Scala Training

v 2.0.1

Trainer: Mario Cartia

Why Scala?

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

Why Scala?

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

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

Why Scala?

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

Why Scala?

- **Scala** is a language that scales well from one-line snippets to *million-line* production codebases, with the convenience of a scripting language and the performance and scalability of a compiled language
- **Scala**'s conciseness makes rapid prototyping a joy, while its optimizing compiler and fast JVM runtime provide great performance to support your heaviest production workloads

Why Scala?

- **Scala**'s functional programming style and type-checking compiler helps rule out entire classes of bugs and defects, saving you time and effort you can instead spend developing features for your users
- Rather than fighting `TypeErrors` and `NullPointerExceptions` in production, **Scala** surfaces mistakes and issues early on during compilation so you can resolve them before they impact your bottom line

Why Scala?

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

History of Scala

The design of Scala started in 2001 at the *École Polytechnique Fédérale de Lausanne* (EPFL) by **Martin Odersky**. It followed on from work on Funnel, a programming language combining ideas from functional programming and Petri nets. Odersky formerly worked on javac, Sun's Java compiler.

After an internal release in late 2003, Scala was released publicly in early 2004 on the Java platformA second version (v2.0) followed in March 2006.

History of Scala

The name Scala is a portmanteau of scalable and language, signifying that it is designed to grow with the demands of its users.

Scala stairs in building BC at EPFL inspired the name and the logo of Scala.

Overview

Before we jump into the examples, here are a few important things to know about Scala:

- It’s a high-level language
- It’s statically typed
- Its syntax is concise but still readable — we call it expressive
- It supports the object-oriented programming (OOP) paradigm
- It supports the functional programming (FP) paradigm
- It has a sophisticated type inference system
- Scala code results in .class files that run on the Java Virtual Machine (JVM)
- It’s easy to use Java libraries in Scala

Hello, world

Ever since the book, C Programming Language, it's been a tradition to begin programming books with a "Hello, world" example, and not to disappoint, this is one way to write that example in Scala:

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

After you save that code to a file named Hello.scala, you can compile it with scalac:

```
$ scalac Hello.scala
```

Hello, world

If you’re coming to Scala from Java, scalac is just like javac, and that command creates two files:

```
Hello$.class
Hello.class
```

These are the same “.class” bytecode files you create with javac, and they’re ready to run in the JVM. You run the Hello application with the scala command:

```
$ scala Hello
```

The Scala REPL

The Scala REPL (“Read-Evaluate-Print-Loop”) is a command-line interpreter that you use as a “playground” area to test your Scala code. We introduce it early here so you can use it with the code examples that follow.

To start a REPL session, just type scala at your operating system command line, and you’ll see something like this:

```
$ scala
Welcome to Scala 2.13.0 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_131).
Type in expressions for evaluation. Or try :help.

scala> _
```

The Scala REPL

Because the REPL is a command-line interpreter, it just sits there waiting for you to type something. Inside the REPL you type Scala expressions to see how they work:

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

scala> val y = x + 1
y: Int = 2
```

As those examples show, after you type your expressions in the REPL, it shows the result of each expression on the line following the prompt.

Two types of variables

Scala has two types of variables:

- `val` is an immutable variable — like final in Java — and should be preferred
- `var` creates a mutable variable, and should only be used when there is a specific reason to use it

Examples:

```
val x = 1   //immutable
var y = 0   //mutable
```

Declaring variable types

In Scala, you typically create variables without declaring their type:

```
val x = 1
val s = "a string"
val p = new Person("Regina")
```

When you do this, Scala can usually infer the data type for you, as shown in these REPL examples:

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

scala> val s = "a string"
val s: String = a string
```

Declaring variable types

This feature is known as *type inference*, and it's a great way to help keep your code concise. You can also explicitly declare a variable's type, but that's not usually necessary:

```
val x: Int = 1
val s: String = "a string"
val p: Person = new Person("Regina")
```

As you can see, that code looks unnecessarily verbose.

