Table of Contents for Programming Scala, 2nd Edition

Chapter 4. Pattern Matching

At first glance, *pattern-matching* expressions look like the familiar `case` statements from your favorite C-like language. In the typical C-like `case` statement you're limited to matching against values of ordinal types and triggering trivial expressions for matches. For example, "In the case that `i` is 5, print a message; in the case that `i` is 6, exit the program."

With Scala's pattern matching, your cases can include types, wildcards, sequences, regular expressions, and even deep inspections of an object's state. This deep inspection follows a protocol that allows the type implementer to control the visibility of internal state. Finally, the exposed state is easy to capture to variables for use. Hence, the terms "extraction" or "destructuring" are sometimes used for this capability.

Pattern matching can be used in several code contexts. We'll start with the most common usage, within `match` clauses. Afterwards, we'll show other uses. We saw two relatively simple examples of `match` clauses in our actor example in A Taste of Concurrency. We also discussed additional uses in Partial Functions.

A Simple Match

To begin with, let's simulate flipping a coin by matching the value of a Boolean:

```
// src/main/scala/progscala2/patternmatching/match-boolean.sc

val bools = Seq(true, false)

for (bool <- bools) {
  bool match {
    case true => println("Got heads")
    case false => println("Got tails")
  }
}
```

It looks just like a C-style `case` statement. As an experiment, try commenting out the second `case false` clause and run the script again. You get a warning and then an error:

```
<console>:12: warning: match may not be exhaustive.
It would fail on the following input: false
               bool match {
               ^
Got heads
scala.MatchError: false (of class
java.lang.Boolean)
  at .<init>(<console>:11)
  at .<clinit>(<console>)
  ...
```

From the type of the list, the compiler knows that there are two possible cases, `true` and `false`. So, it warns that the match isn't exhaustive. Then we see what happens when we try to match on a value for which there is no matching `case` clause. A `MatchError` is thrown.

I should mention that a simpler alternative is an old-fashioned `if` expression in this case:

```
for (bool <- bools) {
  val which = if (bool) "head" else "tails"
          "Got
  println("      + which)
}
```

Values, Variables, and Types in Matches

Let's cover several kinds of matches. The following example matches on specific values, all values of specific types, and it shows one way of writing a "default" clause that matches anything:

```
// src/main/scala/progscala2/patternmatching/match-variable.sc

for {
  x <- Seq(1, 2, 2.7, "one", "two", 'four)                              //
❶
} {
  val str = x match {
//  ❷
    case 1          => "int 1"
//  ❸
                       "other int:
    case i: Int     => "            +i
//  ❹
                       "a double:
    case d: Double  => "          +x
//  ❺
    case "one"      => "string one"
//  ❻
                       "other string:
    case s: String  => "              +s
//  ❼
                       "unexpected value:
    case unexpected => "                   + unexpected             //
❽
  }
  println(str)
//  ❾
}
```

❶

Because of the mix of values, the list is of type `Seq[Any]`.

❷

The `x` is of type `Any`.

❸

Match if `x` equals `1`.

❹

Match any *other* `Int` value. Safely cast the value of `x` to an `Int` and assign to `i`.

❺

Match any `Double`, where the value of `x` is assigned to the `Double` variable `d`.

❻

Match the `String` "one".

❼

Match any other `String`, where the value of `x` is assigned to the `String` variable `s`.

❽

Match all other inputs, where `unexpected` is the variable to which the value of `x` is assigned. Because no type annotation is given, `Any` is inferred. This functions as the "default" clause.

❾

Print the returned `String`.

I lined up the `=>` ("arrows") to make things a bit clearer. Here is the output:

```
int 1
other int: 2
a double 2.7
string one
other string: two
unexpected value: 'four
```

Matches, like all expressions, return a value. In this case, all clauses return strings, so the return type of the whole clause is `String`. The compiler infers the closest supertype (also called the *least upper bound*) for types of values returned by all the `case` clauses.

Because `x` is of type `Any`, we need enough clauses to cover all possible values. (Compare with our first example that matched on Boolean values.). That's why the "default" clause (with `unexpected`) is needed. However, when writing `PartialFunctions`, we *aren't* required to match all possible values, because they are intentionally *partial*.

Matches are eager, so more specific clauses must appear before less specific clauses. Otherwise, the more specific clauses will never get the chance to match. So, the default clause shown must be the last one. The compiler will catch the error, fortunately.

I didn't include a clause with a floating-point literal, because matching on floating-point literals is a bad idea, as rounding errors mean two values that appear to be the same often differ in the least significant digits.

Here is a slight variation of the previous example:

```
// src/main/scala/progscala2/patternmatching/match-variable2.sc

for {
  x <- Seq(1, 2, 2.7, "one", "two", 'four)
} {
  val str = x match {
    case 1          => "int 1"
                       "other int:
    case _: Int     => "            +x
                       "a double:
    case _: Double  => "          +x
    case "one"      => "string one"
                       "other string:
    case _: String  => "               +x
                       "unexpected value:
    case _          => "                   + x
  }
  println(str)
}
```

We replaced the variables `i`, `d`, `s`, and `unexpected` with the placeholder `_`. We don't actually need variables of the specific types, because we're just going to generate strings. So, we can use `x` for all cases.

Tip

Except for `PartialFunctions`, matches must be exhaustive. When the input is of type `Any`, end with a default `case _` or `case some_name` match clause.

There are a few rules and gotchas to keep in mind when writing `case` clauses. The compiler assumes that a term that starts with a capital letter is a type name, while a term that begins with a lowercase letter is assumed to be the name of a variable that will hold an extracted or matched value.

This rule can cause surprises, as shown in the following example:

```
// src/main/scala/progscala2/patternmatching/match-surprise.sc

def checkY(y: Int) = {
  for {
    x <- Seq(99, 100, 101)
  } {
    val str = x match {
                "found
      case y => y!"
                     "int:
      case i: Int => "      +i
    }
    println(str)
  }
}

checkY(100)
```

We want the ability to pass in a specific value for the first `case` clause, rather than hard-code it. So, we might expect the first `case` clause to match when `x` equals `y`, which holds the value of `100`, producing the following output when we run the script:

```
int: 99
found y!
int:
101
```

This is what we actually get:

```
<console>:12: warning: patterns after a variable pattern cannot match (SLS
8.1.1)
If you intended to match against parameter y of method checkY, you must use
backticks, like: case `y` =>
             case y => "found y!"
                  ^
<console>:13: warning: unreachable code due to variable pattern 'y' on line 12
             case i: Int => "int: "+i
                                    ^
<console>:13: warning: unreachable code
             case i: Int => "int: "+i
                                    ^
checkY: (y: Int)Unit
found y!
found y!
found y!
```

The `case y` actually means, “match anything (because there is no type annotation) and assign it to this *new* variable named `y`.” The `y` here is not interpreted as a reference to the method argument `y`. So, we actually wrote a default, match-all clause first, triggering the three warnings that this “variable pattern” will capture everything and so we won’t reach the second `case` expression. Then we get two warnings about unreachable code. The “SLS 8.1.1” refers to Section 8.1.1 in the *Scala Language Specification*.

