Chapter 6

Functional Objects

With the understanding of Scala basics you've gained from previous chapters, you're ready to design more full-featured classes in Scala. In this chapter, the emphasis is on classes that define functional objects, or objects that do not have any mutable state. As a running example, we'll create several variants of a class that models rational numbers as immutable objects. Along the way, we'll show you more aspects of object-oriented programming in Scala: class parameters and constructors, methods and operators, private members, overriding, checking preconditions, overloading, and self references.

6.1 A SPECIFICATION FOR CLASS RATIONAL

A *rational number* is a number that can be expressed as a ratio n/d, where *n* and *d* are integers, except that *d* cannot be zero. *n* is called the numerator and *d* the denominator. Examples of rational numbers are 1/2, 2/3, 112/239, and 2/1. Compared to floating-point numbers, rational numbers have the advantage that fractions are represented exactly, without rounding or approximation.

The class we'll design in this chapter must model the behavior of rational numbers, including allowing them to be added, subtracted, multiplied, and divided. To add two rationals, you must first obtain a common denominator, then add the two numerators. For example, to add1/2 + 2/3, you multiply both parts of the left operand by 3 and both parts of the right operand by 2, which gives you 3/6 + 4/6. Adding the two numerators yields the result, 7/6. To multiply two rational numbers, you can simply multiply their numerators and multiply their denominators. Thus, 1/2 * 2/5 gives 2/10, which can be represented more compactly in its "normalized" form as 1/5. You divide by swapping the numerator and denominator of the right operand and then multiplying. For instance 1/2 / 3/5 is the same as 1/2 * 5/3, or 5/6.

One, maybe rather trivial, observation is that in mathematics, rational numbers do not have mutable state. You can add one rational number to another, but the result will be a new rational number. The original numbers will not have "changed." The immutable Rational class we'll design in this chapter will have the same property. Each rational number will be represented by one Rational object. When you add two Rational objects, you'll create a newRational object to hold the sum.

This chapter will give you a glimpse of some of the ways Scala enables you to write libraries that feel like native language support. For example, at the end of this chapter you'll be able to do this with class Rational:

```
scala> val oneHalf = new Rational(1, 2)
oneHalf: Rational = 1/2

scala> val twoThirds = new Rational(2, 3)
twoThirds: Rational = 2/3

scala> (oneHalf / 7) + (1 - twoThirds)
res0: Rational = 17/42
```

6.2 CONSTRUCTING A RATIONAL

A good place to start designing class Rational is to consider how client programmers will create a new Rational object. Given we've decided to make Rational objects immutable, we'll require that clients provide all data needed by an instance (in this case, a numerator and a denominator) when they construct the instance. Thus, we will start the design with this:

```
class Rational(n: Int, d: Int)
```

One of the first things to note about this line of code is that if a class doesn't have a body, you don't need to specify empty curly braces (though you could, of course, if you wanted to). The identifiers n and d in the parentheses after the class name, Rational, are called *class parameters*. The Scala compiler will gather up these two class parameters and create a*primary constructor* that takes the same two parameters.

IMMUTABLE OBJECT TRADE-OFFS

Immutable objects offer several advantages over mutable objects, and one potential disadvantage. First, immutable objects are often easier to reason about than mutable ones, because they do not have complex state spaces that change over time. Second, you can pass immutable objects around quite freely, whereas you may need to make defensive copies of mutable objects before passing them to other code. Third, there is no way for two threads concurrently accessing an immutable to corrupt its state once it has been properly constructed, because no thread can change the state of an immutable. Fourth, immutable objects make safe hash table keys. If a mutable object is mutated after it is placed into a HashSet, for example, that object may not be found the next time you look into the HashSet.

The main disadvantage of immutable objects is that they sometimes require that a large object graph be copied, whereas an update could be done in its place. In some cases this can be awkward to express and might also cause a performance bottleneck. As a result, it is not uncommon for libraries to provide mutable alternatives to immutable classes. For example, class StringBuilder is a mutable alternative to the immutable String. We'll give you more information on designing mutable objects in Scala in Chapter 18.

