5 Using implicits to write expressive code

In this chapter

- Introduction to implicits
- Mechanics of the implicit resolution system
- Using implicits to enhance classes
- Using implicits to enforce scope rules

The implicit system in Scala allows the compiler to adjust code using a well-defined lookup mechanism. A programmer in Scala can leave out information that the compiler will attempt to infer at compile time. The Scala compiler can infer one of two situations:

- A method call or constructor with a missing parameter.
- Missing conversion from one type to another type. This also applies to method calls on an object that would require a conversion.

In both of these situations, the compiler follows a set of rules to resolve missing data and allow the code to compile. When the programmer leaves out parameters, it's incredibly useful and is done in advanced Scala libraries. When the compiler converts types to ensure that an expression compiles is more dangerous and is the cause of controversy.

The implicit system is one of the greatest assets of the Scala programming language. Using it wisely and conservatively can drastically reduce the size of your code base. It can also be used to elegantly enforce design considerations. Let's look at implicit parameters in Scala.

5.1 Introduction to implicits

Scala provides an `implicit` keyword that can be used in two ways: method or variable definitions, and method parameter lists. If this keyword is used on method or variable definitions, it tells the compiler that those methods or variable definitions can be used during implicit resolution. Implicit resolution is when the compiler determines that a piece of information is missing in code, and it must be looked up. The `implicit` keyword can also be used at the beginning of a method parameter list. This tells the compiler that the parameter list might be missing, in which case the compiler should resolve the parameters via implicit resolution.

Let's look at using the implicit resolution mechanism to resolve a missing parameter list:

```
scala> def findAnInt(implicit x : Int) = x
findAnInt: (implicit x: Int)Int
```

The `findAnInt` method declares a single parameter `x` of type `Int`. This function will return any value that's passed into it. The parameter list is marked with implicit, which means that we don't need to use it. If it's left off, the compiler will look for a variable of type `Int` in the implicit scope. Let's look at the following example:

```
scala> findAnInt
<console>:7: error: could not find implicit value for parameter x: Int
       findAnInt
       ^
```

The `findAnInt` method is called without specifying any argument list. The compiler complains that it can't find an implicit value for the `x` parameter. We'll provide one, as follows:

```
scala> implicit val test = 5
test: Int = 5
```

The `test` value is defined with the `implicit` keyword. This marks it as available for implicit resolution. Since this is in the REPL, the `test` value will be available in the implicit scope for the rest of the REPL session. Here's what happens when we can `findAnInt`:

```
scala> findAnInt
res3: Int = 5
```

The call to `findAnInt` succeeds and returns the value of the `test` value. The compiler was able to successfully complete the function call. We can still provide the parameter if desired.

```
scala> findAnInt(2)
res4: Int = 2
```

This method call passes a parameter with a value of 2. Because the method call is complete, the compiler doesn't need to look up a value using implicits. Remember this, as implicit method parameters can still be explicitly provided. This utility will be discussed further in section 5.6.

To understand how the compiler determines if a variable is available for implicit resolution, it's important to dig into how the compiler deals with identifiers and scope.

5.1.1 Identifiers: A digression

Before delving into the implicit resolution mechanism, it's important to understand how the compiler resolves identifiers within a particular scope. This section references chapter 2 of the Scala Language Specification (SLS), I highly recommend reading through the SLS after you have an understanding of the basics. Identifiers play a crucial role in the selection of implicits, so let's dig into the nuts and bolts of identifiers in Scala.

Scala defines the term *entity* to mean types, values, methods, or classes. These are the things we use to build our programs. We refer to them using identifiers, or names. In Scala this is called a *binding*. For example, in the following code:

```
class Foo {
  def val x = 5
}
```

the `Foo` class itself is an entity, a class containing an `x` method. But we've given this class the name `Foo`, which is the binding. If we declare this class locally within the REPL, we can instantiate it using the name `Foo` because it's locally bound.

```
scala> val y = new Foo
y: Foo = Foo@33262bf4
```

Here we can construct a new variable, named `y`, of type `Foo` using the name `Foo`. Again, this is because the class Foo was defined locally within the REPL and the name `Foo` was bound locally. Let's complicate things by placing `Foo` in a package.

```
package test;

class Foo {
  val x = 5
}
```

The `Foo` class is now a member of the package `test`. If we try to access it with the name `Foo`, it will fail on the REPL:

```
scala> new Foo
<console>:7: error: not found: type Foo
       new Foo
```

Trying to call new `Foo` fails because the name `Foo` isn't bound in our scope. The `Foo` class is now in the `test` package. To access it, we must either use the name `test.Foo` or create a binding of the name `Foo` to the `test.Foo` class in the current scope. For the latter, Scala provides the `import` keyword:

```
scala> import test.Foo
import test.Foo

scala> new Foo
res3: test.Foo = test.Foo@60e1e567
```

The `import` statement takes `test.Foo` entity and binds it in the local scope with the name `Foo`. This allows us to construct a new `test.Foo` instance by calling `new Foo`. This concept should be familiar from Java's `import` statement or C++'s `using` statement. In Scala, things are a bit more flexible.

The `import` statement can be used anywhere in the source file and it will only create a binding in the local scope. This feature allows us to control where imported names are used within our file. This feature can also be used to limit the scope of implicits views or variables. We'll cover this aspect in more detail in section 5.4.

