Chapter 21

Implicit Conversions and Parameters

There's a fundamental difference between your own code and other people's libraries: You can change or extend your own code as you wish, but if you want to use someone else's libraries, you usually have to take them as they are. A number of constructs have sprung up in programming languages to alleviate this problem. Ruby has modules, and Smalltalk lets packages add to each other's classes. These are very powerful but also dangerous, in that you can modify the behavior of a class for an entire application, some parts of which you might not know. C# 3.0 has static extension methods, which are more local but also more restrictive, in that you can only add methods, not fields, to a class, and you can't make a class implement new interfaces.

Scala's answer is implicit conversions and parameters. These can make existing libraries much more pleasant to deal with by letting you leave out tedious, obvious details that obscure the interesting parts of your code. Used tastefully, this results in code that is focused on the interesting, non-trivial parts of your program. This chapter shows you how implicits work, and it presents some of the most common ways they are used.

21.1 IMPLICIT CONVERSIONS

Before delving into the details of implicit conversions, take a look at a typical example of their use. Implicit conversions are often helpful for working with two bodies of software that were developed without each other in mind. Each library has its own way to encode a concept that is essentially the same thing. Implicit conversions help by reducing the number of explicit conversions that are needed from one type to another.

Java includes a library named Swing for implementing cross-platform user interfaces. One of the things Swing does is process events from the operating system, convert them to platform-independent event objects, and pass those events to parts of an application called event listeners.

If Swing had been written with Scala in mind, event listeners would probably have been represented by a function type. Callers could then use the function literal syntax as a lightweight way to specify what should happen for a certain class of events. Since Java doesn't have function literals, Swing uses the next best thing, an inner class that implements a one-method interface. In the case of action listeners, the interface is ActionListener.

Without the use of implicit conversions, a Scala program that uses Swing must use inner classes just like in Java. Here's an example that creates a button and hooks up an action listener to it. The action listener is invoked whenever the button is pressed, at which point it prints the string "pressed!":

```
val button = new JButton
button.addActionListener(
  new ActionListener {
    def actionPerformed(event: ActionEvent) = {
```

```
      println("pressed!")
    }
  }
)
```

This code has a lot of information-free boilerplate. The fact that this listener is anActionListener, the fact that the callback method is named actionPerformed, and the fact that the argument is an ActionEvent are all implied for any argument to addActionListener. The only new information here is the code to be performed, namely the call to println. This new information is drowned out by the boilerplate. Someone reading this code will need to have an eagle's eye to pick through the noise and find the informative part.

A more Scala-friendly version would take a function as an argument, greatly reducing the amount of boilerplate:

```
button.addActionListener( // Type mismatch!
  (_: ActionEvent) => println("pressed!")
)
```

As written so far, this code doesn't work.[1] The addActionListener method wants an action listener but is getting a function. With implicit conversions, however, this code can be made to work.

The first step is to write an implicit conversion between the two types. Here is an implicit conversion from functions to action listeners:

```
implicit def function2ActionListener(f: ActionEvent => Unit) =
  new ActionListener {
    def actionPerformed(event: ActionEvent) = f(event)
  }
```

This is a one-argument method that takes a function and returns an action listener. Like any other one-argument method, it can be called directly and have its result passed on to another expression:

```
button.addActionListener(
  function2ActionListener(
    (_: ActionEvent) => println("pressed!")
  )
)
```

This is already an improvement on the version with the inner class. Note how arbitrary amounts of boilerplate end up replaced by a function literal and a call to a method. It gets better, though, with implicit conversions. Because function2ActionListener is marked as implicit, it can be left out and the compiler will insert it automatically. Here is the result:

```
// Now this works
button.addActionListener(
  (_: ActionEvent) => println("pressed!")
)
```

The way this code works is that the compiler first tries to compile it as is, but it sees a type error. Before giving up, it looks for an implicit conversion that can repair the problem. In this case, it

finds function2ActionListener. It tries that conversion method, sees that it works, and moves on. The compiler works hard here so that the developer can ignore one more fiddly detail. Action listener? Action event function? Either one will work—use the one that's more convenient.

In this section, we illustrated some some of the power of implicit conversions and how they let you dress up existing libraries. In the next sections, you'll learn the rules that determine when implicit conversions are tried and how they are found.

21.2 RULES FOR IMPLICITS

Implicit definitions are those that the compiler is allowed to insert into a program in order to fix any of its type errors. For example, if x + y does not type check, then the compiler might change it to convert(x) + y, where convert is some available implicit conversion. If convertchanges x into something that has a + method, then this change might fix a program so that it type checks and runs correctly. If convert really is just a simple conversion function, then leaving it out of the source code can be a clarification.