The first error message tells us what to do: use “back ticks” to indicate we really want to match against the value held by `y`:

```
// src/main/scala/progscala2/patternmatching/match-surprise-fix.sc

def checkY(y: Int) = {
  for {
    x <- Seq(99, 100, 101)
  } {
    val str = x match {
                  "found               // The only change:
      case `y` => y!"                  `y`
                     "int:
      case i: Int => "      +i
    }
    println(str)
  }
}
checkY(100)
```

Now the output is what we want.

Warning

In `case` clauses, a term that begins with a lowercase letter is assumed to be the name of a new variable that will hold an extracted value. To refer to a previously defined variable, enclose it in back ticks. Conversely, a term that begins with an uppercase letter is assumed to be a type name.

Finally, sometimes we want to handle several different matches with the same code body. To avoid duplication, we could refactor the code body into a method, but `case` clauses also support an "or" construct, using a `|` method:

```
// src/main/scala/progscala2/patternmatching/match-variable2.sc

for {
  x <- Seq(1, 2, 2.7, "one", "two", 'four)
} {
  val str = x match {
                                "a number:
    case _: Int | _: Double => "           +x
    case "one"              => "string one"
                               "other string:
    case _: String          => "               +x
                               "unexpected value:
    case _                  => "                    + x
  }
  println(str)
}
```

Now, both `Int` and `Double` values are matched by the first `case` clause.

Matching on Sequences

`Seq` (for "sequence") is a parent type for the concrete collection types that support iteration over the elements in a deterministic order, such as `List` and `Vector`.

Let's examine the classic idiom for iterating through a `Seq` using pattern matching and recursion, and along the way, learn some useful fundamentals about sequences:

```
// src/main/scala/progscala2/patternmatching/match-seq.sc

val nonEmptySeq    = Seq(1, 2, 3, 4, 5)
// ❶
val emptySeq       = Seq.empty[Int]
val nonEmptyList   = List(1, 2, 3, 4, 5)
// ❷
val emptyList      = Nil
val nonEmptyVector = Vector(1, 2, 3, 4, 5)                              //
❸
val emptyVector    = Vector.empty[Int]
val nonEmptyMap    = Map("one" -> 1, "two" -> 2, "three" -> 3)          //
❹
val emptyMap       = Map.empty[String,Int]

def seqToString[T](seq: Seq[T]): String = seq match {                   //
❺
                        "$head +:
  case head +: tail => s"           + seqToString(tail)                 //
❻
  case Nil => "Nil"
// ❼
}

for (seq <- Seq(
// ❽
    nonEmptySeq, emptySeq, nonEmptyList, emptyList,
    nonEmptyVector, emptyVector, nonEmptyMap.toSeq, emptyMap.toSeq)) {
  println(seqToString(seq))
}
```

❶

Construct a nonempty `Seq[Int]` (a `List` is actually returned), followed by the idiomatic way of constructing an empty `Seq[Int]`.

❷

Construct a nonempty `List[Int]` (a subtype of `Seq`), followed by the special object in the library, `Nil`, that represents an empty `List` of any type parameter.

❸

Construct a nonempty `Vectors[Int]` (a subtype of `Seq`), followed by an empty `Vector[Int]`.

❹

Construct a nonempty `Map[String,Int]`, which *isn't* a subtype of `Seq`. We'll come back to this point in the discussion that follows. The keys are `String`s and the values are `Int`s. Then construct an empty `Map[String,Int]`.

❺

Define a recursive method that constructs a `String` from a `Seq[T]` for some type `T`. The body is one four-line expression that matches on the input `Seq[T]`.

❻

There are two match clauses and they are exhaustive. The first matches on any nonempty `Seq`, extracting the head, the first element, and the tail, which is the rest of the `Seq`. (`Seq` has `head` and `tail` methods, but here these terms are interpreted as variable names as usual for `case` clauses.) The body of the clause constructs a `String` with the head followed by `+:` followed by the result of calling `seqToString` on the tail.

❼

The only other possible case is an empty `Seq`. We can use the special object for empty `List`s, `Nil`, to match all the empty cases. Note that any `Seq` can always be interpreted as terminating with an empty instance of the same type, although only some types, like `List`, are actually implemented that way.

❽

Put the `Seq`s in another `Seq` (calling `toSeq` on the `Map`s to convert them to a sequence of key-value pairs), then iterate through it and print the results of calling `seqToString` on each one.

Here is the output:

```
1 +: 2 +: 3 +: 4 +: 5 +: Nil
Nil
1 +: 2 +: 3 +: 4 +: 5 +: Nil
Nil
1 +: 2 +: 3 +: 4 +: 5 +: Nil
Nil
(one,1) +: (two,2) +: (three,3) +:
Nil
Nil
```

`Map` is not a subtype of `Seq`, because it doesn't guarantee a particular order when you iterate over it. Hence, we called `Map.toSeq` to create sequences of key-value tuples. Still, the resulting `Seq` has the pairs in the insert order. That's a side effect of the implementation for small `Map`s, but not a general guarantee. The empty collections show that `seqToString` works correctly for empty collections.

There are two new kinds of `case` clauses. The first, `head +:` `tail`, matches the head element and the tail `Seq` (the remainder) of a sequence. The operator `+:` is the "cons" (construction) operator for sequences. It is similar to the `::` operator we saw in Precedence Rules for `List`s. Recall that methods that end with a colon (`:`) bind to the *right*, toward the `Seq` tail.

I'm calling them "operators" and "methods," but actually that's not quite right in this context; we'll come back to this expression a bit later and really examine what's going on. For now, let's note a few key points.

First, this `case` clause only matches a nonempty sequence, one with at least a head element, and it extracts that head element and the rest of the sequence into immutable variables named `head` and `tail`, respectively.

Second, to reiterate, the terms `head` and `tail` are arbitrary variable names. However, there are also `head` and `tail` methods on `Seq` for returning the head and tail of a sequence, respectively. Normally, it's clear from the context when the methods are being used. By the way, calling either method on an empty sequence results in a thrown exception.

Because `Seq` behaves conceptually like a linked list, where each head node holds an element and it points to the tail (the rest of the sequence), creating a hierarchical structure that looks schematically like the following for a four-node sequence, an empty sequence is the most natural marker at the end:

```
(node1, (node2, (node3, (node4,
(end))))
```

The Scala library has an object called `Nil` for lists and it matches all empty sequences, which is why we used it. We can use `Nil` even for collections that aren't a `List` because of the way equality for sequences is implemented. The types don't have to match exactly.

This variant adds the parentheses. We'll just use a few of the collections this time:

```
// src/main/scala/progscala2/patternmatching/match-seq-parens.sc

val nonEmptySeq    = Seq(1, 2, 3, 4, 5)
val emptySeq       = Seq.empty[Int]
val nonEmptyMap    = Map("one" -> 1, "two" -> 2, "three" -> 3)

def seqToString2[T](seq: Seq[T]): String = seq match {
                          "($head +:
  case head +: tail => s${seqToString2(tail)})"                        //
❶
  case Nil => "(Nil)"
}

for (seq <- Seq(nonEmptySeq, emptySeq, nonEmptyMap.toSeq)) {
  println(seqToString2(seq))
}
```

❶

Reformatted to add outer parentheses, `(…)`.