Note

This initial Rational example highlights a difference between Java and Scala. In Java, classes have constructors, which can take parameters; whereas in Scala, classes can take parameters directly. The Scala notation is more concise—class parameters can be used directly in the body of the class; there's no need to define fields and write assignments that copy constructor parameters into fields. This can yield substantial savings in boilerplate code, especially for small classes.

The Scala compiler will compile any code you place in the class body, which isn't part of a field or a method definition, into the primary constructor. For example, you could print a debug message like this:

```
class Rational(n: Int, d: Int) {
```

```
  println("Created " + n + "/" + d)
}
```

Given this code, the Scala compiler would place the call to println into Rational's primary constructor. The println call will, therefore, print its debug message whenever you create a new Rational instance:

```
scala> new Rational(1, 2)
Created 1/2
res0: Rational = Rational@2591e0c9
```

6.3 REIMPLEMENTING THE TOSTRING METHOD

When we created an instance of Rational in the previous example, the interpreter printed "Rational@90110a". The interpreter obtained this somewhat funny looking string by callingtoString on the Rational object. By default, class Rational inherits the implementation oftoString defined in class java.lang.Object, which just prints the class name, an @ sign, and a hexadecimal number. The result of toString is primarily intended to help programmers by providing information that can be used in debug print statements, log messages, test failure reports, and interpreter and debugger output. The result currently provided by toString is not especially helpful because it doesn't give any clue about the rational number's value. A more useful implementation of toString would print out the values of the Rational's numerator and denominator. You can *override* the default implementation by adding a method toString to class Rational, like this:

```
class Rational(n: Int, d: Int) {
  override def toString = n + "/" + d
}
```

The override modifier in front of a method definition signals that a previous method definition is overridden (more on this in Chapter 10). Since Rational numbers will display nicely now, we removed the debug println statement we put into the body of previous version of classRational. You can test the new behavior of Rational in the interpreter:

```
scala> val x = new Rational(1, 3)
x: Rational = 1/3

scala> val y = new Rational(5, 7)
y: Rational = 5/7
```

6.4 CHECKING PRECONDITIONS

As a next step, we will turn our attention to a problem with the current behavior of the primary constructor. As mentioned at the beginning of this chapter, rational numbers may not have a zero in the denominator. Currently, however, the primary constructor accepts a zero passed as d:

```
scala> new Rational(5, 0)
res1: Rational = 5/0
```

One of the benefits of object-oriented programming is that it allows you to encapsulate data inside objects so that you can ensure the data is valid throughout its lifetime. In the case of an immutable

object such as Rational, this means that you should ensure the data is valid when the object is constructed. Given that a zero denominator is an invalid state for a Rationalnumber, you should not let a Rational be constructed if a zero is passed in the d parameter.

The best way to approach this problem is to define as a *precondition* of the primary constructor that d must be non-zero. A precondition is a constraint on values passed into a method or constructor, a requirement which callers must fulfill. One way to do that is to userequire,[1] like this:

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  override def toString = n + "/" + d
}
```

The require method takes one boolean parameter. If the passed value is true, require will return normally. Otherwise, require will prevent the object from being constructed by throwing an IllegalArgumentException.

6.5 ADDING FIELDS

Now that the primary constructor is properly enforcing its precondition, we will turn our attention to supporting addition. To do so, we'll define a public add method on class Rationalthat takes another Rational as a parameter. To keep Rational immutable, the add method must not add the passed rational number to itself. Rather, it must create and return a new Rationalthat holds the sum. You might think you could write add this way:

```
class Rational(n: Int, d: Int) { // This won't compile
  require(d != 0)
  override def toString = n + "/" + d
  def add(that: Rational): Rational =
    new Rational(n * that.d + that.n * d, d * that.d)
}
```

However, given this code the compiler will complain:

```
<console>:11: error: value d is not a member of Rational
           new Rational(n * that.d + that.n * d, d * that.d)
                                ^
<console>:11: error: value d is not a member of Rational
           new Rational(n * that.d + that.n * d, d * that.d)
                                                       ^
```

Although class parameters n and d are in scope in the code of your add method, you can only access their value on the object on which add was invoked. Thus, when you say n or d in add's implementation, the compiler is happy to provide you with the values for these class parameters. But it won't let you say that.n or that.d because that does not refer to the Rationalobject on which add was invoked.[2] To access the numerator and denominator on that, you'll need to make them into fields. Listing 6.1 shows how you could add these fields to classRational.[3]

```
  class Rational(n: Int, d: Int) {
    require(d != 0)
    val numer: Int = n
```

```
  val denom: Int = d
  override def toString = numer + "/" + denom
  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
}
```

Listing 6.1 - Rational with fields.

In the version of Rational shown in Listing 6.1, we added two fields named numer and denom, and initialized them with the values of class parameters n and d.[4] We also changed the implementation of toString and add so that they use the fields, not the class parameters. This version of class Rational compiles. You can test it by adding some rational numbers:

```
scala> val oneHalf = new Rational(1, 2)
oneHalf: Rational = 1/2

scala> val twoThirds = new Rational(2, 3)
twoThirds: Rational = 2/3

scala> oneHalf add twoThirds
res2: Rational = 7/6
```

One other thing you can do now that you couldn't do before is access the numerator and denominator values from outside the object. Simply access the public numer and denom fields, like this:

```
scala> val r = new Rational(1, 2)
r: Rational = 1/2

scala> r.numer
res3: Int = 1

scala> r.denom
res4: Int = 2
```

6.6 SELF REFERENCES

The keyword this refers to the object instance on which the currently executing method was invoked, or if used in a constructor, the object instance being constructed. As an example, consider adding a method, lessThan, which tests whether the given Rational is smaller than a parameter:

```
def lessThan(that: Rational) =
  this.numer * that.denom < that.numer * this.denom
```

Here, this.numer refers to the numerator of the object on which lessThan was invoked. You can also leave off the this prefix and write just numer; the two notations are equivalent.

As an example of where you can't do without this, consider adding a max method to classRational that returns the greater of the given rational number and an argument:

```
def max(that: Rational) =
  if (this.lessThan(that)) that else this
```

Here, the first this is redundant. You could have left it off and written: lessThan(that). But the second this represents the result of the method in the case where the test returns false; were you to omit it, there would be nothing left to return!

6.7 AUXILIARY CONSTRUCTORS

Sometimes you need multiple constructors in a class. In Scala, constructors other than the primary constructor are called *auxiliary constructors*. For example, a rational number with a denominator of 1 can be written more succinctly as simply the numerator. Instead of 5/1, for example, you can just write 5. It might be nice, therefore, if instead of writingnew Rational(5, 1), client programmers could simply write new Rational(5). This would require adding an auxiliary constructor to Rational that takes only one argument, the numerator, with the denominator predefined to be 1. Listing 6.2 shows what that would look like.

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  val numer: Int = n
  val denom: Int = d

  def this(n: Int) = this(n, 1) // auxiliary constructor

  override def toString = numer + "/" + denom

  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
}
```

Listing 6.2 - Rational with an auxiliary constructor.

Auxiliary constructors in Scala start with def this(...). The body of Rational's auxiliary constructor merely invokes the primary constructor, passing along its lone argument, n, as the numerator and 1 as the denominator. You can see the auxiliary constructor in action by typing the following into the interpreter:

```
scala> val y = new Rational(3)
y: Rational = 3/1
```

In Scala, every auxiliary constructor must invoke another constructor of the same class as its first action. In other words, the first statement in every auxiliary constructor in every Scala class will have the form "this(...)". The invoked constructor is either the primary constructor (as in the Rational example), or another auxiliary constructor that comes textually before the calling constructor. The net effect of this rule is that every constructor invocation in Scala will end up eventually calling the primary constructor of the class. The primary constructor is thus the single point of entry of a class.

