The Neophyte's Guide to Scala

Daniel Westheide

The Neophyte's Guide to Scala

Daniel Westheide

This book is for sale at http://leanpub.com/theneophytesguidetoscala

This version was published on 2013-10-11

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Daniel Westheide by spreading the word about this book on Twitter!

The suggested hashtag for this book is #scala.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#scala

Contents

Introduction

In the autumn of 2012, more than 50,000 people signed up for Martin Odersky's course "Functional Programming Principles in Scala"[1] at Coursera. That's a huge number of developers for whom this might have been the first contact with Scala, functional programming, or both. In 2013, the course was repeated, introducing even more students and developers to Scala and functional programming.

If you are reading this, maybe you are one of them, or maybe you have started to learn Scala by some other means. In any case, if you have started to learn Scala, you are excited to delve deeper into this beautiful language, but it all still feels a little exotic or foggy to you, then this book is for you.

Even though the Coursera course covered quite a lot of what you need to know about Scala, the given time constraints made it impossible to explain everything in detail. As a result, some Scala features might seem like magic to you if you are new to the language. You are able to use them somehow, but you haven't fully grasped how they work and, more importantly, why they work as they do.

After the first installment of the Coursera course, I started a blog series, intending to clear things up and remove those question marks. This ebook is based on that blog series. Due to the overwhelmingly positive reception, I decided to compile all the articles into an ebook.

In this book, I will also explain some of the features of the Scala language and library that I had trouble with when I started learning the language, partially because I didn't find any good explanations for them, but instead just stumbled upon them in the wild. Where appropriate, I will also try to give guidance on how to use these features in an idiomatic way.

Enough of the introductions. Before we begin, keep in mind that, while having attended the Coursera course is not a prerequisite for studying this book, having roughly the knowledge of Scala as can be acquired in that course is definitely helpful, and I will sometimes refer to the course.

[1] https://www.coursera.org/course/progfun

Extractors

In the Coursera course, you came across one very powerful language feature of Scala: Pattern matching[2]. It allows you to decompose a given data structure, binding the values it was constructed from to variables. It's not an idea that is unique to Scala, though. Other prominent languages in which pattern matching plays an important role are Haskell and Erlang, for instance.

If you followed the video lectures, you saw that you can decompose various kinds of data structures using pattern matching, among them lists, streams, and any instances of case classes. So is this list of data structures that can be destructured fixed, or can you extend it somehow? And first of all, how does this actually work? Is there some kind of magic involved that allows you to write things like the following?

```
1  case class User(firstName: String, lastName: String, score: Int)
2  def advance(xs: List[User]) = xs match {
3    case User(_, _, score1) :: User(_, _, score2) :: _ => score1 - score2
4    case _ => 0
5  }
```

As it turns out, there isn't. At least not much. The reason why you are able to write the above code (no matter how little sense this particular example makes) is the existence of so-called extractors[3].

In its most widely applied form, an extractor has the opposite role of a constructor: While the latter creates an object from a given list of parameters, an extractor extracts the parameters from which an object passed to it was created.

The Scala library contains some predefined extractors, and we will have a look at one of them shortly. Case classes are special because Scala automatically creates a companion object for them: a singleton object that contains not only an `apply` method for creating new instances of the case class, but also an `unapply` method – the method that needs to be implemented by an object in order for it to be an extractor.

Our first extractor, yay!

There is more than one possible signature for a valid `unapply` method, but we will start with the ones that are most widely used. Let's pretend that our `User` class is not a case class after all, but instead a trait, with two classes extending it, and for the moment, it only contains a single field:

[2] http://en.wikipedia.org/wiki/Pattern_matching

[3] http://www.scala-lang.org/node/112

```
trait User {
  def name: String
}
class FreeUser(val name: String) extends User
class PremiumUser(val name: String) extends User
```

We want to implement extractors for the `FreeUser` and `PremiumUser` classes in respective companion objects, just as Scala would have done were these case classes. If your extractor is supposed to only extract a single parameter from a given object, the signature of an `unapply` method looks like this:

```
def unapply(object: S): Option[T]
```

The method expects some object of type `S` and returns an `Option` of type `T`, which is the type of the parameter it extracts.

> Remember that `Option` is Scala's safe alternative to the existence of `null` values. See chapter 5, The Option Type, for more details about it. For now, though, it's enough to know that the `unapply` method returns either `Some[T]` (if it could successfully extract the parameter from the given object) or `None`, which means that the parameters could not be extracted, as per the rules determined by the extractor implementation.

Here are our extractors:

```
trait User {
  def name: String
}
class FreeUser(val name: String) extends User
class PremiumUser(val name: String) extends User

object FreeUser {
  def unapply(user: FreeUser): Option[String] = Some(user.name)
}
object PremiumUser {
  def unapply(user: PremiumUser): Option[String] = Some(user.name)
}
```

We can now use this in the REPL:

```
scala> FreeUser.unapply(new FreeUser("Daniel"))
res0: Option[String] = Some(Daniel)
```

But you wouldn't usually call this method directly. Scala calls an extractor's `unapply` method if the extractor is used as an *extractor pattern*.

If the result of calling `unapply` is `Some[T]`, this means that the pattern matches, and the extracted value is bound to the variable declared in the pattern. If it is `None`, this means that the pattern doesn't match and the next case statement is tested.

Let's use our extractors for pattern matching:

```
val user: User = new PremiumUser("Daniel")
user match {
  case FreeUser(name) => "Hello " + name
  case PremiumUser(name) => "Welcome back, dear " + name
}
```

As you will already have noticed, our two extractors never return `None`. The example shows that this makes more sense than it might seem at first. If you have an object that could be of some type or another, you can check its type and destructure it at the same time.

In the example, the `FreeUser` pattern will not match because it expects an object of a different type than we pass it. Since it wants an object of type `FreeUser`, not one of type `PremiumUser`, this extractor is never even called. Hence, the `user` value is now passed to the `unapply` method of the `PremiumUser` companion object, as that extractor is used in the he second pattern. This pattern will match, and the returned value is bound to the `name` parameter.

Later in this chapter, you will see an example of an extractor that does not always return `Some[T]`.

Extracting several values

Now, let's assume that our classes against which we want to match have some more fields:

```
trait User {
  def name: String
  def score: Int
}
class FreeUser(
    val name: String,
    val score: Int,
    val upgradeProbability: Double)
  extends User
```

```
class PremiumUser(
    val name: String,
    val score: Int)
  extends User
```

If an extractor pattern is supposed to decompose a given data structure into more than one parameter, the signature of the extractor's `unapply` method looks like this:

```
def unapply(object: S): Option[(T1, ..., Tn)]
```

The method expects some object of type `S` and returns an `Option` of type `TupleN`, where `N` is the number of parameters to extract.

Let's adapt our extractors to the modified classes:

```
trait User {
  def name: String
  def score: Int
}
class FreeUser(
    val name: String,
    val score: Int,
    val upgradeProbability: Double)
  extends User
class PremiumUser(
    val name: String,
    val score: Int)
  extends User

object FreeUser {
  def unapply(user: FreeUser): Option[(String, Int, Double)] =
    Some((user.name, user.score, user.upgradeProbability))
}
object PremiumUser {
  def unapply(user: PremiumUser): Option[(String, Int)] =
    Some((user.name, user.score))
}
```

We can now use this extractor for pattern matching, just like we did with the previous version:

```
val user: User = new FreeUser("Daniel", 3000, 0.7d)
user match {
  case FreeUser(name, _, p) =>
    if (p > 0.75) s"$name, what can we do for you today?"
    else s"Hello $name"
  case PremiumUser(name, _) =>
    s"Welcome back, dear $name"
}
```

A Boolean extractor

Sometimes, you don't really have the need to extract parameters from a data structure against which you want to match – instead, you just want to do a simple boolean check. In this case, the third and last of the available `unapply` method signatures comes in handy, which expects a value of type `S` and returns a `Boolean`:

```
def unapply(object: S): Boolean
```

Used in a pattern, the pattern will match if the extractor returns `true`. Otherwise the next case, if available, is tried.

In the previous example, we had some logic that checks whether a free user is likely to be susceptible to being persuaded to upgrade their account. Let's place this logic in its own boolean extractor:

```
object premiumCandidate {
  def unapply(user: FreeUser): Boolean = user.upgradeProbability > 0.75
}
```

As you can see here, it is not necessary for an extractor to reside in the companion object of the class for which it is applicable. Using such a boolean extractor is as simple as this:

```
val user: User = new FreeUser("Daniel", 2500, 0.8d)
user match {
  case freeUser @ premiumCandidate() => initiateSpamProgram(freeUser)
  case _ => sendRegularNewsletter(user)
}
```

This example shows that a boolean extractor is used by just passing it an empty parameter list, which makes sense because it doesn't really extract any parameters to be bound to variables.

There is one other peculiarity in this example: I am pretending that our fictional `initiateSpamProgram` function expects an instance of `FreeUser` because premium users are never to be spammed. Our pattern matching is against any type of `User`, though, so I cannot pass `user` to the `initiateSpamProgram` function – not without ugly type casting anyway.

Luckily, Scala's pattern matching allows to bind the value that is matched to a variable, too, using the type that the used extractor expects. This is done using the `@` operator. Since our `premiumCandidate` extractor expects an instance of `FreeUser`, we have therefore bound the matched value to a variable `freeUser` of type `FreeUser`.

Personally, I haven't used boolean extractors that much, but it's good to know they exist, as sooner or later you will probably find yourself in a situation where they come in handy.

Infix operation patterns

If you followed the Scala course at Coursera, you learned that you can destructure lists and streams in a way that is akin to one of the ways you can create them, using the cons operator, `::` or `#::`, respectively:

```
1  val xs = 58 #:: 43 #:: 93 #:: Stream.empty
2  xs match {
3    case first #:: second #:: _ => first - second
4    case _ => -1
5  }
```

Maybe you have wondered why that is possible. The answer is that as an alternative to the extractor pattern notation we have seen so far, Scala also allows extractors to be used in an infix notation. So, instead of writing `e(p1, p2)`, where `e` is the extractor and `p1` and `p2` are the parameters to be extracted from a given data structure, it's always possible to write `p1 e p2`.

Hence, the *infix operation pattern* `head #:: tail` could also be written as `#::(head, tail)`, and our `PremiumUser` extractor could also be used in a pattern that reads `name PremiumUser score`. However, this is not something you would do in practice. Usage of infix operation patterns is only recommended for extractors that indeed are supposed to read like operators, which is true for the cons operators of `List` and `Stream`, but certainly not for our `PremiumUser` extractor.

A closer look at the Stream extractor

Even though there is nothing special about how the `#::` extractor can be used in pattern matching, let's take a look at it, to better understand what is going on in our pattern matching code above. Also, this is a good example of an extractor that, depending on the state of the passed in data structure, may return `None` and thus not match.

Here is the complete extractor, taken from the sources of Scala 2.9.2:

taken from scala/collection/immutable/Stream.scala, (c) 2003-2011, LAMP/EPFL

```
object #:: {
  def unapply[A](xs: Stream[A]): Option[(A, Stream[A])] =
    if (xs.isEmpty) None
    else Some((xs.head, xs.tail))
}
```

If the given `Stream` instance is empty, it just returns `None`. Thus, `case head #:: tail` will not match for an empty stream. Otherwise, a `Tuple2` is returned, the first element of which is the head of the stream, while the second element of the tuple is the tail, which is itself a `Stream` again. Hence, `case head #:: tail` will match for a stream of one or more elements. If it has only one element, `tail` will be bound to the empty stream.

To understand how this extractor works for our pattern matching example, let's rewrite that example, going from infix operation patterns to the usual extractor pattern notation:

```
val xs = 58 #:: 43 #:: 93 #:: Stream.empty
xs match {
  case #::(first, #::(second, _)) => first - second
  case _ => -1
}
```

First, the extractor is called for the intitial stream `xs` that is passed to the pattern matching block. The extractor returns `Some((xs.head, xs.tail))`, so `first` is bound to `58`, while the tail of `xs` is passed to the extractor again, which is used again inside of the first one. Again, it returns the head and and tail as a `Tuple2` wrapped in a `Some`, so that `second` is bound to the value `43`, while the tail is bound to the wildcard _ and thus thrown away.

Using extractors

So when and how should you actually make use of custom extractors, especially considering that you can get some useful extractors for free if you make use of case classes?

While some people point out that using case classes and pattern matching against them breaks encapsulation, coupling the way you match against data with its concrete representation, this criticism usually stems from an object-oriented point of view. It's a good idea, if you want to do functional programming in Scala, to use case classes as algebraic data types (ADTs)[4] that contain pure data and no behaviour whatsoever.

[4] http://en.wikipedia.org/wiki/Algebraic_data_type

Usually, implementing your own extractors is only necessary if you want to extract something from a type you have no control over, or if you need additional ways of pattern matching against certain data.

Extracting a URL

A common usage of extractors is to extract meaningful values from some string. As an exercise, think about how you would implement and use a `URLExtractor` that takes `String` representations of URLs.

Summary

In this first chapter of the book, we have examined extractors, the workhorse behind pattern matching in Scala. You have learned how to implement your own extractors and how the implementation of an extractor relates to its usage in a pattern.

We haven't covered all there is to say about extractors, though. In the next chapter, you will learn how to implement them if you you want to bind a variable number of extracted parameters in a pattern.

Extracting sequences

In the previous chapter, you learned how to implement your own extractors and how these extractors can be used for pattern matching. However, we only discussed extractors that allow you to destructure a given object into a fixed number of parameters. Yet, for certain kinds of data structures, Scala allows you to do pattern matching expecting an arbitrary number of extracted parameters.

For example, you can use a pattern that only matches a list of exactly two elements, or a list of exactly three elements:

```
val xs = 3 :: 6 :: 12 :: Nil
xs match {
  case List(a, b) => a * b
  case List(a, b, c) => a + b + c
  case _ => 0
}
```

What's more, if you want to match lists the exact length of which you don't care about, you can use a wildcard operator, `_*`:

```
val xs = 3 :: 6 :: 12 :: 24 :: Nil
xs match {
  case List(a, b, _*) => a * b
  case _ => 0
}
```

Here, the first pattern matches, binding the first two elements to the variables `a` and `b`, while simply ignoring the rest of the list, regardless how many remaining elements there are.

Clearly, extractors for these kinds of patterns cannot be implemented with the means introduced in the previous chapter. We need a way to specify that an extractor takes an object of a certain type and destructures it into a sequence of extracted values, where the length of that sequence is unknown at compile time.

Enter `unapplySeq`, an extractor method that allows for doing exactly that. Let's take a look at one of its possible method signatures:

```
def unapplySeq(object: S): Option[Seq[T]]
```

It expects an object of type `S` and returns either `None`, if the object does not match at all, or a sequence of extracted values of type `T`, wrapped in a `Some`.

Example: Extracting given names

Let's make use of this kind of extractor method in an admittedly contrived example. Let's say that in some piece of our application, we are receiving a person's given name as a `String`. This string can contain the person's second or third name, if that person has more than one given name. Hence, possible values could be `"Daniel"`, or `"Catherina Johanna"`, or `"Matthew John Michael"`. We want to be able to match against these names, extracting and binding the individual given names.

Here is a very simple extractor implementation by means of the `unapplySeq` method that will allow us to do that:

```
object GivenNames {
  def unapplySeq(name: String): Option[Seq[String]] = {
    val names = name.trim.split(" ")
    if (names.forall(_.isEmpty)) None else Some(names)
  }
}
```

Given a `String` containing one or more given names, it will extract those as a sequence. If the input name does not contain at least one given name, this extractor will return `None`, and thus, a pattern in which this extractor is used will not match such a string.

We can now put our new extractor to test:

```
def greetWithFirstName(name: String) = name match {
  case GivenNames(firstName, _*) => "Good morning, " + firstName + "!"
  case _ => "Welcome! Please make sure to fill in your name!"
}
```

This nifty little method returns a greeting for a given name, ignoring everything but the first name. `greetWithFirstName("Daniel")` will return `"Good morning, Daniel!"`, while `greetWithFirstName("Catherina Johanna")` will return `"Good morning, Catherina!"`

Combining fixed and variable parameter extraction

Sometimes, you have certain fixed values to be extracted that you know about at compile time, plus an additional optional sequence of values.

Let's assume that in our example, the input name contains the person's complete name, not only the given name. Possible values might be `"John Doe"` or `"Catherina Johanna Peterson"`. We want to be able to match against such strings using a pattern that always binds the person's last name to

the first variable in the pattern and the first name to the second variable, followed by an arbitrary number of additional given names.

This can be achieved by means of a slight modification of our `unapplySeq` method, using a different method signature:

```
def unapplySeq(object: S): Option[(T1, .., Tn-1, Seq[T])]
```

As you can see, `unapplySeq` can also return an `Option` of a `TupleN`, where the last element of the tuple must be the sequence containing the variable parts of the extracted values. This method signature should be somewhat familiar, as it is similar to one of the possible signatures of the `unapply` method that I introduced last week.

Here is an extractor making use of this:

```
object Names {
  def unapplySeq(name: String): Option[(String, String, Seq[String])] = {
    val names = name.trim.split(" ")
    if (names.size < 2) None
    else Some((names.last, names.head, names.drop(1).dropRight(1)))
  }
}
```

Have a close look at the return type and the construction of the `Some`. Our method returns an `Option` of `Tuple3`. That tuple is created with Scala's syntax for tuple literals by just putting the three elements – the last name, the first name, and the sequence of additional given names – in a pair of parentheses.

If this extractor is used in a pattern, the pattern will only match if at least a first and last name is contained in the given input string. The sequence of additional given names is created by dropping the first and the last element from the sequence of names.

We can use this extractor to implement an alternative greeting method:

```
def greet(fullName: String) = fullName match {
  case Names(lastName, firstName, _*) =>
    s"Good morning, $firstName $lastName!"
  case _ =>
    "Welcome! Please make sure to fill in your name!"
}
```

Feel free to play around with this in the REPL or a worksheet.

Summary

In this chapter, you learned how to implement and use extractors that return variable-length sequences of extracted values. Extractors are a pretty powerful mechanism. They can often be reused in flexible ways and provide a powerful way to extend the kinds of patterns you can match against.

In the next chapter, I will give an overview of the different ways in which patterns can be applied in Scala code – there is more to it than just the pattern matching you have seen in the examples so far.

Patterns everywhere

In the first two chapters of this book, we spent quite some time examining what's actually happening when you destructure an instance of a case class in a pattern, and how to write your own extractors, allowing you to destructure any types of objects in any way you desire.

Now it is time to get an overview of where patterns can actually be used in your Scala code, because so far you have only seen one of the different possible ways to make use of patterns. Here we go!

Pattern matching expressions

One place in which patterns can appear is inside of a *pattern matching expression*. This way of using patterns should be very familiar to you after attending the Scala course at Coursera and reading the previous chapters. You have some expression `e`, followed by the `match` keyword and a block, which can contain any number of cases. A case, in turn, consists of the `case` keyword followed by a pattern and, optionally, a guard clause on the left side, plus a block on the right side, to be executed if this pattern matches.

Here is a simple example, making use of patterns and, in one of the cases, a guard clause:

```
1  case class Player(name: String, score: Int)
2
3  def printMessage(player: Player) = player match {
4    case Player(_, score) if score > 100000 => println("Get a job, dude!")
5    case Player(name, _) => println(s"Hey $name, nice to see you again!")
6  }
```

The `printMessage` method has a return type of `Unit`, its sole purpose is to perform a side effect, namely printing a message. It is important to remember that you don't have to use pattern matching as you would use switch statements in languages like Java. What we are using here is called a pattern matching *expression* for a reason. Their return value is what is returned by the block belonging to the first matched pattern.

Usually, it's a good idea to take advantage of this, as it allows you to decouple two things that do not really belong together, making it easier to test your code, too. We could rewrite the example above as follows:

```
def message(player: Player) = player match {
  case Player(_, score) if score > 100000 => "Get a job, dude!"
  case Player(name, _) => s"Hey $name, nice to see you again!"
}
def printMessage(player: Player) = println(message(player))
```

Now, we have a separate `message` method whose return type is `String`. This is essentialy a pure function, returning the result of a pattern matching expression. You could also store the result of such a pattern matching expression as a value or assign it to a variable, of course.

Patterns in value definitions

Another place in which a pattern can occur in Scala is in the left side of a *value definition* (and in a *variable definition*, for that matter, but we want to write our Scala code in a functional style, so you won't see a lot of usage of variables in this book). Let's assume we have a method that returns our current player. We will use a dummy implementation that always returns the same player:

```
def currentPlayer(): Player = Player("Daniel", 3500)
```

Your usual value definition looks like this:

```
val player = currentPlayer()
doSomethingWithTheName(player.name)
```

If you know Python, you are probably familiar with a feature called *sequence unpacking*. The fact that you can use any pattern in the left side of a value definition or variable definition lets you write your Scala code in a similar style. We could change our above code and destructure the given current player while assigning it to the left side:

```
val Player(name, _) = currentPlayer()
doSomethingWithTheName(name)
```

You can do this with any pattern, but generally, it is a good idea to make sure that your pattern always matches. Otherwise, you will be the witness of an exception at runtime. For instance, the following code is problematic. `scores` is a method returning a list of scores. In our code below, this method simply returns an empty list to illustrate the problem:

```
def scores: List[Int] = List()
val best :: rest = scores
println(s"The score of our champion is $best")
```

Oops, we've got a `MatchError`. It seems like our game is not that successful after all, having no scores whatsoever.

A safe and very handy way of using patterns in this way is for destructuring case classes whose type you know at compile time. Also, when working with tuples, this makes your code a lot more readable. Let's say we have a function that returns the name of a player and their score as a tuple, not using the `Player` class we have used so far:

```
def gameResult(): (String, Int) = ("Daniel", 3500)
```

Accessing the fields of a tuple always feels very awkward:

```
val result = gameResult()
println("${result._1}: ${result._2}")
```

It's safe to destructure our tuple in the value definition, as we know we are dealing with a `Tuple2`:

```
val (name, score) = gameResult()
println(s"$name: score")
```

This is much more readable, isn't it?

Patterns in for comprehensions

Patterns also have a very valuable place in for comprehensions. For one, a for comprehension can also contain value definitions. And everything you learnt about the usage of patterns in the left side of value definitions holds true for value definitions in for comprehensions. So if we have a collection of results and want to determine the hall of fame, which in our game is simply a collection of the names of players that have trespassed a certain score threshold, we could do that in a very readable way with a for comprehension:

```
def gameResults(): Seq[(String, Int)] =
  ("Daniel", 3500) :: ("Melissa", 13000) :: ("John", 7000) :: Nil

def hallOfFame = for {
  result <- gameResults()
  (name, score) = result
  if (score > 5000)
} yield name
```

The result is `List("Melissa", "John")`, since the first player does not meet the condition of the guard clause.

This can be written even more concisely, because in for comprehensions, the left side of a *generator* is also a pattern. So, instead of first assigning each game result to `result`, we can directly destructure the result in the left side of the generator:

```
def hallOfFame = for {
  (name, score) <- gameResults()
  if (score > 5000)
} yield name
```

In this example, the pattern `(name, score)` always matches, so if it were not for the guard clause, `if (score > 5000)`, the for comprehension would be equivalent to simply mapping from the tuples to the player names, without filtering anything.