Implicit conversions are governed by the following general rules:

Marking rule: Only definitions marked implicit are available. The implicit keyword is used to mark which declarations the compiler may use as implicits. You can use it to mark any variable, function, or object definition. Here's an example of an implicit function definition:[2]

```
implicit def intToString(x: Int) = x.toString
```

The compiler will only change x + y to convert(x) + y if convert is marked as implicit. This way, you avoid the confusion that would result if the compiler picked random functions that happen to be in scope and inserted them as "conversions." The compiler will only select among the definitions you have explicitly marked as implicit.

Scope rule: An inserted implicit conversion must be in scope as a single identifier, or be associated with the source or target type of the conversion. The Scala compiler will only consider implicit conversions that are in scope. To make an implicit conversion available, therefore, you must in some way bring it into scope. Moreover, with one exception, the implicit conversion must be in scope *as a single identifier*. The compiler will not insert a conversion of the form someVariable.convert. For example, it will not expand x + y tosomeVariable.convert(x) + y. If you want to make someVariable.convert available as an implicit, you would need to import it, which would make it available as a single identifier. Once imported, the compiler would be free to apply it as convert(x) + y. In fact, it is common for libraries to include a Preamble object including a number of useful implicit conversions. Code that uses the library can then do a single "import Preamble._" to access the library's implicit conversions.

There's one exception to the "single identifier" rule. The compiler will also look for implicit definitions in the companion object of the source or expected target types of the conversion. For example, if you're attempting to pass a Dollar object to a method that takes a Euro, the source type is Dollar and the target

type is Euro. You could, therefore, package an implicit conversion from Dollar to Euro in the companion object of either class, Dollar or Euro.

Here's an example in which the implicit definition is placed in Dollar's companion object:

```
object Dollar {
  implicit def dollarToEuro(x: Dollar): Euro = ...
}
class Dollar { ... }
```

In this case, the conversion dollarToEuro is said to be associated to the type Dollar. The compiler will find such an associated conversion every time it needs to convert from an instance of type Dollar. There's no need to import the conversion separately into your program.

The Scope Rule helps with modular reasoning. When you read code in a file, the only things you need to consider from other files are those that are either imported or are explicitly referenced through a fully qualified name. This benefit is at least as important for implicits as for explicitly written code. If implicits took effect system-wide, then to understand a file you would have to know about every implicit introduced anywhere in the program!

One-at-a-time rule: Only one implicit is inserted. The compiler will never rewrite x + yto convert1(convert2(x)) + y. Doing so would cause compile times to increase dramatically on erroneous code, and it would increase the difference between what the programmer writes and what the program actually does. For sanity's sake, the compiler does not insert further implicit conversions when it is already in the middle of trying another implicit. However, it's possible to circumvent this restriction by having implicits take implicit parameters, which will be described later in this chapter.

Explicits-first rule: Whenever code type checks as it is written, no implicits are attempted. The compiler will not change code that already works. A corollary of this rule is that you can always replace implicit identifiers by explicit ones, thus making the code longer but with less apparent ambiguity. You can trade between these choices on a case-by-case basis. Whenever you see code that seems repetitive and verbose, implicit conversions can help you decrease the tedium. Whenever code seems terse to the point of obscurity, you can insert conversions explicitly. The amount of implicits you leave the compiler to insert is ultimately a matter of style.

Naming an implicit conversion

Implicit conversions can have arbitrary names. The name of an implicit conversion matters only in two situations: If you want to write it explicitly in a method application and for determining which implicit conversions are available at any place in the program. To illustrate the second point, say you have an object with two implicit conversions:

```
object MyConversions {
  implicit def stringWrapper(s: String):
      IndexedSeq[Char] = ...
  implicit def intToString(x: Int): String = ...
}
```

In your application, you want to make use of the stringWrapper conversion, but you don't want integers to be converted automatically to strings by means of the intToString conversion. You can achieve this by importing only one conversion, but not the other:

```
import MyConversions.stringWrapper
... // code making use of stringWrapper
```

In this example, it was important that the implicit conversions had names, because only that way could you selectively import one and not the other.

Where implicits are tried

There are three places implicits are used in the language: conversions to an expected type, conversions of the receiver of a selection, and implicit parameters. Implicit conversions to an expected type let you use one type in a context where a different type is expected. For example, you might have a String and want to pass it to a method that requires anIndexedSeq[Char]. Conversions of the receiver let you adapt the receiver of a method call (*i.e.*, the object on which a method is invoked), if the method is not applicable on the original type. An example is "abc".exists, which is converted to stringWrapper("abc").exists because the existsmethod is not available on Strings but is available on IndexedSeqs. Implicit parameters, on the other hand, are usually used to provide more information to the called function about what the caller wants. Implicit parameters are especially useful with generic functions, where the called function might otherwise know nothing at all about the type of one or more arguments. We will examine each of these three kinds of implicits in the next sections.