The output of this script is the following, which shows the hierarchical structure explicitly, where each "sublist" is surrounded by parentheses:

```
(1 +: (2 +: (3 +: (4 +: (5 +: (Nil))))))
(Nil)
((one,1) +: ((two,2) +: ((three,3) +:
(Nil))))
```

So, we process sequences with just two `case` clauses and recursion. Note that this implies something fundamental about all sequences; they are either empty or not. That sounds trite, but once you recognize simple patterns like this, it gives you a surprisingly general tool for "divide and conquer." The idiom used by `processSeq` is widely reusable.

Before Scala 2.10, it was common to use a closely related idiom for `List`s instead:

```
// src/main/scala/progscala2/patternmatching/match-list.sc

val nonEmptyList = List(1, 2, 3, 4, 5)
val emptyList    = Nil

def listToString[T](list: List[T]): String = list match {
                          "($head ::
  case head :: tail => s${listToString(tail)})"                        //
❶
  case Nil => "(Nil)"
}

for (l <- List(nonEmptyList, emptyList)) { println(listToString(l)) }
```

❶

Replaced `+:` with `::`.

The output is similar:

```
(1 :: (2 :: (3 :: (4 :: (5 ::
(Nil))))))
(Nil)
```

It's more conventional now to write code that uses `Seq`, so it can be applied to all subclasses, including `List` and `Vector`.

We can copy and paste the output of the previous examples to reconstruct the original objects:

```
scala> val s1 = (1 +: (2 +: (3 +: (4 +: (5 +: Nil)))))
s1: List[Int] = List(1, 2, 3, 4, 5)

       val l  = (1 :: (2 :: (3 :: (4 :: (5 :: Nil
scala> )))))
l: List[Int] = List(1, 2, 3, 4, 5)

scala> val s2 = (("one",1) +: (("two",2) +: (("three",3) +: Nil)))
s2: List[(String, Int)] = List((one,1), (two,2), (three,3), (four,4
))

       val m  = Map(s2 :_*
scala> )
m: scala.collection.immutable.Map[String,Int] =
  Map(one -> 1, two -> 2, three -> 3, four -> 4)
```

Note that the `Map.apply` factory method expects a variable argument list of two-element tuples. So, in order to use the sequence `s2` to construct a `Map`, we had to use the `:_*` idiom for the compiler to convert it to a variable-argument list.

So, there's an elegant symmetry between construction and pattern matching ("deconstruction") when using `+:` and `::`. We'll explore the implementation and see other examples later in this chapter.

Matching on Tuples

Tuples are also easy to match on, using their literal syntax:

```
// src/main/scala/progscala2/patternmatching/match-tuple.sc

val langs = Seq(
  ("Scala",   "Martin", "Odersky"),
  ("Clojure", "Rich",   "Hickey"),
  ("Lisp",    "John",   "McCarthy"))

for (tuple <- langs) {
  tuple match {
                                    "Found
    case ("Scala", _, _) => println(Scala"       )
// ❶
    case (lang, first, last) =>
// ❷
               "Found other language: $lang ($first,
      println(s$last)"                                    )
  }
}
```

❶

Match a three-element tuple where the first element is the string “Scala” and we ignore the second and third arguments.

❷

Match any three-element tuple, where the elements could be any type, but they are inferred to be `String`s due to the input `langs`. Extract the elements into variables `lang`, `first`, and `last`.

The output is this:

```
Found Scala
Found other language: Clojure (Rich,
Hickey)
Found other language: Lisp (John, McCarthy)
```

A tuple can be taken apart into its constituent elements. We can match on literal values within the tuple, at any positions we want, and we can ignore elements we don’t care about.

Guards in case Clauses

Matching on literal values is very useful, but sometimes you need a little additional logic:

```
// src/main/scala/progscala2/patternmatching/match-guard.sc

for (i <- Seq(1,2,3,4)) {
  i match {
    case _ if i%2 == 0 => println(s"even: $i")
// ❶
    case _             => println(s"odd: $i")
// ❷
  }
}
```

❶

Match only if `i` is even. We use `_` instead of a variable, because we already have `i`.

❷

Match the only other possibility, that `i` is odd.

The output is this:

```
odd: 1
even: 2
odd: 3
even: 4
```

Note that we didn't need parentheses around the condition in the `if` expression, just as we don't need them in `for` comprehensions.

Matching on case Classes

Let's see more examples of *deep matching*, where we examine the contents of instances of case classes:

```
// src/main/scala/progscala2/patternmatching/match-deep.sc

// Simplistic address type. Using all strings is questionable,
too.
case class Address(street: String, city: String, country: String)
case class Person(name: String, age: Int, address: Address)

                                          "1 Scala
val alice   = Person("Alice",   25, Address(Lane"         , "Chicago", "USA"
))
val bob     = Person("Bob",     29, Address("2 Java Ave.",  "Miami",   "USA"
))
                                          "3 Python
val charlie = Person("Charlie", 32, Address(Ct."          , "Boston",  "USA"
))

for (person <- Seq(alice, bob, charlie)) {
  person match {
                                                                  "Hi
    case Person("Alice", 25, Address(_, "Chicago", _) => println(Alice!"    )
    case Person("Bob", 29, Address("2 Java Ave.", "Miami", "USA")) =>
              "Hi
      println(Bob!"    )
    case Person(name, age, _) =>
               "Who are you, $age year-old person named
      println(s$name?"                                        )
  }
}
```

The output is this:

```
Hi Alice!
Hi Bob!
Who are you, 32 year-old person named
Charlie?
```

Note that we could match into nested types. Here's a more real-world example with tuples. Imagine we have a sequence of `(String,Double)` tuples for the names and prices of items in a store and we want to print them with their index. The `Seq.zipWithIndex` method is handy here:

```
// src/main/scala/progscala2/patternmatching/match-deep-tuple.sc

val itemsCosts = Seq(("Pencil", 0.52), ("Paper", 1.35), ("Notebook", 2.43))
val itemsCostsIndices = itemsCosts.zipWithIndex
for (itemCostIndex <- itemsCostsIndices) {
  itemCostIndex match {
                                          "$index: $item costs $cost
    case ((item, cost), index) => println(seach"
)
  }
}
```

Let's run it in the REPL using the `:load` command to see the types, as well as the output (reformatted slightly):

```
scala> :load src/main/scala/progscala2/patternmatching/match-deep-tuple.sc
Loading src/main/scala/progscala2/patternmatching/match-deep-tuple.sc...
itemsCosts: Seq[(String, Double)] =
  List((Pencil,0.52), (Paper,1.35), (Notebook,2.43))
itemsCostsIndices: Seq[((String, Double), Int)] =
  List(((Pencil,0.52),0), ((Paper,1.35),1), ((Notebook,2.43),2))
0: Pencil costs 0.52 each
1: Paper costs 1.35 each
2: Notebook costs 2.43 each
```

Note that the call to `zipWithIndex` returned tuples of the form `((name,cost),index)`. We matched on this form to extract the three elements and print them. I write code like this *a lot*.

unapply Method

So, not only the Scala library types, but also our own case classes can exploit pattern matching and extraction, even with deep nesting.