It is important to know that patterns in the left side of generators can already be used for filtering purposes – if a pattern on the left side of a generator does not match, the respective element is filtered out.

To illustrate, let's say we have a sequence of lists, and we want to return the sizes of all non-empty lists. This means we have to filter out all empty lists and then return the sizes of the ones remaining. Here is one solution:

```
val lists = List(1, 2, 3) :: List.empty :: List(5, 3) :: Nil

for {
  list @ head :: _ <- lists
} yield list.size
```

The pattern on the left side of the generator does not match for empty lists. This will not throw a `MatchError`, but result in any empty list being removed. Hence, we get back `List(3, 2)`.

Patterns and for comprehensions are a very natural and powerful combination, and if you work with Scala for some time, you will see that you'll be using them a lot.

Summary

In this chapter, you have learned about a multitude of ways you can use patterns in your Scala code, aside from the pattern matching expressions you have already been familiar with.

Another usage of patterns is for defining anonymous functions. If you have ever used a `catch` block in order to deal with an exception in Scala, then you have made used of this feature. Head on to the next chapter, Pattern Matching Anonymous Functions, to learn all about this.

Pattern matching anonymous functions

In the previous chapter, you got an overview of the various ways in which patterns can be used in Scala, concluding with a brief mention of anonymous functions as another place in which patterns can be put to use. In this chapter, we are going to take a detailed look at the possibilities opened up by being able to define anonymous functions in this way.

If you have participated in the Scala course at Coursera or have coded in Scala for a while, you will likely have written anonymous functions on a regular basis. For example, given a list of song titles which you want to transform to lower case for your search index, you might want to define an anonymous function that you pass to the `map` method, like this:

```
1  val songTitles =
2    List("The White Hare", "Childe the Hunter", "Take no Rogues")
3  songTitles.map(t => t.toLowerCase)
```

Or, if you like it even shorter, of course, you will probably write it like this, making use of Scala's placeholder syntax:

```
1  songTitles.map(_.toLowerCase)
```

So far so good. However, let's see how this syntax performs for a slightly different example: We have a sequence of pairs, each representing a word and its frequency in some text. Our goal is to filter out those pairs whose frequency is below or above a certain threshold, and then only return the remaining words, without their respective frequencies. We need to write a function `wordsWithoutOutliers(wordFrequencies: Seq[(String, Int)]): Seq[String]`.

Our initial solution makes use of the `filter` and `map` methods, passing anonymous functions to them using our familiar syntax:

```
1  val wordFrequencies = List(
2    ("habitual", 6),
3    ("and", 56),
4    ("consuetudinary", 2),
5    ("additionally", 27),
6    ("homely", 5),
7    ("society", 13))
8
9  def wordsWithoutOutliers(
```

```
10      wordFrequencies: Seq[(String, Int)]): Seq[String] =
11    wordFrequencies.filter(wf => wf._2 > 3 && wf._2 < 25).map(_._1)
12
13  wordsWithoutOutliers(wordFrequencies)
14  // result: List("habitual", "homely", "society")
```

This solution has several problems. The first one is only an aesthetic one – accessing the fields of the tuple looks pretty ugly to me. If only we could destructure the pair, we could make this code a little more pleasant and probably also more readable.

Thankfully, Scala provides an alternative way of writing anonymous functions: A *pattern matching anonymous function* is an anonymous function that is defined as a block consisting of a sequence of cases, surrounded as usual by curly braces, but without a `match` keyword before the block. Let's rewrite our function, making use of this notation:

```
1  def wordsWithoutOutliers(
2      wordFrequencies: Seq[(String, Int)]): Seq[String] =
3    wordFrequencies.filter { case (_, f) =>
4      f > 3 && f < 25 } map { case (w, _) => w }
```

In this example, we have only used a single case in each of our anonymous functions, because we know that this case always matches – we are simply decomposing a data structure whose type we already know at compile time, so nothing can go wrong here. This is a very common way of using pattern matching anonymous functions.

If you try to assign these anonymous functions to values, you will see that they have the expected type:

```
1  val predicate: (String, Int) => Boolean = { case (_, f) =>
2    f > 3 && f < 25 }
3  val transformFn: (String, Int) => String = { case (w, _) => w }
```

Please note that you have to specify the type of the value here, as the Scala compiler cannot infer it for pattern matching anonymous functions.

Nothing prevents you from defining a more complex sequence of cases, of course. However, if you define an anonymous function this way and want to pass it to some other function, such as the ones in our example, you have to make sure that for all possible inputs, one of your cases matches so that your anonymous function always returns a value. Otherwise, you will risk a `MatchError` at runtime.

Partial functions

Sometimes, however, a function that is only defined for specific input values is exactly what you want. In fact, such a function can help us get rid of another problem that we haven't solved yet with our current implementation of the `wordsWithoutOutliers` function: We first filter the given sequence and then map the remaining elements. If we can boil this down to a solution that only has to iterate over the given sequence once, this would not only need fewer CPU cycles but would also make our code shorter and, ultimately, more readable.

If you browse through Scala's collections API, you will notice a method called `collect`, which, for a `Seq[A]`, has the following signature:

```
1  def collect[B](pf: PartialFunction[A, B])
```

This method returns a new sequence by applying the given *partial function* to all of its elements – the partial function both filters and maps the sequence.

So what is a partial function? In short, it's a unary function that is known to be defined only for certain input values and that allows clients to check whether it is defined for a specific input value.

To this end, the `PartialFunction` trait provides an `isDefinedAt` method. As a matter of fact, the `PartialFunction[-A, +B]` type extends the type `(A) => B` (which can also be written as `Function1[A, B]`), and a pattern matching anonymous function is always of type `PartialFunction`.

Due to this inheritance hierarchy, passing a pattern matching anonymous function to a method that expects a `Function1`, like `map` or `filter`, is perfectly fine, as long as that function is defined for all input values, i.e. there is always a matching case.

The `collect` method, however, specifically expects a `PartialFunction[A, B]` that may not be defined for all input values and knows exactly how to deal with that case. For each element in the sequence, it first checks if the partial function is defined for it by calling `isDefinedAt` on the partial function. If this returns `false`, the element is ignored. Otherwise, the result of applying the partial function to the element is added to the result sequence.

Let's first define a partial function that we want to use for refactoring our `wordsWithoutOutliers` function to make use of `collect`:

```
1  val pf: PartialFunction[(String, Int), String] = {
2    case (word, freq) if freq > 3 && freq < 25 => word
3  }
```

We added a guard clause to our case, so that this function will not be defined for word/frequency pairs whose frequency is not within the required range.

Instead of using the syntax for pattern matching anonymous functions, we could have defined this partial function by explicitly extending the `PartialFunction` trait:

```
val pf = new PartialFunction[(String, Int), String] {
  def apply(wordFrequency: (String, Int)) = wordFrequency match {
    case (word, freq) if freq > 3 && freq < 25 => word
  }
  def isDefinedAt(wordFrequency: (String, Int)) = wordFrequency match {
    case (word, freq) if freq > 3 && freq < 25 => true
    case _ => false
  }
}
```

Usually, however, you will want to use the much more concise anonymous function syntax.

Now, if we passed our partial function to the `map` method, this would compile just fine, but result in a `MatchError` at runtime, because our partial function is not defined for all possible input values, thanks to the added guard clause:

```
  wordFrequencies.map(pf) // will throw a MatchError
```

However, we can pass this partial function to the `collect` method, and it will filter and map the sequence as expected:

```
  wordFrequencies.collect(pf) // List("habitual", "homely", "society")
```

The result of this is the same as that of our current implementation of `wordsWithoutOutliers` when passing our dummy `wordFrequencies` sequence to it. So let's rewrite that function:

```
def wordsWithoutOutliers(
    wordFrequencies: Seq[(String, Int)]): Seq[String] =
  wordFrequencies.collect {
    case (word, freq) if freq > 3 && freq < 25 => word }
```

Partial functions have some other very useful properties. For example, they provide the means to be chained, allowing for a neat functional alternative to the chain of responsibility pattern[5] known from object-oriented programming.

Partial functions are also a crucial element of many Scala libraries and APIs. For example, the way an Akka[6] actor processes messages sent to it is defined in terms of a partial function. Hence, it's quite important to know and understand this concept.

[5] http://en.wikipedia.org/wiki/Chain-of-responsibility_pattern

[6] http://akka.io/

Summary

In this chapter, we examined an alternative way of defining anonymous functions, namely as a sequence of cases, which opens up some nice destructuring possibilities in a rather concise way. Moreover, we delved into the topic of partial functions, demonstrating their usefulness by means of a simple use case.

In the next chapter, we are going to dig deeper into the ever-present `Option` type, investigating the reasoning behind its existence and how best to make use of it.

The Option type

For the last couple of chapters, we have pressed ahead and covered a lot of ground concerning some rather advanced techniques, particularly ones related to pattern matching and extractors. Time to shift down a gear and look at one of the more fundamental idiosyncrasies of Scala: the `Option` type.

If you have participated in the Scala course at Coursera, you have already received a brief introduction to this type and seen it in use in the `Map` API. In this book, we have also used it when implementing our own extractors.

And yet, there is still a lot left to be explained about it. You may have wondered what all the fuss is about, what is so much better about options than other ways of dealing with absent values. You might also be at a loss how to actually work with the `Option` type in your own code. The goal of this chapter is to do away with all these question marks and teach you all you really need to know about `Option` as an aspiring Scala novice.

The basic idea

If you have worked with Java at all in the past, it is very likely that you have come across a `NullPointerException` at some time (other languages will throw similarly named errors in such a case). Usually this happens because some method returns `null` when you were not expecting it and thus not dealing with that possibility in your client code. A value of `null` is often abused to represent an absent optional value.

Some languages treat `null` values in a special way or allow you to work safely with values that might be `null`. For instance, Groovy has the null-safe operator for accessing properties, so that `foo?.bar?.baz` will not throw an exception if either `foo` or its `bar` property is `null`, instead directly returning `null`. However, you are screwed if you forget to use this operator, and nothing forces you to do so.

Clojure basically treats its `nil` value like an empty thing, i.e. like an empty list if accessed like a list, or like an empty map if accessed like a map. This means that the `nil` value is bubbling up the call hierarchy. Very often this is okay, but sometimes this just leads to an exception much higher in the call hierchary, where some piece of code isn't that nil-friendly after all.

Scala tries to solve the problem by getting rid of `null` values altogether and providing its own type for representing optional values, i.e. values that may be present or not: the `Option[A]` trait.

`Option[A]` is a container for an optional value of type `A`. If the value of type `A` is present, the `Option[A]` is an instance of `Some[A]`, containing the present value of type `A`. If the value is absent, the `Option[A]` is the object `None`.

By stating that a value may or may not be present *on the type level*, you and any other developers who work with your code are forced by the compiler to deal with this possibility. There is no way you may accidentally rely on the presence of a value that is really optional.

`Option` is mandatory! Do not use `null` to denote that an optional value is absent.

Creating an option

Usually, you can simply create an `Option[A]` for a present value by directly instantiating the `Some` case class:

```
1  val greeting: Option[String] = Some("Hello world")
```

Or, if you know that the value is absent, you simply assign or return the `None` object:

```
1  val greeting: Option[String] = None
```

However, time and again you will need to interoperate with Java libraries or code in other JVM languages that happily make use of `null` to denote absent values. For this reason, the `Option` companion object provides a factory method that creates `None` if the given parameter is `null`, otherwise the parameter wrapped in a `Some`:

```
1  // absentGreeting will be None:
2  val absentGreeting: Option[String] = Option(null)
3  // presentGreeting will be Some("Hello!"):
4  val presentGreeting: Option[String] = Option("Hello!")
```

Working with optional values

This is all pretty neat, but how do you actually work with optional values? It's time for an example. Let's do something boring, so we can focus on the important stuff.

Imagine you are working for one of those hipsterrific startups, and one of the first things you need to implement is a repository of users. We need to be able to find a user by their unique id. Sometimes, requests come in with bogus ids. This calls for a return type of `Option[User]` for our finder method. A dummy implementation of our user repository might look like this:

```
case class User(
  id: Int,
  firstName: String,
  lastName: String,
  age: Int,
  gender: Option[String])

object UserRepository {
  private val users = Map(
    1 -> User(1, "John", "Doe", 32, Some("male")),
  2 -> User(2, "Johanna", "Doe", 30, None))
  def findById(id: Int): Option[User] = users.get(id)
  def findAll = users.values
}
```

Now, if you received an instance of `Option[User]` from the `UserRepository` and need to do something with it, how do you do that?

One way would be to check if a value is present by means of the `isDefined` method of your option, and, if that is the case, get that value via its `get` method:

```
val user1 = UserRepository.findById(1)
if (user1.isDefined) {
  println(user1.get.firstName)
} // will print "John"
```

This is very similar to how the `Optional` type in the Guava library is used in Java. If you think this is clunky and expect something more elegant from Scala, you're on the right track. More importantly, if you use `get`, you might forget about checking with `isDefined` before, leading to an exception at runtime, so you haven't gained a lot over using `null`.

You should stay away from this way of accessing options whenever possible!

Providing a default value

Very often, you want to work with a fallback or default value in case an optional value is absent. This use case is covered pretty well by the `getOrElse` method defined on `Option`:

```
val user = User(2, "Johanna", "Doe", 30, None)
// will print "not specified":
println("Gender: " + user.gender.getOrElse("not specified"))
```

Please note that the default value you can specify as a parameter to the `getOrElse` method is a by-name parameter, which means that it is only evaluated if the option on which you invoke `getOrElse` is indeed `None`. Hence, there is no need to worry if creating the default value is costly for some reason or another – this will only happen if the default value is actually required.

Pattern matching

`Some` is a case class, so it is perfectly possible to use it in a pattern, be it in a regular pattern matching expression or in some other place where patterns are allowed. Let's rewrite the example above using pattern matching:

```
1  val user = User(2, "Johanna", "Doe", 30, None)
2  user.gender match {
3    case Some(gender) => println(s"Gender: $gender")
4    case None => println("Gender: not specified")
5  }
```

Or, if you want to remove the duplicated `println` statement and make use of the fact that you are working with a pattern matching *expression*:

```
1  val user = User(2, "Johanna", "Doe", 30, None)
2  val gender = user.gender match {
3    case Some(gender) => gender
4    case None => "not specified"
5  }
6  println(s"Gender: $gender")
```

You will hopefully have noticed that pattern matching on an `Option` instance is rather verbose, which is also why it is usually not idiomatic to process options this way. So, even if you are all excited about pattern matching, try to use the alternatives when working with options.

There is one quite elegant way of using patterns with options, which you will learn about in the section on for comprehensions, below.

Options can be viewed as collections

So far you haven't seen a lot of elegant or idiomatic ways of working with options. We are coming to that now.

I already mentioned that `Option[A]` is a container for a value of type `A`. More precisely, you may think of it as some kind of collection – some special snowflake of a collection that contains either zero elements or exactly one element of type `A`. This is a very powerful idea!

Even though on the type level, `Option` is not a collection type in Scala, options come with all the goodness you have come to appreciate about Scala collections like `List`, `Set` etc – and if you really need to, you can even transform an option into a `List`, for instance.

So what does this allow you to do?

Performing a side-effect if a value is present

If you need to perform some side-effect only if a specific optional value is present, the `foreach` method you know from Scala's collections comes in handy:

```
1  // prints "Johanna":
2  UserRepository.findById(2).foreach(user => println(user.firstName))
```

The function passed to `foreach` will be called exactly once, if the `Option` is a `Some`, or never, if it is `None`.

Mapping an option

The really good thing about options behaving like a collection is that you can work with them in a very functional way, and the way you do that is exactly the same as for lists, sets etc.

Just as you can map a `List[A]` to a `List[B]`, you can map an `Option[A]` to an `Option[B]`. This means that if your instance of `Option[A]` is defined, i.e. it is `Some[A]`, the result is `Some[B]`, otherwise it is `None`.

If you compare `Option` to `List`, `None` is the equivalent of an empty list: when you map an empty `List[A]`, you get an empty `List[B]`, and when you map an `Option[A]` that is `None`, you get an `Option[B]` that is `None`.

Let's get the age of an optional user:

```
1  val age = UserRepository.findById(1).map(_.age) // age is Some(32)
```

flatMap and options

Let's do the same for the gender:

```
val gender = UserRepository.findById(1).map(_.gender)
```

The type of the resulting `gender` is `Option[Option[String]]`. Why is that?

Think of it like this: You have an `Option` container for a `User`, and *inside* that container you are mapping the `User` instance to an `Option[String]`, since that is the type of the `gender` property on our `User` class.

These nested options are a nuisance? Why, no problem, like all collections, `Option` also provides a `flatMap` method. Just like you can flatten a `List[List[A]]`, you can do the same for an `Option[Option[A]]`:

```
// gender1 is Some("male"):
val gender1 = UserRepository.findById(1).flatMap(_.gender)
// gender2 is None
val gender2 = UserRepository.findById(2).flatMap(_.gender)
// gender3 is None
val gender3 = UserRepository.findById(3).flatMap(_.gender)
```

The result type is now `Option[String]`. If the user is defined *and* its gender is defined, we get it as a flattened `Some`. If either the use or its gender is undefined, we get a `None`.

To understand how this works, let's have a look at what happens when flat mapping a list of lists of strings, always keeping in mind that an `Option` is just a collection, too, like a `List`:

```
val names: List[List[String]] =
  List(
    List("John", "Johanna", "Daniel"),
    List(),
    List("Doe", "Westheide"))
names.map(_.map(_.toUpperCase))
// results in:
// List(
//   List("JOHN", "JOHANNA", "DANIEL"), List(), List("DOE", "WESTHEIDE"))
names.flatMap(_.map(_.toUpperCase))
// results in List("JOHN", "JOHANNA", "DANIEL", "DOE", "WESTHEIDE")
```

If we use `flatMap`, the mapped elements of the inner lists are converted into a single flat list of strings. Obviously, nothing will remain of any empty inner lists.

To lead us back to the `Option` type, consider what happens if you map a list of options of strings:

```
val names: List[Option[String]] =
  List(Some("Johanna"), None, Some("Daniel"))
names.map(_.map(_.toUpperCase))
// result: List(Some("JOHANNA"), None, Some("DANIEL"))
names.flatMap(xs => xs.map(_.toUpperCase))
// result: List("JOHANNA", "DANIEL")
```

If you just map over the list of options, the result type stays `List[Option[String]]`. Using `flatMap`, all elements of the inner collections are put into a flat list: The one element of any `Some[String]` in the original list is unwrapped and put into the result list, whereas any `None` value in the original list does not contain any element to be unwrapped. Hence, `None` values are effectively filtered out.

With this in mind, have a look again at what `flatMap` does on the `Option` type.

Filtering an option

You can filter an option just like you can filter a list. If the instance of `Option[A]` is defined, i.e. it is a `Some[A]`, *and* the predicate passed to `filter` returns `true` for the wrapped value of type `A`, the `Some` instance is returned. If the `Option` instance is already `None` or the predicate returns `false` for the value inside the `Some`, the result is `None`:

```
// None, because age is <= 30:
UserRepository.findById(1).filter(_.age > 30)
// Some(user), because age is > 30:
UserRepository.findById(2).filter(_.age > 30)
// None, because user is already None:
UserRepository.findById(3).filter(_.age > 30)
```

For comprehensions

Now that you know that an `Option` can be treated as a collection and provides `map`, `flatMap`, `filter` and other methods you know from collections, you will probably already suspect that options can be used in *for comprehensions*. Often, this is the most readable way of working with options, especially if you have to chain a lot of `map`, `flatMap` and `filter` invocations. If it's just a single `map`, that may often be preferrable, as it is a little less verbose.

If we want to get the gender for a single user, we can apply the following for comprehension:

```
for {
  user <- UserRepository.findById(1)
  gender <- user.gender
} yield gender // results in Some("male")
```

As you may know from working with lists, this is equivalent to nested invocations of `flatMap`. If the `UserRepository` already returns `None` or the `Gender` is `None`, the result of the for comprehension is `None`. For the user in the example, a gender is defined, so it is returned in a `Some`.

If we wanted to retrieve the genders of all users that have specified it, we could iterate all users, and for each of them yield a gender, if it is defined:

```
for {
  user <- UserRepository.findAll
  gender <- user.gender
} yield gender
```

Since we are effectively flat mapping, the result type is `List[String]`, and the resulting list is `List("male")`, because `gender` is only defined for the first user.

Usage in the left side of a generator

Maybe you remember from chapter three of this book that the left side of a generator in a for comprehension is a pattern. This means that you can also use patterns involving options in for comprehensions.

We could rewrite the previous example as follows:

```
for {
  User(_, _, _, _, Some(gender)) <- UserRepository.findAll
} yield gender
```

Using a `Some` pattern in the left side of a generator has the effect of removing all elements from the result collection for which the respective value is `None`.

Chaining options

Options can also be chained, which is a little similar to chaining partial functions. To do this, you call `orElse` on an `Option` instance, and pass in another `Option` instance as a by-name parameter. If the former is `None`, `orElse` returns the option passed to it, otherwise it returns the one on which it was called.

A good use case for this is finding a resource, when you have several different locations to search for it and an order of preference. In our example, we prefer the resource to be found in the config dir, so we call `orElse` on it, passing in an alternative option:

```
case class Resource(content: String)
val resourceFromConfigDir: Option[Resource] = None
val resourceFromClasspath: Option[Resource] =
  Some(Resource("I was found on the classpath"))
val resource = resourceFromConfigDir orElse resourceFromClasspath
```

This is usually a good fit if you want to chain more than just two options – if you simply want to provide a default value in case a given option is absent, the `getOrElse` method may be a better idea.

Summary

In this chapter, you have received a thorough overview of everything you need to know about the `Option` type in order to use it for your benefit, to understand other people's Scala code and write more readable, functional code.

The most important insight to take away from this chapter is that there is a very basic idea that is common to lists, sets, maps, options, and, as you will see in the following chapters, other data types, and that there is a uniform way of using these types, which is both elegant and very powerful.

In the following chapter, you are going to learn about idiomatic, functional error handling in Scala.

Error handling with Try

When just playing around with a new language, you might get away with simply ignoring the fact that something might go wrong. As soon you want to create anything serious, though, you can no longer run away from handling errors and exceptions in your code. The importance of how well a language supports you in doing so is often underestimated, for some reason or another.

Scala, as it turns out, is pretty well positioned when it comes to dealing with error conditions in an elegant way. In this chapter, I'm going to present Scala's approach to dealing with errors, based on the `Try` type, and the rationale behind it.

Throwing and catching exceptions

Before going straight to Scala's idiomatic approach at error handling, let's first have a look at an approach that is more akin to how you are used to working with error conditions if you come from languages like Java or Ruby. Like these languages, Scala allows you to throw an exception:

```
1  case class Customer(age: Int)
2  class Cigarettes
3  case class UnderAgeException(message: String) extends Exception(message)
4  def buyCigarettes(customer: Customer): Cigarettes =
5    if (customer.age < 16) throw UnderAgeException(
6      s"Customer must be older than 16 but was ${customer.age}")
7    else new Cigarettes
```

Thrown exceptions can be caught and dealt with very similarly to Java, albeit using a partial function to specify the exceptions we want to deal with. Also, Scala's `try/catch` is an expression, so the following code returns the message of the exception:

```
1  val youngCustomer = Customer(15)
2  try {
3    buyCigarettes(youngCustomer)
4    "Yo, here are your cancer sticks! Happy smokin'!"
5  } catch {
6      case UnderAgeException(msg) => msg
7  }
```

Error handling, the functional way

Now, having this kind of exception handling code all over your code base can become ugly very quickly and doesn't really go well with functional programming. It's also a rather bad solution for applications with a lot of concurrency. For instance, if you need to deal with an exception thrown by an Actor that is executed on some other thread, you obviously cannot do that by catching that exception – you will want a possibility to receive a message denoting the error condition.