Control structures

Here's a quick tour of Scala's control structures.

if/else

Scala's if/else control structure is similar to other languages:

```
if (test1) {
    doA()
} else if (test2) {
    doB()
} else if (test3) {
    doC()
} else {
    doD()
}
```

if/else

However, unlike Java and many other languages, the if/else construct returns a value, so, among other things, you can use it as a ternary operator:

```
val x = if (a < b) a else b
```

match expressions

Scala has a `match` expression, which in its most basic use is like a Java switch statement:

```
val result = i match {
    case 1 => "one"
    case 2 => "two"
    case _ => "not 1 or 2"
}
```

match expressions

The `match` expression isn't limited to just integers, it can be used with any data type, including booleans:

```
val booleanAsString = bool match {
    case true => "true"
    case false => "false"
}
```

match expressions

Here's an example of `match` being used as the body of a method, and matching against many different types:

```
def getClassAsString(x: Any):String = x match {
    case s: String => s + " is a String"
    case i: Int => "Int"
    case f: Float => "Float"
    case l: List[_] => "List"
    case p: Person => "Person"
    case _ => "Unknown"
}
```

Powerful match expressions are a big feature of Scala, and we share more examples of it later.

try/catch

Scala's try/catch control structure lets you catch exceptions. It's similar to Java, but its syntax is consistent with match expressions:

```
try {
    writeToFile(text)
} catch {
    case fnfe: FileNotFoundException => println(fnfe)
    case ioe: IOException => println(ioe)
}
```

for loops and expressions

Scala `for` loops (*for-loops*) — look like this:

```
for (arg <- args) println(arg)

// "x to y" syntax
for (i <- 0 to 5) println(i)

// "x to y by" syntax
for (i <- 0 to 10 by 2) println(i)
```

for loops and expressions

You can also add the `yield` keyword to for-loops to create *for-expressions* that yield a result. Here's a for-expression that doubles each value in the sequence 1 to 5:

```
val x = for (i <- 1 to 5) yield i * 2
```

for loops and expressions

Here's another for-expression that iterates over a list of strings:

```
val fruits = List("apple", "banana", "lime", "orange")

val fruitLengths = for {
    f <- fruits
    if f.length > 4
} yield f.length
```

Because Scala code generally just makes sense, we'll imagine that you can guess how this code works, even if you've never seen a for-expression or Scala list until now.

while and do/while

Scala also has `while` and `do/while` loops. Here's their general syntax:

```
// while loop
while(condition) {
    statement(a)
    statement(b)
}

// do-while
do {
   statement(a)
   statement(b)
}
while(condition)
```

Classes

Here's an example of a Scala class:

```
class Person(var firstName: String, var lastName: String) {
    def printFullName() = println(s"$firstName $lastName")
}
```

This is how you use that class:

```
val p = new Person("Julia", "Kern")
println(p.firstName)
p.lastName = "Manes"
p.printFullName()
```

Classes

Notice that there's no need to create "get" and "set" methods to access the fields in the class.

As a more complicated example, here's a `Pizza` class that you'll see later in the examples:

```
class Pizza (
    var crustSize: CrustSize,
    var crustType: CrustType,
    val toppings: ArrayBuffer[Topping]
) {
    def addTopping(t: Topping): Unit = toppings += t
    def removeTopping(t: Topping): Unit = toppings -= t
    def removeAllToppings(): Unit = toppings.clear()
}
```

Classes

```
class Pizza (
    var crustSize: CrustSize,
    var crustType: CrustType,
    val toppings: ArrayBuffer[Topping]
) {
    def addTopping(t: Topping): Unit = toppings += t
    def removeTopping(t: Topping): Unit = toppings -= t
    def removeAllToppings(): Unit = toppings.clear()
}
```

In that code, an `ArrayBuffer` is like Java's `ArrayList`. The `CrustSize`, `CrustType`, and `Topping` classes aren't shown, but you can probably understand how that code works without needing to see those classes.