How does this work? We learned already in A Taste of Concurrency that a case class gets a *companion object* that has a factory method named `apply`, which is used for construction. Using "symmetry" arguments, we might infer that there must be another method generated called `unapply`, which is used for extraction or "deconstruction." Indeed there is such an *extractor* method and it is invoked when a pattern-match expression like this is encountered:

```
person match {
  case Person("Alice", 25, Address(_, "Chicago", _)) => ...
  ...
}
```

Scala looks for `Person.unapply(…)` and `Address.unapply(…)` and calls them. All `unapply` methods return an `Option[TupleN[…]]`, where `N` is the number of values that can be extracted from the object, three for the `Person` case class, and the types that parameterize the tuple match the values that can be extracted, `String`, `Int`, and `Address`, in this case. So, the `Person` companion object that the compiler generates looks like this:

```
object Person {
  def apply(name: String, age: Int, address: Address) =
    new Person(name, age, address)
  def unapply(p: Person): Option[Tuple3[String,Int,Address]] =
    Some((p.name, p.age, p.address))
  ...
}
```

Why is an `Option` used, if the compiler already knows that the object is a `Person`? Scala allows an implementation of `unapply` to "veto" the match for some reason and return `None`, in which case Scala will attempt to use the next `case` clause. Also, we don't have to expose all fields of the instance if we don't want to. We could suppress our `age`, if we're embarrassed by it. We'll explore the details in unapplySeq Method, but for now, just note that the extracted fields are returned in a `Some` wrapping a `Tuple3`. The compiler then extracts those tuple elements for comparison with literal values, like our first clause that looks for "Alice," or it assigns them to variables we've named, or it drops the ones we don't care about when we use `_`.

Note

To gain some performance benefits, Scala 2.11.1 relaxed the requirement that `unapply` return an `Option[T]`. It can now return any type as long as it has the following methods:

```
def isEmpty: Boolean
def get: T
```

The `unapply` methods are invoked recursively, if necessary, as in this case where we have a nested `Address` object in the `Person`. Similarly, our tuple example invoked the corresponding `unapply` methods recursively.

It's no coincidence that the same `case` keyword is used for declaring "special" classes and for `case` expressions in `match` expressions. The features of case classes were designed to enable convenient pattern matching.

Before we move on, note that the return type signature, `Option[Tuple3[String,Int,Address]]`, is a bit wordy. Scala lets us use the tuple literal syntax for the types, too! The following type declarations are all equivalent:

```
val t1: Option[Tuple3[String,Int,Address]] = ...
val t2: Option[(String,Int,Address)] = ...
val t3: Option[ (String, Int, Address) ] = ...
```

The tuple literal syntax is easier to read. Extra whitespace helps, too.

Now let's return to that mysterious `head +: tail` expression and really understand what it means. We saw that the `+:` (cons) operator can be used to construct a new sequence by prepending an element to an existing sequence, and we can construct an entire sequence from scratch this way:

```
val list = 1 +: 2 +: 3 +: 4 +: Nil
```

Because `+:` is a method that binds to the right, we first prepend `4` to `Nil`, then prepend `3` to that list, etc.

Scala wants to support uniform syntax for *construction* and *destruction/extraction*, when possible. We've seen this at work for sequences, lists, and tuples. These operations are *dual*, inverses of each other.

If the construction of sequences is done with a method named `+:`, how is extraction done with the same syntax? We just looked at `unapply` methods, but they could cheat; `Person.unapply` and `TupleN.unapply` already know now many "things" are in their instances, three and N, respectively. Now we want to support nonempty collections of arbitrary length.

To do that, the Scala library defines a special singleton object named `+:`. Yes, the name is "+:". Like methods, types can have names with a wide variety of characters.

It has just one method, the `unapply` method the compiler needs for our extraction `case` statement. The declaration of `unapply` is schematically like this (I've simplified the actual declaration a bit, because we haven't yet covered details about the type system that we need to understand the full signature):

```
def unapply[T, Coll](collection: Coll): Option[(T, Coll
)]
```

The head is of type `T`, which is inferred, and some collection type `Coll`, which represents the type of the input collection and the output tail collection. So, an `Option` of a two-element tuple with the head and tail is returned.

How can the compiler see the expression `case head +: tail =>` … and use a method `+:.unapply(collection)`? We might expect that the `case` clause would have to be written `case +:(head, tail) =>` … to work consistently with the behavior we just examined for pattern matching with `Person`, `Address`, and tuples.

As a matter of fact, we can write it that way:

```
scala> def processSeq2[T](l: Seq[T]): Unit = l match {
     |   case +:(head, tail) =>
                "%s +:
           printf("       , head
     | )
           processSeq2(tail
     | )
       case Nil => print("Nil"
     | )
     | }

scala> processSeq2(List(1,2,3,4,5))
1 +: 2 +: 3 +: 4 +: 5 +: Nil
```

But we can also use *infix* notation, `head +: tail`, because the compiler exploits another bit of syntactic sugar. Types with two type parameters can be written with infix notation and so can `case` clauses. Consider this REPL session:

```
// src/main/scala/progscala2/patternmatching/infix.sc
scala> case class With[A,B](a: A, b: B)
defined class With

scala> val with1: With[String,Int] = With("Foo", 1)
with1: With[String,Int] = With(Foo,1)

       val with2: String With Int  = With("Bar", 2
scala> )
with2: With[String,Int] = With(Bar,2)

scala> Seq(with1, with2) foreach { w =>
         w match
     | {
                                     "$s with
           case s With i => println(s$i"
     | )
           case _        => println(s
       "Unknown:
     | $w"          )

     | }
     | }
Foo with 1
Bar with 2
```

So we can write the type signature one of two ways, `With[String,Int]` or `String With Int`. The latter reads nicely, but it might confuse less experienced Scala programmers. However, note that trying to initialize a value in a similar way doesn't work:

```
// src/main/scala/progscala2/patternmatching/infix.sc
scala> val w = "one" With 2
<console>:7: error: value With is not a member of String
       val w = "one" With 2
                     ^
```

There is also a similar object for `List`s, `::` What if you want to process the sequence elements in reverse? There's an app… err… object for that! The library object `:+` allows you to match on the end elements and work backward:

```
// src/main/scala/progscala2/patternmatching/match-reverse-seq.sc
// Compare to match-
seq.sc

val nonEmptyList   = List(1, 2, 3, 4, 5)
val nonEmptyVector = Vector(1, 2, 3, 4, 5)
val nonEmptyMap    = Map("one" -> 1, "two" -> 2, "three" -> 3)

def reverseSeqToString[T](l: Seq[T]): String = l match {
                                                    " :+
  case prefix :+ end => reverseSeqToString(prefix) + s$end"
  case Nil => "Nil"
}

for (seq <- Seq(nonEmptyList, nonEmptyVector, nonEmptyMap.toSeq)) {
  println(reverseSeqToString(seq))
}
```

The output is this:

```
Nil :+ 1 :+ 2 :+ 3 :+ 4 :+ 5
Nil :+ 1 :+ 2 :+ 3 :+ 4 :+ 5
Nil :+ (one,1) :+ (two,2) :+ (three,3)
```

Note that the `Nils` come first and the methods bind to the left. Also, the same output was generated for the first two inputs, a `List` and a `Vector`.