Hence, in Scala, it's usually preferred to signify that an error has occurred by returning an appropriate value from your function.

Don't worry, we are not going back to C-style error handling, using error codes that we need to check for by convention. Rather, in Scala, we are using a specific type that represents computations that may result in an exception.

In this chapter, we are confining ourselves to the `Try` type that was introduced in Scala 2.10. There is also a similar type, called `Either`, which, even after the introduction of `Try`, can still be very useful, but is more general. An alternative that does not ship with the standard library is `Validation` from the Scalaz[7] project.

The semantics of Try

The semantics of `Try` are best explained by comparing them to those of the `Option` type that was the topic of the previous chapter.

Where `Option[A]` is a container for a value of type `A` that may be present or not, `Try[A]` represents a computation that may result in a value of type `A`, if it is successful, or in some `Throwable` if something has gone wrong. Instances of such a container type for possible errors can easily be passed around between concurrently executing parts of your application.

There are two different types of `Try`: If an instance of `Try[A]` represents a successful computation, it is an instance of `Success[A]`, simply wrapping a value of type `A`. If, on the other hand, it represents a computation in which an error has occurred, it is an instance of `Failure[A]`, wrapping a `Throwable`, i.e. an exception or other kind of error.

If we know that a computation may result in an error, we can simply use `Try[A]` as the return type of our function. This makes the possibility explicit and forces clients of our function to deal with the possibility of an error in some way.

For example, let's assume we want to write a simple web page fetcher. The user will be able to enter the URL of the web page they want to fetch. One part of our application will be a function that parses the entered URL and creates a `java.net.URL` from it:

[7] https://github.com/scalaz/scalaz

```
import scala.util.Try
import java.net.URL
def parseURL(url: String): Try[URL] = Try(new URL(url))
```

As you can see, we return a value of type `Try[URL]`. If the given `url` is syntactically correct, this will be a `Success[URL]`. If the `URL` constructor throws a `MalformedURLException`, however, it will be a `Failure[URL]`.

To achieve this, we are using the `apply` factory method on the `Try` companion object. This method expects a by-name parameter of type `A` (here, `URL`). For our example, this means that the `new URL(url)` is executed inside the `apply` method of the `Try` object. Inside that method, non-fatal exceptions are caught, returning a `Failure` containing the respective exception.

Hence, `parseURL("http://danielwestheide.com")` will result in a `Success[URL]` containing the created URL, whereas `parseURL("garbage")` will result in a `Failure[URL]` containing a `MalformedURLException`.

Working with Try values

Working with `Try` instances is actually very similar to working with `Option` values, so you won't see many surprises here.

You can check if a `Try` is a success by calling `isSuccess` on it and then conditionally retrieve the wrapped value by calling `get` on it. But believe me, there aren't many situations where you will want to do that.

It's also possible to use `getOrElse` to pass in a default value to be returned if the `Try` is a `Failure`:

```
val url = parseURL(Console.readLine("URL: ")).getOrElse(
  new URL("http://duckduckgo.com"))
```

If the URL given by the user is malformed, we use the URL of DuckDuckGo as a fallback.

Chaining operations

One of the most important characteristics of the `Try` type is that, like `Option`, it supports all the higher-order methods you know from other types of collections. As you will see in the examples to follow, this allows you to chain operations on `Try` values and catch any exceptions that might occur, and all that in a very readable manner.

Mapping and flat mapping

Mapping a `Try[A]` that is a `Success[A]` to a `Try[B]` results in a `Success[B]`. If it's a `Failure[A]`, the resulting `Try[B]` will be a `Failure[B]`, on the other hand, containing the same exception as the `Failure[A]`:

```
parseURL("http://danielwestheide.com").map(_.getProtocol)
// results in Success("http")
parseURL("garbage").map(_.getProtocol)
// results in Failure(java.net.MalformedURLException: no protocol: garbage)
```

If you chain multiple `map` operations, this will result in a nested `Try` structure, which is usually not what you want. Consider this method that returns an input stream for a given URL:

```
import java.io.InputStream
def inputStreamForURL(url: String): Try[Try[Try[InputStream]]] =
  parseURL(url).map { u =>
    Try(u.openConnection()).map(conn => Try(conn.getInputStream))
  }
```

Since the anonymous functions passed to the two `map` calls each return a `Try`, the return type is a `Try[Try[Try[InputStream]]]`.

This is where the fact that you can `flatMap` a `Try` comes in handy. The `flatMap` method on a `Try[A]` expects to be passed a function that receives an `A` and returns a `Try[B]`. If our `Try[A]` instance is already a `Failure[A]`, that failure is returned as a `Failure[B]`, simply passing along the wrapped exception along the chain. If our `Try[A]` is a `Success[A]`, `flatMap` unpacks the `A` value in it and maps it to a `Try[B]` by passing this value to the mapping function.

This means that we can basically create a pipeline of operations that require the values carried over in `Success` instances by chaining an arbitrary number of `flatMap` calls. Any exceptions that happen along the way are wrapped in a `Failure`, which means that the end result of the chain of operations is a `Failure`, too.

Let's rewrite the `inputStreamForURL` method from the previous example, this time resorting to `flatMap`:

```
def inputStreamForURL(url: String): Try[InputStream] =
  parseURL(url).flatMap { u =>
    Try(u.openConnection()).flatMap(conn => Try(conn.getInputStream))
  }
```

Now we get a `Try[InputStream]`, which can be a `Failure` wrapping an exception from any of the stages in which one may be thrown, or a `Success` that directly wraps the `InputStream`, the final result of our chain of operations.

Filter and foreach

Of course, you can also filter a `Try` or call `foreach` on it. Both work exactly as you would except after having learned about `Option`.

The `filter` method returns a `Failure` if the `Try` on which it is called is already a `Failure` or if the predicate passed to it returns `false` (in which case the wrapped exception is a `NoSuchElementException`). If the `Try` on which it is called is a `Success` and the predicate returns `true`, that `Succcess` instance is returned unchanged:

```
1  def parseHttpURL(url: String) =
2    parseURL(url).filter(_.getProtocol == "http")
3  // results in a Success[URL]:
4  parseHttpURL("http://apache.openmirror.de")
5  // results in a Failure[URL]:
6  parseHttpURL("ftp://mirror.netcologne.de/apache.org")
```

The function passed to `foreach` is executed only if the `Try` is a `Success`, which allows you to execute a side-effect. The function passed to `foreach` is executed exactly once in that case, being passed the value wrapped by the `Success`:

```
1  parseHttpURL("http://danielwestheide.com").foreach(println)
```

For comprehensions

The support for `flatMap`, `map` and `filter` means that you can also use for comprehensions in order to chain operations on `Try` instances. Usually, this results in more readable code. To demonstrate this, let's implement a method that returns the content of a web page with a given URL using for comprehensions.

```
1  import scala.io.Source
2  def getURLContent(url: String): Try[Iterator[String]] =
3    for {
4      url <- parseURL(url)
5      connection <- Try(url.openConnection())
6      is <- Try(connection.getInputStream)
7      source = Source.fromInputStream(is)
8    } yield source.getLines()
```

There are three places where things can go wrong, all of them covered by usage of the `Try` type. First, the already implemented `parseURL` method returns a `Try[URL]`. Only if this is a `Success[URL]`, we will try to open a connection and create a new input stream from it. If opening the connection

and creating the input stream succeeds, we continue, finally yielding the lines of the web page. Since we effectively chain multiple `flatMap` calls in this for comprehension, the result type is a flat `Try[Iterator[String]]`.

Please note that this could be simplified using `Source#fromURL` and that we fail to close our input stream at the end, both of which are due to the necessity to keep the example focussed on getting across the subject matter at hand.

Pattern Matching

At some point in your code, you will often want to know whether a `Try` instance you have received as the result of some computation represents a success or not and execute different code branches depending on the result. Usually, this is where you will make use of pattern matching. This is easily possible because both `Success` and `Failure` are case classes.

We want to render the requested page if it could be retrieved, or print an error message if that was not possible:

```
1 import scala.util.Success
2 import scala.util.Failure
3 getURLContent("http://danielwestheide.com/foobar") match {
4   case Success(lines) =>
5     lines.foreach(println)
6   case Failure(ex) =>
7     println(s"Problem rendering URL content: ${ex.getMessage}")
8 }
```

Recovering from a Failure

If you want to establish some kind of default behaviour in the case of a `Failure`, you don't have to use `getOrElse`. An alternative is `recover`, which expects a partial function and returns another `Try`. If `recover` is called on a `Success` instance, that instance is returned as is. Otherwise, if the partial function is defined for the given `Failure` instance, its result is returned as a `Success`.

Let's put this to use in order to print a different message depending on the type of the wrapped exception:

```
import java.net.MalformedURLException
import java.io.FileNotFoundException
val content = getURLContent("garbage") recover {
  case e: FileNotFoundException =>
    Iterator("Requested page does not exist")
  case e: MalformedURLException =>
    Iterator("Please make sure to enter a valid URL")
  case _ =>
    Iterator("An unexpected error has occurred. We are so sorry!")
}
```

We could now safely `get` the wrapped value on the `Try[Iterator[String]]` that we assigned to `content`, because we know that it must be a `Success`. Calling `content.get.foreach(println)` would result in `Please make sure to enter a valid URL` being printed to the console.

Summary

Idiomatic error handling in Scala is quite different from the paradigm known from languages like Java or Ruby. The `Try` type allows you to encapsulate computations that result in errors in a container and to chain operations on the computed values in a very elegant way. You can transfer what you know from working with collections and with `Option` values to how you deal with code that may result in errors – all in a uniform way.

We haven't explored all of the methods available on `Try` in this chapter. Like `Option`, `Try` supports the `orElse` method. The `transform` and `recoverWith` methods are also worth having a look at, and I encourage you to do so.

In the next chapter, we are going to deal with `Either`, an alternative type for representing computations that may result in errors, but with a wider scope of application that goes beyond error handling.

The Either type

In the previous chapter, we had a close look at functional error handling using Try, which was introduced in Scala 2.10, mentioning the existence of another, somewhat similar type called `Either`. The latter is what this chapter is about. You will learn how to use it, when to use it, and what its particular pitfalls are.

Speaking of which, at least at the time of this writing, `Either` has some serious design flaws you need to be aware of, so much so that one might argue about whether to use it at all. So why then should you learn about `Either` at all?

For one, people will not all migrate their existing code bases to use `Try` for dealing with exceptions, so it is good to be able to understand the intricacies of this type, too.

Moreover, `Try` is not really an all-out replacement for `Either`, only for one particular usage of it, namely handling exceptions in a functional way. As it stands, `Try` and `Either` really complement each other, each covering different use cases. And, as flawed as `Either` may be, in certain situations it will still be a very good fit.

The semantics

Like `Option` and `Try`, `Either` is a container type. Unlike the aforementioned types, it takes not only one, but two type parameters: An `Either[A, B]` instance can contain either an instance of `A`, or an instance of `B`. This is different from a `Tuple2[A, B]`, which contains both an `A` and a `B` instance.

`Either` has exactly two sub types, `Left` and `Right`. If an `Either[A, B]` object contains an instance of `A`, then the `Either` is a `Left`. Otherwise it contains an instance of `B` and is a `Right`.

There is nothing in the semantics of this type that specifies one or the other sub type to represent an error or a success, respectively. In fact, `Either` is a general-purpose type for use whenever you need to deal with situations where the result can be of one of two possible types. Nevertheless, error handling is a popular use case for it, and by convention, when using it that way, the `Left` represents the error case, whereas the `Right` contains the success value.

Creating an Either

Creating an instance of `Either` is trivial. Both `Left` and `Right` are case classes, so if we want to implement a rock-solid internet censorship feature, we can just do the following:

```
import scala.io.Source
import java.net.URL
def getContent(url: URL): Either[String, Source] =
  if (url.getHost.contains("google"))
    Left("Requested URL is blocked for the good of the people!")
  else
    Right(Source.fromURL(url))
```

Now, if we call `getContent(new URL("http://danielwestheide.com"))`, we will get a `scala.io.Source` wrapped in a `Right`. If we pass in `new URL("https://plus.google.com")`, the return value will be a `Left` containing a `String`.

Working with Either values

Some of the very basic stuff works just as you know from `Option` or `Try`: You can ask an instance of `Either` if it `isLeft` or `isRight`. You can also do pattern matching on it, which is one of the most familiar and convenient ways of working with objects of this type:

```
getContent(new URL("http://google.com")) match {
  case Left(msg) => println(msg)
  case Right(source) => source.getLines.foreach(println)
}
```

Projections

You *cannot*, at least not directly, use an `Either` instance like a collection, the way you are familiar with from `Option` and `Try`. This is because `Either` is designed to be *unbiased*.

`Try` is *success-biased*: it offers you `map`, `flatMap` and other methods that all work under the assumption that the `Try` is a `Success`, and if that's not the case, they effectively don't do anything, returning the `Failure` as-is.

The fact that `Either` is unbiased means that you first have to choose whether you want to work under the assumption that it is a `Left` or a `Right`. By calling `left` or `right` on an `Either` value, you get a `LeftProjection` or `RightProjection`, respectively, which are basically left- or right-biased wrappers for the `Either`.

Mapping

Once you have a projection, you can call `map` on it:

```
val content: Either[String, Iterator[String]] =
  getContent(new URL("http://danielwestheide.com")).right.map(_.getLines())
// content is a Right containing the lines from the Source
// returned by getContent
val moreContent: Either[String, Iterator[String]] =
  getContent(new URL("http://google.com")).right.map(_.getLines)
// moreContent is a Left, as already returned by getContent
```

Regardless of whether the `Either[String, Source]` in this example is a `Left` or a `Right`, it will be mapped to an `Either[String, Iterator[String]]`. If it's called on a `Right`, the value inside it will be transformed. If it's a `Left`, that will be returned unchanged.

We can do the same with a `LeftProjection`, of course:

```
val content: Either[Iterator[String], Source] =
  getContent(new URL("http://danielwestheide.com")).left.map(Iterator(_))
// content is the Right containing a Source, as already
// returned by getContent
val moreContent: Either[Iterator[String], Source] =
  getContent(new URL("http://google.com")).left.map(Iterator(_))
// moreContent is a Left containing the msg returned by getContent
// in an Iterator
```

Now, if the `Either` is a `Left`, its wrapped value is transformed, whereas a `Right` would be returned unchanged. Either way, the result is of type `Either[Iterator[String], Source]`.

Please note that the `map` method is defined on the projection types, not on `Either`, but it does return a value of type `Either`, not a projection. In this, `Either` deviates from the other container types you know. The reason for this has to do with `Either` being unbiased, but as you will see, this can lead to some very unpleasant problems in certain cases. It also means that if you want to chain multiple calls to `map`, `flatMap` and the like, you always have to ask for your desired projection again in between.

Flat mapping

Projections also support flat mapping, avoiding the common problem of creating a convoluted structure of multiple inner and outer `Either` types that you will end up with if you nest multiple calls to `map`.

I'm putting very high requirements on your suspension of disbelief now, coming up with a completely contrived example. Let's say we want to calculate the average number of lines of two of my blog articles. You've always wanted to do that, right? Here's how we could solve this challenging problem:

```
val part5 = new URL("http://t.co/UR1aalX4")
val part6 = new URL("http://t.co/6wlKwTmu")
val content = getContent(part5).right.map(a =>
  getContent(part6).right.map(b =>
    (a.getLines().size + b.getLines().size) / 2))
```

What we'll end up with is an `Either[String, Either[String, Int]]`. Now, `content` being a nested structure of `Rights`, we could flatten it by calling the `joinRight` method on it (you also have `joinLeft` available to flatten a nested structure of `Lefts`).

However, we can avoid creating this nested structure altogether. If we `flatMap` on our outer `RightProjection`, we get a more pleasant result type, unpacking the `Right` of the inner `Either`:

```
val content = getContent(part5).right.flatMap(a =>
  getContent(part6).right.map(b =>
    (a.getLines().size + b.getLines().size) / 2))
```

Now `content` is a flat `Either[String, Int]`, which makes it a lot nicer to work with, for example using pattern matching.

For comprehensions

By now, you have probably learned to love working with for comprehensions in a consistent way on various different data types. You can do that, too, with projections of `Either`, but the sad truth is that it's not quite as nice, and there are things that you won't be able to do without resorting to ugly workarounds, out of the box.

Let's rewrite our `flatMap` example, making use of for comprehensions instead:

```
def averageLineCount(url1: URL, url2: URL): Either[String, Int] =
  for {
    source1 <- getContent(url1).right
    source2 <- getContent(url2).right
  } yield (source1.getLines().size + source2.getLines().size) / 2
```

This is not too bad. Note that we have to call `right` on each `Either` we use in our generators, of course.

Now, let's try to refactor this for comprehension – since the yield expression is a little too involved, we want to extract some parts of it into value definitions inside our for comprehension:

```
def averageLineCountWontCompile(
    url1: URL, url2: URL): Either[String, Int] =
  for {
    source1 <- getContent(url1).right
    source2 <- getContent(url2).right
    lines1 = source1.getLines().size
    lines2 = source2.getLines().size
  } yield (lines1 + lines2) / 2
```

This won't compile! The reason will become clearer if we examine what this for comprehension corresponds to, if you take away the sugar. It translates to something that is similar to the following, albeit much less readable:

```
def averageLineCountDesugaredWontCompile(
    url1: URL, url2: URL): Either[String, Int] =
  getContent(url1).right.flatMap { source1 =>
    getContent(url2).right.map { source2 =>
      val lines1 = source1.getLines().size
      val lines2 = source2.getLines().size
      (lines1, lines2)
    }.map { case (x, y) => x + y / 2 }
  }
```

The problem is that by including a value definition in our for comprehension, a new call to `map` is introduced automatically – on the result of the previous call to `map`, which has returned an `Either`, not a `RightProjection`. As you know, `Either` doesn't define a `map` method, making the compiler a little bit grumpy.

This is where `Either` shows us its ugly trollface. In this example, the value definitions are not strictly necessary. If they are, you can work around this problem, replacing any value definitions by generators, like this:

```
def averageLineCount(url1: URL, url2: URL): Either[String, Int] =
  for {
    source1 <- getContent(url1).right
    source2 <- getContent(url2).right
    lines1 <- Right(source1.getLines().size).right
    lines2 <- Right(source2.getLines().size).right
  } yield (lines1 + lines2) / 2
```

It's important to be aware of these design flaws. They don't make `Either` unusable, but can lead to serious headaches if you don't have a clue what's going on.

Other methods

The projection types have some other useful methods:

You can convert your `Either` instance to an `Option` by calling `toOption` on one of its projections. For example, if you have an `e` of type `Either[A, B]`, `e.right.toOption` will return an `Option[B]`. If your `Either[A, B]` instance is a `Right`, that `Option[B]` will be a `Some`. If it's a `Left`, it will be `None`. The reverse behaviour can be achieved, of course, when calling `toOption` on the `LeftProjection` of your `Either[A, B]`. If you need a sequence of either one value or none, use `toSeq` instead.

Folding

If you want to transform an `Either` value regardless of whether it is a `Left` or a `Right`, you can do so by means of the `fold` method that is defined on `Either`, expecting two transform functions with the same result type, the first one being called if the `Either` is a `Left`, the second one if it's a `Right`.

To demonstrate this, let's combine the two mapping operations we implemented on the `LeftProjection` and the `RightProjection` above:

```
1 val content: Iterator[String] =
2   getContent(new URL("http://danielwestheide.com")).fold(
3     Iterator(_), _.getLines())
4 val moreContent: Iterator[String] =
5   getContent(new URL("http://google.com")).fold(
6     Iterator(_), _.getLines())
```

In this example, we are transforming our `Either[String, Source]` into an `Iterator[String]`, no matter if it's a `Left` or a `Right`. You could just as well return a new `Either` again or execute side-effects and return `Unit` from your two functions. As such, calling `fold` provides a nice alternative to pattern matching.

When to use Either

Now that you have seen how to work with `Either` values and what you have to take care of, let's move on to some specific use cases.

Error handling

You *can* use `Either` for exception handling very much like `Try`. `Either` has one advantage over `Try`: you can have more specific error types at compile time, while `Try` uses `Throwable` all the time. This means that `Either` can be a good choice for expected errors.

You'd have to implement a method like this, delegating to the very useful `Exception` object from the `scala.util.control` package:

```
import scala.util.control.Exception.catching
def handling[Ex <: Throwable, T](
    exType: Class[Ex])(block: => T): Either[Ex, T] =
  catching(exType).either(block).asInstanceOf[Either[Ex, T]]
```

The reason you might want to do that is because while the methods provided by `scala.util.Exception` allow you to catch only certain types of exceptions, the resulting compile-time error type is always `Throwable`.

With this method at hand, you can pass along expected exceptions in an `Either`:

```
import java.net.MalformedURLException
def parseURL(url: String): Either[MalformedURLException, URL] =
  handling(classOf[MalformedURLException])(new URL(url))
```

You will have other expected error conditions, and not all of them result in third-party code throwing an exception you need to handle, as in the example above. In these cases, there is really no need to throw an exception yourself, only to catch it and wrap it in a `Left`. Instead, simply define your own error type, preferably as a case class, and return a `Left` wrapping an instance of that error type in your expected error condition occurs.

Here is an example:

```
case class Customer(age: Int)
class Cigarettes
case class UnderAgeFailure(age: Int, required: Int)
def buyCigarettes(
    customer: Customer): Either[UnderAgeFailure, Cigarettes] =
  if (customer.age < 16) Left(UnderAgeFailure(customer.age, 16))
  else Right(new Cigarettes)
```

You should avoid using `Either` for wrapping unexpected exceptions. `Try` does that better, without all the flaws you have to deal with when working with `Either`.

Processing collections

Generally, `Either` is a pretty good fit if you want to process a collection, where for some items in that collection, this might result in a condition that is problematic, but should not directly result in an exception, which would result in aborting the processing of the rest of the collection.

Let's assume that for our industry-standard web censorship system, we are using some kind of black list:

```
type Citizen = String
case class BlackListedResource(url: URL, visitors: Set[Citizen])

val blacklist = List(
  BlackListedResource(
    new URL("https://google.com"), Set("John Doe", "Johanna Doe")),
  BlackListedResource(
    new URL("http://yahoo.com"), Set.empty),
  BlackListedResource(
    new URL("https://maps.google.com"), Set("John Doe")),
  BlackListedResource(
    new URL("http://plus.google.com"), Set.empty)
)
```

A `BlackListedResource` represents the URL of a black-listed web page plus the citizens who have tried to visit that page.