21.3 IMPLICIT CONVERSION TO AN EXPECTED TYPE

Implicit conversion to an expected type is the first place the compiler will use implicits. The rule is simple. Whenever the compiler sees an X, but needs a Y, it will look for an implicit function that converts X to Y. For example, normally a double cannot be used as an integer because it loses precision:

```
scala> val i: Int = 3.5
<console>:7: error: type mismatch;
 found   : Double(3.5)
 required: Int
       val i: Int = 3.5
                    ^
```

However, you can define an implicit conversion to smooth this over:

```
scala> implicit def doubleToInt(x: Double) = x.toInt
doubleToInt: (x: Double)Int

scala> val i: Int = 3.5
i: Int = 3
```

What happens here is that the compiler sees a Double, specifically 3.5, in a context where it requires an Int. So far, the compiler is looking at an ordinary type error. Before giving up, though, it searches for an implicit conversion from Double to Int. In this case, it finds one:doubleToInt,

because doubleToInt is in scope as a single identifier. (Outside the interpreter, you might bring doubleToInt into scope via an import or possibly through inheritance.) The compiler then inserts a call to doubleToInt automatically. Behind the scenes, the code becomes:

```
val i: Int = doubleToInt(3.5)
```

This is literally an *implicit* conversion. You did not explicitly ask for conversion. Instead, you marked doubleToInt as an available implicit conversion by bringing it into scope as a single identifier, and then the compiler automatically used it when it needed to convert from aDouble to an Int.

Converting Doubles to Ints might raise some eyebrows because, it's a dubious idea to have something that causes a loss in precision happen invisibly. So this is not really a conversion we recommend. It makes much more sense to go the other way, from some more constrained type to a more general one. For instance, an Int can be converted without loss of precision to a Double, so an implicit conversion from Int to Double makes sense. In fact, that's exactly what happens. The scala.Predef object, which is implicitly imported into every Scala program, defines implicit conversions that convert "smaller" numeric types to "larger" ones. For instance, you will find in Predef the following conversion:

```
implicit def int2double(x: Int): Double = x.toDouble
```

That's why in Scala Int values can be stored in variables of type Double. There's no special rule in the type system for this; it's just an implicit conversion that gets applied.[3]

21.4 CONVERTING THE RECEIVER

Implicit conversions also apply to the receiver of a method call, the object on which the method is invoked. This kind of implicit conversion has two main uses. First, receiver conversions allow smoother integration of a new class into an existing class hierarchy. And second, they support writing domain-specific languages (DSLs) within the language.

To see how it works, suppose you write down obj.doIt, and obj does not have a member named doIt. The compiler will try to insert conversions before giving up. In this case, the conversion needs to apply to the receiver, obj. The compiler will act as if the expected "type" of obj was "has a member named doIt." This "has a doIt" type is not a normal Scala type, but it is there conceptually and is why the compiler will insert an implicit conversion in this case.

Interoperating with new types

As mentioned previously, one major use of receiver conversions is allowing smoother integration of new types with existing types. In particular, they allow you to enable client programmers to use instances of existing types as if they were instances of your new type. Take, for example, class Rational shown in Listing 6.5 here. Here's a snippet of that class again:

```
class Rational(n: Int, d: Int) {
  ...
  def + (that: Rational): Rational = ...
  def + (that: Int): Rational = ...
}
```

Class Rational has two overloaded variants of the + method, which take Rationals and Ints, respectively, as arguments. So you can either add two rational numbers or a rational number and an integer:

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

scala> oneHalf + oneHalf
res0: Rational = 1/1

scala> oneHalf + 1
res1: Rational = 3/2
```

What about an expression like 1 + oneHalf? This expression is tricky because the receiver, 1, does not have a suitable + method. So the following gives an error:

```
scala> 1 + oneHalf
<console>:6: error: overloaded method value + with
alternatives (Double)Double <and> ... cannot be applied
to (Rational)
     1 + oneHalf
       ^
```

To allow this kind of mixed arithmetic, you need to define an implicit conversion from Int toRational:

```
scala> implicit def intToRational(x: Int) =
         new Rational(x, 1)
intToRational: (x: Int)Rational
```

With the conversion in place, converting the receiver does the trick:

```
scala> 1 + oneHalf
res2: Rational = 3/2
```

What happens behind the scenes here is that the Scala compiler first tries to type check the expression 1 + oneHalf as it is. This fails because Int has several + methods, but none that takes a Rational argument. Next, the compiler searches for an implicit conversion from Int to another type that has a + method which can be applied to a Rational. It finds your conversion and applies it, which yields:

```
intToRational(1) + oneHalf
```

In this case, the compiler found the implicit conversion function because you entered its definition into the interpreter, which brought it into scope for the remainder of the interpreter session.

Simulating new syntax

The other major use of implicit conversions is to simulate adding new syntax. Recall that you can make a Map using syntax like this:

```
Map(1 -> "one", 2 -> "two", 3 -> "three")
```

Have you wondered how the -> is supported? It's not syntax! Instead, -> is a method of the class ArrowAssoc, a class defined inside the standard Scala preamble (scala.Predef). The preamble also defines an implicit conversion from Any to ArrowAssoc. When you write 1 -> "one", the compiler inserts a conversion from 1 to ArrowAssoc so that the -> method can be found. Here are the relevant definitions:

```
package scala
object Predef {
  class ArrowAssoc[A](x: A) {
    def -> [B](y: B): Tuple2[A, B] = Tuple2(x, y)
  }
  implicit def any2ArrowAssoc[A](x: A): ArrowAssoc[A] =
    new ArrowAssoc(x)
  ...
}
```

This "rich wrappers" pattern is common in libraries that provide syntax-like extensions to the language, so you should be ready to recognize the pattern when you see it. Whenever you see someone calling methods that appear not to exist in the receiver class, they are probably using implicits. Similarly, if you see a class named RichSomething (*e.g.*, RichInt or RichBoolean), that class is likely adding syntax-like methods to type Something.

You have already seen this rich wrappers pattern for the basic types described in Chapter 5. As you can now see, these rich wrappers apply more widely, often letting you get by with an internal DSL defined as a library where programmers in other languages might feel the need to develop an external DSL.

Implicit classes

Implicit classes were added in Scala 2.10 to make it easier to write rich wrapper classes. An implicit class is a class that is preceded by the implicit keyword. For any such class, the compiler generates an implicit conversion from the class's constructor parameter to the class itself. Such a conversion is just what you need if you plan to use the class for the rich wrappers pattern.

For example, suppose you have a class named Rectangle for representing the width and height of a rectangle on the screen:

```
case class Rectangle(width: Int, height: Int)
```

If you use this class very frequently, you might want to use the rich wrappers pattern so you can more easily construct it. Here's one way to do so.

```
implicit class RectangleMaker(width: Int) {
  def x(height: Int) = Rectangle(width, height)
}
```

The above definition defines a RectangleMaker class in the usual manner. In addition, it causes the following conversion to be automatically generated:

```
// Automatically generated
implicit def RectangleMaker(width: Int) =
```

```
    new RectangleMaker(width)
```

As a result, you can create points by putting an x in between two integers:

```
scala> val myRectangle = 3 x 4
  myRectangle: Rectangle = Rectangle(3,4)
```

This is how it works: Since type Int has no method named x, the compiler will look for an implicit conversion from Int to something that does. It will find the generated RectangleMakerconversion, and RectangleMaker does have a method named x. The compiler inserts a call to this conversion, after which the call to x type checks and does what is desired.

As a warning to the adventurous, it might be tempting to think that any class can haveimplicit put in front of it. It's not so. An implicit class cannot be a case class, and its constructor must have exactly one parameter. Also, an implicit class must be located within some other object, class, or trait. In practice, so long as you use implicit classes as rich wrappers to add a few methods onto an existing class, these restrictions should not matter.

21.5 IMPLICIT PARAMETERS

The remaining place the compiler inserts implicits is within argument lists. The compiler will sometimes replace someCall(a) with someCall(a)(b), or new SomeClass(a) with new SomeClass(a)(b), thereby adding a missing parameter list to complete a function call. It is the entire last curried parameter list that's supplied, not just the last parameter. For example, if someCall's missing last parameter list takes three parameters, the compiler might replace someCall(a)with someCall(a)(b, c, d). For this usage, not only must the inserted identifiers, such as b, c, andd in (b, c, d), be marked implicit where they are defined, but also the last parameter list insomeCall's or someClass's definition must be marked implicit.

Here's a simple example. Suppose you have a class PreferredPrompt, which encapsulates a shell prompt string (such as, say "$ " or "> ") that is preferred by a user:

```
class PreferredPrompt(val preference: String)
```

Also, suppose you have a Greeter object with a greet method, which takes two parameter lists. The first parameter list takes a string user name, and the second parameter list takes aPreferredPrompt:

```
object Greeter {
  def greet(name: String)(implicit prompt: PreferredPrompt) = {
    println("Welcome, " + name + ". The system is ready.")
    println(prompt.preference)
  }
}
```

The last parameter list is marked implicit, which means it can be supplied implicitly. But you can still provide the prompt explicitly, like this:

```
scala> val bobsPrompt = new PreferredPrompt("relax> ")
bobsPrompt: PreferredPrompt = PreferredPrompt@714d36d6
```