Note

If you're familiar with Java, you may wonder why Scala's rules for constructors are a bit more restrictive than Java's. In Java, a constructor must either invoke another constructor of the same class, or directly invoke a constructor of the superclass, as its first action. In a Scala class, only the primary constructor can invoke a superclass constructor. The increased restriction in Scala is really a design trade-off that needed to be paid in exchange for the greater conciseness and simplicity of Scala's constructors compared to Java's. Superclasses and the details of how constructor invocation and inheritance interact will be explained in Chapter 10.

6.8 PRIVATE FIELDS AND METHODS

In the previous version of Rational, we simply initialized numer with n and denom with d. As a result, the numerator and denominator of a Rational can be larger than needed. For example, the fraction 66/42 could be normalized to an equivalent reduced form, 11/7, but Rational's primary constructor doesn't currently do this:

```
scala> new Rational(66, 42)
res5: Rational = 66/42
```

To normalize in this way, you need to divide the numerator and denominator by their*greatest common divisor*. For example, the greatest common divisor of 66 and 42 is 6. (In other words, 6 is the largest integer that divides evenly into both 66 and 42.) Dividing both the numerator and denominator of 66/42 by 6 yields its reduced form, 11/7. Listing 6.3 shows one way to do this:

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g

  def this(n: Int) = this(n, 1)

  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  override def toString = numer + "/" + denom

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

Listing 6.3 - Rational with a private field and method.

In this version of Rational, we added a private field, g, and modified the initializers for numerand denom. (An *initializer* is the code that initializes a variable; for example, the "n / g" that initializes numer.) Because g is private, it can be accessed inside the body of the class, but not outside.

We also added a private method, gcd, which calculates the greatest common divisor of two passed Ints. For example, gcd(12, 8) is 4. As you saw in Section 4.1, to make a field or method private you simply place the private keyword in front of its definition. The purpose of the private "helper method" gcd is to factor out code needed by some other part of the class, in this case, the primary constructor. To ensure g is always positive, we pass the absolute value of n and d, which we obtain by invoking abs on them, a method you can invoke on any Int to get its absolute value.

The Scala compiler will place the code for the initializers of Rational's three fields into the primary constructor in the order in which they appear in the source code. Thus, g's initializer,gcd(n.abs, d.abs), will execute before the other two, because it appears first in the source. Fieldg will be initialized with the result, the greatest common divisor of the absolute value of the class parameters, n and d. Field g is then used in the initializers of numer and denom. By dividingn and d by their greatest common divisor, g, every Rational will be constructed in its normalized form:

```
scala> new Rational(66, 42)
res6: Rational = 11/7
```

6.9 DEFINING OPERATORS

The current implementation of Rational addition is OK, but could be made more convenient to use. You might ask yourself why you can write:

```
x + y
```

if x and y are integers or floating-point numbers, but you have to write:

```
x.add(y)
```

or at least:

```
x add y
```

if they are rational numbers. There's no convincing reason why this should be so. Rational numbers are numbers just like other numbers. In a mathematical sense they are even more natural than, say, floating-point numbers.

Why should you not use the natural arithmetic operators on them? In Scala you can do this. In the rest of this chapter, we'll show you how.

The first step is to replace add by the usual mathematical symbol. This is straightforward, as +is a legal identifier in Scala. We can simply define a method with + as its name. While we're at it, we may as well implement a method named * that performs multiplication. The result is shown in Listing 6.4:

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g
```

```
  def this(n: Int) = this(n, 1)

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def * (that: Rational): Rational =
    new Rational(numer * that.numer, denom * that.denom)

  override def toString = numer + "/" + denom

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

Listing 6.4 - Rational with operator methods.