Scala methods

Just like other OOP languages, Scala classes have methods, and this is what the Scala method syntax looks like:

```
def sum(a: Int, b: Int): Int = a + b
def concatenate(s1: String, s2: String): String = s1 + s2
```

You don’t have to declare a method’s return type, so it’s perfectly legal to write those two methods like this, if you prefer:

```
def sum(a: Int, b: Int) = a + b
def concatenate(s1: String, s2: String) = s1 + s2
```

Scala methods

This is how you call those methods:

```
val x = sum(1, 2)
val y = concatenate("foo", "bar")
```

There are more things you can do with methods, such as providing default values for method parameters, but that’s a good start for now.

Traits

Traits in Scala are a lot of fun, and they also let you break your code down into small, modular units. To demonstrate traits, here's an example from later in the book. Given these three traits:

```
trait Speaker {
    def speak(): String  // has no body, so it's abstract
}

trait TailWagger {
    def startTail(): Unit = println("tail is wagging")
    def stopTail(): Unit = println("tail is stopped")
}

trait Runner {
    def startRunning(): Unit = println("I'm running")
    def stopRunning(): Unit = println("Stopped running")
}
```

Traits

You can create a `Dog` class that extends all of those traits while providing behavior for the speak method:

```
class Dog(name: String) extends Speaker with TailWagger with Runner {
    def speak(): String = "Woof!"
}
```

Similarly, here's a `Cat` class that shows how to override multiple trait methods:

```
class Cat extends Speaker with TailWagger with Runner {
    def speak(): String = "Meow"
    override def startRunning(): Unit = println("Yeah ... I don't run")
    override def stopRunning(): Unit = println("No need to stop")
}
```

Collections classes

If you're coming to Scala from Java and you're ready to really jump in and learn Scala, it's possible to use the Java collections classes (see documentation) in Scala, and some people do so for several weeks or months while getting comfortable with Scala. But it's highly recommended that you learn the basic Scala collections classes — `List`, `ListBuffer`, `Vector`, `ArrayBuffer`, `Map`, and `Set` — as soon as possible.

A great benefit of the Scala collections classes is that they offer many powerful methods that you'll want to start using as soon as possible to simplify your code.

Populating lists

There are times when it’s helpful to create sample lists that are populated with data, and Scala offers many ways to populate lists. Here are just a few:

```
val nums = List.range(0, 10)
val nums = (1 to 10 by 2).toList
val letters = ('a' to 'f').toList
val letters = ('a' to 'f' by 2).toList
```

Sequence methods

While there are many sequential collections classes you can use — `Array`, `ArrayBuffer`, `Vector`, `List`, and more — let's look at some examples of what you can do with the List class. Given these two lists:

```
val nums = (1 to 10).toList
val names = List("joel", "ed", "chris", "maurice")
```

This is the `foreach` method:

```
scala> names.foreach(println)
joel
ed
chris
maurice
```

Sequence methods

```
val nums = (1 to 10).toList
val names = List("joel", "ed", "chris", "maurice")
```

Here's the `filter` method, followed by `foreach`:

```
scala> nums.filter(_ < 4).foreach(println)
1
2
3
```

Sequence methods

```
val nums = (1 to 10).toList
val names = List("joel", "ed", "chris", "maurice")
```

Here are some examples of the `map` method:

```
scala> val doubles = nums.map(_ * 2)
doubles: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)

scala> val capNames = names.map(_.capitalize)
capNames: List[String] = List(Joel, Ed, Chris, Maurice)

scala> val lessThanFive = nums.map(_ < 5)
lessThanFive: List[Boolean] = List(true, true, true, true, false, false, false, false, false, false)
```

Sequence methods

Even without any explanation you can see how `map` works: It applies an algorithm you supply to every element in the collection, returning a new, transformed value for each element.