You should compare the implementations of `seqToString` and `reverseSeqToString`, which implement the recursion differently. Make sure you understand how they work.

As before, you could use this output to reconstruct collections (skipping the duplicate second line of the previous output):

```
scala> Nil :+ 1 :+ 2 :+ 3 :+ 4 :+ 5
res0: List[Int] = List(1, 2, 3, 4, 5)
```

Warning

For `List`, the `:+` method for appending elements and the `:+` object for pattern matching both require *O(n)* time, because they have to traverse the list from the head. However, some other sequences, such as `Vector`, are *O(1)*.

unapplySeq Method

What if you want a bit more flexibility to return a sequence of extracted items, rather than a fixed number of them? The `unapplySeq` method lets you do this. It turns out the `Seq` companion object implements `apply` and `unapplySeq`, but not `unapply`:

```
def apply[A](elems: A*): Seq[A]
def unapplySeq[A](x: Seq[A]): Some[Seq[A]]
```

Recall that `A*` means that `elems` is a variable argument list. Matching with `unapplySeq` is invoked in this variation of our previous example for `+:`, where we examine a "sliding window" of pairs of elements at a time:

```
// src/main/scala/progscala2/patternmatching/match-seq-unapplySeq.sc

val nonEmptyList   = List(1, 2, 3, 4, 5)
// ❶
val emptyList      = Nil
val nonEmptyMap    = Map("one" -> 1, "two" -> 2, "three" -> 3)

// Process
pairs
def windows[T](seq: Seq[T]): String = seq match {
  case Seq(head1, head2, _*) =>
// ❷
     "($head1, $head2),
    s"                    + windows(seq.tail)
// ❸
  case Seq(head, _*) =>
     "($head, _),
    s"               + windows(seq.tail)
// ❹
  case Nil => "Nil"
}

for (seq <- Seq(nonEmptyList, emptyList, nonEmptyMap.toSeq)) {
  println(windows(seq))
}
```

❶

Use a nonempty `List`, `Nil`, and a `Map`.

❷

It looks like we're calling `Seq.apply(…)`, but in a match clause, we're actually calling `Seq.unapplySeq`. We grab the first two elements and ignore the rest of the variable arguments with `_*`. Think of the `*` as matching zero to many, like in regular expressions.

❸

Format a string with the first two elements, then move the "window" one over by calling `seq.tail`. Note that we didn't capture this tail in the match.

❹

We also need a match for a one-element sequence or we won't have exhaustive matching. Use `_` for the nonexistent "second" element. We actually know that this call to `windows(seq.tail)` will simply return `Nil`, but rather than duplicate the string, we take the extra performance hit by calling the method again.

Note the *sliding window* output:

```
(1, 2), (2, 3), (3, 4), (4, 5), (5, _), Nil
Nil
((one,1), (two,2)), ((two,2), (three,3)), ((three,3), _),
Nil
```

We could still use the `+:` matching we saw before, which is more elegant:

```
// src/main/scala/progscala2/patternmatching/match-seq-without-unapplySeq.sc

val nonEmptyList   = List(1, 2, 3, 4, 5)
val emptyList      = Nil
val nonEmptyMap    = Map("one" -> 1, "two" -> 2, "three" -> 3)

// Process
pairs
def windows2[T](seq: Seq[T]): String = seq match {
                                    "($head1, $head2),
  case head1 +: head2 +: tail => s"                    + windows2(seq.tail)
                         "($head, _),
  case head +: tail => s"              + windows2(tail)
  case Nil => "Nil"
}

for (seq <- Seq(nonEmptyList, emptyList, nonEmptyMap.toSeq)) {
  println(windows2(seq))
}
```

Working with sliding windows is actually so useful that `Seq` gives us two methods to create them:

```
scala> val seq = Seq(1,2,3,4,5)
seq: Seq[Int] = List(1, 2, 3, 4, 5)

scala> val slide2 = seq.sliding(2)
slide2: Iterator[Seq[Int]] = non-empty iterator

scala> slide2.toSeq
res0: Seq[Seq[Int]] = res56: Seq[Seq[Int]] = Stream(List(1, 2), ?)

scala> slide2.toList
res1: List[Seq[Int]] = List(List(1, 2), List(2, 3), List(3, 4), List(4, 5))

scala> seq.sliding(3,2).toList
res2: List[Seq[Int]] = List(List(1, 2, 3), List(3, 4, 5))
```

Both `sliding` methods return an iterator, meaning they are “lazy” and don’t immediately make a copy of the list, which would be expensive for large sequences. Calling `toSeq` converts the iterator into a `collection.immutable.Stream`, a lazy list that evaluates its head eagerly, but only evaluates the tail elements on demand. In contrast, calling `toList` evaluates the whole iterator eagerly, creating a `List`.

Note there is a slight difference in the results. We don’t end with `(5, _)`, for example.

Matching on Variable Argument Lists

In Inferring Type Information, we described how Scala supports variable argument lists for methods. For example, suppose I'm writing a tool for interoperating with SQL and I want a case class to represent the `WHERE foo IN (val1, val2, …)` SQL clause (this example is inspired by a real project, not open source). Here's an example case class with a variable argument list to handle the list of values in this clause. I've also included some other definitions for the `WHERE x OP y` clauses, where `OP` is one of the SQL comparison operators:

```
// src/main/scala/progscala2/patternmatching/match-vararglist.sc

// Operators for WHERE
clauses
object Op extends Enumeration {
// ❶
  type Op = Value

  val EQ   = Value("=")
  val NE   = Value("!=")
  val LTGT = Value("<>")
  val LT   = Value("<")
  val LE   = Value("<=")
  val GT   = Value(">")
  val GE   = Value(">=")
}
import Op._

// Represent a SQL "WHERE x op value" clause, where +op+ is
a
// comparison operator: =, !=, <>, <, <=, >, or
>=.
case class WhereOp[T](columnName: String, op: Op, value: T)          //
❷

// Represent a SQL "WHERE x IN (a, b, c, ...)"
clause.
case class WhereIn[T](columnName: String, val1: T, vals: T*)         //
❸

val wheres = Seq(
// ❹
  WhereIn("state", "IL", "CA", "VA"),
  WhereOp("state", EQ, "IL"),
  WhereOp("name", EQ, "Buck Trends"),
  WhereOp("age", GT, 29))

for (where <- wheres) {
  where match {
    case WhereIn(col, val1, vals @ _*) =>
// ❺
                                              ",
      val valStr = (val1 +: vals).mkString("   )
                "WHERE $col IN
      println (s($valStr)"               )
                                             "WHERE $col $op
    case WhereOp(col, op, value) => println (s$value"                )
                        "ERROR: Unknown expression:
    case _ => println (s$where"                            )
  }
}
```

❶

An enumeration for the comparison operators, where we assign a “name” that’s the string representation of

the operator in SQL.

