## Chapter 26

# Extractors

By now you have probably grown accustomed to the concise way data can be decomposed and analyzed using pattern matching. This chapter shows you how to generalize this concept further. Until now, constructor patterns were linked to case classes. For instance, Some(x) is a valid pattern because Some is a case class. Sometimes you might wish that you could write patterns like this without creating an associated case class. In fact, you might wish to be able to create your own kinds of patterns. Extractors give you a way to do so. This chapter explains what extractors are and how you can use them to define patterns that are decoupled from an object's representation.

### 26.1 AN EXAMPLE: EXTRACTING EMAIL ADDRESSES

To illustrate the problem extractors solve, imagine that you need to analyze strings that represent email addresses. Given a string, you want to decide whether it is an email address or not, and, if it is, you want to access the user and domain parts of the address. The traditional way to do this uses three helper functions:

```
def isEMail(s: String): Boolean
def domain(s: String): String
def user(s: String): String
```

With these functions, you could parse a given string s as follows:

```
if (isEMail(s)) println(user(s) + " AT " + domain(s))
else println("not an email address")
```

This works, but is kind of clumsy. What's more, things would become more complicated if you combined several such tests. For instance you might want to find two successive strings in a list that are both email addresses with the same user. You can try this yourself with the access functions defined previously to see what would be involved.

You saw already in Chapter 15 that pattern matching is ideal for attacking problems like this. Let's assume for the moment that you could match a string with a pattern:

```
EMail(user, domain)
```

The pattern would match if the string contained an embedded at sign (@). In that case it would bind variable user to the part of the string before the @ and variable domain to the part after it. Postulating a pattern like this, the previous expression could be written more clearly like this:

```
s match {
  case EMail(user, domain) => println(user + " AT " + domain)
  case _ => println("not an email address")
}
```

The more complicated problem of finding two successive email addresses with the same user part would translate to the following pattern:

```
ss match {
  case EMail(u1, d1) :: EMail(u2, d2) :: _ if (u1 == u2) => ...
  ...
}
```

This is much more legible than anything that could be written with access functions. However, the problem is that strings are not case classes; they do not have a representation that conforms to EMail(user, domain). This is where Scala's extractors come in: they let you define new patterns for pre-existing types, where the pattern need not follow the internal representation of the type.

## 26.2 EXTRACTORS

An extractor in Scala is an object that has a method called unapply as one of its members. The purpose of that unapply method is to match a value and take it apart. Often, the extractor object also defines a dual method apply for building values, but this is not required. As an example, Listing 26.1 shows an extractor object for email addresses:

```
object EMail {

  // The injection method (optional)
  def apply(user: String, domain: String) = user + "@" + domain

  // The extraction method (mandatory)
  def unapply(str: String): Option[(String, String)] = {
    val parts = str split "@"
    if (parts.length == 2) Some(parts(0), parts(1)) else None
  }
}
```

**Listing 26.1 - The EMail string extractor object.**

This object defines both apply and unapply methods. The apply method has the same meaning as always: it turns EMail into an object that can be applied to arguments in parentheses in the same way a method is applied. So you can write EMail("John", "epfl.ch") to construct the string"John@epfl.ch". To make this more explicit, you could also let EMail inherit from Scala's function type, like this:

```
object EMail extends ((String, String) => String) { ... }
```

**Note**

The "(String, String) => String" portion of the previous object declaration means the same as Function2[String, String, String], which declares an abstract apply method that EMail implements. As a result of this declaration, you could, for example, passEMail to a method expecting a Function2[String, String, String].

The unapply method is what turns EMail into an extractor. In a sense, it reverses the construction process of apply. Where apply takes two strings and forms an email address string out of

them, unapply takes an email address and returns potentially two strings: the user and the domain of the address. But unapply must also handle the case where the given string is not an email address. That's why unapply returns an Option-type over pairs of strings. Its result is either Some(user, domain) if the string str is an email address with the given user and domainparts,[1] or None, if str is not an email address. Here are some examples:

```
unapply("John@epfl.ch")  equals  Some("John", "epfl.ch")
unapply("John Doe")  equals  None
```

Now, whenever pattern matching encounters a pattern referring to an extractor object, it invokes the extractor's unapply method on the selector expression. For instance, executing the code:

```
selectorString match { case EMail(user, domain) => ... }
```

would lead to the call:

```
EMail.unapply(selectorString)
```

As you saw previously, this call to EMail.unapply will return either None or Some(u, d), for some values u for the user part of the address and d for the domain part. In the None case, the pattern does not match, and the system tries another pattern or fails with a MatchError exception. In the Some(u, d) case, the pattern matches and its variables are bound to the elements of the returned value. In the previous match, user would be bound to u and domain would be bound tod.