With class Rational defined in this manner, you can now write:

```
scala> val x = new Rational(1, 2)
x: Rational = 1/2

scala> val y = new Rational(2, 3)
y: Rational = 2/3

scala> x + y
res7: Rational = 7/6
```

As always, the operator syntax on the last input line is equivalent to a method call. You could also write:

```
scala> x.+(y)
res8: Rational = 7/6
```

but this is not as readable.

Another thing to note is that given Scala's rules for operator precedence, which were described in Section 5.9, the * method will bind more tightly than the + method for Rationals. In other words, expressions involving + and * operations on Rationals will behave as expected.For example, x + x * y will execute as x + (x * y), not (x + x) * y:

```
scala> x + x * y
res9: Rational = 5/6

scala> (x + x) * y
res10: Rational = 2/3

scala> x + (x * y)
res11: Rational = 5/6
```

6.10 IDENTIFIERS IN SCALA

You have now seen the two most important ways to form an identifier in Scala: alphanumeric and operator. Scala has very flexible rules for forming identifiers. Besides the two forms you have seen there are also two others. All four forms of identifier formation are described in this section.

An alphanumeric identifier starts with a letter or underscore, which can be followed by further letters, digits, or underscores. The `$' character also counts as a letter; however, it is reserved for identifiers generated by the Scala compiler. Identifiers in user programs should not contain `$' characters, even though it will compile; if they do, this might lead to name clashes with identifiers generated by the Scala compiler.

Scala follows Java's convention of using camel-case[5] identifiers, such as toString andHashSet. Although underscores are legal in identifiers, they are not used that often in Scala programs, in part to be consistent with Java, but also because underscores have many other non-identifier uses in Scala code. As a result, it is best to avoid identifiers like to_string,__init__, or name_. Camel-case names of fields, method parameters, local variables, and functions should start with a lower case letter, for example: length, flatMap, and s. Camel-case names of classes and traits should start with an upper case letter, for example: BigInt, List, and UnbalancedTreeMap.[6]

Note

One consequence of using a trailing underscore in an identifier is that if you attempt, for example, to write a declaration like this, "val name_: Int = 1", you'll get a compiler error. The compiler will think you are trying to declare a val named "name_:". To get this to compile, you would need to insert an extra space before the colon, as in: "val name_ : Int = 1".

One way in which Scala's conventions depart from Java's involves constant names. In Scala, the word *constant* does not just mean val. Even though a val does remain constant after it is initialized, it is still a variable. For example, method parameters are vals, but each time the method is called those vals can hold different values. A constant is more permanent. For example, scala.math.Pi is defined to be the double value closest to the real value of π, the ratio of a circle's circumference to its diameter. This value is unlikely to change ever; thus, Pi is clearly a constant. You can also use constants to give names to values that would otherwise be*magic numbers* in your code: literal values with no explanation, which in the worst case appear in multiple places. You may also want to define constants for use in pattern matching, a use case that will be described in Section 15.2. In Java, the convention is to give constants names that are all upper case, with underscores separating the words, such as MAX_VALUE or PI. In Scala, the convention is merely that the first character should be upper case. Thus, constants named in the Java style, such as X_OFFSET, will work as Scala constants, but the Scala convention is to use camel case for constants, such as XOffset.

An operator identifier consists of one or more operator characters. Operator characters are printable ASCII characters such as +, :, ?, ~ or #.[7] Here are some examples of operator identifiers:

+ ++ ::: <?> :->

The Scala compiler will internally "mangle" operator identifiers to turn them into legal Java identifiers with embedded $ characters. For instance, the identifier :-> would be represented internally as $colon$minus$greater. If you ever wanted to access this identifier from Java code, you'd need to use this internal representation.

Because operator identifiers in Scala can become arbitrarily long, there is a small difference between Java and Scala. In Java, the input x<-y would be parsed as four lexical symbols, so it would be equivalent to x < - y. In Scala, <- would be parsed as a single identifier, giving x <- y. If you want the first interpretation, you need to separate the < and the - characters by a space. This is unlikely to be a problem in practice, as very few people would write x<-y in Java without inserting spaces or parentheses between the operators.