If you're ready to see one of the most powerful collections methods, here's `foldLeft`:

```
scala> nums.foldLeft(0)(_ + _)
res0: Int = 55

scala> nums.foldLeft(1)(_ * _)
res1: Int = 3628800
```

Sequence methods

```
scala> nums.foldLeft(0)(_ + _)
res0: Int = 55

scala> nums.foldLeft(1)(_ * _)
res1: Int = 3628800
```

Once you know that the first parameter to `foldLeft` is a seed value, you can guess that the first example yields the sum of the numbers in `nums`, and the second example returns the *product* of all those numbers.

Sequence methods

There are many (many!) more methods available to Scala collections classes, and many of them will be demonstrated in the collections lessons that follow, but hopefully this gives you an idea of their power.

Tuples

Tuples let you put a heterogenous collection of elements in a little container. A tuple can contain between two and 22 values, and all of the values can have different types. For example, this is a tuple that holds three different types, an `Int`, a `Double`, and a `String`:

```
(11, 11.0, "Eleven")
```

This is known as a `Tuple3`, because it contains three elements.

Tuples

Tuples are convenient in many places, such as where you might use an ad-hoc class in other languages. For instance, you can return a tuple from a method instead of returning a class:

```
def getAaplInfo(): (String, BigDecimal, Long) = {
    // get the stock symbol, price, and volume
    ("AAPL", BigDecimal(123.45), 101202303L)
}
```

Then you can assign the result of the method to a variable:

```
val t = getAaplInfo()
```

Tuples

Once you have a tuple variable, you can access its values by number, preceded by an underscore:

```
t._1
t._2
t._3
```

The REPL demonstrates the results of accessing those fields:

```
scala> t._1
res0: String = AAPL

scala> t._2
res1: scala.math.BigDecimal = 123.45

scala> t._3
res2: Long = 101202303
```

Tuples

The values of a tuple can also be extracted using pattern matching. In this next example, the fields inside the tuple are assigned to the variables `symbol`, `price`, and `volume`:

```
val (symbol, price, volume) = getAaplInfo()
```

Once again, the REPL shows the result:

```
scala> val (symbol, price, volume) = getAaplInfo()
symbol: String = AAPL
price: scala.math.BigDecimal = 123.45
volume: Long = 101202303
```

Tuples

Tuples are nice for those times when you want to quickly (and temporarily) group some things together. If you notice that you are using the same tuples multiple times, it could be useful to declare a dedicated case class, such as:

```
case class StockInfo(symbol: String, price: BigDecimal, volume: Long)
```

Exercise 1

1. Write a program that prints ‘Hello World’ to the screen.
2. Write a program that asks the user for their name and greets them with their name.
3. Modify the previous program such that only the users Alice and Bob are greeted with their names.
4. Write a program that asks the user for a number n and prints the sum of the numbers 1 to n
5. Modify the previous program such that only multiples of three or five are considered in the sum, e.g. 3, 5, 6, 9, 10, 12, 15 for n=17

6. Write a program that asks the user for a number n and gives them the possibility to choose between computing the sum and computing the product of 1,…,n.
7. Write a program that that asks the user for a number n and prints first n prime numbers.
8. Write a guessing game where the user has to guess a secret number. After every guess the program tells the user whether their number was too large or too small. At the end the number of tries needed should be printed. It counts only as one try if they input the same number multiple times consecutively.
9. Write a function that tests whether a string is a palindrome

10. Write a function that takes a list of strings an prints them, one per line, in a rectangular frame. For example the list ["Hello", "World", "in", "a", "frame"] gets printed as:

```
*********
* Hello *
* World *
* in    *
* a     *
* frame *
*********
```

Exercise 2

1. Write a function that returns the largest element in a list.
2. Write a custom function that reverses a list (in place or not using a boolean parameter).
3. Write a function that returns the elements on odd positions in a list.
4. Write three functions that compute the sum of the numbers in a list: using a for-loop, a while-loop and recursion.
5. Write a custom function that concatenates two lists. [a,b,c], [1,2,3] → [a,b,c,1,2,3]