In the EMail pattern matching example, the type String of the selector expression,selectorString, conformed to unapply's argument type (which in the example was also String). This is quite common, but not necessary. It would also be possible to use the EMail extractor to match selector expressions for more general types. For instance, to find out whether an arbitrary value x was an email address string, you could write:

```
val x: Any = ...
x match { case EMail(user, domain) => ... }
```

Given this code, the pattern matcher will first check whether the given value x conforms toString, the parameter type of EMail's unapply method. If it does conform, the value is cast toString and pattern matching proceeds as before. If it does not conform, the pattern fails immediately.

In object EMail, the apply method is called an injection, because it takes some arguments and yields an element of a given set (in our case: the set of strings that are email addresses). Theunapply method is called an extraction, because it takes an element of the same set and extracts some of its parts (in our case: the user and domain substrings). Injections and extractions are often grouped together in one object, because then you can use the object's name for both a constructor and a pattern, which simulates the convention for pattern matching with case classes. However, it is also possible to define an extraction in an object without a corresponding injection. The object itself is called an extractor, regardless of whether or not it has an apply method.

If an injection method is included, it should be the dual to the extraction method. For instance, a call of:

```
EMail.unapply(EMail.apply(user, domain))
```

should return:

```
Some(user, domain)
```

*i.e.*, the same sequence of arguments wrapped in a Some. Going in the other direction means running first the unapply and then the apply, as shown in the following code:

```
EMail.unapply(obj) match {
  case Some(u, d) => EMail.apply(u, d)
}
```

In that code, if the match on obj succeeds, you'd expect to get back that same object from theapply. These two conditions for the duality of apply and unapply are good design principles. They are not enforced by Scala, but it's recommended to keep to them when designing your extractors.

## 26.3 PATTERNS WITH ZERO OR ONE VARIABLES

The unapply method of the previous example returned a pair of element values in the success case. This is easily generalized to patterns of more than two variables. To bind N variables, anunapply would return an N-element tuple, wrapped in a Some.

The case where a pattern binds just one variable is treated differently, however. There is no one-tuple in Scala. To return just one pattern element, the unapply method simply wraps the element itself in a Some. For example, the extractor object shown in Listing 26.2 defines applyand unapply for strings that consist of the same substring appearing twice in a row:

```
object Twice {
  def apply(s: String): String = s + s
  def unapply(s: String): Option[String] = {
    val length = s.length / 2
    val half = s.substring(0, length)
    if (half == s.substring(length)) Some(half) else None
  }
}
```

**Listing 26.2 - The Twice string extractor object.**

It's also possible that an extractor pattern does not bind any variables. In that case the corresponding unapply method returns a boolean—true for success and false for failure. For instance, the extractor object shown in Listing 26.3 characterizes strings consisting of all uppercase characters:

```
object UpperCase {
  def unapply(s: String): Boolean = s.toUpperCase == s
}
```

**Listing 26.3 - The UpperCase string extractor object.**

This time, the extractor only defines an unapply, but not an apply. It would make no sense to define an apply, as there's nothing to construct.

The following userTwiceUpper function applies all previously defined extractors together in its pattern matching code:

```
def userTwiceUpper(s: String) = s match {
  case EMail(Twice(x @ UpperCase()), domain) =>
    "match: " + x + " in domain " + domain
  case _ =>
    "no match"
}
```

The first pattern of this function matches strings that are email addresses whose user part consists of two occurrences of the same string in uppercase letters. For instance:

```
scala> userTwiceUpper("DIDI@hotmail.com")
res0: String = match: DI in domain hotmail.com

scala> userTwiceUpper("DIDO@hotmail.com")
res1: String = no match

scala> userTwiceUpper("didi@hotmail.com")
res2: String = no match
```

Note that UpperCase in function userTwiceUpper takes an empty parameter list. This cannot be omitted as otherwise the match would test for equality with the object UpperCase! Note also that, even though UpperCase() itself does not bind any variables, it is still possible to associate a variable with the whole pattern matched by it. To do this, you use the standard scheme of variable binding explained in Section 15.2: the form x @ UpperCase() associates the variable xwith the pattern matched by UpperCase(). For instance, in the first userTwiceUpper invocation above, x was bound to "DI", because that was the value against which the UpperCase() pattern was matched.