A mixed identifier consists of an alphanumeric identifier, which is followed by an underscore and an operator identifier. For example, unary_+ used as a method name defines a unary +operator. Or, myvar_= used as method name defines an assignment operator. In addition, the mixed identifier form myvar_= is generated by the Scala compiler to support properties (more on that in Chapter 18).

A literal identifier is an arbitrary string enclosed in back ticks (` ...`). Some examples of literal identifiers are:

`x` `<clinit>` `yield`

The idea is that you can put any string that's accepted by the runtime as an identifier between back ticks. The result is always a Scala identifier. This works even if the name contained in the back ticks would be a Scala reserved word. A typical use case is accessing the static yieldmethod in Java's Thread class. You cannot write Thread.yield() because yield is a reserved word in Scala. However, you can still name the method in back ticks, *e.g.*, Thread.`yield`().

6.11 METHOD OVERLOADING

Back to class Rational. With the latest changes, you can now do addition and multiplication operations in a natural style on rational numbers. But one thing still missing is mixed arithmetic. For instance, you cannot multiply a rational number by an integer because the operands of * always have to be Rationals. So for a rational number r you can't write r * 2. You must write r * new Rational(2), which is not as nice.

To make Rational even more convenient, we'll add new methods to the class that perform mixed addition and multiplication on rational numbers and integers. While we're at it, we'll add methods for subtraction and division too. The result is shown in Listing 6.5.

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g
```

```
  def this(n: Int) = this(n, 1)

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)

  def - (that: Rational): Rational =
    new Rational(
      numer * that.denom - that.numer * denom,
      denom * that.denom
    )

  def - (i: Int): Rational =
    new Rational(numer - i * denom, denom)

  def * (that: Rational): Rational =
    new Rational(numer * that.numer, denom * that.denom)

  def * (i: Int): Rational =
    new Rational(numer * i, denom)

  def / (that: Rational): Rational =
    new Rational(numer * that.denom, denom * that.numer)

  def / (i: Int): Rational =
    new Rational(numer, denom * i)

  override def toString = numer + "/" + denom

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

Listing 6.5 - Rational with overloaded methods.

There are now two versions each of the arithmetic methods: one that takes a rational as its argument and another that takes an integer. In other words, each of these method names is*overloaded* because each name is now being used by multiple methods. For example, the name + is used by one method that takes a Rational and another that takes an Int. In a method call, the compiler picks the version of an overloaded method that correctly matches the types of the arguments. For instance, if the argument y in x.+(y) is a Rational, the compiler will pick the method + that takes a Rational parameter. But if the argument is an integer, the compiler will pick the method + that takes an Int parameter instead. If you try this:

```
scala> val x = new Rational(2, 3)
x: Rational = 2/3

scala> x * x
res12: Rational = 4/9

scala> x * 2
```

```
res13: Rational = 4/3
```

You'll see that the * method invoked is determined in each case by the type of the right operand.

Note

Scala's process of overloaded method resolution is very similar to Java's. In every case, the chosen overloaded version is the one that best matches the static types of the arguments. Sometimes there is no unique best matching version; in that case the compiler will give you an "ambiguous reference" error.

6.12 IMPLICIT CONVERSIONS

Now that you can write r * 2, you might also want to swap the operands, as in 2 * r. Unfortunately this does not work yet:

```
scala> 2 * r
<console>:10: error: overloaded method value * with
alternatives:
  (x: Double)Double <and>
  (x: Float)Float <and>
  (x: Long)Long <and>
  (x: Int)Int <and>
  (x: Char)Int <and>
  (x: Short)Int <and>
  (x: Byte)Int
 cannot be applied to (Rational)
              2 * r
                ^
```

The problem here is that 2 * r is equivalent to 2.*(r), so it is a method call on the number 2, which is an integer. But the Int class contains no multiplication method that takes a Rationalargument—it couldn't because class Rational is not a standard class in the Scala library.