Now we want to process this black list, where our main purpose is to identify problematic citizens, i.e. those that have tried to visit blocked pages. At the same time, we want to identify suspicous web pages – if not a single citizen has tried to visit a black-listed page, we must assume that our subjects are bypassing our filter somehow, and we need to investigate that.

Here is how we can process our black list:

```
val checkedBlacklist: List[Either[URL, Set[Citizen]]] =
  blacklist.map(resource =>
    if (resource.visitors.isEmpty) Left(resource.url)
    else Right(resource.visitors))
```

We have created a sequence of `Either` values, with the `Left` instances representing suspicious URLs and the `Right` ones containing sets of problem citizens. This makes it almost a breeze to identify both our problem citizens and our suspicious web pages:

```
val suspiciousResources =
  checkedBlacklist.flatMap(_.left.toOption)
val problemCitizens =
  checkedBlacklist.flatMap(_.right.toOption).flatten.toSet
```

These more general use cases beyond exception handling are where `Either` really shines.

Summary

You have learned how to make use of `Either`, what its pitfalls are, and when to put it to use in your code. It's a type that is not without flaws, and whether you want to have to deal with them and incorporate it in your own code is ultimately up to you.

In practice, you will notice that, now that we have `Try` at our hands, there won't be terribly many use cases for it. Also, many people prefer Scalaz[8] `Validations` for dealing with errors. Nevertheless, it's good to know about it, both for those situations where it will be your perfect tool and to understand pre-2.10 Scala code you come across where it's used for error handling.

[8] https://github.com/scalaz/scalaz

Welcome to the Future

As an aspiring and enthusiastic Scala developer, you will likely have heard of Scala's approach at dealing with concurrency – or maybe that was even what attracted you in the first place. Said approach makes reasoning about concurrency and writing well-behaved concurrent programs a lot easier than the rather low-level concurrency APIs you are confronted with in most other languages.

One of the two cornerstones of this approach is the *Future*, the other being the *Actor*. The former shall be the subject of this chapter. I will explain what futures are good for and how you can make use of them in a functional way.

Please make sure that you have version 2.10.0 or later if you want to get your hands dirty and try out the examples yourself. The futures we are discussing here were only incorporated into the Scala core distribution with the 2.10.0 release. Originally, with a slightly different API, they were part of the Akka concurrency toolkit.

Why sequential code can be bad

Suppose you want to prepare a cappuccino. You could simply execute the following steps, one after another:

1. Grind the required coffee beans
2. Heat some water
3. Brew an espresso using the ground coffee and the heated water
4. Froth some milk
5. Combine the espresso and the frothed milk to a cappuccino

Translated to Scala code, you would do something like this:

```
1  import scala.util.Try
2  // Some type aliases, just for getting more meaningful method signatures:
3  type CoffeeBeans = String
4  type GroundCoffee = String
5  case class Water(temperature: Int)
6  type Milk = String
7  type FrothedMilk = String
8  type Espresso = String
9  type Cappuccino = String
```

```
// dummy implementations of the individual steps:
def grind(beans: CoffeeBeans): GroundCoffee = s"ground coffee of $beans"
def heatWater(water: Water): Water = water.copy(temperature = 85)
def frothMilk(milk: Milk): FrothedMilk = s"frothed $milk"
def brew(coffee: GroundCoffee, heatedWater: Water): Espresso = "espresso"
def combine(espresso: Espresso, frothedMilk: FrothedMilk): Cappuccino =
  "cappuccino"
// some exceptions for things that might go wrong in the individual steps
// (we'll need some of them later, use the others when experimenting
// with the code):
case class GrindingException(msg: String) extends Exception(msg)
case class FrothingException(msg: String) extends Exception(msg)
case class WaterBoilingException(msg: String) extends Exception(msg)
case class BrewingException(msg: String) extends Exception(msg)
// going through these steps sequentially:
def prepareCappuccino(): Try[Cappuccino] = for {
  ground   <- Try(grind("arabica beans"))
  water    <- Try(heatWater(Water(25)))
  espresso <- Try(brew(ground, water))
  foam     <- Try(frothMilk("milk"))
} yield combine(espresso, foam)
```

Doing it like this has several advantages: You get a very readable step-by-step instruction of what to do. Moreover, you will likely not get confused while preparing the cappuccino this way, since you are avoiding context switches.

On the downside, preparing your cappuccino in such a step-by-step manner means that your brain and body are on wait during large parts of the whole process. While waiting for the ground coffee, you are effectively blocked. Only when that's finished, you're able to start heating some water, and so on.

This is clearly a waste of valuable resources. It's very likely that you would want to initiate multiple steps and have them execute concurrently. Once you see that the water and the ground coffee is ready, you'd start brewing the espresso, in the meantime already starting the process of frothing the milk.

It's really no different when writing a piece of software. A web server only has so many threads for processing requests and creating appropriate responses. You don't want to block these valuable threads by waiting for the results of a database query or a call to another HTTP service. Instead, you want an asynchronous programming model and non-blocking IO, so that, while the processing of one request is waiting for the response from a database, the web server thread handling that request can serve the needs of some other request instead of idling along.

Of course, you already knew all that - what with Node.js being all the rage among the cool kids for a while now. The approach used by Node.js and some others is to communicate via callbacks,

exclusively. Unfortunately, this can very easily lead to a convoluted mess of callbacks within callbacks within callbacks, making your code hard to read and debug.

Scala's `Future` allows callbacks, too, as you will see very shortly, but it provides much better alternatives, so it's likely you won't need them a lot.

You might also be familiar with other `Future` implementations, most notably the one provided by Java. There is not really much you can do with a Java future other than checking if it's completed or simply blocking until it is completed. In short, they are nearly useless and definitely not a joy to work with.

If you think that Scala's futures are anything like that, get ready for a surprise. Here we go!

Semantics of Future

Scala's `Future[T]`, residing in the `scala.concurrent` package, is a container type, representing a computation that is supposed to *eventually* result in a value of type `T`. Alas, the computation might go wrong or time out, so when the future is completed, it may not have been successful after all, in which case it contains an exception instead.

`Future` is a write-once container – after a future has been completed, it is effectively immutable. Also, the `Future` type only provides an interface for *reading* the value to be computed. The task of *writing* the computed value is achieved via a `Promise`. Hence, there is a clear separation of concerns in the API design. In this chapter, we are focussing on the former, postponing the use of the `Promise` type to the next one.

Working with Futures

There are several ways you can work with Scala futures, which we are going to examine by rewriting our cappuccino example to make use of the `Future` type. First, we need to rewrite all of the functions that can be executed concurrently so that they immediately return a `Future` instead of computing their result in a blocking way:

```
1  import scala.concurrent.future
2  import scala.concurrent.Future
3  import scala.concurrent.ExecutionContext.Implicits.global
4  import scala.concurrent.duration._
5  import scala.util.Random
6
7  def grind(beans: CoffeeBeans): Future[GroundCoffee] = future {
8    println("start grinding...")
9    Thread.sleep(Random.nextInt(2000))
10   if (beans == "baked beans") throw GrindingException("are you joking?")
```

```
11    println("finished grinding...")
12    s"ground coffee of $beans"
13  }
14
15  def heatWater(water: Water): Future[Water] = future {
16    println("heating the water now")
17    Thread.sleep(Random.nextInt(2000))
18    println("hot, it's hot!")
19    water.copy(temperature = 85)
20  }
21
22  def frothMilk(milk: Milk): Future[FrothedMilk] = future {
23    println("milk frothing system engaged!")
24    Thread.sleep(Random.nextInt(2000))
25    println("shutting down milk frothing system")
26    s"frothed $milk"
27  }
28
29  def brew(coffee: GroundCoffee, heatedWater: Water): Future[Espresso] =
30    future {
31      println("happy brewing :)")
32      Thread.sleep(Random.nextInt(2000))
33      println("it's brewed!")
34      "espresso"
35    }
```

There are several things that require an explanation here.

First off, there is `future`, a method defined in the `scala.concurrent` package that requires two arguments:

```
1  def future[T](body: => T)(implicit execctx: ExecutionContext): Future[T]
```

The computation to be computed asynchronously is passed in as the `body` by-name parameter. The second argument, in its own argument list, is an *implicit* one, which means we don't have to specify one if a matching implicit value is defined somewhere in scope. We make sure this is the case by importing the global execution context.

An `ExecutionContext` is something that can execute our future, and you can think of it as something like a thread pool. Since the `ExecutionContext` is available implicitly, we only have a single one-element argument list remaining. This allows us to use curly braces instead of parentheses, making it look a little bit like we are using a feature of the language when we are calling `future`. The `ExecutionContext` is an implicit parameter for virtually all of the `Future` API.

Furthermore, of course, in this simple example, we don't actually compute anything, which is why we are putting in some random sleep, simply for demonstration purposes. We also print to the console before and after our "computation" to make the non-deterministic and concurrent nature of our code clearer when trying it out.

The computation of the value to be returned by a `Future` will start at some non-deterministic time after that `Future` instance has been created, by some thread assigned to it by the `ExecutionContext`.

Callbacks

Sometimes, when things are simple, using a callback can be perfectly fine. Callbacks for futures are partial functions. You can pass a callback to the `onSuccess` method. It will only be called if the `Future` completes successfully, and if so, it receives the computed value as its input:

```
grind("arabica beans").onSuccess { case ground =>
  println("okay, got my ground coffee")
}
```

Similarly, you could register a failure callback with the `onFailure` method. Your callback will receive a `Throwable`, but it will only be called if the `Future` did not complete successfully.

Usually, it's better to combine these two and register a completion callback that will handle both cases. The input parameter for that callback is a `Try`:

```
import scala.util.{Success, Failure}
grind("baked beans").onComplete {
  case Success(ground) =>
    println(s"got my $ground")
  case Failure(ex) =>
    println("This grinder needs a replacement, seriously!")
}
```

Since we are passing in baked beans, an exception occurs in the `grind` method, leading to the `Future` completing with a `Failure`.

Composing futures

Using callbacks can be quite painful if you have to start nesting callbacks. Thankfully, you don't have to do that! The real power of the Scala futures is that they are composable.

You will have noticed that all the container types we discussed made it possible for you to map them, flat map them, or use them in for comprehensions and that I mentioned that `Future` is a container type, too. Hence, the fact that Scala's `Future` type allows you to do all that will not come as a surprise at all.

The real question is: What does it really mean to perform these operations on something that hasn't even finished computing yet?

Mapping the future

Haven't you always wanted to be a traveller in time who sets out to map the future? As a Scala developer you can do exactly that! Suppose that once your water has heated you want to check if its temperature is okay. You can do so by mapping your `Future[Water]` to a `Future[Boolean]`:

```
val temperatureOkay: Future[Boolean] = heatWater(Water(25)).map { water =>
  println("we're in the future!")
  (80 to 85).contains(water.temperature)
}
```

The `Future[Boolean]` assigned to `temperatureOkay` will eventually contain the successfully computed boolean value. Go change the implementation of `heatWater` so that it throws an exception (maybe because your water heater explodes or something) and watch how `we're in the future` will never be printed to the console.

When you are writing the function you pass to `map`, you're in the future, or rather in a possible future. That mapping function gets executed as soon as your `Future[Water]` instance has completed successfully. However, the timeline in which that happens might not be the one you live in. If your instance of `Future[Water]` fails, what's taking place in the function you passed to `map` will never happen. Instead, the result of calling `map` will be a `Future[Boolean]` containing a `Failure`.

Keeping the future flat

If the computation of one `Future` depends on the result of another, you'll likely want to resort to `flatMap` to avoid a deeply nested structure of futures.

For example, let's assume that the process of actually measuring the temperature takes a while, so you want to determine whether the temperature is okay asynchronously, too. You have a function that takes an instance of `Water` and returns a `Future[Boolean]`:

```
def temperatureOkay(water: Water): Future[Boolean] = future {
  (80 to 85).contains(water.temperature)
}
```

Use `flatMap` instead of `map` in order to get a `Future[Boolean]` instead of a `Future[Future[Boolean]]`:

```
val nestedFuture: Future[Future[Boolean]] = heatWater(Water(25)).map {
  water => temperatureOkay(water)
}
val flatFuture: Future[Boolean] = heatWater(Water(25)).flatMap {
  water => temperatureOkay(water)
}
```

Again, the mapping function is only executed after (and if) the `Future[Water]` instance has been completed successfully, hopefully with an acceptable temperature.

For comprehensions

Instead of calling `flatMap`, you'll usually want to write a for comprehension, which is essentially the same, but reads a lot clearer. Our example above could be rewritten like this:

```
val acceptable: Future[Boolean] = for {
  heatedWater <- heatWater(Water(25))
  okay        <- temperatureOkay(heatedWater)
} yield okay
```

If you have multiple computations that can be computed in parallel, you need to take care that you already create the corresponding `Future` instances outside of the for comprehension.

```
def prepareCappuccinoSequentially(): Future[Cappuccino] = {
  for {
    ground   <- grind("arabica beans")
    water    <- heatWater(Water(20))
    foam     <- frothMilk("milk")
    espresso <- brew(ground, water)
  } yield combine(espresso, foam)
}
```

This reads nicely, but since a for comprehension is just another representation for nested `flatMap` calls, this means that the `Future[Water]` created in `heatWater` is only really instantiated after the `Future[GroundCoffee]` has completed successfully. You can check this by watching the sequential console output coming from the functions we implemented above.

Hence, make sure to instantiate all your independent futures before the for comprehension:

```
def prepareCappuccino(): Future[Cappuccino] = {
  val groundCoffee = grind("arabica beans")
  val heatedWater = heatWater(Water(20))
  val frothedMilk = frothMilk("milk")
  for {
    ground   <- groundCoffee
    water    <- heatedWater
    foam     <- frothedMilk
    espresso <- brew(ground, water)
  } yield combine(espresso, foam)
}
```

Now, the three futures we create before the for comprehension start being completed immediately and execute concurrently. If you watch the console output, you will see that it's non-deterministic. The only thing that's certain is that the `"happy brewing"` output will come last. Since the method in which it is called requires the values coming from two other futures, it is only created inside our for comprehension, i.e. after those futures have completed successfully.

Failure projections

You will have noticed that `Future[T]` is success-biased, allowing you to use `map`, `flatMap`, `filter` etc. under the assumption that it will complete successfully. Sometimes, you may want to be able to work in this nice functional way for the timeline in which things go wrong. By calling the `failed` method on an instance of `Future[T]`, you get a failure projection of it, which is a `Future[Throwable]`. Now you can `map` that `Future[Throwable]`, for example, and your mapping function will only be executed if the original `Future[T]` has completed with a failure.

Summary

You have seen the `Future`, and it looks bright! The fact that it's just another container type that can be composed and used in a functional way makes working with it very pleasant.

Making blocking code concurrent can be pretty easy by wrapping it in a call to `future`. However, it's better to be non-blocking in the first place. To achieve this, one has to make a `Promise` to complete a `Future`. This and using the futures in practice will be the topic of the next chapter.

Promises and Futures in practice

In the previous chapter, you got an introduction to the `Future` type, its underlying paradigm, and how to put it to use to write highly readable and composable asynchronously executing code.

You also learned that `Future` is really only one piece of the puzzle: It's a read-only type that allows you to work with the values it will compute and handle failure to do so in an elegant way. In order for you to be able to read a computed value from a `Future`, however, there must be a way for some other part of your software to put that value there. In this chapter, you will see how this is done by means of the `Promise` type, followed by some guidance on how to use futures and promises in practice.

Promises

In the previous chapter, we had a sequential block of code that we passed to the `future` method from the `scala.concurrent` package, and, given an `ExecutionContext` was in scope, it magically executed that code block asynchronously, returning its result as a `Future`.

While this is an easy way to get a `Future` when you want one, there is an alternative way to create `Future` instances and have them complete with a success or failure. Where `Future` provides an interface exclusively for *querying*, `Promise` is a companion type that allows you to complete a `Future` by putting a value into it. This can be done exactly once. Once a `Promise` has been completed, it's not possible to change it any more.

A `Promise` instance is always linked to exactly one instance of `Future`. If you call the `future` method again in the REPL, you will indeed notice that the `Future` returned is a `Promise`, too:

```
1  import concurrent.Future
2  import concurrent.future
3  import concurrent.ExecutionContext.Implicits.global
4  val f: Future[String] = future { "Hello world!" }
5  // REPL output:
6  // f: scala.concurrent.Future[String] =
7  //    scala.concurrent.impl.Promise$DefaultPromise@793e6657
```

The object you get back is a `DefaultPromise`, which implements both `Future` and `Promise`. This is an implementation detail, however. The `Future` and the `Promise` to which it belongs may very well be separate objects.

What this little example shows is that there is obviously no way to complete a `Future` other than through a `Promise` – the `future` method is just a nice helper function that shields you from this.

Now, let's see how we can get our hands dirty and work with the `Promise` type directly.

Promising a rosy future

When talking about promises that may be fulfilled or not, an obvious example domain is that of politicians, elections, campaign pledges, and legislative periods.

Suppose the politicians that then got elected into office promised their voters a tax cut. This can be represented as a `Promise[TaxCut]`, which you can create by calling the `apply` method on the `Promise` companion object, like so:

```
import concurrent.Promise
case class TaxCut(reduction: Int)
// either give the type as a type parameter to the factory method:
val taxcut = Promise[TaxCut]()
// or give the compiler a hint by specifying the type of your val:
val taxcut2: Promise[TaxCut] = Promise()
```

Once you have created a `Promise`, you can get the `Future` belonging to it by calling the `future` method on the `Promise` instance:

```
val taxcutF: Future[TaxCut] = taxcut.future
```

The returned `Future` might not be the same object as the `Promise`, but calling the `future` method of a `Promise` multiple times will definitely always return the same object to make sure the one-to-one relationship between a `Promise` and its `Future` is preserved.

Completing a Promise

Once you have made a `Promise` and told the world that you will deliver on it in the forseeable `Future`, you better do your very best to make it happen.

In Scala, you can complete a `Promise` either with a success or a failure.

Delivering on your Promise

To complete a `Promise` with a success, you call its `success` method, passing it the value that the `Future` associated with it is supposed to have:

```
taxcut.success(TaxCut(20))
```

Once you have done this, that `Promise` instance is no longer writable, and future attempts to do so will lead to an exception.

Also, completing your `Promise` like this leads to the successful completion of the associated `Future`. Any success or completion handlers on that future will now be called, or if, for instance, you are mapping that future, the mapping function will now be executed.

Usually, the completion of the `Promise` and the processing of the completed `Future` will not happen in the same thread. It's more likely that you create your `Promise`, start computing its value in another thread and immediately return the uncompleted `Future` to the caller.

To illustrate, let's do something like that for our taxcut promise:

```
1  object Government {
2    def redeemCampaignPledge(): Future[TaxCut] = {
3      val p = Promise[TaxCut]()
4      future {
5        println("Starting the new legislative period.")
6        Thread.sleep(2000)
7        p.success(TaxCut(20))
8        println("We reduced the taxes! You must reelect us!!!!1111")
9      }
10     p.future
11   }
12 }
```

Please don't get confused by the usage of the `future` method from the `scala.concurrent` package in this example. I'm just using it because it is so convenient for executing a block of code asynchronously. I could just as well have implemented the computation of the result (which involves a lot of sleeping) in a `Runnable` that is executed asynchrously by an `ExecutorService`, with a lot more boilerplate code. The point is that the `Promise` is not completed in the caller thread.

Let's redeem our campaign pledge then and add an `onComplete` callback function to our future:

```
1  import scala.util.{Success, Failure}
2  val taxCutF: Future[TaxCut] = Government.redeemCampaignPledge()
3    println("Let's see if they remember their promises...")
4    taxCutF.onComplete {
5      case Success(TaxCut(reduction)) =>
6        println(s"Yay! They cut our taxes by $reduction percentage points!")
7      case Failure(ex) =>
8        println(s"They broke their promises! Because of a ${ex.getMessage}")
9    }
```

If you try this out multiple times, you will see that the order of the console output is not deterministic. Eventually, the completion handler will be executed and run into the success case.

Breaking Promises like a sir

As a politician, you are pretty much used to not keeping your promises. As a Scala developer, you sometimes have no other choice, either. If that happens, you can still complete your `Promise` instance gracefully, by calling its `failure` method and passing it an exception:

```
case class LameExcuse(msg: String) extends Exception(msg)
object Government {
  def redeemCampaignPledge(): Future[TaxCut] = {
       val p = Promise[TaxCut]()
       future {
         println("Starting the new legislative period.")
         Thread.sleep(2000)
         p.failure(LameExcuse("global economy crisis"))
         println("We didn't fulfill our promises, so what?")
       }
       p.future
     }
}
```

This implementation of the `redeemCampaignPledge()` method will to lots of broken promises. Once you have completed a `Promise` with the `failure` method, it is no longer writable, just as is the case with the `success` method. The associated `Future` will now be completed with a `Failure`, too, so the callback function above would run into the failure case.

If you already have a `Try`, you can also complete a `Promise` by calling its `complete` method. If the `Try` is a `Success`, the associated `Future` will be completed successfully, with the value inside the `Success`. If it's a `Failure`, the `Future` will completed with that failure.

Future-based programming in practice

If you want to make use of the future-based paradigm in order to increase the scalability of your application, you have to design your application to be non-blocking from the ground-up, which basically means that the functions in all your application layers are asynchronous and return futures.

A likely use case these days is that of developing a web application. If you are using a modern Scala web framework, it will allow you to return your responses as something like a `Future[Response]` instead of blocking and then returning your finished `Response`. This is important since it allows your web server to handle a huge number of open connections with a relatively low number of threads. By always giving your web server `Future[Response]`, you maximize the utilization of the web server's dedicated thread pool.

In the end, a service in your application might make multiple calls to your database layer and/or some external web service, receiving multiple futures, and then compose them to return a new

`Future`, all in a very readable for comprehension, such as the one you saw in the previous chapter. The web layer will turn such a `Future` into a `Future[Response]`.

However, how do you implement this in practice? There are are three different cases you have to consider:

Non-blocking IO

Your application will most certainly involve a lot of IO. For example, your web application will have to talk to a database, and it might act as a client that is calling other web services.

If at all possible, make use of libraries that are based on Java's non-blocking IO capabilities, either by using Java's NIO API directly or through a library like Netty. Such libraries, too, can serve many connections with a reasonably-sized dedicated thread pool.

Developing such a library yourself is one of the few places where directly working with the `Promise` type makes a lot of sense.

Blocking IO

Sometimes, there is no NIO-based library available. For instance, most database drivers you'll find in the Java world nowadays are using blocking IO. If you made a query to your database with such a driver in order to respond to a HTTP request, that call would be made on a web server thread. To avoid that, place all the code talking to the database inside a `future` block, like so:

```
1  import concurrent.future
2  // get back a Future[ResultSet] or something similar:
3  future {
4    queryDB(query)
5  }
```

So far, we always used the implicitly available global `ExecutionContext` to execute such future blocks. It's probably a good idea to create a dedicated `ExecutionContext` that you will have in scope in your database layer.