```
scala> Greeter.greet("Bob")(bobsPrompt)
Welcome, Bob. The system is ready.
relax>
```

To let the compiler supply the parameter implicitly, you must first define a variable of the expected type, which in this case is PreferredPrompt. You could do this, for example, in a preferences object:

```
object JoesPrefs {
  implicit val prompt = new PreferredPrompt("Yes, master> ")
}
```

Note that the val itself is marked implicit. If it wasn't, the compiler would not use it to supply the missing parameter list. It will also not use it if it isn't in scope as a single identifier, as shown in this example:

```
scala> Greeter.greet("Joe")
<console>:13: error: could not find implicit value for
parameter prompt: PreferredPrompt
              Greeter.greet("Joe")
                            ^
```

Once you bring it into scope via an import, however, it will be used to supply the missing parameter list:

```
scala> import JoesPrefs._
import JoesPrefs._

scala> Greeter.greet("Joe")
Welcome, Joe. The system is ready.
Yes, master>
```

Note that the implicit keyword applies to an entire parameter list, not to individual parameters. Listing 21.1 shows an example in which the last parameter list of Greeter's greetmethod, which is again marked implicit, has two parameters: prompt (of type PreferredPrompt) and drink (of type PreferredDrink).

Singleton object JoesPrefs declares two implicit vals, prompt of type PreferredPrompt and drink of type PreferredDrink. As before, however, so long as these are not in scope as single identifiers, they won't be used to fill in a missing parameter list to greet:

```
scala> Greeter.greet("Joe")
<console>:19: error: could not find implicit value for
parameter prompt: PreferredPrompt
              Greeter.greet("Joe")
                            ^
```

You can bring both implicit vals into scope with an import:

```
scala> import JoesPrefs._
import JoesPrefs._
```

Because both prompt and drink are now in scope as single identifiers, you can use them to supply the last parameter list explicitly, like this:

```
scala> Greeter.greet("Joe")(prompt, drink)
Welcome, Joe. The system is ready.
But while you work, why not enjoy a cup of tea?
Yes, master>
```

And because all the rules for implicit parameters are now met, you can alternatively let the Scala compiler supply prompt and drink for you by leaving off the last parameter list:

```
scala> Greeter.greet("Joe")
Welcome, Joe. The system is ready.
But while you work, why not enjoy a cup of tea?
Yes, master>

class PreferredPrompt(val preference: String)
class PreferredDrink(val preference: String)

object Greeter {
  def greet(name: String)(implicit prompt: PreferredPrompt,
      drink: PreferredDrink) = {

    println("Welcome, " + name + ". The system is ready.")
    print("But while you work, ")
    println("why not enjoy a cup of " + drink.preference + "?")
    println(prompt.preference)
  }
}

object JoesPrefs {
  implicit val prompt = new PreferredPrompt("Yes, master> ")
  implicit val drink = new PreferredDrink("tea")
}
```

Listing 21.1 - An implicit parameter list with multiple parameters.

One thing to note about the previous examples is that we didn't use String as the type of promptor drink, even though ultimately it was a String that each of them provided through theirpreference fields. Because the compiler selects implicit parameters by matching types of parameters against types of values in scope, implicit parameters usually have "rare" or "special" enough types that accidental matches are unlikely. For example, the typesPreferredPrompt and PreferredDrink in Listing 21.1 were defined solely to serve as implicit parameter types. As a result, it is unlikely that implicit variables of these types will be in scope if they aren't intended to be used as implicit parameters to Greeter.greet.

Another thing to know about implicit parameters is that they are perhaps most often used to provide information about a type mentioned *explicitly* in an earlier parameter list, similar to the type classes of Haskell.

As an example, consider the maxListOrdering function shown in Listing 21.2, which returns the maximum element of the passed list.

```
def maxListOrdering[T](elements: List[T])
      (ordering: Ordering[T]): T =
  elements match {
    case List() =>
      throw new IllegalArgumentException("empty list!")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxListOrdering(rest)(ordering)
      if (ordering.gt(x, maxRest)) x
      else maxRest
  }
```

Listing 21.2 - A function with an upper bound.

The signature of maxListOrdering is similar to that of orderedMergeSort, shown in Listing 19.12here: It takes a List[T] as its argument, and now it takes an additional argument of typeOrdering[T]. This additional argument specifies which ordering to use when comparing elements of type T. As such, this version can be used for types that don't have a built-in ordering. Additionally, this version can be used for types that *do* have a built-in ordering, but for which you occasionally want to use some other ordering.

This version is more general, but it's also more cumbersome to use. Now a caller must specify an explicit ordering even if T is something like String or Int that has an obvious default ordering. To make the new method more convenient, it helps to make the second argument implicit. This approach is shown in Listing 21.3.

The ordering parameter in this example is used to describe the ordering of Ts. In the body ofmaxListImpParm, this ordering is used in two places: a recursive call to maxListImpParm, and an ifexpression that checks whether the head of the list is larger than the maximum element of the rest of the list.