❷

A case class for `WHERE x OP y` clauses.

❸

A case class for `WHERE x IN (val1, val2, …)` clauses.

❹

Some example objects to parse.

❺

Note the syntax for matching on a variable argument: `name @ _*`.

The syntax for pattern matching on a variable argument list, `name @ _*`, is not that intuitive. Occasionally, you'll need it. Here is the output:

```
WHERE state IN (IL, CA,
VA)
WHERE state = IL
WHERE name = Buck Trends
WHERE age > 29
```

Matching on Regular Expressions

Regular expressions (or *regexes*) are convenient for extracting data from strings that have a particular structure.

Scala wraps Java's regular expressions.[] Here is an example:

```
// src/main/scala/progscala2/patternmatching/match-regex.sc

                      """Book: title=([^,]+),\s+author=
val BookExtractorRE = (.+)"""                                    .r     //
❶
                           """Magazine: title=([^,]+),\s+issue=
val MagazineExtractorRE = (.+)"""                                 .r

val catalog = Seq(
  "Book: title=Programming Scala Second Edition, author=Dean
  Wampler"                                                          ,
  "Magazine: title=The New Yorker, issue=January
  2014"                                                ,
  "Unknown: text=Who put this
  here??"
)

for (item <- catalog) {
  item match {
    case BookExtractorRE(title, author) =>
// ❷
               """Book "$title", written by
      println(s$author"""                                )
    case MagazineExtractorRE(title, issue) =>
               """Magazine "title", issue
      println(s$issue"""                               )
                           "Unrecognized entry:
    case entry => println(s$entry"                      )
  }
}
```

❶

Match a book string, with two *capture groups* (note the parentheses), one for the title and one for the author. Calling the `r` method on a string creates a regex from it. Match a magazine string, with *capture groups* for the title and issue (date).

❷

Use them like case classes, where the string matched by each capture group is assigned to a variable.

The output is:

```
Book "Programming Scala Second Edition", written by Dean
Wampler
Magazine "The New Yorker", issue January 2014
Unrecognized entry: Unknown: text=Who put this here??
```

We use triple-quoted strings for the regexes. Otherwise, we would have to escape the regex “backslash” constructs, e.g, `\\s` instead of `\s`. You can also define regular expressions by creating new instances of the `Regex` class, as in `new Regex("""\W""")`, but this isn’t very common.

Warning

Using interpolation in triple-quoted strings doesn't work cleanly for the regex escape sequences. You still need to escape these sequences, e.g., `s"""$first\\s+$second""".r` instead of `s"""$first\s+$second""".r`. If you aren't using interpolation, escaping isn't necessary.

`scala.util.matching.Regex` defines several methods for other manipulations, such as finding and replacing matches.

More on Binding Variables in case Clauses

Suppose you want to extract values from an object, but you also want to assign a variable to the whole object itself.

Let's modify our previous example matching on fields in `Person` instances:

```
// src/main/scala/progscala2/patternmatching/match-deep2.sc

case class Address(street: String, city: String, country: String)
case class Person(name: String, age: Int, address: Address)

                                          "1 Scala
val alice   = Person("Alice",   25, Address(Lane"         , "Chicago", "USA"
))
val bob     = Person("Bob",     29, Address("2 Java Ave.",  "Miami",   "USA"
))
                                          "3 Python
val charlie = Person("Charlie", 32, Address(Ct."          , "Boston",  "USA"
))

for (person <- Seq(alice, bob, charlie)) {
  person match {
                                                    "Hi Alice!
    case p @ Person("Alice", 25, address) => println(s$p"           )
    case p @ Person("Bob", 29, a @ Address(street, city, country)) =>
               "Hi ${p.name}! age ${p.age}, in
      println(s${a.city}"                                  )
    case p @ Person(name, age, _) =>
               "Who are you, $age year-old person named $name?
      println(s$p"                                              )
  }
}
```

The `p @` … syntax assigns to `p` the whole `Person` instance and similarly for `a @` … and an `Address`. Here is the output now (reformatted to fit the page):

```
Hi Alice! Person(Alice,25,Address(1 Scala Lane,Chicago,USA))
Hi Bob! age 29, in Miami
Who are you, 32 year-old person named Charlie?
Person(Charlie,32,
  Address(3 Python Ct.,Boston,USA))
```

Keep in mind that if we aren't extracting fields from the `Person` instance, we can just write `p: Person => …` .

More on Type Matching

Consider the following example, where we attempt to discriminate between `List[Double]` and `List[String]` inputs:

```
// src/main/scala/progscala2/patternmatching/match-types.sc
scala> for {
         x <- Seq(List(5.5,5.6,5.7), List("a", "b"
     | ))
     | } yield (x match {
         case seqd: Seq[Double] => ("seq double", seqd
     | )
         case seqs: Seq[String] => ("seq string", seqs
     | )
         case _                 => ("unknown!",
     | x)
     | })
<console>:12: warning: non-variable type argument Double in type pattern
Seq[Double] (the underlying of Seq[Double]) is unchecked since it is
eliminated by erasure
                case seqd: Seq[Double] => ("seq double", seqd)
                           ^
<console>:13: warning: non-variable type argument String in type pattern
Seq[String] (the underlying of Seq[String]) is unchecked since it is
eliminated by erasure
                case seqs: Seq[String] => ("seq string", seqs)
                           ^
<console>:13: warning: unreachable code
                case seqs: Seq[String] => ("seq string", seqs)
                                          ^
res0: List[(String, List[Any])] =
  List((seq double,List(5.5, 5.6, 5.7)),(seq double,List(a, b)))
```

What do the warnings mean? Scala runs on the JVM and these warnings result from the JVM's *type erasure*, a historical legacy of Java's introduction of *generics* in Java 5. In order to avoid breaking older code, the JVM byte code doesn't retain information about the actual type parameters that were used for instances of generic (parameterized) types, like `List`.

So, the compiler is warning us that, while it can check that a given object is a `List`, it can't check at *runtime* that it's a `List[Double]` or a `List[String]`. In fact, it considers the second `case` clause for `List[String]` to be unreachable code, meaning the previous `case` clause for `List[Double]` will match *any* `List`. Note the output, which shows that "seq double" was written for both inputs.

One ugly, but effective workaround is to match on the collection first, then use a nested match on the head element to determine the type. We now have to handle an empty sequence, too:

```
// src/main/scala/progscala2/patternmatching/match-types2.sc

def doSeqMatch[T](seq: Seq[T]): String = seq match {
  case Nil => "Nothing"
  case head +: _ => head match {
    case _ : Double => "Double"
    case _ : String => "String"
              "Unmatched seq
    case _ => element"
  }
}

for {
  x <- Seq(List(5.5,5.6,5.7), List("a", "b"), Nil)
} yield {
  x match {
                          "seq
    case seq: Seq[_] => (s${doSeqMatch(seq)}"     , seq)
    case _           => ("unknown!", x)
  }
}
```

This script returns the desired result,
`Seq((seq Double,List(5.5, 5.6, 5.7)), (seq String,List(a, b)), (seq Nothing,List()))` .