6. Write a custom function that combines two lists by alternatingly taking elements, e.g. [a,b,c], [1,2,3] → [a,1,b,2,c,3].
7. Write a function that merges two sorted lists into a new sorted list. [1,4,6],[2,3,5] → [1,2,3,4,5,6].
8. Write a function that takes a number and returns a list of its digits. So for 2342 it should return [2,3,4,2].
9. Implement the following sorting algorithm: Merge sort (iterative) https://en.wikipedia.org/wiki/Merge_sort
10. Implement the following sorting algorithm: Quick sort (recursive) https://en.wikipedia.org/wiki/Quicksort

Project 1 - TO-DO list application

Implements a simple application for managing task lists. The application must have the following features:

- implement six commands: `help, list, add, remove, save` and `load`
- the `help` command must show the list of available commands
- the `list` command must show the list of previously entered activities. For example:

```
1. Lorem ipsum
2. Bla bla bla
3. Aeiou
```

- the `add` and `remove` commands must allow the addition and deletion of list elements (for deletion an index corresponding to the position of the element to be removed in the list will be passed)
- the `save` and `load` commands will save and load data on a plain text comma delimited (id, text) format

Scala Collections

Scala collections systematically distinguish between mutable and immutable collections. A *mutable* collection can be updated or extended in place. This means you can change, add, or remove elements of a collection as a side effect.

Immutable collections, by contrast, never change. You have still operations that simulate additions, removals, or updates, but those operations will in each case return a new collection and leave the old collection unchanged.

Scala Collections

All collection classes are found in the package `scala.collection` or one of its sub-packages `mutable`, `immutable`, and `generic`. Most collection classes needed by client code exist in three variants, which are located in packages `scala.collection`, `scala.collection.immutable`, and `scala.collection.mutable`, respectively. Each variant has different characteristics with respect to mutability.

Scala Collections

A collection in package `scala.collection.immutable` is guaranteed to be immutable for everyone. Such a collection will never change after it is created. Therefore, you can rely on the fact that accessing the same collection value repeatedly at different points in time will always yield a collection with the same elements.

A collection in package `scala.collection.mutable` is known to have some operations that change the collection in place.

So dealing with mutable collection means you need to understand which code changes which collection when.

Scala Collections

A collection in package `scala.collection` can be either mutable or immutable. For instance, collection.IndexedSeq[T] is a superclass of both collection.immutable.IndexedSeq[T] and collection.mutable.IndexedSeq[T]

Generally, the root collections in package `scala.collection` define the same interface as the immutable collections, and the mutable collections in package `scala.collection.mutable` typically add some side-effecting modification operations to this immutable interface.

Scala Collections

By default, Scala always picks immutable collections. For instance, if you just write `Set` without any prefix or without having imported `Set` from somewhere, you get an immutable set, and if you write `Iterable` you get an immutable iterable collection, because these are the default bindings imported from the `scala` package. To get the mutable default versions, you need to write explicitly `collection.mutable.Set`, or `collection.mutable.Iterable`.

Scala Collections

A useful convention if you want to use both mutable and immutable versions of collections is to import just the package `collection.mutable`.

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

Then a word like `Set` without a prefix still refers to an immutable collection, whereas `mutable.Set` refers to the mutable counterpart.