## 26.4 VARIABLE ARGUMENT EXTRACTORS

The previous extraction methods for email addresses all returned a fixed number of element values. Sometimes, this is not flexible enough. For example, you might want to match on a string representing a domain name, so that every part of the domain is kept in a different sub-pattern. This would let you express patterns such as the following:

```
dom match {
  case Domain("org", "acm") => println("acm.org")
  case Domain("com", "sun", "java") => println("java.sun.com")
  case Domain("net", _*) => println("a .net domain")
}
```

In this example things were arranged so that domains are expanded in reverse order—from the top-level domain down to the sub-domains. This was done so that you could better profit from sequence patterns. You saw in Section 15.2 that a sequence wildcard pattern, _*, at the end of an argument list matches any remaining elements in a sequence. This feature is more useful if the top-level domain comes first, because then you can use sequence wildcards to match sub-domains of arbitrary depth.

The question remains how an extractor can support vararg matching as shown in the previous example, where patterns can have a varying number of sub-patterns. The unapplymethods encountered so far are not sufficient, because they each return a fixed number of sub-elements in the success case. To handle this case, Scala lets you define a different extraction method specifically for vararg matching. This method is called unapplySeq. To see how it is written, have a look at the Domain extractor, shown in Listing 26.4:

```
object Domain {

  // The injection method (optional)
  def apply(parts: String*): String =
    parts.reverse.mkString(".")

  // The extraction method (mandatory)
  def unapplySeq(whole: String): Option[Seq[String]] =
    Some(whole.split("\\.").reverse)
}
```

**Listing 26.4 - The Domain string extractor object.**

The Domain object defines an unapplySeq method that first splits the string into parts separated by periods. This is done using Java's split method on strings, which takes a regular expression as its argument. The result of split is an array of substrings. The result ofunapplySeq is then that array with all elements reversed and wrapped in a Some.

The result type of an unapplySeq must conform to Option[Seq[T]], where the element type T is arbitrary. As you saw in Section 17.1, Seq is an important class in Scala's collection hierarchy. It's a common superclass of several classes describing different kinds of sequences: Lists,Arrays, WrappedString, and several others.

For symmetry, Domain also has an apply method that builds a domain string from a variable argument parameter of domain parts starting with the top-level domain. As always, the applymethod is optional.

You can use the Domain extractor to get more detailed information out of email strings. For instance, to search for an email address named "tom" in some ".com" domain, you could write the following function:

```
def isTomInDotCom(s: String): Boolean = s match {
  case EMail("tom", Domain("com", _*)) => true
  case _ => false
}
```

This gives the expected results:

```
scala> isTomInDotCom("tom@sun.com")
res3: Boolean = true

scala> isTomInDotCom("peter@sun.com")
res4: Boolean = false

scala> isTomInDotCom("tom@acm.org")
res5: Boolean = false
```

It's also possible to return some fixed elements from an unapplySeq together with the variable part. This is expressed by returning all elements in a tuple, where the variable part comes last, as usual. As an example, Listing 26.5 shows a new extractor for emails where the domain part is already expanded into a sequence:

```
object ExpandedEMail {
  def unapplySeq(email: String)
      : Option[(String, Seq[String])] = {
    val parts = email split "@"
    if (parts.length == 2)
      Some(parts(0), parts(1).split("\\.").reverse)
    else
      None
  }
}
```

**Listing 26.5 - The ExpandedEMail extractor object.**

The unapplySeq method in ExpandedEMail returns an optional value of a pair (a Tuple2). The first element of the pair is the user part. The second element is a sequence of names representing the domain. You can match on this as usual:

```
scala> val s = "tom@support.epfl.ch"
s: String = tom@support.epfl.ch

scala> val ExpandedEMail(name, topdom, subdoms @ _*) = s
name: String = tom
topdom: String = ch
subdoms: Seq[String] = WrappedArray(epfl, support)
```

## 26.5 EXTRACTORS AND SEQUENCE PATTERNS

You saw in Section 15.2 that you can access the elements of a list or an array using sequence patterns such as:

```
List()
List(x, y, _*)
Array(x, 0, 0, _)
```

In fact, these sequence patterns are all implemented using extractors in the standard Scala library. For instance, patterns of the form List(...) are possible because the scala.Listcompanion object is an extractor that defines an unapplySeq method. Listing 26.6 shows the relevant definitions:

```
package scala
object List {
  def apply[T](elems: T*) = elems.toList
  def unapplySeq[T](x: List[T]): Option[Seq[T]] = Some(x)
  ...
}
```

**Listing 26.6 - An extractor that defines an unapplySeq method.**

The List object contains an apply method that takes a variable number of arguments. That's what lets you write expressions such as:

```
List()
List(1, 2, 3)
```

It also contains an unapplySeq method that returns all elements of the list as a sequence. That's what supports List(...) patterns. Very similar definitions exist in the object scala.Array. These support analogous injections and extractions for arrays.

