Daniel Westheide

The Neophyte's Guide to Scala

The Neophyte's Guide to Scala

Daniel Westheide

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

This version was published on 2019-12-15

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:

#scala

Contents

Extractors

In the Coursera course, you came across one very powerful language feature of Scala: Pattern matching[1]. 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[2].

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:

[1] http://en.wikipedia.org/wiki/Pattern_matching
[2] 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 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)[3] that contain pure data and no behaviour whatsoever.

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.

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

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.