You can create an `ExecutionContext` from a Java `ExecutorService`, which means you will be able to tune the thread pool for executing your database calls asynchronously independently from the rest of your application:

```
import java.util.concurrent.Executors
import concurrent.ExecutionContext
val executorService = Executors.newFixedThreadPool(4)
val executionContext =
  ExecutionContext.fromExecutorService(executorService)
```

Long-running computations

Depending on the nature of your application, it will occasionally have to call long-running tasks that don't involve any IO at all, which means they are CPU-bound. These, too, should not be executed by a web server thread. Hence, you should turn them into `Futures`, too:

```
import concurrent.future
future {
  longRunningComputation(data, moreData)
}
```

Again, if you have long-running computations, having them run in a separate `ExecutionContext` for CPU-bound tasks is a good idea. How to tune your various thread pools is highly dependent on your individual application and beyond the scope of this book.

Summary

In this chapter, we looked at promises, the writable part of the future-based concurrency paradigm, and how to use them to complete a `Future`, followed by some advice on how to put futures to use in practice.

In the next chapter, we are taking a step back from concurrency issues and examine how functional programming in Scala can help you to make your code more reusable, a claim that has long been associated with object-oriented programming.

Staying DRY with higher-order functions

In the previous chapters, you heard a lot about the composable nature of Scala's container types. As it turns out, being composable is a quality that you will not only find in `Future`, `Try`, and other container types, but also in functions, which are first class citizens in the Scala language.

Composability naturally entails reusability. While the latter has often been claimed to be one of the big advantages of object-oriented programming, it's a trait that is definitely true for pure functions, i.e. functions that do not have any side-effects and are referentially transparent.

One obvious way is to implement a new function by calling already existing functions in its body. However, there are other ways to reuse existing functions: In this chapter, I will discuss some fundementals of functional programming that we have been missing out on so far. You will learn how to follow the DRY[9] principle by leveraging higher-order functions in order to reuse existing code in new contexts.

On higher-order functions

A higher-order function, as opposed to a first-order function, can have one of three forms:

1. One or more of its parameters is a function, and it returns some value.
2. It returns a function, but none of its parameters is a function.
3. Both of the above: One or more of its parameters is a function, and it returns a function.

Throughout this book, you have seen a lot of usages of higher-order functions of the first type: We called methods like `map`, `filter`, or `flatMap` and passed a function to it that was used to transform or filter a collection in some way. Very often, the functions we passed to these methods were anonymous functions, sometimes involving a little bit of duplication.

In this chapter, we are only concerned with what the other two types of higher-order functions can do for us: The first of them allows us to produce new functions based on some input data, whereas the other gives us the power and flexibility to compose new functions that are somehow based on existing functions. In both cases, we can eliminate code duplication.

[9] http://en.wikipedia.org/wiki/Don%27t_repeat_yourself

And out of nowhere, a function was born

You might think that the ability to create new functions based on some input data is not terribly useful. While we mainly want to deal with how to compose new functions based on existing ones, let's first have a look at how a function that produces new functions may be used.

Let's assume we are implementing a freemail service where users should be able to configure when an email is supposed to be blocked. We are representing emails as instances of a simple case class:

```
case class Email(
  subject: String,
  text: String,
  sender: String,
  recipient: String)
```

We want to be able to filter new emails by the criteria specified by the user, so we have a filtering function that makes use of a predicate, a function of type `Email => Boolean` to determine whether the email is to be blocked. If the predicate is `true`, the email is accepted, otherwise it will be blocked:

```
type EmailFilter = Email => Boolean
def newMailsForUser(mails: Seq[Email], f: EmailFilter) = mails.filter(f)
```

Note that we are using a type alias for our function, so that we can work with more meaningful names in our code.

Now, in order to allow the user to configure their email filter, we can implement some factory functions that produce `EmailFilter` functions configured to the user's liking:

```
val sentByOneOf: Set[String] => EmailFilter =
  senders => email => senders.contains(email.sender)
val notSentByAnyOf: Set[String] => EmailFilter =
  senders => email => !senders.contains(email.sender)
val minimumSize: Int => EmailFilter = n => email => email.text.size >= n
val maximumSize: Int => EmailFilter = n => email => email.text.size <= n
```

Each of these four `val`s is a function that returns an `EmailFilter`, the first two taking as input a `Set[String]` representing senders, the other two an `Int` representing the length of the email body.

We can use any of these functions to create a new `EmailFilter` that we can pass to the `newMailsForUser` function:

```
val emailFilter: EmailFilter = notSentByAnyOf(Set("johndoe@example.com"))
val mails = Email(
  subject = "It's me again, your stalker friend!",
  text = "Hello my friend! How are you?",
  sender = "johndoe@example.com",
  recipient = "me@example.com") :: Nil
newMailsForUser(mails, emailFilter) // returns an empty list
```

This filter removes the one mail in the list because our user decided to put the sender on their black list. We can use our factory functions to create arbitrary `EmailFilter` functions, depending on the user's requirements.

Reusing existing functions

There are two problems with the current solution. First of all, there is quite a bit of duplication in the predicate factory functions above, when initially I told you that the composable nature of functions made it easy to stick to the DRY principle. So let's get rid of the duplication.

To do that for the `minimumSize` and `maximumSize`, we introduce a function `sizeConstraint` that takes a predicate that checks if the size of the email body is okay. That size will be passed to the predicate by the `sizeConstraint` function:

```
type SizeChecker = Int => Boolean
val sizeConstraint: SizeChecker => EmailFilter =
  f => email => f(email.text.size)
```

Now we can express `minimumSize` and `maximumSize` in terms of `sizeConstraint`:

```
val minimumSize: Int => EmailFilter = n => sizeConstraint(_ >= n)
val maximumSize: Int => EmailFilter = n => sizeConstraint(_ <= n)
```

Function composition

For the other two predicates, `sentByOneOf` and `notSentByAnyOf`, we are going to introduce a very generic higher-order function that allows us to express one of the two functions in terms of the other.

Let's implement a function `complement` that takes a predicate `A => Boolean` and returns a new function that always returns the opposite of the given predicate:

```
def complement[A](predicate: A => Boolean) = (a: A) => !predicate(a)
```

Now, for an existing predicate `p` we could get the complement by calling `complement(p)`. However, `sentByAnyOf` is not a predicate, but it *returns* one, namely an `EmailFilter`.

Scala functions provide two composing functions that will help us here: Given two functions `f` and `g`, `f.compose(g)` returns a new function that, when called, will first call `g` and then apply `f` on the result of it. Similarly, `f.andThen(g)` returns a new function that, when called, will apply `g` to the result of `f`.

We can put this to use to create our `notSentByAnyOf` predicate without code duplication:

```
val notSentByAnyOf = sentByOneOf andThen(g => complement(g))
```

What this means is that we ask for a new function that first applies the `sentByOneOf` function to its arguments (a `Set[String]`) and then applies the `complement` function to the `EmailFilter` predicate returned by the former function. Using Scala's placeholder syntax for anonymous functions, we could write this more concisely as:

```
val notSentByAnyOf = sentByOneOf andThen(complement(_))
```

Of course, you will now have noticed that, given a `complement` function, you could also implement the `maximumSize` predicate in terms of `minimumSize` instead of extracting a `sizeConstraint` function. However, the latter is more flexible, allowing you to specify arbitrary checks on the size of the mail body.

Composing predicates

Another problem with our email filters is that we can currently only pass a single `EmailFilter` to our `newMailsForUser` function. Certainly, our users want to configure multiple criteria. We need a way to create a composite predicate that returns `true` if either any, none or all of the predicates it consists of return `true`.

Here is one way to implement these functions:

```
def any[A](predicates: (A => Boolean)*): A => Boolean =
  a => predicates.exists(pred => pred(a))
def none[A](predicates: (A => Boolean)*) =
  complement(any(predicates: _*))
def every[A](predicates: (A => Boolean)*) =
  none(predicates.view.map(complement(_)): _*)
```

The any function returns a new predicate that, when called with an input a, checks if at least one of its predicates holds true for the value a. Our none function simply returns the complement of the predicate returned by any – if at least one predicate holds true, the condition for none is not satisfied. Finally, our every function works by checking that none of the complements to the predicates passed to it holds true.

We can now use this to create a composite `EmailFilter` that represents the user's configuration:

```
val filter: EmailFilter = every(
    notSentByAnyOf(Set("johndoe@example.com")),
    minimumSize(100),
    maximumSize(10000)
  )
```

Composing a transformation pipeline

As another example of function composition, consider our example scenario again. As a freemail provider, we want not only to allow user's to configure their email filter, but also do some processing on emails sent by our users. These are simple functions `Email => Email`. Some possible transformations are the following:

```
val addMissingSubject = (email: Email) =>
  if (email.subject.isEmpty) email.copy(subject = "No subject")
  else email
val checkSpelling = (email: Email) =>
  email.copy(text = email.text.replaceAll("your", "you're"))
val removeInappropriateLanguage = (email: Email) =>
  email.copy(
    text = email.text.replaceAll("dynamic typing", "**CENSORED**"))
val addAdvertismentToFooter = (email: Email) =>
  email.copy(
    text = email.text + "\nThis mail sent via Super Awesome Free Mail")
```

Now, depending on the weather and the mood of our boss, we can configure our pipeline as required, either by multiple `andThen` calls, or, having the same effect, by using the `chain` method defined on the `Function` companion object:

```
val pipeline = Function.chain(Seq(
  addMissingSubject,
  checkSpelling,
  removeInappropriateLanguage,
  addAdvertismentToFooter))
```

Higher-order functions and partial functions

We won't cover them detail here, but now that you know more about how you can compose or reuse functions by means of higher-order functions, you might want to have a look at partial functions again.

Chaining partial functions

In the chapter on pattern matching anonymous functions, I mentioned that partial functions can be used to create a nice alternative to the chain of responsibility pattern: The `orElse` method defined on the `PartialFunction` trait allows you to chain an arbitrary number of partial functions, creating a composite partial function. The first one, however, will only pass on to the next one if it isn't defined for the given input. Hence, you can do something like this:

```
val handler = fooHandler orElse barHandler orElse bazHandler
```

Lifting partial functions

Also, sometimes a `PartialFunction` is not what you need. If you think about it, another way to represent the fact that a function is not defined for all input values is to have a standard function whose return type is an `Option[A]` – if the function is not defined for an input value, it will return `None`, otherwise a `Some[A]`.

If that's what you need in a certain context, given a `PartialFunction` named `pf`, you can call `pf.lift` to get the normal function returning an `Option`. If you have one of the latter and require a partial function, call `Function.unlift(f)`.

Summary

In this chapter, we have seen the value of higher-order functions, which allow you to reuse existing functions in new, unforeseen contexts and compose them in a very flexible way. While in the examples, you didn't save much in terms of lines of code, because the functions I showed were rather tiny, the real point is to illustrate the increase in flexibility. Also, composing and reusing functions is something that has benefits not only for small functions, but also on an architectural level.

In the next chapter, we will continue to examine ways to combine functions by means of partial function application and currying.

Currying and partially applied functions

The previous chapter was all about avoiding code duplication, either by lifting existing functions to match your new requirements or by composing them. In this chapter, we are going to have a look at two other mechanisms the Scala language provides in order to enable you to reuse your functions: Partial application of functions and currying.

Partially applied functions

Scala, like many other languages following the functional programming paradigm, allows you to apply a function *partially*. What this means is that, when applying the function, you do not pass in arguments for *all* of the parameters defined by the function, but only for some of them, leaving the remaining ones blank. What you get back is a new function whose parameter list only contains those parameters from the original function that were left blank.

Do not confuse partially applied functions with *partially defined functions*, which are represented by the `PartialFunction` type in Scala and were covered in chapter 4.

To illustrate how partial function application works, let's revisit our example from the previous chapter: For our imaginary free mail service, we wanted to allow the user to configure a filter so that only emails meeting certain criteria would show up in their inbox, with all others being blocked.

Our `Email` case class still looks like this:

```
1  case class Email(
2    subject: String,
3    text: String,
4    sender: String,
5    recipient: String)
6  type EmailFilter = Email => Boolean
```

The criteria for filtering emails were represented by a predicate `Email => Boolean`, which we aliased to the type `EmailFilter`, and we were able to generate new predicates by calling appropriate factory functions.

Two of the factory functions from the previous chapter created `EmailFiter` instances that checked if the email text satisfied a given minimum or maximum length. This time we want to make use of partial function application to implement these factory functions. We want to have a general method `sizeConstraint` and be able to create more specific size constraints by fixing some of its parameters.

Here is our revised `sizeConstraint` method:

```
type IntPairPred = (Int, Int) => Boolean
def sizeConstraint(pred: IntPairPred, n: Int, email: Email) =
  pred(email.text.size, n)
```

We also define an alias `IntPairPred` for the type of predicate that checks a pair of integers (the value `n` and the text size of the email) and returns whether the email text size is okay, given `n`.

Note that unlike our `sizeConstraint` function from the previous chapter, this one does not return a new `EmailFilter` predicate, but simply evaluates all the arguments passed to it, returning a `Boolean`. The trick is to get such a predicate of type `EmailFilter` by partially applying `sizeConstraint`.

First, however, because we take the DRY principle very seriously, let's define all the commonly used instances of `IntPairPred`. Now, when we call `sizeConstraint`, we don't have to repeatedly write the same anonymous functions, but can simply pass in one of those:

```
val gt: IntPairPred = _ > _
val ge: IntPairPred = _ >= _
val lt: IntPairPred = _ < _
val le: IntPairPred = _ <= _
val eq: IntPairPred = _ == _
```

Finally, we are ready to do some partial application of the `sizeConstraint` function, fixing its first parameter with one of our `IntPairPred` instances:

```
val minimumSize: (Int, Email) => Boolean =
  sizeConstraint(ge, _: Int, _: Email)
val maximumSize: (Int, Email) => Boolean =
  sizeConstraint(le, _: Int, _: Email)
```

As you can see, you have to use the placeholder _ for all parameters not bound to an argument value. Unfortunately, you also have to specify the type of those arguments, which makes partial function application in Scala a bit tedious.

The reason is that the Scala compiler cannot infer these types, at least not in all cases – think of overloaded methods where it's impossible for the compiler to know which of them you are referring to.

On the other hand, this makes it possible to bind or leave out arbitrary parameters. For example, we can leave out the first one and pass in the size to be used as a constraint:

```
val constr20: (IntPairPred, Email) => Boolean =
  sizeConstraint(_: IntPairPred, 20, _: Email)
val constr30: (IntPairPred, Email) => Boolean =
  sizeConstraint(_: IntPairPred, 30, _: Email)
```

Now we have two functions that take an `IntPairPred` and an `Email` and compare the email text size to `20` and `30`, respectively, but the comparison logic has not been specified yet, as that's exactly what the `IntPairPred` is good for.

This shows that, while quite verbose, partial function application in Scala is a little more flexible than in Clojure, for example, where you have to pass in arguments from left to right, but can't leave out any in the middle.

From methods to function objects

When doing partial application on a method, you can also decide to not bind any parameters whatsoever. The parameter list of the returned function object will be the same as for the method. You have effectively turned a method into a function that can be assigned to a `val` or passed around:

```
val sizeConstraintFn: (IntPairPred, Int, Email) => Boolean =
  sizeConstraint _
```

Producing those EmailFilters

We still haven't got any functions that adhere to the `EmailFilter` type or that return new predicates of that type – like `sizeConstraint`, `minimumSize` and `maximumSize` don't return a new predicate, but a `Boolean`, as their signature clearly shows.

However, our email filters are just another partial function application away. By fixing the integer parameter of `minimumSize` and `maximumSize`, we can create new functions of type `EmailFilter`:

```
val min20: EmailFilter = minimumSize(20, _: Email)
val max20: EmailFilter = maximumSize(20, _: Email)
```

Of course, we could achieve the same by partially applying our `constr20` we created above:

```
val min20: EmailFilter = constr20(ge, _: Email)
val max20: EmailFilter = constr20(le, _: Email)
```

Spicing up your functions

Maybe you find partial function application in Scala a little too verbose, or simply not very elegant to write and look at. Lucky you, because there is an alternative.

As you should know by now, methods in Scala can have more than one parameter list. Let's define our `sizeConstraint` method such that each parameter is in its own parameter list:

```
def sizeConstraint(pred: IntPairPred)(n: Int)(email: Email): Boolean =
  pred(email.text.size, n)
```

If we turn this into a function object that we can assign or pass around, the signature of that function looks like this:

```
val sizeConstraintFn: IntPairPred => Int => Email => Boolean =
  sizeConstraint _
```

Such a chain of one-parameter functions is called a *curried* function, so named after Haskell Curry, who re-discovered this technique and got all the fame for it. In fact, in the Haskell programming language, all functions are in curried form by default.

In our example, it takes an `IntPairPred` and returns a function that takes an `Int` and returns a new function. That last function, finally, takes an `Email` and returns a `Boolean`.

Now, if we we want to bind the `IntPairPred`, we simply apply `sizeConstraintFn`, which takes exactly this one parameter and returns a new one-parameter function:

```
val minSize: Int => Email => Boolean = sizeConstraint(ge)
val maxSize: Int => Email => Boolean = sizeConstraint(le)
```

There is no need to use any placeholders for parameters left blank, because we are in fact not doing any partial function application.

We can now create the same `EmailFilter` predicates as we did using partial function application before, by applying these curried functions:

```
val min20: Email => Boolean = minSize(20)
val max20: Email => Boolean = maxSize(20)
```

Of course, it's possible to do all this in one step if you want to bind several parameters at once. It just means that you immediately apply the function that was returned from the first function application, without assigning it to a `val` first:

```
val min20: Email => Boolean = sizeConstraintFn(ge)(20)
val max20: Email => Boolean = sizeConstraintFn(le)(20)
```

Currying existing functions

It's not always the case the you know beforehand whether writing your functions in curried form makes sense or not – after all, the usual function application looks a little more verbose than for functions that have only declared a single parameter list for all their parameters. Also, you'll sometimes want to work with third-party functions in curried form, when they are written with a single parameter list.

Transforming a function with multiple parameters in one list to curried form is, of course, just another higher-order function, generating a new function from an existing one. This transformation logic is available in form of the `curried` method on functions with more than one parameter. Hence, if we have a function `sum`, taking two parameters, we can get the curried version simply calling calling its `curried` method:

```
1  val sum: (Int, Int) => Int = _ + _
2  val sumCurried: Int => Int => Int = sum.curried
```

If you need to do the reverse, you have `Function.uncurried` at your fingertips, which you need to pass the curried function to get it back in uncurried form.

Injecting your dependencies the functional way

To close this chapter, let's have a look at what role curried functions can play in the large. If you come from the enterprisy Java or .NET world, you will be very familiar with the necessity to use more or less fancy dependency injection containers that take the heavy burden of providing all your objects with their respective dependencies off you. In Scala, you don't really need any external tool for that, as the language already provides numerous features that make it much less of a pain to this all on your own.

When programming in a very functional way, you will notice that there is still a need to inject dependencies: Your functions residing in the higher layers of your application will have to make calls to other functions. Simply hard-coding the functions to call in the body of your functions makes it difficult to test them in isolation. Hence, you will need to pass the functions your higher-layer function depends on as arguments to that function.

It would not be DRY at all to always pass the same dependencies to your function when calling it, would it? Curried functions to the rescue! Currying and partial function application are one of several ways of injecting dependencies in functional programming.

The following, very simplified example illustrates this technique:

```
case class User(name: String)
trait EmailRepository {
  def getMails(user: User, unread: Boolean): Seq[Email]
}
trait FilterRepository {
  def getEmailFilter(user: User): EmailFilter
}
trait MailboxService {
  def getNewMails
      (emailRepo: EmailRepository)
      (filterRepo: FilterRepository)
      (user: User) =
    emailRepo.getMails(user, true).filter(filterRepo.getEmailFilter(user))
  val newMails: User => Seq[Email]
}
```

We have a service that depends on two different repositories, These dependencies are declared as parameters to the `getNewMails` method, each in their own parameter list.

The `MailboxService` already has a concrete implementation of that method, but is lacking one for the `newMails` field. The type of that field is `User => Seq[Email]` – that's the function that components depending on the `MailboxService` will call.

We need an object that extends `MailboxService`. The idea is to implement `newMails` by currying the `getNewMails` method and fixing it with concrete implementations of the dependencies, `EmailRepository` and `FilterRepository`:

```
object MockEmailRepository extends EmailRepository {
  def getMails(user: User, unread: Boolean): Seq[Email] = Nil
}
object MockFilterRepository extends FilterRepository {
  def getEmailFilter(user: User): EmailFilter = _ => true
}
object MailboxServiceWithMockDeps extends MailboxService {
  val newMails: (User) => Seq[Email] =
    getNewMails(MockEmailRepository)(MockFilterRepository) _
}
```

We can now call `MailboxServiceWithMoxDeps.newMails(User("daniel"))` without having to specify the two repositories to be used. In a real application, of course, we would very likely not use a direct reference to a concrete implementation of the service, but have this injected, too.

This is probably not the most powerful and scaleable way of injecting your dependencies in Scala, but it's good to have this in your tool belt, and it's a very good example of the benefits that partial

function application and currying can provide in the wild. If you want know more about this, I recommend to have a look at the excellent slides for the presentation "Dependency Injection in Scala[10]" by Debasish Ghosh, which is also where I first came across this technique.

Summary

In this chapter, we discussed two additional functional programming techniques that help you keep your code free of duplication and, on top of that, give you a lot of flexibility, allowing you to reuse your functions in manifold ways. Partial function application and currying have more or less the same effect, but sometimes one of them is the more elegant solution.

In the next chapter, we will continue to look at ways to keep things flexible, discussing the what and how of type classes in Scala.

[10] http://de.slideshare.net/debasishg/dependency-injection-in-scala-beyond-the-cake-pattern

Type classes

After having discussed several functional programming techniques for keeping things DRY and flexible in the last two chapters, in particular function composition, partial function application, and currying, we are going to stick with the general notion of making your code as flexible as possible.

However, this time, we are not looking so much at how you can leverage functions as first-class objects to achieve this goal. Instead, this chapter is all about using the type system in such a manner that it's not in the way, but rather supports you in keeping your code extensible: You're going to learn about *type classes*.

You might think that this is some exotic idea without practical relevance, brought into the Scala community by some vocal Haskell fanatics. This is clearly not the case. Type classes have become an important part of the Scala standard library and even more so of many popular and commonly used third-party open-source libraries, so it's generally a good idea to make yourself familiar with them.

I will discuss the idea of type classes, why they are useful, how to benefit from them as a client, and how to implement your own type classes and put them to use for great good.

The problem

Instead of starting off by giving an abstract explanation of what type classes are, let's tackle this subject by means of an – admittedly simplified, but nevertheless resonably practical – example.

Imagine that we want to write a fancy statistics library. This means we want to provide a bunch of functions that operate on collections of numbers, mostly to compute some aggregate values for them. Imagine further that we are restricted to accessing an element from such a collection by index and to using the `reduce` method defined on Scala collections. We impose this restriction on ourselves because we are going to re-implement a little bit of what the Scala standard library already provides – simply because it's a nice example without many distractions, and it's small enough for this book. Finally, our implementation assumes that the values we get are already sorted.

