Chapter 15

Case Classes and Pattern Matching

This chapter introduces *case classes* and *pattern matching*, twin constructs that support you when writing regular, non-encapsulated data structures. These two constructs are particularly helpful for tree-like recursive data.

If you have programmed in a functional language before, then you will probably recognize pattern matching. But case classes will be new to you. Case classes are Scala's way to allow pattern matching on objects without requiring a large amount of boilerplate. Generally, all you need to do is add a single case keyword to each class that you want to be pattern matchable.

This chapter starts with a simple example of case classes and pattern matching. It then goes through all of the kinds of patterns that are supported, talks about the role of *sealed* classes, discusses the Option type, and shows some non-obvious places in the language where pattern matching is used. Finally, a larger, more realistic example of pattern matching is shown.

15.1 A SIMPLE EXAMPLE

Before delving into all the rules and nuances of pattern matching, it is worth looking at a simple example to get the general idea. Let's say you need to write a library that manipulates arithmetic expressions, perhaps as part of a domain-specific language you are designing.

A first step to tackling this problem is the definition of the input data. To keep things simple, we'll concentrate on arithmetic expressions consisting of variables, numbers, and unary and binary operations. This is expressed by the hierarchy of Scala classes shown in Listing 15.1.

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
    left: Expr, right: Expr) extends Expr
```

Listing 15.1 - Defining case classes.

The hierarchy includes an abstract base class Expr with four subclasses, one for each kind of expression being considered.[1] The bodies of all five classes are empty. As mentioned previously, in Scala you can leave out the braces around an empty class body if you wish, soclass C is the same as class C {}.

Case classes

The other noteworthy thing about the declarations of Listing 15.1 is that each subclass has acase modifier. Classes with such a modifier are called case classes. Using the modifier makes the Scala compiler add some syntactic conveniences to your class.

First, it adds a factory method with the name of the class. This means that, for instance, you can write Var("x") to construct a Var object, instead of the slightly longer new Var("x"):

```
scala> val v = Var("x")
v: Var = Var(x)
```

The factory methods are particularly nice when you nest them. Because there are no noisy newkeywords sprinkled throughout the code, you can take in the expression's structure at a glance:

```
scala> val op = BinOp("+", Number(1), v)
op: BinOp = BinOp(+,Number(1.0),Var(x))
```

The second syntactic convenience is that all arguments in the parameter list of a case class implicitly get a val prefix, so they are maintained as fields:

```
scala> v.name
res0: String = x

scala> op.left
res1: Expr = Number(1.0)
```

Third, the compiler adds "natural" implementations of methods toString, hashCode, and equalsto your class. They will print, hash, and compare a whole tree consisting of the class and (recursively) all its arguments. Since == in Scala always delegates to equals, this means that elements of case classes are always compared structurally:

```
scala> println(op)
BinOp(+,Number(1.0),Var(x))

scala> op.right == Var("x")
res3: Boolean = true
```

Finally, the compiler adds a copy method to your class for making modified copies. This method is useful for making a new instance of the class that is the same as another one except that one or two attributes are different. The method works by using named and default parameters (see Section 8.8). You specify the changes you'd like to make by using named parameters. For any parameter you don't specify, the value from the old object is used. As an example, here is how you can make an operation just like op except that the operator has changed:

```
scala> op.copy(operator = "-")
res4: BinOp = BinOp(-,Number(1.0),Var(x))
```

All these conventions add a lot of convenience—at a small price. You have to write the case modifier, and your classes and objects become a bit larger. They are larger because additional methods are generated and an implicit field is added for each constructor parameter. However, the biggest advantage of case classes is that they support pattern matching.

Pattern matching

Say you want to simplify arithmetic expressions of the kinds just presented. There is a multitude of possible simplification rules. The following three rules just serve as an illustration:

```
UnOp("-", UnOp("-", null))  => null   // Double negation
BinOp("+", null, Number(0)) => null   // Adding zero
BinOp("*", null, Number(1)) => null   // Multiplying by one
```

Using pattern matching, these rules can be taken almost as they are to form the core of a simplification function in Scala, as shown in Listing 15.2. The function, simplifyTop, can be used like this:

```
scala> simplifyTop(UnOp("-", UnOp("-", Var("x"))))
res4: Expr = Var(x)
```

```
  def simplifyTop(expr: Expr): Expr = expr match {
    case UnOp("-", UnOp("-", e))  => e   // Double negation
    case BinOp("+", e, Number(0)) => e   // Adding zero
    case BinOp("*", e, Number(1)) => e   // Multiplying by one
    case _ => expr
  }
```

Listing 15.2 - The simplifyTop function, which does a pattern match.

The right-hand side of simplifyTop consists of a match expression. match corresponds to switch in Java, but it's written after the selector expression. In other words, it's:

```
selector match { alternatives }
```

instead of:

```
switch (selector) { alternatives }
```

A pattern match includes a sequence of *alternatives*, each starting with the keyword case. Each alternative includes a *pattern* and one or more expressions, which will be evaluated if the pattern matches. An arrow symbol => separates the pattern from the expressions.

A match expression is evaluated by trying each of the patterns in the order they are written. The first pattern that matches is selected, and the part following the arrow is selected and executed.

A *constant pattern* like "+" or 1 matches values that are equal to the constant with respect to==.
A *variable pattern* like e matches every value. The variable then refers to that value in the right hand side of the case clause. In this example, note that the first three alternatives evaluate to e, a variable that is bound within the associated pattern. The *wildcard pattern* (_) also matches every value, but it does not introduce a variable name to refer to that value. InListing 15.2, notice how the match ends with a default case that does nothing to the expression. Instead, it just results in expr, the expression matched upon.

A *constructor pattern* looks like UnOp("-", e). This pattern matches all values of type UnOpwhose first argument matches "-" and whose second argument matches e. Note that the arguments to the

constructor are themselves patterns. This allows you to write deep patterns using a concise notation. Here's an example:

```
UnOp("-", UnOp("-", e))
```

Imagine trying to implement this same functionality using the visitor design pattern![2]Almost as awkward, imagine implementing it as a long sequence of if statements, type tests, and type casts.

match compared to switch

Match expressions can be seen as a generalization of Java-style switches. A Java-style switchcan be naturally expressed as a match expression, where each pattern is a constant and the last pattern may be a wildcard (which represents the default case of the switch).

However, there are three differences to keep in mind: First, match is an expression in Scala (*i.e.*, it always results in a value). Second, Scala's alternative expressions never "fall through" into the next case. Third, if none of the patterns match, an exception named MatchError is thrown. This means you always have to make sure that all cases are covered, even if it means adding a default case where there's nothing to do.

```
expr match {
  case BinOp(op, left, right) =>
    println(expr + " is a binary operation")
  case _ =>
}
```

Listing 15.3 - A pattern match with an empty "default" case.

Listing 15.3 shows an example. The second case is necessary because without it, the matchexpression would throw a MatchError for every expr argument that is not a BinOp. In this example, no code is specified for that second case, so if that case runs it does nothing. The result of either case is the unit value `()', which is also the result of the entire matchexpression.

15.2 KINDS OF PATTERNS

The previous example showed several kinds of patterns in quick succession. Now take a minute to look at each pattern in detail.

The syntax of patterns is easy, so do not worry about that too much. All patterns look exactly like the corresponding expression. For instance, given the hierarchy of Listing 15.1, the pattern Var(x) matches any variable expression, binding x to the name of the variable. Used as an expression, Var(x)—exactly the same syntax—recreates an equivalent object, assuming x is already bound to the variable's name. Since the syntax of patterns is so transparent, the main thing to pay attention to is just what kinds of patterns are possible.

Wildcard patterns

The wildcard pattern (_) matches any object whatsoever. You have already seen it used as a default, catch-all alternative, like this:

```
expr match {
  case BinOp(op, left, right) =>
    println(expr + " is a binary operation")
  case _ => // handle the default case
}
```

Wildcards can also be used to ignore parts of an object that you do not care about. For example, the previous example does not actually care what the elements of a binary operation are; it just checks whether or not it is a binary operation. Thus, the code can just as well use the wildcard pattern for the elements of the BinOp, as shown in Listing 15.4.

```
expr match {
  case BinOp(_, _, _) => println(expr + " is a binary operation")
  case _ => println("It's something else")
}
```

Listing 15.4 - A pattern match with wildcard patterns.

Constant patterns

A constant pattern matches only itself. Any literal may be used as a constant. For example, 5,true, and "hello" are all constant patterns. Also, any val or singleton object can be used as a constant. For example, Nil, a singleton object, is a pattern that matches only the empty list.Listing 15.5 shows some examples of constant patterns:

```
  def describe(x: Any) = x match {
    case 5 => "five"
    case true => "truth"
    case "hello" => "hi!"
    case Nil => "the empty list"
    case _ => "something else"
  }
```

Listing 15.5 - A pattern match with constant patterns.

Here is how the pattern match shown in Listing 15.5 looks in action:

```
scala> describe(5)
res6: String = five

scala> describe(true)
res7: String = truth

scala> describe("hello")
res8: String = hi!

scala> describe(Nil)
res9: String = the empty list

scala> describe(List(1,2,3))
res10: String = something else
```

Variable patterns

A variable pattern matches any object, just like a wildcard. But unlike a wildcard, Scala binds the variable to whatever the object is. You can then use this variable to act on the object further. For example, Listing 15.6 shows a pattern match that has a special case for zero, and a default case for all other values. The default case uses a variable pattern so that it has a name for the value, no matter what it is.

```
expr match {
  case 0 => "zero"
  case somethingElse => "not zero: " + somethingElse
}
```

Listing 15.6 - A pattern match with a variable pattern.

Variable or constant?

Constant patterns can have symbolic names. You saw this already when we used Nil as a pattern. Here is a related example, where a pattern match involves the constants E(2.71828...) and Pi (3.14159...):

```
scala> import math.{E, Pi}
import math.{E, Pi}

scala> E match {
         case Pi => "strange math? Pi = " + Pi
         case _ => "OK"
       }
res11: String = OK
```

As expected, E does not match Pi, so the "strange math" case is not used.

How does the Scala compiler know that Pi is a constant imported from scala.math, and not a variable that stands for the selector value itself? Scala uses a simple lexical rule for disambiguation: a simple name starting with a lowercase letter is taken to be a pattern variable; all other references are taken to be constants. To see the difference, create a lowercase alias for pi and try with that:

```
scala> val pi = math.Pi
pi: Double = 3.141592653589793

scala> E match {
         case pi => "strange math? Pi = " + pi
       }
res12: String = strange math? Pi = 2.718281828459045
```

Here the compiler will not even let you add a default case at all. Since pi is a variable pattern, it will match all inputs, and so no cases following it can be reached:

```
scala> E match {
         case pi => "strange math? Pi = " + pi
         case _ => "OK"
       }
<console>:12: warning: unreachable code
                case _ => "OK"
                          ^
```

You can still use a lowercase name for a pattern constant, if you need to, by using one of two tricks. First, if the constant is a field of some object, you can prefix it with a qualifier. For instance, pi is a variable pattern, but this.pi or obj.pi are constants even though they start with lowercase letters. If that does not work (because pi is a local variable, say), you can alternatively enclose the variable name in back ticks. For instance, `pi` would again be interpreted as a constant, not as a variable:

```
scala> E match {
         case `pi` => "strange math? Pi = " + pi
         case _ => "OK"
       }
res14: String = OK
```

As you can see, the back-tick syntax for identifiers is used for two different purposes in Scala to help you code your way out of unusual circumstances. Here you see that it can be used to treat a lowercase identifier as a constant in a pattern match. Earlier on, in Section 6.10, you saw that it can also be used to treat a keyword as an ordinary identifier, *e.g.*, writingThread.`yield`() treats yield as an identifier rather than a keyword.

Constructor patterns

Constructors are where pattern matching becomes really powerful. A constructor pattern looks like "BinOp("+", e, Number(0))". It consists of a name (BinOp) and then a number of patterns within parentheses: "+", e, and Number(0). Assuming the name designates a case class, such a pattern means to first check that the object is a member of the named case class, and then to check that the constructor parameters of the object match the extra patterns supplied.

These extra patterns mean that Scala patterns support *deep matches*. Such patterns not only check the top-level object supplied, but also the contents of the object against further patterns. Since the extra patterns can themselves be constructor patterns, you can use them to check arbitrarily deep into an object. For example, the pattern shown in Listing 15.7 checks that the top-level object is a BinOp, that its third constructor parameter is a Number, and that the value field of that number is 0. This pattern is one line long yet checks three levels deep.

```
expr match {
  case BinOp("+", e, Number(0)) => println("a deep match")
  case _ =>
}
```

Listing 15.7 - A pattern match with a constructor pattern.

Sequence patterns

You can match against sequence types, like List or Array, just like you match against case classes. Use the same syntax, but now you can specify any number of elements within the pattern. Listing 15.8 shows a pattern that checks for a three-element list starting with zero.

```
expr match {
  case List(0, _, _) => println("found it")
  case _ =>
```

```
}
```

Listing 15.8 - A sequence pattern with a fixed length.

If you want to match against a sequence without specifying how long it can be, you can specify _* as the last element of the pattern. This funny-looking pattern matches any number of elements within a sequence, including zero elements. Listing 15.9 shows an example that matches any list that starts with zero, regardless of how long the list is.

```
expr match {
  case List(0, _*) => println("found it")
  case _ =>
}
```

Listing 15.9 - A sequence pattern with an arbitrary length.

Tuple patterns

You can match against tuples too. A pattern like (a, b, c) matches an arbitrary 3-tuple. An example is shown in Listing 15.10.

```
def tupleDemo(expr: Any) =
  expr match {
    case (a, b, c)  =>  println("matched " + a + b + c)
    case _ =>
  }
```

Listing 15.10 - A pattern match with a tuple pattern.

If you load the tupleDemo method shown in Listing 15.10 into the interpreter, and pass to it a tuple with three elements, you'll see:

```
scala> tupleDemo(("a ", 3, "-tuple"))
matched a 3-tuple
```

Typed patterns

You can use a *typed pattern* as a convenient replacement for type tests and type casts. Listing 15.11 shows an example.

```
def generalSize(x: Any) = x match {
  case s: String => s.length
  case m: Map[_, _] => m.size
  case _ => -1
}
```

Listing 15.11 - A pattern match with typed patterns.

Here are a few examples of using generalSize in the Scala interpreter:

```
scala> generalSize("abc")
res16: Int = 3

scala> generalSize(Map(1 -> 'a', 2 -> 'b'))
res17: Int = 2
```

```
scala> generalSize(math.Pi)
res18: Int = -1
```

The generalSize method returns the size or length of objects of various types. Its argument is of type Any, so it could be any value. If the argument is a String, the method returns the string's length. The pattern "s: String" is a typed pattern; it matches every (non-null) instance ofString. The pattern variable s then refers to that string.

Note that even though s and x refer to the same value, the type of x is Any, while the type of s isString. So you can write s.length in the alternative expression that corresponds to the pattern, but you could not write x.length, because the type Any does not have a length member. An equivalent but more long-winded way that achieves the effect of a match against a typed pattern employs a type test followed by a type cast. Scala uses a different syntax than Java for these. To test whether an expression expr has type String, say, you write:

```
expr.isInstanceOf[String]
```

To cast the same expression to type String, you use:

```
expr.asInstanceOf[String]
```

Using a type test and cast, you could rewrite the first case of the previous match expression as shown in Listing 15.12.

```
if (x.isInstanceOf[String]) {
  val s = x.asInstanceOf[String]
  s.length
} else ...
```

Listing 15.12 - Using isInstanceOf and asInstanceOf (poor style).

The operators isInstanceOf and asInstanceOf are treated as predefined methods of class Any that take a type parameter in square brackets. In fact, x.asInstanceOf[String] is a special case of a method invocation with an explicit type parameter String.

As you will have noted by now, writing type tests and casts is rather verbose in Scala. That's intentional because it is not encouraged practice. You are usually better off using a pattern match with a typed pattern. That's particularly true if you need to do both a type test and a type cast, because both operations are then rolled into a single pattern match.

The second case of the match expression in Listing 15.11 contains the typed pattern "m: Map[_, _]". This pattern matches any value that is a Map of some arbitrary key and value types, and lets m refer to that value. Therefore, m.size is well typed and returns the size of the map. The underscores in the type pattern[3] are like wildcards in other patterns. You could have also used (lowercase) type variables instead.

Type erasure

Can you also test for a map with specific element types? This would be handy, say, for testing whether a given value is a map from type Int to type Int. Let's try:

```
scala> def isIntIntMap(x: Any) = x match {
         case m: Map[Int, Int] => true
         case _ => false
       }
<console>:9: warning: non-variable type argument Int in type
pattern scala.collection.immutable.Map[Int,Int] (the
underlying of Map[Int,Int]) is unchecked since it is
eliminated by erasure
         case m: Map[Int, Int] => true
                 ^
```

Scala uses the erasure model of generics, just like Java does. This means that no information about type arguments is maintained at runtime. Consequently, there is no way to determine at runtime whether a given Map object has been created with two Int arguments, rather than with arguments of different types. All the system can do is determine that a value is a Map of some arbitrary type parameters. You can verify this behavior by applying isIntIntMap to arguments of different instances of class Map:

```
scala> isIntIntMap(Map(1 -> 1))
res19: Boolean = true

scala> isIntIntMap(Map("abc" -> "abc"))
res20: Boolean = true
```

The first application returns true, which looks correct, but the second application also returnstrue, which might be a surprise. To alert you to the possibly non-intuitive runtime behavior, the compiler emits unchecked warnings like the one shown previously.

The only exception to the erasure rule is arrays, because they are handled specially in Java as well as in Scala. The element type of an array is stored with the array value, so you can pattern match on it. Here's an example:

```
scala> def isStringArray(x: Any) = x match {
         case a: Array[String] => "yes"
         case _ => "no"
       }
isStringArray: (x: Any)String

scala> val as = Array("abc")
as: Array[String] = Array(abc)

scala> isStringArray(as)
res21: String = yes

scala> val ai = Array(1, 2, 3)
ai: Array[Int] = Array(1, 2, 3)

scala> isStringArray(ai)
res22: String = no
```

Variable binding

In addition to the standalone variable patterns, you can also add a variable to any other pattern. You simply write the variable name, an at sign (@), and then the pattern. This gives you a variable-binding pattern, which means the pattern is to perform the pattern match as normal, and if the pattern succeeds, set the variable to the matched object just as with a simple variable pattern.

As an example, Listing 15.13 shows a pattern match that looks for the absolute value operation being applied twice in a row. Such an expression can be simplified to only take the absolute value one time.

```
expr match {
  case UnOp("abs", e @ UnOp("abs", _)) => e
  case _ =>
}
```

Listing 15.13 - A pattern with a variable binding (via the @ sign).

The example shown in Listing 15.13 includes a variable-binding pattern with e as the variable and UnOp("abs", _) as the pattern. If the entire pattern match succeeds, then the portion that matched the UnOp("abs", _) part is made available as variable e. The result of the case is just e, because e has the same value as expr but with one less absolute value operation.

15.3 PATTERN GUARDS

Sometimes, syntactic pattern matching is not precise enough. For instance, say you are given the task of formulating a simplification rule that replaces sum expressions with two identical operands, such as e + e, by multiplications of two (*e.g.*, e * 2). In the language of Expr trees, an expression like:

```
BinOp("+", Var("x"), Var("x"))
```

would be transformed by this rule to:

```
BinOp("*", Var("x"), Number(2))
```

You might try to define this rule as follows:

```
scala> def simplifyAdd(e: Expr) = e match {
         case BinOp("+", x, x) => BinOp("*", x, Number(2))
         case _ => e
       }
<console>:14: error: x is already defined as value x
         case BinOp("+", x, x) => BinOp("*", x, Number(2))
                          ^
```

This fails because Scala restricts patterns to be linear: a pattern variable may only appear once in a pattern. However, you can re-formulate the match with a *pattern guard*, as shown in Listing 15.14:

```
scala> def simplifyAdd(e: Expr) = e match {
         case BinOp("+", x, y) if x == y =>
           BinOp("*", x, Number(2))
         case _ => e
       }
simplifyAdd: (e: Expr)Expr
```

Listing 15.14 - A match expression with a pattern guard.

A pattern guard comes after a pattern and starts with an if. The guard can be an arbitrary boolean expression, which typically refers to variables in the pattern. If a pattern guard is present, the match succeeds only if the guard evaluates to true. Hence, the first case above would only match binary operations with two equal operands.

Some other examples of guarded patterns are:

```
// match only positive integers
case n: Int if 0 < n => ...

// match only strings starting with the letter `a'
case s: String if s(0) == 'a' => ...
```

15.4 PATTERN OVERLAPS

Patterns are tried in the order in which they are written. The version of simplify shown inListing 15.15 presents an example where the order of the cases matters.

```
def simplifyAll(expr: Expr): Expr = expr match {
  case UnOp("-", UnOp("-", e)) =>
    simplifyAll(e)   // `-' is its own inverse
  case BinOp("+", e, Number(0)) =>
    simplifyAll(e)   // `0' is a neutral element for `+'
  case BinOp("*", e, Number(1)) =>
    simplifyAll(e)   // `1' is a neutral element for `*'
  case UnOp(op, e) =>
    UnOp(op, simplifyAll(e))
  case BinOp(op, l, r) =>
    BinOp(op, simplifyAll(l), simplifyAll(r))
  case _ => expr
}
```

Listing 15.15 - Match expression in which case order matters.

The version of simplify shown in Listing 15.15 will apply simplification rules everywhere in an expression, not just at the top, as simplifyTop did. It can be derived from simplifyTop by adding two more cases for general unary and binary expressions (cases four and five in Listing 15.15).

The fourth case has the pattern UnOp(op, e); *i.e.*, it matches every unary operation. The operator and operand of the unary operation can be arbitrary. They are bound to the pattern variables op and e, respectively. The alternative in this case applies simplifyAll recursively to the operand e and then rebuilds the same unary operation with the (possibly) simplified operand. The fifth case for BinOp is analogous: it is a "catch-all" case for arbitrary binary operations, which recursively applies the simplification method to its two operands.

In this example, it is important that the catch-all cases come after the more specific simplification rules. If you wrote them in the other order, then the catch-all case would be run in favor of the more specific rules. In many cases, the compiler will even complain if you try. For example, here's a match expression that won't compile because the first case will match anything that would be matched by the second case:

```
scala> def simplifyBad(expr: Expr): Expr = expr match {
         case UnOp(op, e) => UnOp(op, simplifyBad(e))
         case UnOp("-", UnOp("-", e)) => e
       }
<console>:21: warning: unreachable code
         case UnOp("-", UnOp("-", e)) => e
                                         ^
```

15.5 SEALED CLASSES

Whenever you write a pattern match, you need to make sure you have covered all of the possible cases. Sometimes you can do this by adding a default case at the end of the match, but that only applies if there is a sensible default behavior. What do you do if there is no default? How can you ever feel safe that you covered all the cases?

You can enlist the help of the Scala compiler in detecting missing combinations of patterns in a match expression. To do this, the compiler needs to be able to tell which are the possible cases. In general, this is impossible in Scala because new case classes can be defined at any time and in arbitrary compilation units. For instance, nothing would prevent you from adding a fifth case class to the Expr class hierarchy in a different compilation unit from the one where the other four cases are defined.

The alternative is to make the superclass of your case classes sealed. A sealed class cannot have any new subclasses added except the ones in the same file. This is very useful for pattern matching because it means you only need to worry about the subclasses you already know about. What's more, you get better compiler support as well. If you match against case classes that inherit from a sealed class, the compiler will flag missing combinations of patterns with a warning message.

If you write a hierarchy of classes intended to be pattern matched, you should consider sealing them. Simply put the sealed keyword in front of the class at the top of the hierarchy. Programmers using your class hierarchy will then feel confident in pattern matching against it. The sealed keyword, therefore, is often a license to pattern match. Listing 15.16 shows an example in which Expr is turned into a sealed class.

```
sealed abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
    left: Expr, right: Expr) extends Expr
```

Listing 15.16 - A sealed hierarchy of case classes.

Now define a pattern match where some of the possible cases are left out:

```
def describe(e: Expr): String = e match {
  case Number(_) => "a number"
  case Var(_)    => "a variable"
}
```

You will get a compiler warning like the following:

```
warning: match is not exhaustive!
missing combination           UnOp
missing combination          BinOp
```

Such a warning tells you that there's a risk your code might produce a MatchError exception because some possible patterns (UnOp, BinOp) are not handled. The warning points to a potential source of runtime faults, so it is usually a welcome help in getting your program right.

However, at times you might encounter a situation where the compiler is too picky in emitting the warning. For instance, you might know from the context that you will only ever apply the describe method above to expressions that are either Numbers or Vars, so you know that no MatchError will be produced. To make the warning go away, you could add a third catch-all case to the method, like this:

```
def describe(e: Expr): String = e match {
  case Number(_) => "a number"
  case Var(_) => "a variable"
  case _ => throw new RuntimeException // Should not happen
}
```

That works, but it is not ideal. You will probably not be very happy that you were forced to add code that will never be executed (or so you think), just to make the compiler shut up.

A more lightweight alternative is to add an @unchecked annotation to the selector expression of the match. This is done as follows:

```
def describe(e: Expr): String = (e: @unchecked) match {
  case Number(_) => "a number"
  case Var(_)    => "a variable"
}
```

Annotations are described in Chapter 27. In general, you can add an annotation to an expression in the same way you add a type: follow the expression with a colon and the name of the annotation (preceded by an at sign). For example, in this case you add an @uncheckedannotation to the variable e, with "e: @unchecked". The @unchecked annotation has a special meaning for pattern matching. If a match's selector expression carries this annotation, exhaustivity checking for the patterns that follow will be suppressed.

15.6 THE OPTION TYPE

Scala has a standard type named Option for optional values. Such a value can be of two forms:Some(x), where x is the actual value, or the None object, which represents a missing value.

Optional values are produced by some of the standard operations on Scala's collections. For instance, the get method of Scala's Map produces Some(value) if a value corresponding to a given key has been found, or None if the given key is not defined in the Map. Here's an example:

```
scala> val capitals =
```

```
         Map("France" -> "Paris", "Japan" -> "Tokyo")
capitals: scala.collection.immutable.Map[String,String] =
Map(France -> Paris, Japan -> Tokyo)

scala> capitals get "France"
res23: Option[String] = Some(Paris)

scala> capitals get "North Pole"
res24: Option[String] = None
```

The most common way to take optional values apart is through a pattern match. For instance:

```
scala> def show(x: Option[String]) = x match {
         case Some(s) => s
         case None => "?"
       }
show: (x: Option[String])String

scala> show(capitals get "Japan")
res25: String = Tokyo

scala> show(capitals get "France")
res26: String = Paris

scala> show(capitals get "North Pole")
res27: String = ?
```

The Option type is used frequently in Scala programs. Compare this to the dominant idiom in Java of using null to indicate no value. For example, the get method of java.util.HashMapreturns either a value stored in the HashMap or null if no value was found. This approach works for Java but is error prone because it is difficult in practice to keep track of which variables in a program are allowed to be null.

If a variable is allowed to be null, then you must remember to check it for null every time you use it. When you forget to check, you open the possibility that a NullPointerException may result at runtime. Because such exceptions may not happen very often, it can be difficult to discover the bug during testing. For Scala, the approach would not work at all because it is possible to store value types in hash maps, and null is not a legal element for a value type. For instance, aHashMap[Int, Int] cannot return null to signify "no element."

By contrast, Scala encourages the use of Option to indicate an optional value. This approach to optional values has several advantages over Java's. First, it is far more obvious to readers of code that a variable whose type is Option[String] is an optional String than a variable of typeString, which may sometimes be null. But most importantly, that programming error described earlier of using a variable that may be null without first checking it for null becomes a type error in Scala. If a variable is of type Option[String] and you try to use it as a String, your Scala program will not compile.

15.7 PATTERNS EVERYWHERE

Patterns are allowed in many parts of Scala, not just in standalone match expressions. Take a look at some other places you can use patterns.

Patterns in variable definitions

Anytime you define a val or a var, you can use a pattern instead of a simple identifier. For example, you can take apart a tuple and assign each of its parts to its own variable, as shown in Listing 15.17:

```
scala> val myTuple = (123, "abc")
myTuple: (Int, String) = (123,abc)

scala> val (number, string) = myTuple
number: Int = 123
string: String = abc
```

Listing 15.17 - Defining multiple variables with one assignment.

This construct is quite useful when working with case classes. If you know the precise case class you are working with, then you can deconstruct it with a pattern. Here's an example:

```
scala> val exp = new BinOp("*", Number(5), Number(1))
exp: BinOp = BinOp(*,Number(5.0),Number(1.0))

scala> val BinOp(op, left, right) = exp
op: String = *
left: Expr = Number(5.0)
right: Expr = Number(1.0)
```

Case sequences as partial functions

A sequence of cases (*i.e.*, alternatives) in curly braces can be used anywhere a function literal can be used. Essentially, a case sequence *is* a function literal, only more general. Instead of having a single entry point and list of parameters, a case sequence has multiple entry points, each with their own list of parameters. Each case is an entry point to the function, and the parameters are specified with the pattern. The body of each entry point is the right-hand side of the case.

Here is a simple example:

```
val withDefault: Option[Int] => Int = {
  case Some(x) => x
  case None => 0
}
```

The body of this function has two cases. The first case matches a Some, and returns the number inside the Some. The second case matches a None, and returns a default value of zero. Here is this function in use:

```
scala> withDefault(Some(10))
res28: Int = 10

scala> withDefault(None)
res29: Int = 0
```

This facility is quite useful for the Akka actors library, because it allows its receive method to be defined as a series of cases:

```
var sum = 0
```

```
def receive = {

  case Data(byte) =>
    sum += byte

  case GetChecksum(requester) =>
    val checksum = ~(sum & 0xFF) + 1
    requester ! checksum
}
```

One other generalization is worth noting: a sequence of cases gives you a *partial* function. If you apply such a function on a value it does not support, it will generate a run-time exception. For example, here is a partial function that returns the second element of a list of integers:

```
val second: List[Int] => Int = {
  case x :: y :: _ => y
}
```

When you compile this, the compiler will correctly warn that the match is not exhaustive:

```
<console>:17: warning: match is not exhaustive!
missing combination            Nil
```

This function will succeed if you pass it a three-element list, but not if you pass it an empty list:

```
scala> second(List(5, 6, 7))
res24: Int = 6

scala> second(List())
scala.MatchError: List()
      at $anonfun$1.apply(<console>:17)
      at $anonfun$1.apply(<console>:17)
```

If you want to check whether a partial function is defined, you must first tell the compiler that you know you are working with partial functions. The type List[Int] => Int includes all functions from lists of integers to integers, whether or not the functions are partial. The type that only includes *partial* functions from lists of integers to integers is writtenPartialFunction[List[Int],Int]. Here is the second function again, this time written with a partial function type:

```
val second: PartialFunction[List[Int],Int] = {
  case x :: y :: _ => y
}
```

Partial functions have a method isDefinedAt, which can be used to test whether the function is defined at a particular value. In this case, the function is defined for any list that has at least two elements:

```
scala> second.isDefinedAt(List(5,6,7))
res30: Boolean = true

scala> second.isDefinedAt(List())
res31: Boolean = false
```

The typical example of a partial function is a pattern matching function literal like the one in the previous example. In fact, such an expression gets translated by the Scala compiler to a partial function by translating the patterns twice—once for the implementation of the real function, and once to test whether the function is defined or not.

For instance, the function literal { case x :: y :: _ => y } gets translated to the following partial function value:

```
new PartialFunction[List[Int], Int] {
  def apply(xs: List[Int]) = xs match {
    case x :: y :: _ => y
  }
  def isDefinedAt(xs: List[Int]) = xs match {
    case x :: y :: _ => true
    case _ => false
  }
}
```

This translation takes effect whenever the declared type of a function literal is PartialFunction. If the declared type is just Function1, or is missing, the function literal is instead translated to acomplete function.

In general, you should try to work with complete functions whenever possible, because using partial functions allows for runtime errors that the compiler cannot help you with. Sometimes partial functions are really helpful though. You might be sure that an unhandled value will never be supplied. Alternatively, you might be using a framework that expects partial functions and so will always check isDefinedAt before calling the function. An example of the latter is the react example given above, where the argument is a partially defined function, defined precisely for those messages that the caller wants to handle.

Patterns in for expressions

You can also use a pattern in a for expression, as shown in Listing 15.18. This for expression retrieves all key/value pairs from the capitals map. Each pair is matched against the pattern(country, city), which defines the two variables country and city.

```
scala> for ((country, city) <- capitals)
         println("The capital of " + country + " is " + city)
The capital of France is Paris
The capital of Japan is Tokyo
```

Listing 15.18 - A for expression with a tuple pattern.

The pair pattern shown in Listing 15.18 was special because the match against it can never fail. Indeed, capitals yields a sequence of pairs, so you can be sure that every generated pair can be matched against a pair pattern. But it is equally possible that a pattern might not match a generated value. Listing 15.19 shows an example where that is the case.