Sealed Hierarchies and Exhaustive Matches

Let's revisit the need for exhaustive matches and consider the situation where we have a `sealed` class hierarchy, which we discussed in Sealed Class Hierarchies. As an example, suppose we define the following code to represent the allowed message types or "methods" for HTTP:

```
//
src/main/scala/progscala2/patternmatching/http.sc

sealed abstract class HttpMethod() {
// ❶
    def body: String
// ❷
    def bodyLength = body.length
}

case class Connect(body: String) extends HttpMethod                  //
❸
case class Delete (body: String) extends HttpMethod
case class Get    (body: String) extends HttpMethod
case class Head   (body: String) extends HttpMethod
case class Options(body: String) extends HttpMethod
case class Post   (body: String) extends HttpMethod
case class Put    (body: String) extends HttpMethod
case class Trace  (body: String) extends HttpMethod

def handle (method: HttpMethod) = method match {                     //
❹
```

```
                          "connect: (length: ${method.bodyLength})
  case Connect (body) => s$body"
                          "delete:  (length: ${method.bodyLength})
  case Delete  (body) => s$body"
                          "get:     (length: ${method.bodyLength})
  case Get     (body) => s$body"
                          "head:    (length: ${method.bodyLength})
  case Head    (body) => s$body"
                          "options: (length: ${method.bodyLength})
  case Options (body) => s$body"
                          "post:    (length: ${method.bodyLength})
  case Post    (body) => s$body"
                          "put:     (length: ${method.bodyLength})
  case Put     (body) => s$body"
                          "trace:   (length: ${method.bodyLength})
  case Trace   (body) => s$body"
}

val methods = Seq(
          "connect
  Connect(body..."         ),
          "delete
  Delete (body..."        ),
          "get
  Get    (body..."     ),
          "head
  Head   (body..."      ),
          "options
  Options(body..."         ),
          "post
  Post   (body..."      ),
          "put
  Put    (body..."     ),
          "trace
  Trace  (body..."       ))

methods foreach (method => println(handle(method)))
```

❶

Define a sealed, abstract base class `HttpMethod`. Because it is declared `sealed`, the only allowed subtypes must be defined in this file.

❷

Define a method for the body of the HTTP message.

❸

Define eight case classes that extend `HttpMethod`. Note that each declares a constructor argument `body: String`, which is a `val` because each of these types is a case class. This `val` *implements* the abstract `def` method in `HttpMethod`.

❹

An *exhaustive* pattern-match expression, even though we don't have a default clause, because the `method` argument can only be an instance of one of the eight case classes we've defined.

Tip

When pattern matching on an instance of a sealed base class, the match is exhaustive if the `case` clauses cover all the derived types defined in the same source file. Because no user-defined derived types are allowed, the match can never become nonexhaustive as the project evolves, since users are prevented from defining new types.

A corollary is to avoid using `sealed` if the type hierarchy is at all likely to change. Instead, rely on your traditional object-oriented inheritance principles, including polymorphic methods. What if you added a new derived type, either in this file or in another file, and you removed the `sealed` keyword on `HttpMethod`? You would have to find and fix all pattern-match clauses in your code base *and* your client's code bases that match on `HttpMethod` instances.

As a side note, we are exploiting a useful feature for implementing certain methods. An abstract, no-argument method declaration in a parent type can be implemented by a `val` in a subtype. This is because a `val` has a single, fixed value (of course), whereas a no-argument method returning the same type can return any value of the type. Hence, the `val` implementation is more restrictive in the return type, which means using it where the method is "called" is always just as safe as calling a method. In fact, this is an application of *referential transparency*, where we are substituting a value for an expression that *should* always return the same value!

Tip

An abstract, no-argument method declaration in a parent type can be implemented by a `val` in a subtype. A recommended practice is to declare abstract, no-argument methods instead of `val`s in abstract parent types, leaving subtype implementers greater freedom to implement the member with either a method or a `val`.

Running this script yields the following output:

```
connect: (length: 15) connect
body...
delete:  (length: 14) delete body...
get:     (length: 11) get body...
head:    (length: 12) head body...
options: (length: 15) options
body...
post:    (length: 12) post body...
put:     (length: 11) put body...
trace:   (length: 13) trace body...
```

The `HTTPMethod` case classes are small, so we could in principle also use an `Enumeration` for them, but there's a big drawback. The compiler can't tell whether or not the match clauses on `Enumeration` values are exhaustive. If we converted this example to use `Enumeration` and forgot a match clause for `Trace`, we would only know at runtime when a `MatchError` is thrown.

Warning

Avoid enumerations when pattern matching is required. The compiler can't tell if the matches are exhaustive.

Other Uses of Pattern Matching

Fortunately, this powerful feature is not limited to `case` clauses. You can use pattern matching when defining values,

including in `for` comprehensions:

```
scala> case class Address(street: String, city: String, country: String)
defined class Address

scala> case class Person(name: String, age: Int, address: Address)
defined class Person

scala> val Person(name, age, Address(_, state, _)) =
                                      "1 Scala
     |   Person("Dean", 29, Address(Way"         , "CA", "USA"))
name: String = Dean
age: Int = 29
state: String = CA
```

Yes, in one step we extracted all those fields from the `Person` object, and skipped a few of them. It works for `List` s, too:

```
scala> val head +: tail = List(1,2,3)
head: Int = 1
tail: List[Int] = List(2, 3)

scala> val head1 +: head2 +: tail = Vector(1,2,3)
head1: Int = 1
head2: Int = 2
tail: scala.collection.immutable.Vector[Int] = Vector(3)

scala> val Seq(a,b,c) = List(1,2,3)
a: Int = 1
b: Int = 2
c: Int = 3

scala> val Seq(a,b,c) = List(1,2,3,4)
scala.MatchError: List(1, 2, 3, 4) (of class collection.immutable.$colon$colon)
  ... 43 elided
```

Very handy. Try some of your own examples.

We can use pattern matching in `if` expressions:

```
                                          "1 Scala
scala> val p = Person("Dean", 29, Address(Way"         , "CA", "USA"))
p: Person = Person(Dean,29,Address(1 Scala Way,CA,USA))

       if (p == Person("Dean", 29,
                       "1 Scala
            |    Address(Way"         , "CA", "USA"))) "yes" else
scala> "no"
res0: String = yes

       if (p == Person("Dean", 29,
                       "1 Scala
            |    Address(Way"         , "CA", "USSR"))) "yes" else
scala> "no"
res1: String = no
```

However, the _ placeholders don't work here:

```
scala> if (p == Person(_, 29, Address(_, _, "USA"))) "yes" else "no"
<console>:13: error: missing parameter type for expanded function
 ((x$1) => p.$eq$eq(Person(x$1,29,((x$2,x$3) => Address(x$2,x$3,"USA")))))
             if (p == Person(_, 29, Address(_, _, "USA"))) "yes" else
"no"
                                ^
 ...
```

There's an internal function called `$eq$eq` that's used for the `==` test. We've opened the magic kimono a bit. Because the JVM specification only allows alphanumeric characters, _, and `$` in identifiers, Scala "mangles" nonalphanumeric characters to something acceptable to the JVM. In this case, `=` becomes `$eq`. All these mappings are listed in Table 22-1 in Chapter 22.