Scala Collections

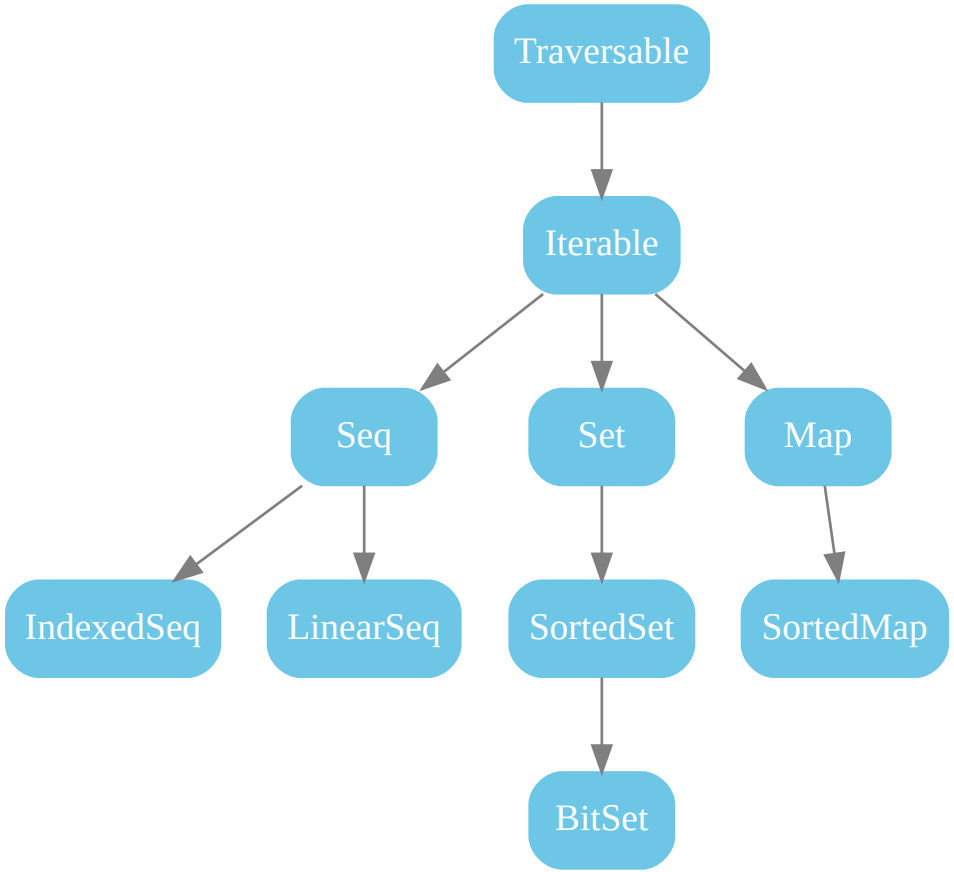
The last package in the collection hierarchy is `collection.generic`. This package contains building blocks for implementing collections. Typically, collection classes defer the implementations of some of their operations to classes in `generic`. Users of the collection framework on the other hand should need to refer to classes in `generic` only in exceptional circumstances.