We will start with a very crude implementation of `median`, `quartiles`, and `iqr` numbers of type `Double`:

```
object Statistics {
  def median(xs: Vector[Double]): Double = xs(xs.size / 2)
  def quartiles(xs: Vector[Double]): (Double, Double, Double) =
    (xs(xs.size / 4), median(xs), xs(xs.size / 4 * 3))
  def iqr(xs: Vector[Double]): Double = quartiles(xs) match {
    case (lowerQuartile, _, upperQuartile) =>
      upperQuartile - lowerQuartile
  }
  def mean(xs: Vector[Double]): Double = {
    xs.reduce(_ + _) / xs.size
  }
}
```

The median cuts a data set in half, whereas the lower and upper quartile (first and third element of the tuple returned by our `quartile` method), split the lowest and highest 25 percent of the data, respectively. Our `iqr` method returns the interquartile range, which is the difference between the upper and lower quartile.

Now, of course, we want to support more than just double numbers. So let's implement all these methods again for `Int` numbers, right?

Well, no! First of all, we can't simply overload the methods defined above for `Vector[Int]` without some dirty tricks, because the type parameter suffers from *type erasure*. Also, that would be a tiny little bit repetitious, wouldn't it?

If only `Int` and `Double` would extend from a common base class or implement a common trait like `Number`! We could be tempted to change the type required and returned by our methods to that more general type. Our method signatures would look like this:

```
object Statistics {
  def median(xs: Vector[Number]): Number = ???
  def quartiles(xs: Vector[Number]): (Number, Number, Number) = ???
  def iqr(xs: Vector[Number]): Number = ???
  def mean(xs: Vector[Number]): Number = ???
}
```

Thankfully, in this case there is no such common trait, so we aren't tempted to walk this road at all. However, in other cases, that might very well be the case – and still be a bad idea. Not only do we drop previously available type information, we also close our API against future extensions to types whose sources we don't control: We cannot make some new number type coming from a third party extend the `Number` trait.

Ruby's answer to that problem is *monkey patching*, polluting the global namespace with an extension to that new type, making it act like a `Number` after all. Java developers who have got beaten up by the Gang of Four in their youth, on the other hand, will think that an *Adapter* may solve all of their problems:

```
object Statistics {
  trait NumberLike[A] {
    def get: A
    def plus(y: NumberLike[A]): NumberLike[A]
    def minus(y: NumberLike[A]): NumberLike[A]
    def divide(y: Int): NumberLike[A]
  }
  case class NumberLikeDouble(x: Double) extends NumberLike[Double] {
    def get: Double = x
    def minus(y: NumberLike[Double]) = NumberLikeDouble(x - y.get)
    def plus(y: NumberLike[Double]) = NumberLikeDouble(x + y.get)
    def divide(y: Int) = NumberLikeDouble(x / y)
  }
  type Quartile[A] = (NumberLike[A], NumberLike[A], NumberLike[A])
  def median[A](xs: Vector[NumberLike[A]]): NumberLike[A] =
    xs(xs.size / 2)
  def quartiles[A](xs: Vector[NumberLike[A]]): Quartile[A] =
    (xs(xs.size / 4), median(xs), xs(xs.size / 4 * 3))
  def iqr[A](xs: Vector[NumberLike[A]]): NumberLike[A] =
    quartiles(xs) match {
      case (lowerQuartile, _, upperQuartile) =>
        upperQuartile.minus(lowerQuartile)
  }
  def mean[A](xs: Vector[NumberLike[A]]): NumberLike[A] =
    xs.reduce(_.plus(_)).divide(xs.size)
}
```

Now we have solved the problem of extensibility: Users of our library can pass in a `NumberLike` adapter for `Int` (which we would likely provide ourselves) or for any possible type that might behave like a number, without having to recompile the module in which our statistics methods are implemented.

However, always wrapping your numbers in an adapter is not only tiresome to write and read, it also means that you have to create a lot of instances of your adapter classes when interacting with our library.

Type classes to the rescue!

A powerful alternative to the approaches outlined so far is, of course, to define and use a type class. Type classes, one of the prominent features of the Haskell language, despite their name, haven't got anything to do with classes in object-oriented programming.

A type class `C` defines some behaviour in the form of operations that must be supported by a type `T` for it to be a member of type class `C`. Whether the type `T` is a member of the type class `C` is not inherent in the type. Rather, any developer can declare that a type is a member of a type class simply by providing implementations of the operations the type must support. Now, once `T` is made a member of the type class `C`, functions that have constrained one or more of their parameters to be members of `C` can be called with arguments of type `T`.

As such, type classes allow ad-hoc and retroactive polymorphism. Code that relies on type classes is open to extension without the need to create adapter objects.

Creating a type class

In Scala, type classes can be implemented and used by a combination of techniques. It's a little more involved than in Haskell, but also gives developers more control.

Creating a type class in Scala involves several steps. First, let's define a trait. This is the actual type class:

```
1 object Math {
2   trait NumberLike[T] {
3     def plus(x: T, y: T): T
4     def divide(x: T, y: Int): T
5     def minus(x: T, y: T): T
6   }
7 }
```

We have created a type class called `NumberLike`. Type classes always take one or more type parameters, and they are usually designed to be stateless, i.e. the methods defined on our `NumberLike` trait operate only on the passed in arguments. In particular, where our adapter above operated on its member of type `T` and one argument, the methods defined for our `NumberLike` type class take two parameters of type `T` each – the member has become the first parameter of the operations supported by `NumberLike`.

Providing default members

The second step in implementing a type class is usually to provide some default implementations of your type class trait in its companion object. We will see in a moment why this is generally a good strategy. First, however, let's do this, too, by making `Double` and `Int` members of our `NumberLike` type class:

```
object Math {
  trait NumberLike[T] {
    def plus(x: T, y: T): T
    def divide(x: T, y: Int): T
    def minus(x: T, y: T): T
  }
  object NumberLike {
    implicit object NumberLikeDouble extends NumberLike[Double] {
      def plus(x: Double, y: Double): Double = x + y
      def divide(x: Double, y: Int): Double = x / y
      def minus(x: Double, y: Double): Double = x - y
    }
    implicit object NumberLikeInt extends NumberLike[Int] {
      def plus(x: Int, y: Int): Int = x + y
      def divide(x: Int, y: Int): Int = x / y
      def minus(x: Int, y: Int): Int = x - y
    }
  }
}
```

Two things: First, you see that the two implementations are basically identical. That is not always the case when creating members of a type classes. Our `NumberLike` type class is just a rather narrow domain. Later in the chapter, I will give examples of type classes where there is a lot less room for duplication when implementing them for multiple types. Second, please ignore the fact that we are losing precision in `NumberLikeInt` by doing integer division. It's all to keep things simple for this example.

As you can see, members of type classes are usually singleton objects. Also, please note the `implicit` keyword before each of the type class implementations. This is one of the crucial elements for making type classes possible in Scala, making type class members implicitly available under certain conditions. More about that in the next section.

Coding against type classes

Now that we have our type class and two default implementations for common types, we want to code against this type class in our statistics module. Let's focus on the `mean` method for now:

```
object Statistics {
  import Math.NumberLike
  def mean[T](xs: Vector[T])(implicit ev: NumberLike[T]): T =
    ev.divide(xs.reduce(ev.plus(_, _)), xs.size)
}
```

This may look a little intimidating at first, but it's actually quite simple. Our method takes a type parameter `T` and a single parameter of type `Vector[T]`.

The idea to constrain a parameter to types that are members of a specific type class is realized by means of the `implicit` second parameter list. What does this mean? Basically, that a value of type `NumberLike[T]` must be implicitly available in the current scope. This is the case if an *implicit value* has been declared and made available in the current scope, very often by importing the package or object in which that implicit value is defined.

If and only if no other implicit value can be found, the compiler will look in the companion object of the type of the implicit parameter. Hence, as a library designer, putting your default type class implementations in the companion object of your type class trait means that users of your library can easily override these implementations with their own ones, which is exactly what you want. Users can also pass in an explicit value for an implicit parameter to override the implicit values that are in scope.

Let's see if the default type class implementations can be resolved:

```
val numbers =
  Vector[Double](13, 23.0, 42, 45, 61, 73, 96, 100, 199, 420, 900, 3839)
println(Statistics.mean(numbers))
```

Wonderful! If we try this with a `Vector[String]`, we get an error at compile time, stating that no implicit value could be found for parameter `ev: NumberLike[String]`. If you don't like this error message, you can customize it by annotating your type class trait with the `@implicitNotFound` annotation:

```
object Math {
  import annotation.implicitNotFound
  @implicitNotFound("No member of type class NumberLike in scope for ${T}")
  trait NumberLike[T] {
    def plus(x: T, y: T): T
    def divide(x: T, y: Int): T
    def minus(x: T, y: T): T
  }
}
```

Context bounds

A second, implicit parameter list on all methods that expect a member of a type class can be a little verbose. As a shortcut for implicit parameters with only one type parameter, Scala provides so-called *context bounds*. To show how those are used, we are going to implement our other statistics methods using those instead:

```
object Statistics {
  import Math.NumberLike
  def mean[T](xs: Vector[T])(implicit ev: NumberLike[T]): T =
    ev.divide(xs.reduce(ev.plus(_, _)), xs.size)
  def median[T : NumberLike](xs: Vector[T]): T = xs(xs.size / 2)
  def quartiles[T: NumberLike](xs: Vector[T]): (T, T, T) =
    (xs(xs.size / 4), median(xs), xs(xs.size / 4 * 3))
  def iqr[T: NumberLike](xs: Vector[T]): T = quartiles(xs) match {
    case (lowerQuartile, _, upperQuartile) =>
      implicitly[NumberLike[T]].minus(upperQuartile, lowerQuartile)
  }
}
```

A context bound `T : NumberLike` means that an implicit value of type `NumberLike[T]` must be available, and so is really equivalent to having a second implicit parameter list with a `NumberLike[T]` in it. If you want to access that implicitly available value, however, you need to call the `implicitly` method, as we do in the `iqr` method. If your type class requires more than one type parameter, you cannot use the context bound syntax.

Custom type class members

As a user of a library that makes use of type classes, you will sooner or later have types that you want to make members of those type classes. For instance, we might want to use the statistics library for instances of the Joda Time `Duration` type. To do that, we need Joda Time on our classpath, of course:

```
libraryDependencies += "joda-time" % "joda-time" % "2.1"

libraryDependencies += "org.joda" % "joda-convert" % "1.3"
```

Now we just have to create an implicitly available implementation of `NumberLike` (please make sure you have Joda Time on your classpath when trying this out):

```
object JodaImplicits {
  import Math.NumberLike
  import org.joda.time.Duration
  implicit object NumberLikeDuration extends NumberLike[Duration] {
    def plus(x: Duration, y: Duration): Duration = x.plus(y)
    def divide(x: Duration, y: Duration): Duration =
      Duration.millis(x.getMillis / y)
    def minus(x: Duration, y: Duration): Duration = x.minus(y)
  }
}
```

If we import the package or object containing this `NumberLike` implementation, we can now compute the mean value for a bunch of durations:

```
import Statistics._
import JodaImplicits._
import org.joda.time.Duration._

val durations = Vector(
  standardSeconds(20),
  standardSeconds(57),
  standardMinutes(2),
  standardMinutes(17),
  standardMinutes(30),
  standardMinutes(58),
  standardHours(2),
  standardHours(5),
  standardHours(8),
  standardHours(17),
  standardDays(1),
  standardDays(4))
println(mean(durations).getStandardHours)
```

Use cases

Our `NumberLike` type class was a nice exercise, but Scala already ships with the `Numeric` type class, which allows you to call methods like `sum` or `product` on collections for whose type `T` a `Numeric[T]` is available. Another type class in the standard library that you will use a lot is `Ordering`, which allows you to provide an implicit ordering for your own types, available to the `sort` method on Scala's collections.

There are more type classes in the standard library, but not all of them are ones you have to deal with on a regular basis as a Scala developer.

A very common use case in third-party libraries is that of object serialization and deserialization, most notably to and from JSON. By making your classes members of an appropriate formatter type class, you can customize the way your classes are serialized to JSON, XML or whatever format is currently the new black.

Mapping between Scala types and ones supported by your database driver is also commonly made customizable and extensible via type classes.

Summary

Once you start to do some serious work with Scala, you will inevitably stumble upon type classes. I hope that after reading this chapter, you are prepared to take advantage of this powerful technique.

Scala type classes allow you to develop your Scala code in such a way that it's open for retroactive extension while retaining as much concrete type information as possible. In contrast to approaches from other languages, they give developers full control, as default type class implementations can be overridden without much hassle, and type classes implementations are not made available in the global namespace.

You will see that this technique is especially useful when writing libraries intended to be used by others, but type classes also have their use in application code to decrease coupling between modules.

Path-dependent types

In the previous chapter, you were introduced you to the idea of type classes – a pattern that allows you to design your programs to be open for extension without giving up important information about concrete types. In this chapter, we will stick with Scala's type system and deal with one of its features that distinguishes it from most other mainstream programming languages: Scala's form of dependent types, in particular path-dependent types and dependent method types.

One of the most widely used arguments against static typing is that *"the compiler is just in the way"* and that in the end, it's all only data, so why care about building up a complex hierarchy of types?

In the end, having static types is all about preventing bugs by allowing the Ã¼ber-smart compiler to humiliate you on a regular basis, making sure you're doing the right thing before it's too late.

Using path-dependent types is one powerful way to help the compiler prevent you from introducing bugs, as it places logic that is usually only available at runtime into types.

Sometimes, accidentally introducing path-dependent types can lead to frustation, though, especially if you have never heard of them. Hence, it's definitely a good idea to get familiar with them, whether you decide to put them to use or not.

The problem

We will start with a problem that path-dependent types can help us solving: In the realm of fan fiction, the most atrocious things happen – usually, the involved characters will end up making out with each other, regardless how inappropriate it is. There is even crossover fan fiction, in which two characters from different franchises are making out with each other.

However, elitist fan fiction writers look down on this. Surely there is a way to prevent such wrongdoing! Here is a first version of our domain model:

```
1  object Franchise {
2     case class Character(name: String)
3   }
4  class Franchise(name: String) {
5    import Franchise.Character
6    def createFanFiction(
7      lovestruck: Character,
8      objectOfDesire: Character): (Character, Character) =
9        (lovestruck, objectOfDesire)
10 }
```

Characters are represented by instances of the `Character` case class, and the `Franchise` class has a method to create a new piece of fan fiction about two characters. Let's create two franchises and some characters:

```
val starTrek = new Franchise("Star Trek")
val starWars = new Franchise("Star Wars")

val quark = Franchise.Character("Quark")
val jadzia = Franchise.Character("Jadzia Dax")

val luke = Franchise.Character("Luke Skywalker")
val yoda = Franchise.Character("Yoda")
```

Unfortunately, at the moment we are unable to prevent bad things from happening:

```
starTrek.createFanFiction(lovestruck = jadzia, objectOfDesire = luke)
```

Horrors of horrors! Someone has created a piece of fan fiction in which Jadzia Dax is making out with Luke Skywalker. Preposterous! Clearly, we should not allow this. Your first intuition might be to somehow check at runtime that two characters making out are from the same franchise. For example, we could change the model like so:

```
object Franchise {
  case class Character(name: String, franchise: Franchise)
}
class Franchise(name: String) {
  import Franchise.Character
  def createFanFiction(
      lovestruck: Character,
      objectOfDesire: Character): (Character, Character) = {
    require(lovestruck.franchise == objectOfDesire.franchise)
    (lovestruck, objectOfDesire)
  }
}
```

Now, the `Character` instances have a reference to their `Franchise`, and trying to create a fan fiction with characters from different franchises will lead to an `IllegalArgumentException` (feel free to try this out in a REPL).

Safer fiction with path-dependent types

This is pretty good, isn't it? It's the kind of fail-fast behaviour we have been indoctrinated with for years. However, with Scala, we can do better. There is a way to fail even faster – not at runtime, but at compile time. To achieve that, we need to encode the connection between a `Character` and its `Franchise` at the type level.

Luckily, the way Scala's nested types work allow us to do that. In Scala, a nested type is bound to a specific instance of the outer type, not to the outer type itself. This means that if you try to use an instance of the inner type outside of the instance of the enclosing type, you will face a compile error:

```
class A {
  class B
  var b: Option[B] = None
}
val a1 = new A
val a2 = new A
val b1 = new a1.B
val b2 = new a2.B
a1.b = Some(b1)
a2.b = Some(b1) // does not compile
```

You cannot simply assign an instance of the `B` that is bound to `a2` to the field on `a1` – the one is an `a2.B`, the other expects an `a1.B`. The dot syntax represents the path to the type, going along concrete instances of other types. Hence the name, path-dependent types.

We can put these to use in order to prevent characters from different franchises making out with each other:

```
class Franchise(name: String) {
  case class Character(name: String)
  def createFanFictionWith(
    lovestruck: Character,
    objectOfDesire: Character): (Character, Character) =
      (lovestruck, objectOfDesire)
}
```

Now, the type `Character` is nested in the type `Franchise`, which means that it is dependent on a specific enclosing instance of the `Franchise` type.

Let's create our example franchises and characters again:

```
val starTrek = new Franchise("Star Trek")
val starWars = new Franchise("Star Wars")

val quark = starTrek.Character("Quark")
val jadzia = starTrek.Character("Jadzia Dax")

val luke = starWars.Character("Luke Skywalker")
val yoda = starWars.Character("Yoda")
```

You can already see in how our `Character` instances are created that their types are bound to a specific franchise. Let's see what happens if we try to put some of these characters together:

```
starTrek.createFanFictionWith(lovestruck = quark, objectOfDesire = jadzia)
starWars.createFanFictionWith(lovestruck = luke, objectOfDesire = yoda)
```

These two compile, as expected. They *are* tasteless, but what can we do?

Now, let's see what happens if we try to create some fan fiction about Jadzia Dax and Luke Skywalker:

```
starTrek.createFanFictionWith(lovestruck = jadzia, objectOfDesire = luke)
```

Et voilÃ : The thing that should not be does not even compile! The compiler complains about a type mismatch:

```
found   : starWars.Character
required: starTrek.Character
starTrek.createFanFictionWith(lovestruck = jadzia, objectOfDesire = luke)
                                                                    ^
```

This technique also works if our method is not defined on the `Franchise` class, but in some other module. In this case, we can make use of dependent method types, where the type of one parameter depends on a previous parameter:

```
def createFanFiction
    (f: Franchise)(lovestruck: f.Character, objectOfDesire: f.Character) =
  (lovestruck, objectOfDesire)
```

As you can see, the type of the `lovestruck` and `objectOfDesire` parameters depends on the `Franchise` instance passed to the method. Note that this only works if the instance on which other types depend is in its own parameter list.

Abstract type members

Often, dependent method types are used in conjunction with abstract type members. Suppose we want to develop a hipsterrific key-value store. It will only support setting and getting the value for a key, but in a typesafe manner. Here is our oversimplified implementation:

```
object AwesomeDB {
  abstract class Key(name: String) {
    type Value
  }
}
import AwesomeDB.Key
class AwesomeDB {
  import collection.mutable.Map
  val data = Map.empty[Key, Any]
  def get(key: Key): Option[key.Value] =
    data.get(key).asInstanceOf[Option[key.Value]]
  def set(key: Key)(value: key.Value): Unit =
    data.update(key, value)
}
```

We have defined a class `Key` with an abstract type member `Value`. The methods on `AwesomeDB` refer to that type without ever knowing or caring about the specific manifestation of this abstract type.

We can now define some concrete keys that we want to use:

trait IntValued extends Key { type Value = Int } trait StringValued extends Key { type Value = String } object Keys { val foo = new Key(“foo”) with IntValued val bar = new Key(“bar”) with StringValued } {% endcodeblock %}

Now we can set and get key/value pairs in a typesafe manner:

```
val dataStore = new AwesomeDB
dataStore.set(Keys.foo)(23)
val i: Option[Int] = dataStore.get(Keys.foo)
dataStore.set(Keys.foo)("23") // does not compile
```

Path-dependent types in practice

While path-dependent types are not necessarily omnipresent in your typical Scala code, they do have a lot of practical value beyond modelling the domain of fan fiction.

One of the most widespread uses is probably seen in combination with the *cake pattern*, which is a technique for composing your components and managing their dependencies, relying solely on features of the language. See the excellent articles by Debasish Ghosh[11] and Precog's Daniel Spiewak[12] to learn more about both the cake pattern and how it can be improved by incorporating path-dependent types.

Summary

In general, whenever you want to make sure that objects created or managed by a specific instance of another type cannot accidentally or purposely be interchanged or mixed, path-dependent types are the way to go.

Path-dependent types and dependent method types play a crucial role for attempts to encode information into types that is typically only known at runtime, for instance heterogenous lists, type-level representations of natural numbers and collections that carry their size in their type. Miles Sabin is exploring the limits of Scala's type system in this respect in his excellent library Shapeless[13].

[11] http://debasishg.blogspot.ie/2013/02/modular-abstractions-in-scala-with.html
[12] http://precog.com/blog/Existential-Types-FTW/
[13] https://github.com/milessabin/shapeless

The Actor approach to concurrency

After several chapters about how you can leverage the Scala type system to achieve a great amount of both flexibility and compile-time safety, we are now shifting back to a topic that we already tackled previously in this book: Scala's take on concurrency.

In these earlier chapters, you learned about an approach that allows you to work asynchronously by making use of composable Futures.

This approach is a very good fit for numerous problems. However, it's not the only one Scala has to offer. A second cornerstone of Scala concurrency is the *Actor* model. It provides an approach to concurrency that is entirely based on passing messages between processes.

Actors are not a new idea – the most prominent implementation of this model can be found in Erlang. The Scala core library has had its own actors library for a long time, but it faces the destiny of deprecation in the coming Scala version 2.11, when it will ultimately be replaced by the actors implementation provided by the Akka[14] toolkit, which has been a de-facto standard for actor-based development with Scala for quite a while.

In this chapter, you will be introduced to the rationale behind Akka's actor model and learn the basics of coding within this paradigm using the Akka toolkit. It is by no means an in-depth discussion of everything you need to know about Akka actors, and in that, it differs from most of the previous chapters in this book. Rather, the intention is to familiarize you with the Akka mindset and serve as an initial spark to get you excited about it.

The problems with shared mutable state

The predominant approach to concurrency today is that of shared mutable state – a large number of stateful objects whose state can be changed by multiple parts of your application, each running in their own thread. Typically, the code is interspersed with read and write locks, to make sure that the state can only be changed in a controlled way and prevent multiple threads from mutating it simultaneously. At the same time, we are trying hard not to lock too big a block of code, as this can drastically slow down the application.

More often than not, code like this has originally been written without having concurrency in mind at all – only to be made fit for a multi-threaded world once the need arose. While writing software without the need for concurrency like this leads to very straightforward code, adapting it to the needs of a concurrent world leads to code that is really, really difficult to read and understand.

The core problem is that low-level synchronization constructs like locks and threads are *very hard to reason about*. As a consequence, it's very hard to get it right: If you can't easily reason about

[14] http://akka.io

what's going on, you can be sure that nasty bugs will ensue, from race conditions to deadlocks or just strange behaviour – maybe you'll only notice after some months, long after your code has been deployed to your production servers.

Also, working with these low-level constructs makes it a real challenge to achieve an acceptable performance.

The Actor model

The Actor programming model is aimed at avoiding all the problems described above, allowing you to write highly performant concurrent code that is easy to reason about. Unlike the widely used approach of shared mutable state, it requires you to design and write your application from the ground up with concurrency in mind – it's not really possible to add support for it later on.

