpURLConnectionBackend` is "the default synchronous backend."

> I share several sttp recipes in the Scala Cookbook[2], but this code shows how to make the most basic `GET` request with sttp.

Feel free to change that URL to see how this process works with other websites.

Making an HTTP POST request

Making an HTTP `POST` request with sttp is very similar. Just put the following code in a file named `HttpPost.scala` and run it as shown, and you'll see its results:

[2] https://amzn.to/3du1pMR

```
//> using scala "3"
//> using lib "com.softwaremill.sttp.client3::core::3.7.2"

// run me with 'scala-cli HttpPost.scala'

import sttp.client3.*

@main
def doPost() =
    val backend = HttpURLConnectionBackend()
    val response = basicRequest
        .body("Hello, world!")
        .post(uri"https://httpbin.org/post?hello=world")
        .send(backend)
    println(response)
```

This example uses the `post` method of the `basicRequest` type, which corresponds to the HTTP `POST` protocol. It sends a simple "body" to the URL, and then receives and prints the server response

Discussion

Those examples show the basics of making HTTP requests with Scala 3 and the sttp library. Everything after this, including manually setting timeouts, dealing with cookies, passing parameters, etc., just builds on top of these examples. If you're interested in learning more, see the sttp website, or my examples in the Scala Cookbook, 2nd Edition[3].

[3] https://amzn.to/3du1pMR

82

Example: Create a GUI Application with Scala and Swing

To give you an idea of how seamless it is to use Java libraries from Scala, this example uses the Java "Swing" libraries for GUI "thick client" application development. Swing was introduced many years ago with Java 2, and you can still use it for GUI development today.

The following code creates a Swing text area, puts some text in it, adds that to a scroll pane, adds the scroll pane to a frame, and then displays it all:

```
//> using scala "3"

// run me with 'scala-cli Swing.scala'

import java.awt.{BorderLayout, Dimension}
import javax.swing.{JFrame, JScrollPane, JTextArea, WindowConstants}

@main def SwingExample =
    // create a text area
    val textArea = JTextArea("Hello, Swing world")

    // put that text area in a scroll pane
    val scrollPane = JScrollPane(textArea)

    // create a JFrame
    val frame = JFrame("Hello, Swing")
    // put the scroll pane in the frame
    frame.getContentPane.add(scrollPane, BorderLayout.CENTER)
    // set some frame properties
    frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE)
    frame.setSize(new Dimension(600, 400))
    // this tells swing to center the window
    frame.setLocationRelativeTo(null)
    // finally, make it visible
    frame.setVisible(true)
```

Figure 82.1: A Scala Swing example application.

I tested that code using Scala 3.1 with Java 11, and it works as desired to display a Java `JFrame` on my display. Assuming that you put that code in a file named `Swing.scala`, you can run it like this:

```
$ scala-cli Swing.scala
```

you should see a window (`JFrame`) that looks like this:

More information

If you're interested in writing Swing/GUI code with Scala, here are a couple of tutorials I've written:

- A semi-transparent popup dialog (JFrame, actually) in Scala[1]
- Passing an anonymous function to SwingUtilities.invokeLater (in Scala)[2]

[1] https://alvinalexander.com/source-code/scala/semi-transparent-popup-dialog-jframe-actually-scala-or-java/
[2] https://alvinalexander.com/source-code/swingutilities-invokelater-anonymous-function-lambda

The second tutorial is important because when you work with Swing you *really* need to run code with `SwingUtilities.invokeLater`, as shown there.

JavaFX is a more modern Java GUI-building framework, and I've also written about it in these blog posts and tutorials:

- A Scala version of the JavaFX "Hello, world" example[3]
- A sample Scala/JavaFX application[4]
- A Scala/JavaFX WebSocket client[5]

[3] https://alvinalexander.com/scala/scala-javafx-hello-world-example/
[4] https://alvinalexander.com/source-code/scala-javafx-application-launch-scene-stylesheets/
[5] https://alvinalexander.com/source-code/scala-javafx-websocket-client/

83

The End (and What's Next)

That brings us to the end of this book. I hope it's been helpful!

If enough people like this book and would like to see a follow-up to it, I'll be glad to write that next book (i.e., "Book 2" in this series). Until then, hopefully my other books — both of which are over 700 pages long — can be helpful on your learning journey:

- I wrote Functional Programming, Simplified[1] for *Scala 2*, but if you're interested in learning FP, many people have told me they are happy with it. The book takes you from the Scala 2 basics all the way into FP concepts like monads.
- The Scala Cookbook, 2nd Edition[2] was written for *Scala 3* and released in late 2021. For the time being that's the best "next step" I can offer you.

Functional Programming, Simplified is written in the same style as this book, with over 100 small lessons. But its material is significantly different from this book as it focuses entirely on FP.

The *Scala Cookbook* is also written in a similar style to this book, and includes over 250 recipes for common Scala problems. Its table of contents is eight pages long, and you can see that index here on Amazon.com[3]. But briefly, I can tell you that what that book covers that this book does not includes:

- More coverage of the collections classes, their performance, and their methods
- Parallel and concurrent programming
- Web services
- Apache Spark

[1] https://alvinalexander.com/scala/functional-programming-simplified-book
[2] https://amzn.to/3du1pMR
[3] https://amzn.to/3du1pMR

- Scala.js
- *Much* more on data types (inheritance, generics)
- "Best practices" for writing Scala 3 code
- More domain modeling information
- Much more!

If you want to learn more about Scala 3 today, that's my best option for you at the moment.

Thank you!

Thank you again for taking the time to read *Learn Scala 3 The Fast Way! (Book 1)*. If you like the book — or sadly, even if you didn't — I'd appreciate a review to let others know about it. You can find the two current versions of the book here:

- The Kindle version on Amazon.com[4]
- The PDF version on Gumroad.com[5]

And if you're interested, you can find me at these locations:

- twitter.com/alvinalexander[6]
- linkedin.com/in/alvinalexander[7]

All the best,
Alvin Alexander
alvinalexander.com[8]

[4] https://www.amazon.com/dp/B0BDWQ75YC
[5] https://alvinalexander.gumroad.com/l/learn-scala3-fast
[6] https://twitter.com/alvinalexander
[7] https://www.linkedin.com/in/alvinalexander/
[8] https://alvinalexander.com