However, there is another way to solve this problem in Scala: You can create an implicit conversion that automatically converts integers to rational numbers when needed. Try adding this line in the interpreter:

```
scala> implicit def intToRational(x: Int) = new Rational(x)
```

This defines a conversion method from Int to Rational. The implicit modifier in front of the method tells the compiler to apply it automatically in a number of situations. With the conversion defined, you can now retry the example that failed before:

```
scala> val r = new Rational(2,3)
r: Rational = 2/3

scala> 2 * r
res15: Rational = 4/3
```

For an implicit conversion to work, it needs to be in scope. If you place the implicit method definition inside class Rational, it won't be in scope in the interpreter. For now, you'll need to define it directly in the interpreter.

As you can glimpse from this example, implicit conversions are a very powerful technique for making libraries more flexible and more convenient to use. Because they are so powerful, they can also be easily misused. You'll find out more on implicit conversions, including ways to bring them into scope where they are needed, in Chapter 21.

6.13 A WORD OF CAUTION

As this chapter has demonstrated, creating methods with operator names and defining implicit conversions can help you design libraries for which client code is concise and easy to understand. Scala gives you a great deal of power to design such easy-to-use libraries. But please bear in mind that with power comes responsibility.

If used unartfully, both operator methods and implicit conversions can give rise to client code that is hard to read and understand. Because implicit conversions are applied implicitly by the compiler, not explicitly written down in the source code, it can be non-obvious to client programmers what implicit conversions are being applied. And although operator methods will usually make client code more concise, they will only make it more readable to the extent client programmers will be able to recognize and remember the meaning of each operator.

The goal you should keep in mind as you design libraries is not merely enabling concise client code, but readable, understandable client code. Conciseness will often be a big part of that readability, but you can take conciseness too far. By designing libraries that enable tastefully concise and at the same time understandable client code, you can help those client programmers work productively.

6.14 CONCLUSION

In this chapter, you saw more aspects of classes in Scala. You saw how to add parameters to a class, define several constructors, define operators as methods, and customize classes so that they are natural to use. Maybe most importantly, you saw that defining and using immutable objects is a quite natural way to code in Scala.

Although the final version of Rational shown in this chapter fulfills the requirements set forth at the beginning of the chapter, it could still be improved. We will in fact return to this example later in the book. For example, in Chapter 30, you'll learn how to override equals andhashcode to allow Rationals to behave better when compared with == or placed into hash tables. In Chapter 21, you'll learn how to place implicit method definitions in a companion object forRational, so they can be more easily placed into scope when client programmers are working with Rationals.

Footnotes for Chapter 6:

[1] The require method is defined in standalone object, Predef. As mentioned in Section 4.4,Predef's members are imported automatically into every Scala source file.

[2] Actually, you could add a Rational to itself, in which case that would refer to the object on which add was invoked. But because you can pass any Rational object to add, the compiler still won't let you say that.n.

[3] In Section 10.6 you'll find out about *parametric fields*, which provide a shorthand for writing the same code.

[4] Even though n and d are used in the body of the class, given they are only used inside constructors, the Scala compiler will not emit fields for them. Thus, given this code the Scala compiler will generate a class with two Int fields, one for numer and one for denom.

[5] This style of naming identifiers is called *camel case* because the identifiersHaveHumps consisting of the embedded capital letters.

[6] In Section 16.5, you'll see that sometimes you may want to give a special kind of class known as a *case class* a name consisting solely of operator characters. For example, the Scala API contains a class named ::, which facilitates pattern matching on Lists.

[7] More precisely, an operator character belongs to the Unicode set of mathematical symbols(Sm) or other symbols(So), or to the 7-bit ASCII characters that are not letters, digits, parentheses, square brackets, curly braces, single or double quote, or an underscore, period, semi-colon, comma, or back tick character.