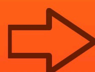
FREE PREVIEW!

# LEARN SCALA 3 THE FAST WAY!

1) READ A SHORT LESSON



2) WORK THE EXAMPLES

3) TAKE THE ONLINE EXERCISES

BOOK 1: THE ADVENTURE BEGINS

# 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)*

Copyright 2022 Alvin J. Alexander[1]

All rights reserved. No part of this book may be reproduced without prior written permission from the author.

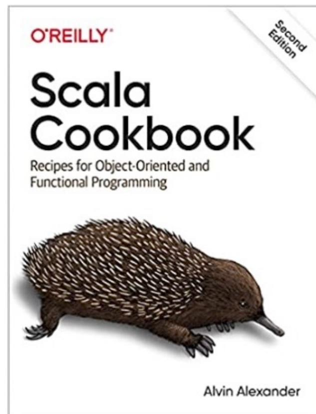
This book is presented solely for educational purposes. While best efforts have been made to prepare this book, the author makes no representations or warranties of any kind and assumes no liabilities of any kind with respect to the accuracy or completeness of the contents, and specifically disclaims any implied warranties of merchantability or fitness of use for a particular purpose. The author shall not be held liable or responsible to any person or entity with respect to any loss or incidental or consequential damages caused, or alleged to have been caused, directly or indirectly, by the information or programs contained herein. Any use of this information is at your own risk.

Version 0.1, published August 29, 2022

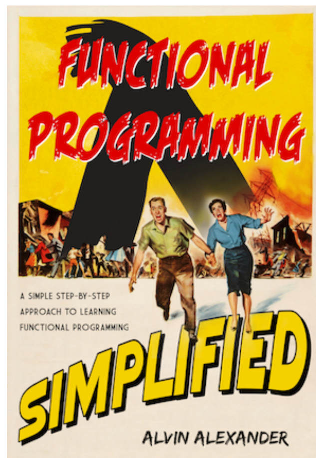
Buy the book and find/report issues:
alvinalexander.com/scala/learn-scala-3-the-fast-way-book[2]

[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 (alvinalexander.com)[4]

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

# Contents


1 About This Early Release 1
2 About The Author 3
3 Welcome to Scala 3! 5
4 Why Learn Scala 3? 9
5 Your Setup For This Book 15
6 Note 1: Significant Indentation Syntax 17
7 Note 2: Comments 19
8 Beginning: The Scala REPL 21
9 Beginning: Printing With println 25
10 Variables: val Fields 27
11 Variables: var Fields 31
12 Variables: Explicitly Declaring The Data Type 33
13 Strings: Common Methods 35
14 Strings: Interpolators 37
15 Strings: Multiline Strings 41
16 Numeric Data Types 43
17 Constructs: Mathematical Expressions 47
18 Constructs: `if/then`/else 49
19 Expression-Oriented Programming (EOP) 51
20 Tuples 55
21 Collections: The List Class 57
22 Collections: Updating List Elements 61
23 Collections: Other Sequence Classes 63
24 Constructs: for Loops 67
25 Constructs: for Expressions 71

# 1

# About This Early Release

This is an *early release* of this book, by which I mean that it's like an alpha or beta of a software application. I refer to this as a Version 0.1 release, and I anticipate that there will be at least one more release before it's complete.

The current issues I'm aware of are:

- The Github repository is not ready for release yet
- The online exercises are ready, but I'd like to add some more exercises (maybe 10-20% more)
- When you publish a book you need to go through a "final formatting" stage, and I haven't done that yet
- I have a list of about 10-20 small issues that I'm aware of that I want to improve

Other than that, the book is about 270 pages long right now, and I don't anticipate that it will grow by more than 10 additional pages, so it's close … but not quite done yet.

Note that if you buy the PDF version of this early release here on Gumroad.com[1], you will be notified of updates as I make them available.

Thanks for reading, and I hope this is helpful!