Scala Collections

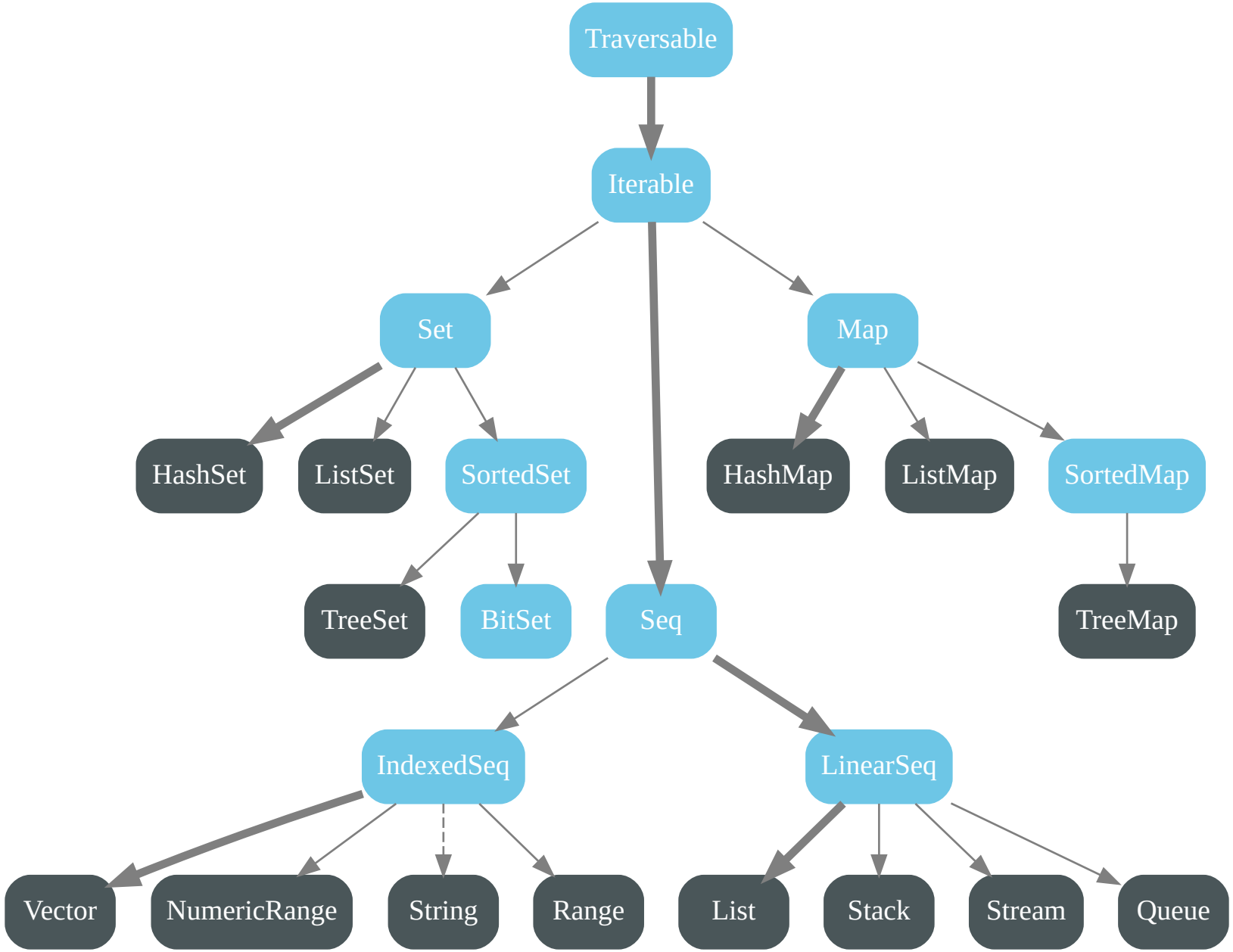
For convenience and backwards compatibility some important types have aliases in the `scala` package, so you can use them by their simple names without needing an import. An example is the `List` type, which can be accessed alternatively as

```
scala.collection.immutable.List   // that's where it is defined
scala.List                        // via the alias in the scala package
List                              // because scala._
                                  // is always automatically imported
```