```
def maxListImpParm[T](elements: List[T])
      (implicit ordering: Ordering[T]): T =

  elements match {
    case List() =>
      throw new IllegalArgumentException("empty list!")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxListImpParm(rest)(ordering)
      if (ordering.gt(x, maxRest)) x
      else maxRest
  }
```

Listing 21.3 - A function with an implicit parameter.

The maxListImpParm function is an example of an implicit parameter used to provide more information about a type mentioned explicitly in an earlier parameter list. To be specific, the implicit parameter ordering, of type Ordering[T], provides more information about type T—in this case, how to order Ts. Type T is mentioned in List[T], the type of parameter elements, which appears in the earlier parameter list. Because elements must always be provided explicitly in any invocation

of maxListImpParm, the compiler will know T at compile time and can therefore determine whether an implicit definition of type Ordering[T] is available. If so, it can pass in the second parameter list, ordering, implicitly.

This pattern is so common that the standard Scala library provides implicit "ordering" methods for many common types. You could therefore use this maxListImpParm method with a variety of types:

```
scala> maxListImpParm(List(1,5,10,3))
res9: Int = 10

scala> maxListImpParm(List(1.5, 5.2, 10.7, 3.14159))
res10: Double = 10.7

scala> maxListImpParm(List("one", "two", "three"))
res11: String = two
```

In the first case, the compiler inserted an ordering for Ints; in the second case, for Doubles; in the third case, for Strings.

A style rule for implicit parameters

As a style rule, it is best to use a custom named type in the types of implicit parameters. For example, the types of prompt and drink in the previous example was not String, butPreferredPrompt and PreferredDrink, respectively. As a counterexample, consider that themaxListImpParm function could just as well have been written with the following type signature:

```
def maxListPoorStyle[T](elements: List[T])
      (implicit orderer: (T, T) => Boolean): T
```

To use this version of the function, though, the caller would have to supply an ordererparameter of type (T, T) => Boolean. This is a fairly generic type that includes any function from two Ts to a Boolean. It does not indicate anything at all about what the type is for; it could be an equality test, a less-than test, a greater-than test, or something else entirely.

The actual code for maxListImpParm, given in Listing 21.3, shows better style. It uses an orderingparameter of type Ordering[T]. The word Ordering in this type indicates exactly what the implicit parameter is used for: it is for ordering elements of T. Because this ordering type is more explicit, it's no trouble to add implicit providers for this type in the standard library. To contrast, imagine the chaos that would ensue if you added an implicit of type (T, T) => Booleanin the standard library, and the compiler started sprinkling it around in people's code. You would end up with code that compiles and runs, but that does fairly arbitrary tests against pairs of items! Thus the style rule: Use at least one role-determining name within the type of an implicit parameter.

21.6 CONTEXT BOUNDS

The previous example showed an opportunity to use an implicit but did not. Note that when you use implicit on a parameter, not only will the compiler try to *supply* that parameter with an implicit

value, but the compiler will also *use* that parameter as an available implicit in the body of the method! Thus, the first use of ordering within the body of the method can be left out.

```
def maxList[T](elements: List[T])
      (implicit ordering: Ordering[T]): T =

  elements match {
    case List() =>
      throw new IllegalArgumentException("empty list!")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxList(rest)     // (ordering) is implicit
      if (ordering.gt(x, maxRest)) x  // this ordering is
      else maxRest                    // still explicit
  }
```

Listing 21.4 - A function that uses an implicit parameter internally.

When the compiler examines the code in Listing 21.4, it will see that the types do not match up. The expression maxList(rest) only supplies one parameter list, but maxList requires two. Since the second parameter list is implicit, the compiler does not give up type checking immediately. Instead, it looks for an implicit parameter of the appropriate type, in this caseOrdering[T]. In this case, it finds one and rewrites the call to maxList(rest)(ordering), after which the code type checks.

There is also a way to eliminate the second use of ordering. It involves the following method defined in the standard library:

```
def implicitly[T](implicit t: T) = t
```

The effect of calling implicitly[Foo] is that the compiler will look for an implicit definition of type Foo. It will then call the implicitly method with that object, which in turn returns the object right back. Thus you can write implicitly[Foo] whenever you want to find an implicit object of type Foo in the current scope. For example, Listing 21.5 shows a use ofimplicitly[Ordering[T]] to retrieve
the ordering parameter by its type.

Look closely at this last version of maxList. There is not a single mention of the orderingparameter in the text of the method. The second parameter could just as well be named "comparator":