## 26.6 EXTRACTORS VERSUS CASE CLASSES

Even though they are very useful, case classes have one shortcoming: they expose the concrete representation of data. This means that the name of the class in a constructor pattern corresponds to the concrete representation type of the selector object. If a match against:

```
case C(...)
```

succeeds, you know that the selector expression is an instance of class C.

Extractors break this link between data representations and patterns. You have seen in the examples in this section that they enable patterns that have nothing to do with the data type of the object that's selected on. This property is called representation independence. In open systems of large size, representation independence is very important because it allows you to change an implementation type used in a set of components without affecting clients of these components.

If your component had defined and exported a set of case classes, you'd be stuck with them because client code could already contain pattern matches against these case classes. Renaming some case classes or changing the class hierarchy would affect client code. Extractors do not share this problem, because they represent a layer of indirection between a data representation and the way it is viewed by clients. You could still change a concrete representation of a type, as long as you update all your extractors with it.

Representation independence is an important advantage of extractors over case classes. On the other hand, case classes also have some advantages of their own over extractors. First, they are much easier to set up and to define, and they require less code. Second, they usually lead to more efficient pattern matches than extractors, because the Scala compiler can optimize patterns over case classes much better than patterns over extractors. This is because the mechanisms of case classes are fixed, whereas an unapply or unapplySeq method in an extractor could do almost anything. Third, if your case classes inherit from a sealed base class, the Scala compiler will check your pattern matches for exhaustiveness and will complain if some combination of possible values is not covered by a pattern. No such exhaustiveness checks are available for extractors.

So which of the two methods should you prefer for your pattern matches? It depends. If you write code for a closed application, case classes are usually preferable because of their advantages in conciseness,

speed and static checking. If you decide to change your class hierarchy later, the application needs to be refactored, but this is usually not a problem. On the other hand, if you need to expose a type to unknown clients, extractors might be preferable because they maintain representation independence.

Fortunately, you need not decide right away. You could always start with case classes and then, if the need arises, change to extractors. Because patterns over extractors and patterns over case classes look exactly the same in Scala, pattern matches in your clients will continue to work.

Of course, there are also situations where it's clear from the start that the structure of your patterns does not match the representation type of your data. The email addresses discussed in this chapter were one such example. In that case, extractors are the only possible choice.

## 26.7 REGULAR EXPRESSIONS

One particularly useful application area of extractors are regular expressions. Like Java, Scala provides regular expressions through a library, but extractors make it much nicer to interact with them.

### Forming regular expressions

Scala inherits its regular expression syntax from Java, which in turn inherits most of the features of Perl. We assume you know that syntax already; if not, there are many accessible tutorials, starting with the Javadoc documentation of class java.util.regex.Pattern. Here are just some examples that should be enough as refreshers:

| | |
|---|---|
| ab? | An `a', possibly followed by a `b'. |
| [1em] \d+ | A number consisting of one or more digits represented by \d. |
| [a-dA-D]\w* | A word starting with a letter between a and d in lower or upper case, followed by a sequence of zero or more "word characters" denoted by \w. (A word character is a letter, digit, or underscore.) |
| [1em](-)? (\d+)(\.\d*)? | A number consisting of an optional minus sign, followed by one or more digits, optionally followed by a period and zero or more digits. The number contains three groups, *i.e.*, the minus sign, the part before the decimal point, and the fractional part including the decimal point. Groups are enclosed in parentheses. |

Scala's regular expression class resides in package scala.util.matching.

```
scala> import scala.util.matching.Regex
```

A new regular expression value is created by passing a string to the Regex constructor. For instance:

```
scala> val Decimal = new Regex("(-)?(\\d+)(\\.\\d*)?")
Decimal: scala.util.matching.Regex = (-)?(\d+)(\.\d*)?
```

Note that, compared to the regular expression for decimal numbers given previously, every backslash appears twice in the string above. This is because in Java and Scala a single backslash is an escape character in a string literal, not a regular character that shows up in the string. So instead of `\' you need to write `\\' to get a single backslash in the string.