All the best,
Alvin Alexander

Longmont, Colorado
August 29, 2022

[1] https://alvinalexander.gumroad.com/l/learn-scala3-fast

# 2

# 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

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[3], which receives millions of page views every year, and I occasionally post small Scala tips on my Twitter account[4].

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.com
[4] 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 that includes most of the code shown, along with a few small projects that are good for beginners.
- Online exercises for every lesson.

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

# 3

# 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 you can download and experiment with, and a companion website with exercises for each lesson.

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 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.

## Exercises and code examples

At the end of each lesson you'll find a link to that lesson's online exercises. The exercises won't take long, and they often present the material in a way that's different from the book to test your understanding.

You'll also find that these exercises are a great way to test your knowledge in the future.

[1] https://amzn.to/3CpnJ6t

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* rememeber.

In addition to the online exercises, the book will eventually also have a corresponding Github repository of example code that you can use and modify. (In this early release that repo is TBD.)

## 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:

- 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 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)[2], which was also written for Scala 3.

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

# 4

# 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.

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

## 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 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 `{}` sym-

bols, 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).

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.

## 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 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
```

[7] https://scala-cli.virtuslab.org

```
    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[8].

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

# 5

# Your Setup For This Book

As we get close to the first programming lesson, let's start getting you 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, you Scala-CLI has an option to export your settings to a Scala build tool like *sbt* or *Mill*

[1] https://scala-cli.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 use Scala-CLI[2], just use the Installation link on their website.

## 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 this approach, so I recommend it instead.

> 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.

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

# 6

# 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.

# 7

# 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
```

# 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 on your system, 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

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

run Scala code in your browser. You just enter your code, press Run, and see the output.

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!

# 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.

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

# 10

# Variables: val Fields

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

```
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 *ton* 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"
```

# 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 event 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.

# 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.

# 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("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.

## More information

If you ever want to write your own string interpolator, see my tutorial, How to create your own Scala 3 String interpolator[1].

[1] https://alvinalexander.com/scala/scala-3-how-create-custom-string-interpolator-extension-methods

# 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
```

# 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 also 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        // Int
val b = 1_000_000    // Int
val c = 1_000_000L   // Long (created with the 'L' after the number)
val d = 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]

[1] https://www.scala-lang.org/api/current/scala/math/BigInt.html
[2] https://www.scala-lang.org/api/current/scala/math/BigDecimal.html

# 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 are *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.

# 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.

# 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 then `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 peak 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`.

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

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.

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*.

# 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 they are good points to know.

## 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!

# 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.

# 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].

---

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

# 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.

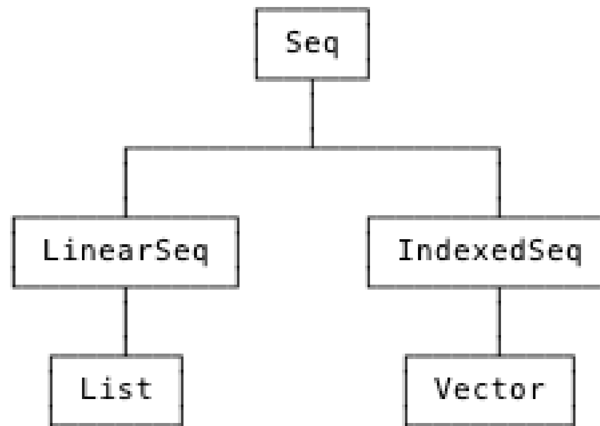



*Figure 23.1: A subset of Scala's immutable sequence classes*

## 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.

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

# 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.

# 25

# Constructs: for Expressions

As you get better and better at using Scala you'll find that you won't use `for` *loops* very often, but you will start to use a similar construct: `for` *expressions*.

A `for` expression is similar to a `for` loop, except that it really is an expression, and 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` (pronounced "exes"), 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")

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]`.