```
def maxList[T](elements: List[T])
      (implicit comparator: Ordering[T]): T = // same body...

  def maxList[T](elements: List[T])
        (implicit ordering: Ordering[T]): T =

    elements match {
      case List() =>
        throw new IllegalArgumentException("empty list!")
      case List(x) => x
      case x :: rest =>
        val maxRest = maxList(rest)
        if (implicitly[Ordering[T]].gt(x, maxRest)) x
        else maxRest
    }
```

Listing 21.5 - A function that uses implicitly.

For that matter, this version works as well:

```
def maxList[T](elements: List[T])
      (implicit iceCream: Ordering[T]): T = // same body...
```

Because this pattern is common, Scala lets you leave out the name of this parameter and shorten the method header by using a *context bound*. Using a context bound, you would write the signature of maxList as shown in Listing 21.6. The syntax [T : Ordering] is a context bound, and it does two things. First, it introduces a type parameter T as normal. Second, it adds an implicit parameter of type Ordering[T]. In previous versions of maxList, that parameter was called ordering, but when using a context bound you don't know what the parameter will be called. As shown earlier, you often don't need to know what the parameter is called.

Intuitively, you can think of a context bound as saying something *about* a type parameter. When you write [T <: Ordered[T]] you are saying that a T *is* an Ordered[T]. To contrast, when you write [T : Ordering] you are not so much saying what T is; rather, you are saying that there is some form of ordering associated with T. Thus, a context bound is quite flexible. It allows you to use code that requires orderings—or any other property of a type—without having to change the definition of that type.

```
def maxList[T : Ordering](elements: List[T]): T =
  elements match {
    case List() =>
      throw new IllegalArgumentException("empty list!")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxList(rest)
      if (implicitly[Ordering[T]].gt(x, maxRest)) x
      else maxRest
  }
```

Listing 21.6 - A function with a context bound.

21.7 WHEN MULTIPLE CONVERSIONS APPLY

It can happen that multiple implicit conversions are in scope and each would work. For the most part, Scala refuses to insert a conversion in such a case. Implicits work well when the conversion left out is completely obvious and pure boilerplate. If multiple conversions apply, then the choice isn't so obvious after all.

Here's a simple example. There is a method that takes a sequence, a conversion that turns an integer into a range, and a conversion that turns an integer into a list of digits:

```
scala> def printLength(seq: Seq[Int]) = println(seq.length)
printLength: (seq: Seq[Int])Unit

scala> implicit def intToRange(i: Int) = 1 to i
intToRange: (i:
Int)scala.collection.immutable.Range.Inclusive
```

```
scala> implicit def intToDigits(i: Int) =
         i.toString.toList.map(_.toInt)
intToDigits: (i: Int)List[Int]

scala> printLength(12)
<console>:26: error: type mismatch;
 found   : Int(12)
 required: Seq[Int]
Note that implicit conversions are not applicable because
they are ambiguous:
 both method intToRange of type (i:
Int)scala.collection.immutable.Range.Inclusive
 and method intToDigits of type (i: Int)List[Int]
 are possible conversion functions from Int(12) to Seq[Int]
              printLength(12)
                          ^
```

The ambiguity here is real. Converting an integer to a sequence of digits is completely different from converting it to a range. In this case, the programmer should specify which one is intended and be explicit. Up through Scala 2.7, that was the end of the story. Whenever multiple implicit conversions applied, the compiler refused to choose between them. The situation was just as with method overloading. If you try to call foo(null) and there are two different foo overloads that accept null, the compiler will refuse. It will say that the method call's target is ambiguous.

Scala 2.8 loosened this rule. If one of the available conversions is strictly *more specific* than the others, then the compiler will choose the more specific one. The idea is that whenever there is a reason to believe a programmer would always choose one of the conversions over the others, don't require the programmer to write it explicitly. After all, method overloading has the same relaxation. Continuing the previous example, if one of the available foo methods takes a String while the other takes an Any, then choose the String version. It's clearly more specific.

To be more precise, one implicit conversion is *more specific* than another if one of the following applies:

- The argument type of the former is a subtype of the latter's.
- Both conversions are methods, and the enclosing class of the former extends the enclosing class of the latter.

The motivation to revisit this issue and revise the rule was to improve interoperation between Java collections, Scala collections, and strings.

Here's a simple example:

```
val cba = "abc".reverse
```

What is the type inferred for cba? Intuitively, the type should be String. Reversing a string should yield another string, right? However, in Scala 2.7, what happened was that "abc" was converted to a Scala collection. Reversing a Scala collection yields a Scala collection, so the type of cba would be a collection. There's also an implicit conversion back to a string, but that didn't patch up every problem. For example, in versions prior to Scala 2.8,"abc" == "abc".reverse.reverse was false!

With Scala 2.8, the type of cba is String. The old implicit conversion to a Scala collection (now named WrappedString) is retained. However, there is a more specific conversion supplied fromString to a new type called StringOps. StringOps has many methods such as reverse, but instead of returning a collection, they return a String. The conversion to StringOps is defined directly inPredef, whereas the conversion to a Scala collection is defined in a new class,LowPriorityImplicits, which is extended by Predef. Whenever a choice exists between these two conversions, the compiler chooses the conversion to StringOps, because it's defined in a subclass of the class where the other conversion is defined.

21.8 DEBUGGING IMPLICITS

Implicits are a powerful feature in Scala, but one that's sometimes difficult to get right. This section contains a few tips for debugging implicits.

Sometimes you might wonder why the compiler did not find an implicit conversion that you think should apply. In that case it helps to write the conversion out explicitly. If that also gives an error message, you then know why the compiler could not apply your implicit.

For instance, assume that you mistakenly took wrapString to be a conversion from Strings toLists, instead of IndexedSeqs. You would wonder why the following code does not work:

```
scala> val chars: List[Char] = "xyz"
<console>:24: error: type mismatch;
 found   : String("xyz")
 required: List[Char]
       val chars: List[Char] = "xyz"
                               ^
```

Again, it helps to write the wrapString conversion explicitly to find out what went wrong:

```
scala> val chars: List[Char] = wrapString("xyz")
<console>:24: error: type mismatch;
 found   : scala.collection.immutable.WrappedString
 required: List[Char]
       val chars: List[Char] = wrapString("xyz")
                                          ^
```

With this, you have found the cause of the error: wrapString has the wrong return type. On the other hand, it's also possible that inserting the conversion explicitly will make the error go away. In that case you know that one of the other rules (such as the Scope Rule) was preventing the implicit conversion from being applied.

When you are debugging a program, it can sometimes help to see what implicit conversions the compiler is inserting. The -Xprint:typer option to the compiler is useful for this. If you runscalac with this option, the compiler will show you what your code looks like after all implicit conversions have been added by the type checker. An example is shown in Listing 21.7 andListing 21.8. If you look at the last statement in each of these listings, you'll see that the second parameter list to enjoy, which was

left off in the code in Listing 21.7, "enjoy("reader")," was inserted by the compiler, as shown in Listing 21.8:

```
  Mocha.this.enjoy("reader")(Mocha.this.pref)
```

If you are brave, try scala -Xprint:typer to get an interactive shell that prints out the post-typing source code it uses internally. If you do so, be prepared to see an enormous amount of boilerplate surrounding the meat of your code.

```
  object Mocha extends App {

    class PreferredDrink(val preference: String)

    implicit val pref = new PreferredDrink("mocha")

    def enjoy(name: String)(implicit drink: PreferredDrink) = {
      print("Welcome, " + name)
      print(". Enjoy a ")
      print(drink.preference)
      println("!")
    }

    enjoy("reader")
  }
```

Listing 21.7 - Sample code that uses an implicit parameter.

```
  $ scalac -Xprint:typer mocha.scala
  [[syntax trees at end of typer]]
// Scala source: mocha.scala
  package <empty> {
    final object Mocha extends java.lang.Object with Application
        with ScalaObject {


// ...

      private[this] val pref: Mocha.PreferredDrink =
        new Mocha.this.PreferredDrink("mocha");
      implicit <stable> <accessor>
        def pref: Mocha.PreferredDrink = Mocha.this.pref;
      def enjoy(name: String)
          (implicit drink: Mocha.PreferredDrink): Unit = {
        scala.this.Predef.print("Welcome, ".+(name));
        scala.this.Predef.print(". Enjoy a ");
        scala.this.Predef.print(drink.preference);
        scala.this.Predef.println("!")
      };
      Mocha.this.enjoy("reader")(Mocha.this.pref)
    }
  }
```

Listing 21.8 - Sample code after type checking and insertion of implicits.

21.9 CONCLUSION

Implicits are a powerful, code-condensing feature of Scala. This chapter has shown you Scala's rules about implicits and several common programming situations where you can profit from using implicits.

As a word of warning, implicits can make code confusing if they are used too frequently. Thus, before adding a new implicit conversion, first ask whether you can achieve a similar effect through other means, such as inheritance, mixin composition, or method overloading. If all of these fail, however, and you feel like a lot of your code is still tedious and redundant, then implicits might just be able to help you out.

Footnotes for Chapter 21:

[1] As will be explained in Section 31.5, it does work in Scala 2.12.

[2] Variables and singleton objects marked implicit can be used as *implicit parameters*. This use case will be described later in this chapter.

[3] The Scala compiler backend will treat the conversion specially, however, translating it to a special "i2d" bytecode. So the compiled image is the same as in Java.