If a regular expression contains many backslashes this might be a bit painful to write and to read. Scala's raw strings provide an alternative. As you saw in Section 5.2, a raw string is a sequence of characters between triple quotes. The difference between a raw and a normal string is that all characters in a raw string appear exactly as they are typed. This includes backslashes, which are not treated as escape characters. So you could write equivalently and somewhat more legibly:

```
scala> val Decimal = new Regex("""(-)?(\d+)(\.\d*)?""")
Decimal: scala.util.matching.Regex = (-)?(\d+)(\.\d*)?
```

As you can see from the interpreter's output, the generated result value for Decimal is exactly the same as before.

Another, even shorter way to write a regular expression in Scala is this:

```
scala> val Decimal = """(-)?(\d+)(\.\d*)?""".r
Decimal: scala.util.matching.Regex = (-)?(\d+)(\.\d*)?
```

In other words, simply append a .r to a string to obtain a regular expression. This is possible because there is a method named r in class StringOps, which converts a string to a regular expression. The method is defined as shown in Listing 26.7:

```
package scala.runtime
import scala.util.matching.Regex

class StringOps(self: String) ... {
  ...
  def r = new Regex(self)
}
```

**Listing 26.7 - How the r method is defined in StringOps.**

### Searching for regular expressions

You can search for occurrences of a regular expression in a string using several different operators:

regex findFirstIn str

Finds first occurrence of regular expression regex in string str, returning the result in an Optiontype.

regex findAllIn str

Finds all occurrences of regular expression regex in string str, returning the results in anIterator.

regex findPrefixOf str

Finds an occurrence of regular expression regex at the start of string str, returning the result in an Option type.

For instance, you could define the input sequence below and then search decimal numbers in it:

```
scala> val Decimal = """(-)?(\d+)(\.\d*)?""".r
Decimal: scala.util.matching.Regex = (-)?(\d+)(\.\d*)?

scala> val input = "for -1.0 to 99 by 3"
input: String = for -1.0 to 99 by 3

scala> for (s <- Decimal findAllIn input)
       println(s)
-1.0
99
3

scala> Decimal findFirstIn input
res7: Option[String] = Some(-1.0)

scala> Decimal findPrefixOf input
res8: Option[String] = None
```

**Extracting with regular expressions**

What's more, every regular expression in Scala defines an extractor. The extractor is used to identify substrings that are matched by the groups of the regular expression. For instance, you could decompose a decimal number string as follows:

```
scala> val Decimal(sign, integerpart, decimalpart) = "-1.23"
sign: String = -
integerpart: String = 1
decimalpart: String = .23
```

In this example, the pattern, Decimal(...), is used in a val definition, as described in Section 15.7. What happens here is that the Decimal regular expression value defines an unapplySeqmethod. That method matches every string that corresponds to the regular expression syntax for decimal numbers. If the string matches, the parts that correspond to the three groups in the regular expression (-)?(\d+)(\.\d*)? are returned as elements of the pattern and are then matched by the three pattern variables sign, integerpart, and decimalpart. If a group is missing, the element value is set to null, as can be seen in the following example:

```
scala> val Decimal(sign, integerpart, decimalpart) = "1.0"
sign: String = null
integerpart: String = 1
decimalpart: String = .0
```

It's also possible to mix extractors with regular expression searches in a for expression. For instance, the following expression decomposes all decimal numbers it finds in the inputstring:

```
scala> for (Decimal(s, i, d) <- Decimal findAllIn input)
         println("sign: " + s + ", integer: " +
             i + ", decimal: " + d)
sign: -, integer: 1, decimal: .0
sign: null, integer: 99, decimal: null
```

```
sign: null, integer: 3, decimal: null
```

## 26.8 CONCLUSION

In this chapter you saw how to generalize pattern matching with extractors. Extractors let you define your own kinds of patterns, which need not correspond to the type of the expressions you select on. This gives you more flexibility in the kinds of patterns you can use for matching. In effect it's like having different possible views on the same data. It also gives you a layer between a type's representation and the way clients view it. This lets you do pattern matching while maintaining representation independence, a property which is very useful in large software systems.

Extractors are one more element in your tool box that let you define flexible library abstractions. They are used heavily in Scala's libraries, for instance, to enable convenient regular expression matching.

**Footnotes for Chapter 26:**

[1] As demonstrated here, where Some is applied to the tuple, (user, domain), you can leave off one pair of parentheses when passing a tuple to a function that takes a single argument. Thus,Some(user, domain) means the same as Some((user, domain)).