Suppose we have a function that takes a sequence of integers and returns the sum and count of the elements in a tuple:

```
scala> def sum_count(ints: Seq[Int]) = (ints.sum, ints.size)

scala> val (sum, count) = sum_count(List(1,2,3,4,5))
sum: Int = 15
count: Int = 5
```

I use this idiom frequently. We saw a teaser example in Expanded Scope and Value Definitions that used pattern matching in a `for` comprehension. Here's the relevant part from that example again:

```
// src/main/scala/progscala2/patternmatching/scoped-option-
for.sc

                                                "Yorkshire
val dogBreeds = Seq(Some("Doberman"), None, Some(Terrier"           ),
                                                 "Scottish
                    Some("Dachshund"), None, Some(Terrier"          ),
                              "Great              "Portuguese Water
                    None, Some(Dane"       ), Some(Dog"
))

        "second
println(pass:"        )
for {
  Some(breed) <- dogBreeds
  upcasedBreed = breed.toUpperCase()
} println(upcasedBreed)
```

As before, the output is the following:

```
DOBERMAN
YORKSHIRE TERRIER
DACHSHUND
SCOTTISH TERRIER
GREAT DANE
PORTUGUESE WATER
DOG
```

A particularly convenient use of pattern matching and `case` clauses is to make function literals of complex arguments easier to use:

```
// src/main/scala/progscala2/patternmatching/match-fun-args.sc

case class Address(street: String, city: String, country: String)
case class Person(name: String, age: Int)

val as = Seq(
          "1 Scala
  Address(Lane"         , "Anytown", "USA"),
          "2 Clojure
  Address(Lane"           , "Othertown", "USA"))
val ps = Seq(
  Person("Buck Trends", 29),
  Person("Clo Jure", 28))

val pas = ps zip as

// Ugly way:
pas map { tup =>
  val Person(name, age) = tup._1
  val Address(street, city, country) = tup._2
   "$name (age: $age) lives at $street, $city, in
  s$country"
}

// Nicer
way:
pas map {
  case (Person(name, age), Address(street, city, country)) =>
     "$name (age: $age) lives at $street, $city, in
    s$country"
}
```

Note that the type of the zipped list is `Seq[(Person,Address)]`. So, the function we pass to `map` must have the `(Person,Address) =>` type `String`. We show two functions. The first is a “regular” function that takes a tuple argument, then pattern matches to extract the fields from the two elements of the tuple.

The second function is a *partial function*, as discussed in Partial Functions. The syntax is more concise, especially for extracting values from tuples and more complex structures. Just remember that because the function given is actually a `PartialFunction`, the `case` expressions must match the inputs exactly or a `MatchError` will be thrown at runtime.

In both cases, the resulting sequence of strings is the following:

```
List(
  "Buck Trends (age: 29) lives at 1 Scala Lane, Anytown, in
  USA"                                                    ,
  "Clo Jure (age: 28) lives at 2 Clojure Lane, Othertown, in
  USA"                                                     )
```

Finally, we can use pattern matching on a regular expression to decompose a string. Here’s an example extracted from tests I once wrote for parsing (simple!) SQL strings:

```
// src/main/scala/progscala2/patternmatching/regex-assignments.sc
scala> val cols = """\*|[\w, ]+""" // for columns
cols: String = \*|[\w, ]+

scala> val table = """\w+"""       // for table names
table: String = \w+

scala> val tail = """.*"""         // for other clauses
tail: String = .*

scala> val selectRE =
     | s"""SELECT\\s*(DISTINCT)?\\s+($cols)\\s*FROM\\s+($table)\\s*($tail)?;""".
r
selectRE: scala.util.matching.Regex = \
  SELECT\s*(DISTINCT)?\s+(\*|[\w, ]+)\s*FROM\s+(\w+)\s*(.*)?;

scala> val selectRE(distinct1, cols1, table1, otherClauses) =
     |
"SELECT DISTINCT * FROM
atable;"
distinct1: String = DISTINCT
cols1: String = *
table1: String = atable
otherClauses: String = ""

scala> val selectRE(distinct2, cols2, table2, otherClauses) =
     |
"SELECT col1, col2 FROM
atable;"
distinct2: String = null
                "col1, col2
cols2: String = "
table2: String = atable
otherClauses: String = ""

scala> val selectRE(distinct3, cols3, table3, otherClauses) =
     |
"SELECT DISTINCT col1, col2 FROM
atable;"
distinct3: String = DISTINCT
                "col1, col2
cols3: String = "
table3: String = atable
otherClauses: String = ""

scala> val selectRE(distinct4, cols4, table4, otherClauses) =
     |
"SELECT DISTINCT col1, col2 FROM atable WHERE col1 =
'foo';"
distinct4: String = DISTINCT
                "col1, col2
cols4: String = "
table4: String = atable
otherClauses: String = WHERE col1 = 'foo'
```

Note that we had to add extra backslashes, e.g., `\\s` instead of `\s`, in the regular expression string, because we used interpolation.

Obviously, using regular expressions to parse complex text, like XML or programming languages, has its limits. Beyond simple cases, consider a parser library, like the ones we'll discuss in Chapter 20.

Pattern matching is a powerful "protocol" for extracting data inside data structures. One of the unintended consequences of the *JavaBeans* model was that it encouraged developers to expose fields in their objects through getters and setters, often ignoring concerns that state should be encapsulated and only exposed as appropriate, especially for mutable fields. Access to state information should be carefully designed to reflect the abstraction exposed.

Consider using pattern matching for those times when you need to extract information in a controlled way. You can customize `unapply` methods, as we saw in unapply Method, to control the state exposed. These methods let you extract that information while hiding the implementation details. In fact, the information returned by `unapply` might be a transformation of the actual fields in the type.

Finally, when designing pattern-matching statements, be wary of relying on a default `case` clause. Under what circumstances would "none of the above" be the correct answer? It may indicate that the design should be refined so you know more precisely all the possible matches that might occur.

Along with `for` comprehensions, pattern matching makes idiomatic Scala code concise, yet powerful. It's not unusual for Scala programs to have 10 times fewer lines of code than comparable programs written in Java.

So, even though Java 8 added anonymous functions ("lambdas"), which was an enormous improvement, tools like pattern matching and `for` comprehensions are compelling reasons to switch to Scala.

Recap and What's Next

Pattern matching is a hallmark of many functional languages. It is a flexible and concise technique for extracting data from data structures. We saw examples of pattern matching in `case` clauses and how to use pattern matching in other expressions.

The next chapter discusses a unique, powerful, but controversial feature in Scala, *implicits*, which are a set of tools for building intuitive DSLs (domain-specific languages), reducing boilerplate, and making APIs both easier to use and more amenable to customization.