```
scala> val results = List(Some("apple"), None,
           Some("orange"))
```

```
results: List[Option[String]] = List(Some(apple), None,
Some(orange))

scala> for (Some(fruit) <- results) println(fruit)
apple
orange
```

Listing 15.19 - Picking elements of a list that match a pattern.

As you can see from this example, generated values that do not match the pattern are discarded. For instance, the second element None in the results list does not match the patternSome(fruit); therefore it does not show up in the output.

15.8 A LARGER EXAMPLE

After having learned the different forms of patterns, you might be interested in seeing them applied in a larger example. The proposed task is to write an expression formatter class that displays an arithmetic expression in a two-dimensional layout. Divisions such as "x / (x + 1)" should be printed vertically, by placing the numerator on top of the denominator, like this:

```
 x
-----
x + 1
```

As another example, here's the expression ((a / (b * c) + 1 / n) / 3) in two dimensional layout:

```
 a    1
----- + -
b * c   n
---------
   3
```

From these examples it looks like the class (we'll call it ExprFormatter) will have to do a fair bit of layout juggling, so it makes sense to use the layout library developed in Chapter 10. We'll also use the Expr family of case classes you saw previously in this chapter, and place bothChapter 10's layout library and this chapter's expression formatter into named packages. The full code for the example will be shown in Listings 15.20 and 15.21.

A useful first step is to concentrate on horizontal layout. A structured expression like:

```
BinOp("+",
      BinOp("*",
            BinOp("+", Var("x"), Var("y")),
            Var("z")),
      Number(1))
```

should print (x + y) * z + 1. Note that parentheses are mandatory around x + y, but would be optional around (x + y) * z. To keep the layout as legible as possible, your goal should be to omit parentheses wherever they are redundant, while ensuring that all necessary parentheses are present.

To know where to put parentheses, the code needs to know about the relative precedence of each operator, so it's a good idea to tackle this first. You could express the relative precedence directly as a map literal of the following form:

```
Map(
  "|" -> 0, "||" -> 0,
  "&" -> 1, "&&" -> 1, ...
)
```

However, this would involve some amount of pre-computation of precedences on your part. A more convenient approach is to just define groups of operators of increasing precedence and then calculate the precedence of each operator from that. Listing 15.20 shows the code.

```
package org.stairwaybook.expr
import org.stairwaybook.layout.Element.elem

sealed abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
    left: Expr, right: Expr) extends Expr

class ExprFormatter {

  // Contains operators in groups of increasing precedence
  private val opGroups =
    Array(
      Set("|", "||"),
      Set("&", "&&"),
      Set("^"),
      Set("==", "!="),
      Set("<", "<=", ">", ">="),
      Set("+", "-"),
      Set("*", "%")
    )

  // A mapping from operators to their precedence
  private val precedence = {
    val assocs =
      for {
        i <- 0 until opGroups.length
        op <- opGroups(i)
      } yield op -> i
    assocs.toMap
  }

  private val unaryPrecedence = opGroups.length
  private val fractionPrecedence = -1

  // continued in Listing 15.21...
```

Listing 15.20 - The top half of the expression formatter.

```
// ...continued from Listing 15.20

import org.stairwaybook.layout.Element
```

```
  private def format(e: Expr, enclPrec: Int): Element =

    e match {

      case Var(name) =>
        elem(name)

      case Number(num) =>
        def stripDot(s: String) =
          if (s endsWith ".0") s.substring(0, s.length - 2)
          else s
        elem(stripDot(num.toString))

      case UnOp(op, arg) =>
        elem(op) beside format(arg, unaryPrecedence)

      case BinOp("/", left, right) =>
        val top = format(left, fractionPrecedence)
        val bot = format(right, fractionPrecedence)
        val line = elem('-', top.width max bot.width, 1)
        val frac = top above line above bot
        if (enclPrec != fractionPrecedence) frac
        else elem(" ") beside frac beside elem(" ")

      case BinOp(op, left, right) =>
        val opPrec = precedence(op)
        val l = format(left, opPrec)
        val r = format(right, opPrec + 1)
        val oper = l beside elem(" " + op + " ") beside r
        if (enclPrec <= opPrec) oper
        else elem("(") beside oper beside elem(")")
    }

    def format(e: Expr): Element = format(e, 0)
  }
```

Listing 15.21 - The bottom half of the expression formatter.

The precedence variable is a map from operators to their precedences, which are integers starting with 0. It is calculated using a for expression with two generators. The first generator produces every index i of the opGroups array. The second generator produces every operator opin opGroups(i). For each such operator the for expression yields an association from the operator op to its index i. Hence, the relative position of an operator in the array is taken to be its precedence.

Associations are written with an infix arrow, *e.g.*, op -> i. So far you have seen associations only as part of map constructions, but they are also values in their own right. In fact, the association op -> i is nothing else but the pair (op, i).

Now that you have fixed the precedence of all binary operators except /, it makes sense to generalize this concept to also cover unary operators. The precedence of a unary operator is higher than the precedence of every binary operator. Thus we can set unaryPrecedence (shown in Listing 15.20) to the length of the opGroups array, which is one more than the precedence of the * and % operators. The precedence of a fraction is treated differently from the other operators because fractions use vertical

layout. However, it will prove convenient to assign to the division operator the special precedence value -1, so we'll initialize fractionPrecedence to -1 (shown in Listing 15.20).

After these preparations, you are ready to write the main format method. This method takes two arguments: an expression e, of type Expr, and the precedence enclPrec of the operator directly enclosing the expression e. (If there's no enclosing operator, enclPrec should be zero.) The method yields a layout element that represents a two-dimensional array of characters.

Listing 15.21 shows the remainder of class ExprFormatter, which includes three methods. The first method, stripDot, is a helper method. The next method, the private format method, does most of the work to format expressions. The last method, also named format, is the lone public method in the library, which takes an expression to format. The private format method does its work by performing a pattern match on the kind of expression. The match expression has five cases. We'll discuss each case individually.

The first case is:

```
case Var(name) =>
  elem(name)
```

If the expression is a variable, the result is an element formed from the variable's name.

The second case is:

```
case Number(num) =>
  def stripDot(s: String) =
    if (s endsWith ".0") s.substring(0, s.length - 2)
    else s
  elem(stripDot(num.toString))
```

If the expression is a number, the result is an element formed from the number's value. ThestripDot function cleans up the display of a floating-point number by stripping any ".0" suffix from a string.

The third case is:

```
case UnOp(op, arg) =>
  elem(op) beside format(arg, unaryPrecedence)
```

If the expression is a unary operation UnOp(op, arg) the result is formed from the operation opand the result of formatting the argument arg with the highest-possible environment precedence.[4] This means that if arg is a binary operation (but not a fraction) it will always be displayed in parentheses.

The fourth case is:

```
case BinOp("/", left, right) =>
  val top = format(left, fractionPrecedence)
  val bot = format(right, fractionPrecedence)
  val line = elem('-', top.width max bot.width, 1)
  val frac = top above line above bot
  if (enclPrec != fractionPrecedence) frac
  else elem(" ") beside frac beside elem(" ")
```

If the expression is a fraction, an intermediate result frac is formed by placing the formatted operands left and right on top of each other, separated by an horizontal line element. The width of the horizontal line is the maximum of the widths of the formatted operands. This intermediate result is also the final result unless the fraction appears itself as an argument of another fraction. In the latter case, a space is added on each side of frac. To see the reason why, consider the expression "(a / b) / c".

Without the widening correction, formatting this expression would give:

```
a
-
b
-
c
```

The problem with this layout is evident—it's not clear where the top-level fractional bar is. The expression above could mean either "(a / b) / c" or "a / (b / c)". To disambiguate, a space should be added on each side to the layout of the nested fraction "a / b". Then the layout becomes unambiguous:

```
 a
 -
 b
---
 c
```

The fifth and last case is:

```
  case BinOp(op, left, right) =>
    val opPrec = precedence(op)
    val l = format(left, opPrec)
    val r = format(right, opPrec + 1)
    val oper = l beside elem(" " + op + " ") beside r
    if (enclPrec <= opPrec) oper
    else elem("(") beside oper beside elem(")")
```

This case applies for all other binary operations. Since it comes after the case starting with:

```
  case BinOp("/", left, right) => ...
```

you know that the operator op in the pattern BinOp(op, left, right) cannot be a division. To format such a binary operation, one needs to format first its operands left and right. The precedence parameter for formatting the left operand is the precedence opPrec of the operatorop, while for the right operand it is one more than that. This scheme ensures that parentheses also reflect the correct associativity.

For instance, the operation:

```
  BinOp("-", Var("a"), BinOp("-", Var("b"), Var("c")))
```

would be correctly parenthesized as "a - (b - c)". The intermediate result oper is then formed by placing the formatted left and right operands side-by-side, separated by the operator. If the precedence of the

current operator is smaller than the precedence of the enclosing operator, oper is placed between parentheses; otherwise, it is returned as is.

```
import org.stairwaybook.expr._

object Express extends App {

  val f = new ExprFormatter

  val e1 = BinOp("*", BinOp("/", Number(1), Number(2)),
                      BinOp("+", Var("x"), Number(1)))

  val e2 = BinOp("+", BinOp("/", Var("x"), Number(2)),
                      BinOp("/", Number(1.5), Var("x")))

  val e3 = BinOp("/", e1, e2)

  def show(e: Expr) = println(f.format(e)+ "\n\n")

  for (e <- Array(e1, e2, e3)) show(e)
}
```

Listing 15.22 - An application that prints formatted expressions.

This finishes the design of the private format function. The only remaining method is the public format method, which allows client programmers to format a top-level expression without passing a precedence argument. Listing 15.22 shows a demo program that exercisesExprFormatter.

Note that, even though this program does not define a main method, it is still a runnable application because it inherits from the App trait. You can run the Express program with the command:

```
scala Express
```

This will give the following output:

```
1
- * (x + 1)
2


x  1.5
- + ---
2   x


1
- * (x + 1)
2
----------
 x  1.5
```

```
- + ---
2    x
```

15.9 CONCLUSION

In this chapter, you learned about Scala's case classes and pattern matching in detail. By using them, you can take advantage of several concise idioms not normally available in object-oriented languages. However, Scala's pattern matching goes further than this chapter describes. If you want to use pattern matching on one of your classes, but you do not want to open access to your classes the way case classes do, you can use the *extractors* described inChapter 26. In the next chapter, we'll turn our attention to lists.

Footnotes for Chapter 15:

[1] Instead of an abstract class, we could have also chosen to model the root of that class hierarchy as a trait. Modeling it as an abstract class may be slightly more efficient.

[2] Gamma, *et. al.*, *Design Patterns* [Gam95]

[3] In the typed pattern, m: Map[_, _], the "Map[_, _]" portion is called a type pattern.

[4] The value of unaryPrecedence is the highest possible precedence, because it was initialized to one more than the precedence of the * and % operators.