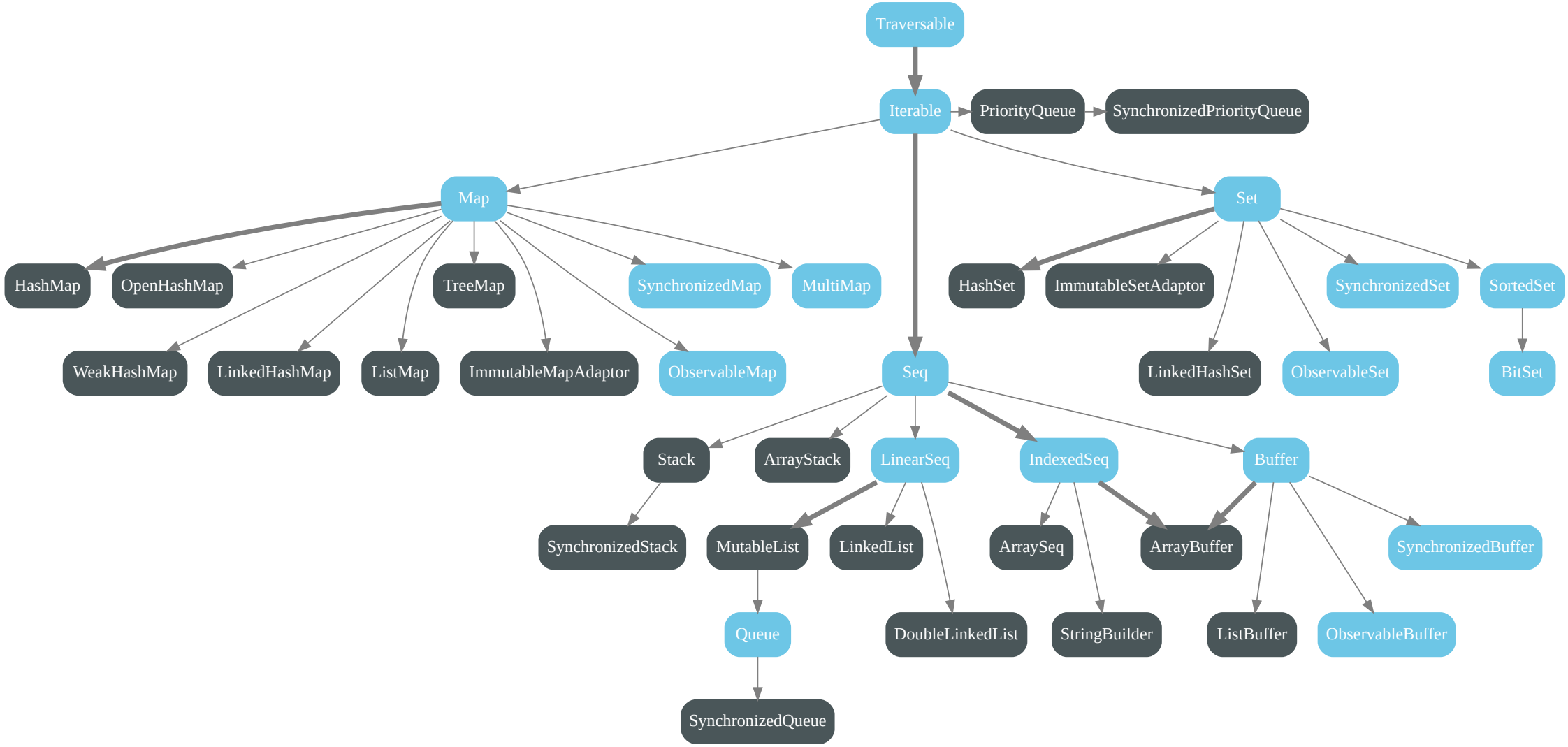
The following figure shows all collections in package `scala.collection`. These are all high-level abstract classes or traits, which generally have mutable as well as immutable implementations.



The following figure shows all collections in package `scala.collection.immutable`.



And the following figure shows all collections in package `scala.collection.mutable`.



An Overview of the Collections API

The most important collection classes are shown in the figures above. There is quite a bit of commonality shared by all these classes. For instance, every kind of collection can be created by the same uniform syntax, writing the collection class name followed by its elements:

```
Traversable(1, 2, 3)
Iterable("x", "y", "z")
Map("x" -> 24, "y" -> 25, "z" -> 26)
Set(Color.red, Color.green, Color.blue)
SortedSet("hello", "world")
Buffer(x, y, z)
IndexedSeq(1.0, 2.0)
LinearSeq(a, b, c)
```

An Overview of the Collections API

The same principle also applies for specific collection implementations, such as:

```
List(1, 2, 3)
HashMap("x" -> 24, "y" -> 25, "z" -> 26)
```

All these collections get displayed with `toString` in the same way they are written above.

An Overview of the Collections API

All collections support the API provided by `Traversable`, but specialize types wherever this makes sense. For instance the `map` method in class `Traversable` returns another `Traversable` as its result. But this result type is overridden in subclasses. For instance, calling `map` on a `List` yields again a `List`, calling it on a `Set` yields again a `Set` and so on.

```
scala> List(1, 2, 3) map (_ + 1)
res0: List[Int] = List(2, 3, 4)
scala> Set(1, 2, 3) map (_ * 2)
res0: Set[Int] = Set(2, 4, 6)
```

This behavior which is implemented everywhere in the collections libraries is called the *uniform return type* principle.