The idea is that your application consists of lots of light-weight entities called actors. Each of these actors is responsible for only a very small task, and is thus easy to reason about. A more complex business logic arises out of the interaction between several actors, delegating tasks to others or passing messages to collaborators for other reasons.

The Actor System

Actors are pitiful creatures: They cannot live on their own. Rather, each and every actor in Akka resides in and is created by an *actor system*. Aside from allowing you to create and find actors, an `ActorSystem` provides for a whole bunch of additional functionality, none of which shall concern us right now.

In order to try out the example code, please add the following resolver and dependency to your SBT-based Scala 2.10 project first:

```
resolvers +=
  "Typesafe Releases" at "http://repo.typesafe.com/typesafe/releases"

libraryDependencies += "com.typesafe.akka" %% "akka-actor" % "2.1.0"
```

Now, let's create an `ActorSystem`. We'll need it as an environment for our actors:

```
import akka.actor.ActorSystem
object Barista extends App {
  val system = ActorSystem("Barista")
  system.shutdown()
}
```

We created a new instance of `ActorSystem` and gave it the name `"Barista"` – we are returning to the domain of coffee, which should be familiar from the chapter on composable futures.

Finally, we are good citizens and shut down our actor system once we no longer need it.

Defining an actor

Whether your application consists of a few dozen or a few million actors totally depends on your use case, but Akka is absolutely okay with a few million. You might be baffled by this insanely high number. It's important to understand that there is not a one-to-one relationship between an actor and a thread. You would soon run out of memory if that were the case. Rather, due to the non-blocking nature of actors, one thread can execute many actors – switching between them depending on which of them has messages to be processed.

To understand what is actually happening, let's first create a very simple actor, a `Barista` that can receive orders but doesn't really do anything apart from printing messages to the console:

```
1  sealed trait CoffeeRequest
2  case object CappuccinoRequest extends CoffeeRequest
3  case object EspressoRequest extends CoffeeRequest
4
5  import akka.actor.Actor
6  class Barista extends Actor {
7    def receive = {
8      case CappuccinoRequest => println("I have to prepare a cappuccino!")
9      case EspressoRequest => println("Let's prepare an espresso.")
10   }
11 }
```

First, we define the types of messages that our actor understands. Typically, case classes are used for messages sent between actors if you need to pass along any parameters. If all the actor needs is an unparameterized message, this message is typically represented as a case object – which is exactly what we are doing here.

In any case, it's crucial that your messages are immutable, or else bad things will happen.

Next, let's have a look at our class `Barista`, which is the actual actor, extending the aptly named `Actor` trait. Said trait defines a method `receive` which returns a value of type `Receive`. The latter is really only a type alias for `PartialFunction[Any, Unit]`.

Processing messages

So what's the meaning of this `receive` method? The return type, `PartialFunction[Any, Unit]` may seem strange to you in more than one respect.

In a nutshell, the partial function returned by the `receive` method is responsible for processing your messages. Whenever another part of your software – be it another actor or not – sends your actor a message, Akka will *eventually* let it process this message by calling the partial function returned by your actor's `receive` method, passing it the message as an argument.

Side-effecting

When processing a message, an actor can do whatever you want it to, apart from returning a value.

Wat!?

As the return type of `Unit` suggests, your partial function is side-effecting. This might come as a bit of a shock to you after we emphasized the usage of pure functions all the time. For a concurrent programming model, this actually makes a lot of sense. Actors are where your state is located, and having some clearly defined places where side-effects will occur in a controllable manner is totally fine – each message your actor receives is processed in isolation, one after another, so there is no need to reason about synchronization or locks.

Untyped

But… this partial function is not only side-effecting, it's also as untyped as you can get in Scala, expecting an argument of type `Any`. Why is that, when we have such a powerful type system at our fingertips?

This has a lot to do with some important design choices in Akka that allow you to do things like forwarding messages to other actors, installing load balancing or proxying actors without the sender having to know anything about them and so on.

In practice, this is usually not a problem. With the messages themselves being strongly typed, you typically use pattern matching for processing those types of messages you are interested in, just as we did in our tiny example above.

Sometimes though, the weakly typed actors can indeed lead to nasty bugs the compiler can't catch for you. If you have grown to love the benefits of a strong type system and think you don't want to go away from that at any costs for some parts of your application, you may want to look at Akka's new experimental Typed Channels[15] feature.

Asynchronous and non-blocking

I wrote above that Akka would let your actor *eventually* process a message sent to it. This is important to keep in mind: Sending a message and processing it is done in an asynchronous and non-blocking fashion. The sender will not be blocked until the message has been processed by the receiver. Instead, they can immediately continue with their own work. Maybe they expect to get a messsage from your actor in return after a while, or maybe they are not interested in hearing back from your actor at all.

What really happens when some component sends a message to an actor is that this message is delivered to the actor's *mailbox*, which is basically a queue. Placing a message in an actor's mailbox is a non-blocking operation, i.e. the sender doesn't have to wait until the message is actually enqueued in the recipient's mailbox.

[15] http://doc.akka.io/docs/akka/snapshot/scala/typed-channels.html

The *dispatcher* will notice the arrival of a new message in an actor's mailbox, again asynchronously. If the actor is not already processing a previous message, it is now allocated to one of the threads available in the execution context. Once the actor is done processing any previous messages, the dispatcher sends it the next message from its mailbox for processing.

The actor blocks the thread to which it is allocated for as long as it takes to process the message. While this doesn't block the sender of the message, it means that lengthy operations degrade overall performance, as all the other actors have to be scheduled for processing messages on one of the remaining threads.

Hence, a core principle to follow for your `Receive` partial functions is to spend as little time inside them as possible. Most importantly, avoid calling blocking code inside your message processing code, if possible at all.

Of course, this is something you can't prevent doing completely – the majority of database drivers nowadays is still blocking, and you will want to be able to persist data or query for it from your actor-based application. There are solutions to this dilemma, but we won't cover them in this introductory chapter.

Creating an actor

Defining an actor is all well and good, but how do we actually use our `Barista` actor in our application? To do that, we have to create a new instance of our `Barista` actor. You might be tempted to do it the usual way, by calling its constructor like so:

```
1  val barista = new Barista // will throw exception
```

This will not work! Akka will thank you with an `ActorInitializationException`. The thing is, in order for the whole actor thingie to work properly, your actors need to be managed by the `ActorSystem` and its components. Hence, you have to ask the actor system for a new instance of your actor:

```
1  import akka.actor.{ActorRef, Props}
2  val barista: ActorRef = system.actorOf(Props[Barista], "Barista")
```

The `actorOf` method defined on `ActorSystem` expects a `Props` instance, which provides a means of configuring newly created actors, and, optionally, a name for your actor instance. We are using the simplest form of creating such a `Props` instance, providing the `apply` method of the companion object with a type parameter. Akka will then create a new instance of the actor of the given type by calling its default constructor.

Be aware that the type of the object returned by `actorOf` is not `Barista`, but `ActorRef`. Actors never communicate with another directly and hence there are supposed to be no direct references to actor

instances. Instead, actors or other components of your application aquire references to the actors they need to send messages to.

Thus, an `ActorRef` acts as some kind of proxy to the actual actor. This is convenient because an `ActorRef` can be serialized, allowing it to be a proxy for a remote actor on some other machine. For the component aquiring an `ActorRef`, the location of the actor – local in the same JVM or remote on some other machine – is completely transparent. We call this property *location transparency*.

Please note that `ActorRef` is not parameterized by type. Any `ActorRef` can be exchanged for another, allowing you to send arbitrary messages to any `ActorRef`. This is by design and, as already mentioned, allows for easily modifying the topology of your actor system wihout having to make any changes to the senders.

Sending messages

Now that we have created an instance of our `Barista` actor and got an `ActorRef` linked to it, we can send it a message. This is done by calling the `!` method on the `ActorRef`:

```
1 barista ! CappuccinoRequest
2 barista ! EspressoRequest
3 println("I ordered a cappuccino and an espresso")
```

Calling the `!` is a fire-and-forget operation: You *tell* the `Barista` that you want a cappuccino, but you don't wait for their response. It's the most common way in Akka for interacting with other actors. By calling this method, you tell Akka to enqueue your message in the recipient's mailbox. As described above, this doesn't block, and eventually the recipient actor will process your message.

Due to the asynchronous nature, the result of the above code is not deterministic. It might look like this:

```
1 I have to prepare a cappuccino!
2 I ordered a cappuccino and an espresso
3 Let's prepare an espresso.
```

Even though we first sent the two messages to the `Barista` actor's mailbox, between the processing of the first and second message, our own output is printed to the console.

Answering to messages

Sometimes, being able to tell others what to do just doesn't cut it. You would like to be able to answer by in turn sending a message to the sender of a message you got – all asynchronously of course.

To enable you to do that and lots of other things that are of no concern to us right now, actors have a method called `sender`, which returns the `ActorRef` of the sender of the last message, i.e. the one you are currently processing.

But how does it know about that sender? The answer can be found in the signature of the `!` method, which has a second, implicit parameter list:

```
1  def !(message: Any)(implicit sender: ActorRef = Actor.noSender): Unit
```

When called from an actor, its `ActorRef` is passed on as the implicit `sender` argument.

Let's change our `Barista` so that they immediately send a `Bill` to the sender of a `CoffeeRequest` before printing their usual output to the console:

```
1  case class Bill(cents: Int)
2  case object ClosingTime
3  class Barista extends Actor {
4    def receive = {
5      case CappuccinoRequest =>
6        sender ! Bill(250)
7        println("I have to prepare a cappuccino!")
8      case EspressoRequest =>
9        sender ! Bill(200)
10       println("Let's prepare an espresso.")
11     case ClosingTime => context.system.shutdown()
12   }
13 }
```

While we are at it, we are introducing a new message, `ClosingTime`. The `Barista` reacts to it by shutting down the actor system, which they, like all actors, can access via their `ActorContext`.

Now, let's introduce a second actor representing a customer:

```
case object CaffeineWithdrawalWarning
class Customer(caffeineSource: ActorRef) extends Actor {
  def receive = {
    case CaffeineWithdrawalWarning => caffeineSource ! EspressoRequest
    case Bill(cents) => println(s"I have to pay $cents cents, or else!")
  }
}
```

This actor is a real coffee junkie, so it needs to be able to order new coffee. We pass it an `ActorRef` in the constructor – for the `Customer`, this is simply its `caffeineSource` – it doesn't know whether this `ActorRef` points to a `Barista` or something else. It knows that it can send `CoffeeRequest` messages to it, and that is all that matters to them.

Finally, we need to create these two actors and send the customer a `CaffeineWithdrawalWarning` to get things rolling:

```
val barista = system.actorOf(Props[Barista], "Barista")
val customer = system.actorOf(Props(new Customer(barista)), "Customer")
customer ! CaffeineWithdrawalWarning
barista ! ClosingTime
```

Here, for the `Customer` actor, we are using a different factory method for creating a `Props` instance: As a by-name parameter, we pass it a code block that creates a new instance of the `Customer` actor. We need to do this because we want to pass the `ActorRef` of our `Barista` actor to the constructor of the `Customer` actor.

Sending the `CaffeineWithdrawalWarning` to the customer makes it send an `EspressoRequest` to the barista who will then send a `Bill` back to the customer. The output of this may look like this:

```
Let's prepare an espresso.
I have to pay 200 cents, or else!
```

First, while processing the `EspressoRequest` message, the `Barista` sends a message to the sender of that message, the `Customer` actor. However, this operation doesn't block until the latter processes it. The `Barista` actor can continue processing the `EspressoRequest` immediately, and does this by printing to the console. Shortly after, the `Customer` starts to process the `Bill` message and in turn prints to the console.

Asking questions

Sometimes, sending an actor a message and expecting a message in return at some later time isn't an option – the most common place where this is the case is in components that need to interface

with actors, but are not actors themselves. Living outside of the actor world, they cannot receive messages.

For situations such as these, there is Akka's *ask* support, which provides some sort of bridge between actor-based and future-based concurrency. From the client perspective, it works like this:

```
1  import akka.pattern.ask
2  import akka.util.Timeout
3  import scala.concurrent.duration._
4  implicit val timeout = Timeout(2.second)
5  implicit val ec = system.dispatcher
6  val f: Future[Any] = barista2 ? CappuccinoRequest
7  f.onSuccess {
8    case Bill(cents) => println(s"Will pay $cents cents for a cappuccino")
9  }
```

First, you need to import support for the *ask* syntax and create an implicit timeout for the `Future` returned by the `?` method. Also, the `Future` needs an `ExecutionContext`. Here, we simply use the default dispatcher of our `ActorSystem`, which is conveniently also an `ExecutionContext`.

As you can see, the returned future is untyped – it's a `Future[Any]`. This shouldn't come as a surprise, since it's really a received message from an actor, and those are untyped, too.

For the actor that is being asked, this is actually the same as sending some message to the sender of a processed message. This is why asking our `Barista` works out of the box without having to change anything in our `Barista` actor.

Once the actor being asked sends a message to the sender, the `Promise` belonging to the returned `Future` is completed.

Generally, telling is preferable to asking, because it's more resource-sparing. Akka is not for polite people! However, there are situations where you really need to ask, and then it's perfectly fine to do so.

Stateful actors

Each actor may maintain an internal state, but that's not strictly necessary. Sometimes, a large part of the overall application state consists of the information carried by the immutable messages passed between actors.

An actor only ever processes one message at a time. While doing so, it may modify its internal state. This means that there is some kind of mutable state in an actor, but since each message is processed in isolation, there is no way the internal state of our actor can get messed up due to concurrency problems.

To illustrate, let's turn our stateless `Barista` into an actor carrying state, by simply counting the number of orders:

```
1  class Barista extends Actor {
2    var cappuccinoCount = 0
3    var espressoCount = 0
4    def receive = {
5      case CappuccinoRequest =>
6        sender ! Bill(250)
7        cappuccinoCount += 1
8        println(s"I have to prepare cappuccino #$cappuccinoCount")
9      case EspressoRequest =>
10       sender ! Bill(200)
11       espressoCount += 1
12       println(s"Let's prepare espresso #$espressoCount.")
13     case ClosingTime => context.system.shutdown()
14   }
15 }
```

We introduced two `vars`, `cappuccinoCount` and `espressoCount` that are incremented with each respective order. This is actually the first time in this book that we have used a `var`. While to be avoided in functional programming, they are really the only way to allow your actors to carry state. Since each message is processed in isolation, our above code is similar to using `AtomicInteger` values in a non-actor environment.

Summary

And here ends our introduction to the actor programming model for concurrency and how to work within this paradigm using Akka. While we have really only scratched the surface and have ignored some important concepts of Akka, I hope to have given enough of an insight into this approach to concurrency to give you a basic understanding and get you interested in learning more.

In the next chapter, we will elaborate our little example, adding some meaningful behaviour to it while introducing more of the ideas behind Akka actors, among them the question of how errors are handled in an actor system.

Dealing with failure in actor systems

In the previous chapter, you were introduced to the second cornerstone of Scala concurrency: The *actor* model, which complements the model based on composable *futures* backed by *promises*. You learnt how to define and create actors, how to send messages to them and how an actor processes these messages, possibly modifying its internal state as a result or asynchronously sending a response message to the sender.

While that was hopefully enough to get you interested in the actor model for concurrency, we left out some crucial concepts you will want to know about before starting to develop actor-based applications that consist of more than a simple echo actor.

The actor model is meant to help you achieve a high level of fault tolerance. In this chapter, we are going to have a look at how to deal with failure in an actor-based application, which is fundamentally different from error handling in a traditional layered server architecture.

The way you deal with failure is closely linked to some core Akka concepts and to some of the elements an actor system in Akka consists of. Hence, this chapter will also serve as a guide to those ideas and components.

Actor hierarchies

Before going into what happens when an error occurs in one of your actors, it's essential to introduce one crucial idea underlying the actor approach to concurrency – an idea that is the very foundation for allowing you to build fault-tolerant concurrent applications: Actors are organized in a hierarchy.

So what does this mean? First of all, it means that every single of your actors has got a parent actor, and that each actor can create child actors. Basically, you can think of an actor system as a pyramid of actors. Parent actors watch over their children, just as in real life, taking care that they get back on their feet if they stumble. You will see shortly how exactly this is done.

The guardian actor

In the previous chapter, we only had two different actors, a `Barista` actor and a `Customer` actor. I will not repeat their rather trivial implementations, but focus on how we created instances of these actor types:

```
import akka.actor.ActorSystem
val system = ActorSystem("Coffeehouse")
val barista = system.actorOf(Props[Barista], "Barista")
val customer = system.actorOf(Props(new Customer(barista)), "Customer")
```

As you can see, we create these two actors by calling the `actorOf` method defined on the `ActorSystem` type.

So what is the parent of these two actors? Is it the actor system? Not quite, but close. The actor system is not an actor itself, but it has got a so-called *guardian actor* that serves as the parent of all root-level user actors, i.e. actors we create by calling `actorOf` on our actor system.

There shouldn't be a whole lot of actors in your system that are children of the guardian actor. It really makes more sense to have only a few top-level actors, each of them delegating most of the work to their children.

Actor paths

The hierarchical structure of an actor system becomes apparent when looking at the *actor paths* of the actors you create. These are basically URLs by which actors can be addressed. You can get an actor's path by calling `path` on its `ActorRef`:

```
barista.path
// => akka.actor.ActorPath = akka://Coffeehouse/user/Barista
customer.path
// => akka.actor.ActorPath = akka://Coffeehouse/user/Customer
```

The `akka` protocol is followed by the name of our actor system, the name of the user guardian actor and, finally, the name we gave our actor when calling `actorOf` on the system. In the case of remote actors, running on different machines, you would additionally see a host and a port.

Actor paths can be used to look up another actor. For example, instead of requiring the barista reference in its constructor, the `Customer` actor could call the `actorFor` method on its `ActorContext`, passing in a relative path to retrieve a reference to the barista:

```
context.actorFor("../Barista")
```

However, while being able to look up an actor by its path can sometimes come in handy, it's often a much better idea to pass in dependencies in the constructor, just as we did before. Too much intimate knowledge about where your dependencies are located in the actor system makes your system more susceptible to bugs, and it will be difficult to refactor later on.

An example hierarchy

To illustrate how parents watch over their children and what this has got to do with keeping your system fault-tolerant, I'm going to stick to the domain of the coffeehouse. Let's give the `Barista` actor a child actor to which it can delegate some of the work involved in running a coffeehouse.

If we really were to model the work of a barista, we were likely giving them a bunch of child actors for all the various subtasks. But to keep this chapter focused, we have to be a little simplistic with our example.

Let's assume that the barista has got a register. It processes transactions, printing appropriate receipts and incrementing the day's sales so far. Here is a first version of it:

```
1  import akka.actor._
2  object Register {
3    sealed trait Article
4    case object Espresso extends Article
5    case object Cappuccino extends Article
6    case class Transaction(article: Article)
7  }
8  class Register extends Actor {
9    import Register._
10   import Barista._
11   var revenue = 0
12   val prices = Map[Article, Int](Espresso -> 150, Cappuccino -> 250)
13   def receive = {
14     case Transaction(article) =>
15       val price = prices(article)
16       sender ! createReceipt(price)
17       revenue += price
18   }
19   def createReceipt(price: Int): Receipt = Receipt(price)
20 }
```

It contains an immutable map of the prices for each article, and an integer variable representing the revenue. Whenever it receives a `Transaction` message, it increments the revenue accordingly and returns a printed receipt to the sender.

The `Register` actor, as already mentioned, is supposed to be a child actor of the `Barista` actor, which means that we will not create it from our actor system, but from within our `Barista` actor. The initial version of our actor-come-parent looks like this:

```
object Barista {
  case object EspressoRequest
  case object ClosingTime
  case class EspressoCup(state: EspressoCup.State)
  object EspressoCup {
    sealed trait State
    case object Clean extends State
    case object Filled extends State
    case object Dirty extends State
  }
  case class Receipt(amount: Int)
}
class Barista extends Actor {
  import Barista._
  import Register._
  import EspressoCup._
  import context.dispatcher
  import akka.util.Timeout
  import akka.pattern.ask
  import akka.pattern.pipe
  import concurrent.duration._

  implicit val timeout = Timeout(4.seconds)
  val register = context.actorOf(Props[Register], "Register")
  def receive = {
    case EspressoRequest =>
      val receipt = register ? Transaction(Espresso)
      receipt.map((EspressoCup(Filled), _)).pipeTo(sender)
    case ClosingTime => context.stop(self)
  }
}
```

First off, we define the message types that our `Barista` actor is able to deal with. An `EspressoCup` can have one out of a fixed set of states, which we ensure by using a sealed trait.

The more interesting part is to be found in the implementation of the `Barista` class. The `dispatcher`, `ask`, and `pipe` imports as well as the implicit timeout are required because we make use of Akka's ask syntax and futures in our `Receive` partial function: When we receive an `EspressoRequest`, we ask the `Register` actor for a `Receipt` for our `Transaction`. This is then combined with a filled espresso cup and piped to the sender, which will thus receive a tuple of type `(EspressoCup, Receipt)`. This kind of delegating subtasks to child actors and then aggregating or amending their work is typical for actor-based applications.

Also, note how we create our child actor by calling `actorOf` on our `ActorContext` instead of the

`ActorSystem`. By doing so, the actor we create becomes a child actor of the one who called this method, instead of a top-level actor whose parent is the guardian actor.

Finally, here is our `Customer` actor, which, like the `Barista` actor, will sit at the top level, just below the guardian actor:

```
object Customer {
  case object CaffeineWithdrawalWarning
}
class Customer(coffeeSource: ActorRef) extends Actor with ActorLogging {
  import Customer._
  import Barista._
  import EspressoCup._
  def receive = {
    case CaffeineWithdrawalWarning => coffeeSource ! EspressoRequest
    case (EspressoCup(Filled), Receipt(amount)) =>
      log.info(s"yay, caffeine for ${self}!")
  }
}
```

It is not terribly interesting for our tutorial, which focuses more on the `Barista` actor hierarchy. What's new is the use of the `ActorLogging` trait, which allows us to write to the log instead of printing to the console.

Now, if we create our actor system and populate it with a `Barista` and two `Customer` actors, we can happily feed our two under-caffeinated addicts with a shot of black gold:

```
import Customer._
val system = ActorSystem("Coffeehouse")
val barista = system.actorOf(Props[Barista], "Barista")
val customerJohnny = system.actorOf(Props(new Customer(barista)), "Johnny")
val customerAlina = system.actorOf(Props(new Customer(barista)), "Alina")
customerJohnny ! CaffeineWithdrawalWarning
customerAlina ! CaffeineWithdrawalWarning
```

If you try this out, you should see two log messages from happy customers.

To crash or not to crash?

Of course, what we are really interested in, at least in this chapter, is not happy customers, but the question of what happens if things go wrong.

Our register is a fragile device – its printing functionality is not as reliable as it should be. Every so often, a paper jam causes it to fail. Let's add a `PaperJamException` type to the `Register` companion object:

```
class PaperJamException(msg: String) extends Exception(msg)
```

Then, let's change the `createReceipt` method in our `Register` actor accordingly:

```
def createReceipt(price: Int): Receipt = {
  import util.Random
  if (Random.nextBoolean())
    throw new PaperJamException("OMG, not again!")
  Receipt(price)
}
```

Now, when processing a `Transaction` message, our `Register` actor will throw a `PaperJamException` in about half of the cases.

What effect does this have on our actor system, or on our whole application? Luckily, Akka is very robust and not affected by exceptions in our code at all. What happens, though, is that the parent of the misbehaving child is notified – remember that parents are watching over their children, and this is the situation where they have to decide what to do.

Supervisor strategies

The whole act of being notified about exceptions in child actors, however, is not handled by the parent actor's `Receive` partial function, as that would confound the parent actor's own behaviour with the logic for dealing with failure in its children. Instead, the two responsibilities are clearly separated.

Each actor defines its own *supervisor strategy*, which tells Akka how to deal with certain types of errors occurring in your children.

There are basically two different types of supervisor strategy, the `OneForOneStrategy` and the `AllForOneStrategy`. Choosing the former means that the way you want to deal with an error in one of your children will only affect the child actor from which the error originated, whereas the latter will affect all of your child actors. Which of those strategies is best depends a lot on your individual application.

Regardless of which type of `SupervisorStrategy` you choose for your actor, you will have to specify a `Decider`, which is a `PartialFunction[Throwable, Directive]` – this allows you to match against certain subtypes of `Throwable` and decide for each of them what's supposed to happen to your problematic child actor (or all your child actors, if you chose the all-for-one strategy).

Directives

Here is a list of the available directives:

```
sealed trait Directive
case object Resume extends Directive
case object Restart extends Directive
case object Stop extends Directive
case object Escalate extends Directive
```

- **Resume**: If you choose to `Resume`, this probably means that you think of your child actor as a little bit of a drama queen. You decide that the exception was not so exceptional after all – the child actor or actors will simply resume processing messages as if nothing extraordinary had happened.
- **Restart**: The `Restart` directive causes Akka to create a new instance of your child actor or actors. The reasoning behind this is that you assume that the internal state of the child/children is corrupted in some way so that it can no longer process any further messages. By restarting the actor, you hope to put it into a clean state again.
- **Stop**: You effectively kill the actor. It will not be restarted.
- **Escalate**: If you choose to `Escalate`, you probably don't know how to deal with the failure at hand. You delegate the decision about what to do to your own parent actor, hoping they are wiser than you. If an actor escalates, they may very well be restarted themselves by their parent, as the parent will only decide about its own child actors.

The default strategy

You don't have to specify your own supervisor strategy in each and every actor. In fact, we haven't done that so far. This means that the default supervisor strategy will take effect. It looks like this:

```
final val defaultStrategy: SupervisorStrategy = {
  def defaultDecider: Decider = {
    case _: ActorInitializationException ⇒ Stop
    case _: ActorKilledException         ⇒ Stop
    case _: Exception                    ⇒ Restart
  }
  OneForOneStrategy()(defaultDecider)
}
```

This means that for exceptions other than `ActorInitializationException` or `ActorKilledException`, the respective child actor in which the exception was thrown will be restarted.

Hence, when a `PaperJamException` occurs in our `Register` actor, the supervisor strategy of the parent actor (the barista) will cause the `Register` to be restarted, because we haven't overridden the default strategy.

If you try this out, you will likely see an exception stacktrace in the log, but nothing about the `Register` actor being restarted.

Let's verify that this is really happening. To do so, however, you will need to learn about the *actor lifecycle*.

The actor lifecycle

To understand what the directives of a supervisor strategy actually do, it's crucial to know a little bit about an actor's lifecycle. Basically, it boils down to this: when created via `actorOf`, an actor is *started*. It can then be *restarted* an arbitrary number of times, in case there is a problem with it. Finally, an actor can be *stopped*, ultimately leading to its death.

There are numerous lifecycle hook methods that an actor implementation can override. It's also important to know their default implementations. Let's go through them briefly:

- **preStart**: Called when an actor is started, allowing you to do some initialization logic. The default implementation is empty.
- **postStop**: Empty by default, allowing you to clean up resources. Called after `stop` has been called for the actor.
- **preRestart**: Called right before a crashed actor is restarted. By default, it stops all children of that actor and then calls `postStop` to allow cleaning up of resources.
- **postRestart**: Called immediately after an actor has been restarted. Simply calls `preStart` by default.

This means that by default, restarting an actor entails a restart of its children. This may be exactly what you want, depending on your specific actor and use case. If it's not what you want, these hook methods allow you to change that behaviour.

Let's see if our `Register` gets indeed restarted upon failure by simply adding some log output to its `postRestart` method. Make the `Register` type extend the `ActorLogging` trait and add the following method to it:

```
override def postRestart(reason: Throwable) {
  super.postRestart(reason)
  log.info(s"Restarted because of ${reason.getMessage}")
}
```

Now, if you send the two `Customer` actors a bunch of `CaffeineWithdrawalWarning` messages, you should see the one or the other of those log outputs, confirming that our `Register` actor has been restarted.

Death of an actor

Often, it doesn't make sense to restart an actor again and again – think of an actor that talks to some other service over the network, and that service has been unreachable for a while. In such cases, it is a very good idea to tell Akka how often to restart an actor within a certain period of time. If that limit is exceeded, the actor is instead stopped and hence dies. Such a limit can be configured in the constructor of the supervisor strategy:

```
import scala.concurrent.duration._
import akka.actor.OneForOneStrategy
import akka.actor.SupervisorStrategy.Restart
OneForOneStrategy(10, 2.minutes) {
  case _ => Restart
}
```

The self-healing system?

So, is our system running smoothly, healing itself whenever this damn paper jam occurs? Let's change our log output:

```
override def postRestart(reason: Throwable) {
  super.postRestart(reason)
  log.info(s"Restarted, and revenue is $revenue cents")
}
```

And while we are at it, let's also add some more logging to our `Receive` partial function, making it look like this:

```
def receive = {
  case Transaction(article) =>
    val price = prices(article)
    sender ! createReceipt(price)
    revenue += price
    log.info(s"Revenue incremented to $revenue cents")
}
```

Ouch! Something is clearly not as it should be. In the log, you will see the revenue increasing, but as soon as there is a paper jam and the `Register` actor restarts, it is reset to `0`. This is because restarting indeed means that the old instance is discarded and a new one created as per the `Props` we initially passed to `actorOf`.

Of course, we could change our supervisor strategy, so that it resumes in case of a `PaperJamException`. We would have to add this to the `Barista` actor:

```
val decider: PartialFunction[Throwable, Directive] = {
  case _: PaperJamException => Resume
}
override def supervisorStrategy: SupervisorStrategy =
  OneForOneStrategy()(
    decider.orElse(SupervisorStrategy.defaultStrategy.decider))
```

Now, the actor is not restarted upon a `PaperJamException`, so its state is not reset.

Error kernel

So we just found a nice solution to preserve the state of our `Register` actor, right?

Well, sometimes, simply resuming might be the best thing to do. But let's assume that we really *have* to restart it, because otherwise the paper jam will not disappear. We can simulate this by maintaining a boolean flag that says if we are in a paper jam situation or not. Let's change our `Register` like so:

```
class Register extends Actor with ActorLogging {
  import Register._
  import Barista._
  var revenue = 0
  val prices = Map[Article, Int](Espresso -> 150, Cappuccino -> 250)
  var paperJam = false
  override def postRestart(reason: Throwable) {
    super.postRestart(reason)
    log.info(s"Restarted, and revenue is $revenue cents")
  }
  def receive = {
    case Transaction(article) =>
      val price = prices(article)
      sender ! createReceipt(price)
      revenue += price
      log.info(s"Revenue incremented to $revenue cents")
  }
  def createReceipt(price: Int): Receipt = {
    import util.Random
    if (Random.nextBoolean()) paperJam = true
    if (paperJam) throw new PaperJamException("OMG, not again!")
    Receipt(price)
  }
}
```

Also remove the supervisor strategy we added to the `Barista` actor.

Now, the paper jam remains forever, until we have restarted the actor. Alas, we cannot do that without also losing important state regarding our revenue.

This is where the *error kernel* pattern comes in. Basically, it is just a simple guideline you should always try to follow, stating that if an actor carries important internal state, then it should delegate dangerous tasks to child actors, so as to prevent the state-carrying actor from crashing. Sometimes, it may make sense to spawn a new child actor for each such task, but that's not a necessity.

The essence of the pattern is to keep important state as far at the top of the actor hierarchy as possible, while pushing error-prone tasks as far to the bottom of the hierarchy as possible.

Let's apply this pattern to our `Register` actor. We will keep the revenue state in the `Register` actor, but move the error-prone behaviour of printing the receipt to a new child actor, which we appropriately enough call `ReceiptPrinter`. Here is the latter:

```
1  object ReceiptPrinter {
2    case class PrintJob(amount: Int)
3    class PaperJamException(msg: String) extends Exception(msg)
4  }
5  class ReceiptPrinter extends Actor with ActorLogging {
6    var paperJam = false
7    override def postRestart(reason: Throwable) {
8      super.postRestart(reason)
9      log.info(s"Restarted, paper jam == $paperJam")
10   }
11   def receive = {
12     case PrintJob(amount) => sender ! createReceipt(amount)
13   }
14   def createReceipt(price: Int): Receipt = {
15     if (Random.nextBoolean()) paperJam = true
16     if (paperJam) throw new PaperJamException("OMG, not again!")
17     Receipt(price)
18   }
19 }
```

Again, we simulate the paper jam with a boolean flag and throw an exception each time someone asks us to print a receipt while in a paper jam. Other than the new message type, `PrintJob`, this is really just extracted from the `Register` type.

This is a good thing, not only because it moves away this dangerous operation from the stateful `Register` actor, but it also makes our code simpler and consequently easier to reason about: The `ReceiptPrinter` actor is responsible for exactly one thing, and the `Register` actor has become simpler, too, now being only responsible for managing the revenue, delegating the remaining functionality to a child actor:

```
class Register extends Actor with ActorLogging {
  import akka.pattern.ask
  import akka.pattern.pipe
  import context.dispatcher
  implicit val timeout = Timeout(4.seconds)
  var revenue = 0
  val prices = Map[Article, Int](Espresso -> 150, Cappuccino -> 250)
  val printer = context.actorOf(Props[ReceiptPrinter], "Printer")
  override def postRestart(reason: Throwable) {
    super.postRestart(reason)
    log.info(s"Restarted, and revenue is $revenue cents")
  }
  def receive = {
    case Transaction(article) =>
      val price = prices(article)
      val requester = sender
      (printer ? PrintJob(price)).map((requester, _)).pipeTo(self)
    case (requester: ActorRef, receipt: Receipt) =>
      revenue += receipt.amount
      log.info(s"revenue is $revenue cents")
      requester ! receipt
  }
}
```

We don't spawn a new `ReceiptPrinter` for each `Transaction` message we get. Instead, we use the default supervisor strategy to have the printer actor restart upon failure.

One part that merits explanation is the weird way we increment our revenue: First we ask the printer for a receipt. We map the future to a tuple containing the answer as well as the requester, which is the sender of the `Transaction` message and pipe this to ourselves. When processing that message, we finally increment the revenue and send the receipt to the requester.

The reason for that indirection is that we want to make sure that we only increment our revenue if the receipt was successfully printed. Since it is vital to never ever modify the internal state of an actor inside of a future, we have to use this level of indirection. It helps us make sure that we only change the revenue within the confines of our actor, and not on some other thread.

Assigning the `sender` to a `val` is necessary for similar reasons: When mapping a future, we are no longer in the context of our actor either – since `sender` is a method, it would now likely return the reference to some other actor that has sent us a message, not the one we intended.

Now, our `Register` actor is safe from constantly being restarted, yay!

Of course, the very idea of having the printing of the receipt and the management of the revenue in one place is questionable. Having them together came in handy for demonstrating the error kernel

pattern. Yet, it would certainly be a lot better to seperate the receipt printing from the revenue management altogether, as these are two concerns that don't really belong together.

Timeouts

Another thing that we may want to improve upon is the handling of timeouts. Currently, when an exception occurs in the `ReceiptPrinter`, this leads to an `AskTimeoutException`, which, since we are using the ask syntax, comes back to the `Barista` actor in an unsuccessfully completed `Future`.

Since the `Barista` actor simply maps over that future (which is success-biased) and then pipes the transformed result to the customer, the customer will also receive a `Failure` containing an `AskTimeoutException`.

The `Customer` didn't ask for anything, though, so it is certainly not expecting such a message, and in fact, it currently doesn't handle these messages. Let's be friendly and send customers a `ComebackLater` message – this is a message they already understand, and it makes them try to get an espresso at a later point. This is clearly better, as the current solution means they will never know that they will not get their espresso.

To achieve this, let's recover from `AskTimeoutException` failures by mapping them to `ComebackLater` messages. The `Receive` partial function of our `Barista` actor thus now looks like this:

```
1  def receive = {
2    case EspressoRequest =>
3      val receipt = register ? Transaction(Espresso)
4      receipt.map((EspressoCup(Filled), _)).recover {
5        case _: AskTimeoutException => ComebackLater
6      } pipeTo(sender)
7    case ClosingTime => context.system.shutdown()
8  }
```

Now, the `Customer` actors know they can try their luck later, and after trying often enough, they should finally get their eagerly anticipated espresso.

Death Watch

Another principle that is important in order to keep your system fault-tolerant is to keep a watch on important dependencies – dependencies as opposed to children.

Sometimes, you have actors that depend on other actors without the latter being their children. This means that they can't be their supervisors. Yet, it is important to keep a watch on their state and be notified if bad things happen.

Think, for instance, of an actor that is responsible for database access. You will want actors that require this actor to be alive and healthy to know when that is no longer the case. Maybe you want

to switch your system to a maintenance mode in such a situation. For other use cases, simply using some kind of backup actor as a replacement for the dead actor may be a viable solution.

In any case, you will need to place a watch on an actor you depend on in order to get the sad news of its passing away. This is done by calling the `watch` method defined on `ActorContext`. To illustrate, let's have our `Customer` actors watch the `Barista` – they are highly addicted to caffeine, so it's fair to say they depend on the barista:

```
class Customer(coffeeSource: ActorRef) extends Actor with ActorLogging {
  import context.dispatcher

  context.watch(coffeeSource)

  def receive = {
    case CaffeineWithdrawalWarning => coffeeSource ! EspressoRequest
    case (EspressoCup(Filled), Receipt(amount)) =>
      log.info(s"yay, caffeine for ${self}!")
    case ComebackLater =>
      log.info("grumble, grumble")
      context.system.scheduler.scheduleOnce(300.millis) {
        coffeeSource ! EspressoRequest
      }
    case Terminated(barista) =>
      log.info("Oh well, let's find another coffeehouse...")
  }
}
```

We start watching our `coffeeSource` in our constructor, and we added a new case for messages of type `Terminated` – this is the kind of message we will receive from Akka if an actor we watch dies.

Now, if we send a `ClosingTime` to the message and the `Barista` tells its context to stop itself, the `Customer` actors will be notified. Give it a try, and you should see their output in the log.

Instead of simply logging that we are not amused, this could just as well initiate some failover logic, for instance.

Summary

In this chapter, you got to know some of the important components of an actor system, all while learning how to put the tools provided by Akka and the ideas behind it to use in order to make your system more fault-tolerant.

While there is still a lot more to learn about the actor model and Akka, we shall leave it at that for now, as this would go beyond the scope of this book. In the next and final chapter, I will point you

to a bunch of Scala resources you may want to peruse to continue your journey through Scala land, and if actors and Akka got you excited, there will be something in there for you, too.

Where to go from here

Over the course of the last fifteen chapters, we have delved into numerous language and library features of Scala, hopefully deepening your understanding of those features and their underlying ideas and concepts.

As such, I hope this guide has served you well as a supplement to whatever introductory resources you have been using to learn Scala, whether you attended the Scala course at Coursera[16] or learned from a book. I have tried to cover all the quirks I stumbled over back when I learned the language – things that were only mentioned briefly or not covered at all in the books and tutorials available to me – and I hope that especially those explanations were of value to you.

With every chapter, we ventured into more advanced territory, covering ideas like type classes and path-dependent types. While I could have gone on writing about more and more arcane features, I felt like this would go against the idea of this guide, which is clearly targeted at aspiring neophytes.

Hence, I will conclude this book with some suggestions of where to go from here if you want more. Rest assured that I will continue writing about Scala at my blog[17].

How you want to continue your journey with Scala, of course, depends a lot on your individual preferences: Maybe you are now at a point where you would like to **teach Scala to others**, or maybe you are excited about Scala's type system and would like to explore some of the language's more arcane features by delving into **type-level programming**.

More often than not, a good way to really get comfortable with a new language and its whole ecosystem of libraries is to use it for **creating something useful**, i.e. a real application that is more than just toying around. Personally, I have also gained a lot from **contributing** to open source projects early on.

In the following, I will elaborate those four paths, which, of course, are not mutually exclusive, and provide you with numerous links to highly recommendable additional resources.

Teaching Scala

Having studies this book and worked through the examples, you should be familiar enough with Scala to be able to teach the basics. Maybe you are in a Java or Ruby shop and want to get your coworkers excited about Scala and functional programming.

Great, then why not organize a workshop? A nice way of introducing people to a new language is to not do a talk with lots of slides, but to teach by example, introducing a language in small steps by solving tiny problems together. Active participation is key!

[16] https://www.coursera.org/course/progfun

[17] http://danielwestheide.com

If that's something you'd like to do, the Scala community has you covered. Have a look at Scala Koans[18], a collection of small lessons, each of which provides a problem to be solved by fixing an initially failing test. The project is inspired by the Ruby Koans[19] project and is a really good resource for teaching the language to others in small collaborative coding sessions.

Another amazing project ideally suited for workshops or other events is Scalatron[20], a game in which bots fight against each other in a virtual arena. Why not teach the language by developing such a bot together in a workshop, that will then fight against the computer? Once the participants are familiar enough with the language, organize a tournament, where each participant will develop their own bot.

Mastering arcane powers

We have only seen a tiny bit of what the Scala type system allows you to do. If this small hint at the high wizardry that's possible got you excited and you want to master the arcane powers of type-level programming, a good starting resource is the blog series Type-Level Programming in Scala[21] by Mark Harrah.

After that, I recommend to have a look at Shapeless[22], a library in which Miles Sabin explores the limits of the Scala language in terms of generic and polytypic programming.

Creating something useful

Reading books, doing tutorials and toying around with a new language is all fine to get a certain understanding of it, but in order to become really comfortable with Scala and its paradigm and learn how to think the Scala way, I highly recommend that you start creating something useful with it – something that is clearly more than a toy application (this is true for learning any language, in my opinion).

By tackling a real-world problem and trying to create a useful and usable application, you will also get a good overview of the ecosystem of libraries and get a feeling for which of those can be of service to you in specific situations.

In order to find relevant libraries or get updates of ones you are interested in, you should subscribe to implicit.ly[23] and regularly take a look at Scala projects on GitHub[24].

[18] http://www.scalakoans.org/
[19] http://rubykoans.com/
[20] http://scalatron.github.com/
[21] http://apocalisp.wordpress.com/2010/06/08/type-level-programming-in-scala/
[22] https://github.com/milessabin/shapeless
[23] http://notes.implicit.ly/
[24] https://github.com/languages/Scala

It's all about the web

These days, most applications written in Scala will be some kind of server applications, often with a RESTful interface exposed via HTTP and a web frontend.

If the actor model for concurrency is a good fit for your use case and you hence choose to use the Akka[25] toolkit, an excellent choice for exposing a REST API via HTTP is Spray Routing[26]. This is a great tool if you don't need a web frontend, or if you want to develop a single-page web application that will talk to your backend by means of a REST API.

If you need something less minimalistic, of course, Play[27], which is part of the Typesafe stack, is a good choice, especially if you seek something that is widely adopted and hence well supported.

Living in a concurrent world

If after the two chapters on actors and Akka, you think that Akka is a good fit for your application, you will likely want to learn a lot more about it before getting serious with it.

While the Akka documentation[28] is pretty exhaustive and thus serves well as a reference, I think that the best choice for actually *learning* Akka is Derek Wyatt's book Akka Concurrency[29].

Once you have got serious with Akka, you should definitely subscribe to Let It Crash[30], which provides you with news and advanced tips and tricks and regarding all things Akka.

If actors are not your thing and you prefer a concurrency model allowing you to leverage the composability of *Futures*, your library of choice is probably Twitter's Finagle[31]. It allows you to modularize your application as a bunch of small remote services, with support for numerous popular protocols out of the box.

Contributing

Another really great way to learn a lot about Scala quickly is to start contributing to one or more open source projects – preferably to libraries you have been using while working on your own application.

Of course, this is nothing that is specific to Scala, but still I think it deserves to be mentioned. If you have only just learned Scala and are not using it at your day job already, it's nearly the only choice you have to learn from other, more experienced Scala developers.

[25] http://akka.io
[26] http://spray.io/documentation/spray-routing/
[27] http://playframework.org
[28] http://akka.io/docs/
[29] http://www.artima.com/shop/akka_concurrency
[30] http://letitcrash.com/
[31] http://twitter.github.com/finagle

It forces you to read a lot of Scala code from other people, discovering how to do things differently, possibly more idiomatically, and you can have those experienced developers review your code in pull requests.

I have found the Scala community at large to be very friendly and helpful, so don't shy away from contributing, even if you think you're too much of a rookie when it comes to Scala.

While some projects might have their own way of doing things, it's certainly a good idea to study the Scala Style Guide[32] to get familiar with common coding conventions.

Connecting

By contributing to open source projects, you have already started connecting with the Scala community. However, you may not have the time to do that, or you may prefer other ways of connecting to like-minded people.

Try finding a local Scala user group or meetup. Scala Tribes[33] provides an overview of Scala communities across the globe, and the Scala topic at Lanyrd[34] keeps you up-to-date on any kind of Scala-related event, from conferences to meetups.

If you don't like connecting in meatspace, the scala-user mailing list/Google group[35] and the Scala IRC channel on Freenode[36] may be good alternatives.

Other resources

Regardless of which of the paths outlined above you follow, there are a few resources I would like to recommend:

- Functional Programming in Scala[37] by Paul Chiusano and RÃºnar Bjarnason, which is currently available as an Early Access Edition and teaches you a lot more about functional programming and thinking about problems with a functional mindset.
- The Scala Documentation Site[38], which, for some reason, is not linked very prominently from the main Scala website, especially the available guides[39] and tutorials[40].
- Resources for Getting Started With Functional Programming and Scala[41] by Kelsey Innis contains a lot more helpful links about some of the topics we covered in this book.

[32] http://docs.scala-lang.org/style/
[33] http://www.scala-tribes.org
[34] http://lanyrd.com/topics/scala/
[35] https://groups.google.com/forum/#!forum/scala-user
[36] irc://irc.freenode.net/scala
[37] http://www.manning.com/bjarnason/
[38] http://docs.scala-lang.org/
[39] http://docs.scala-lang.org/overviews/
[40] http://docs.scala-lang.org/tutorials/
[41] http://nerd.kelseyinnis.com/blog/2013/01/07/resources-for-getting-started-with-functional-programming-and-scala/

Conclusion

I hope you have enjoyed this book and that I could get you excited about Scala. While this book is coming to an end, I seriously hope that it's just the beginning of your journey through Scala land. Let me know how your journey went so far and where you think it will go from here.