Scala is also more flexible in binding entities with arbitrary names. In Java or C#, one can only bring the name bound in some other scope, or package, into the current one. For example, the `test.Foo` class could only be imported locally with the name `Foo`. The Scala `import` statement can give arbitrary names to imported entities using the `{OriginalBinding=>NewBinding}` syntax. Let's import our `test.Foo` entity with a different name:

```
scala> import test.{Foo=>Bar}
import test.{Foo=>Bar}

scala> new Bar
res1: test.Foo = test.Foo@596b753
```

The first `import` statement binds the `test.Foo` class to the current scope using the name `Bar`. The next line constructs a new instance of `test.Foo` by calling `new Bar`. You can use this renaming to avoid conflicts in classes imported from different packages. A good example is with `java.util.List` and `scala.List`. To avoid confusion within Scala, it's common to see `import java.util.{List=>JList}` in code that interacts with Java.

> **RENAMING PACKAGES** Scala's import statement can also be used to alter the names of packages. This can be handy when interacting with Java libraries. For example, when using the `java.io` package, I frequently do the following:
>
> ```
> import java.{io=>jio}
> def someMethod( input : jio.InputStream ) = ...
> ```

Binding entities allows us to name them within a particular scope. But it's important to understand what constitutes a scope and what bindings are found in a scope.

5.1.2 Scope and bindings

A scope is a lexical boundary in which bindings are available. A scope could be anything from the body of a class to the body of a method to an anonymous block. As a general rule, anytime you use the `{}` characters you're creating a new scope.

In Scala, scopes can be nested. This means I can construct a new scope inside another scope. When creating a new scope, the bindings from the outer scope are still available. This allows us to do the following:

```
class Foo(x : Int) {
  def tmp = {
    x
  }
}
```

The `Foo` class is defined with the constructor parameter `x`. We then define the `tmp` method with a nested scope. We can still access the constructor parameter inside this scope with the name `x`. This nested scope has access to bindings in its parent scope, however we can create new bindings that shadow the parent. In this case, the `tmp` method can create a new binding called `x` that does not refer to the constructor parameter `x`. Let's take a look:

```
scala> class Foo(x : Int) {
     |   def tmp = {
     |     val x = 2
     |     x
     |   }
     | }
defined class Foo
```

The `Foo` class is defined the same as before, but the `tmp` method defines a variable named x in the nested scope. This binding *shadows* the constructor parameter `x`. Shadowing means that the local binding is visible and the constructor parameter is no longer accessible, at least using the name `x`. In Scala, bindings of higher precedence shadow bindings of lower precedence within the same scope. Also, bindings of higher or the same precedence shadow bindings in an outer scope.

Scala defines the following precedence on bindings:

1. Definitions and declarations that are local, inherited, or made available by a package clause in the same source file where the definition occurs have highest precedence.
2. Explicit imports have next highest precedence.
3. Wildcard imports (`import foo._`) have next highest precedence.
4. Definitions made available by a package clause not in the source file where the definition occurs have lowest precedence.

Let's look at an example of this precedence. First, let's define a `test` package and an

Bindings and Shadowing

In Scala, a binding shadows bindings of lower precedence within the same scope. A binding shadows bindings of the same or lower precedence in an outer scope. This is what allows us to write:

```
class Foo(x : Int) {
 def tmp = {
 val x = 2
 x
 }
}
```

And have calls to `tmp` return the value `2`.

object x within it in a source file called `externalbindings.scala`, as shown in the following listing:

Listing 5.1 externalbindings.scala

```
package test;

object x {
  override def toString = "Externally bound x object in package test"
}
```

This file defines a package `test` with the `x` object inside it. The `x` object overrides the `toString` method so we can easily call `toString` on it. This means that for the purposes of our test, the x object should have the lowest binding precedence with binding rules. Now, let's create a file that will test the binding rules:

Listing 5.2 Implicit binding test file

```
package test;

object Test {
  def main(args : Array[String]) : Unit = {
    testSamePackage()
    testWildcardImport()
    testExplicitImport()
    testInlineDefinition()
  }
  ...
}
```

First, we declare the contents of the file to be in the same test package as our earlier definition. Next, we define a main method that will call four testing methods, one for each binding precedence rule. Let's fill the first one in now:

```
def testSamePackage() {
  println(x)
}
```

This method calls `println` on an entity called `x`. Because the `Test` object is defined within the `test` package, the `x` object created earlier is available and used for this method. To prove this, look at the output of this method:

```
scala> test.Test.testSamePackage()
Externally bound x object in package test
```

Calling the `testSamePackage` method produces the string we defined for the object `x`. Now let's see what happens if we add a `Wildcard` import:

Listing 5.3 `Wildcard` imports

```
object Wildcard {
  def x = "Wildcard Import x"
}

def testWildcardImport() {
```

```
  import Wildcard._
  println(x)
}
```

The `Wildcard` object is a nested object used to contain the `x` entity so that it can later be imported. `The entity x` is defined as a method that returns the string `"Wildcard Import x"`. The `testWildcardImport` method first calls `import Wildcard._`. This is a wildcard import that will bind all the names/entities from the `Wildcard` object into the current scope. Because wildcard imports have higher precedence than resources made available from the same package but in a different source file, the `Wildcard.x` entity will be used instead of the `test.x` entity. We see this when we run the `testWildcardImport` function:

```
scala> test.Test.testWildcardImport()
Wildcard Import x
```

When calling the `testWildcardImport` method, the string `Wildcard Import x` is returned—exactly what we expect from the binding precedence. Things get more interesting when we add explicit imports.

Listing 5.4 Explicit imports

```
object Explicit {
  def x = "Explicit Import x"
}

def testExplicitImport() {
  import Explicit.x
  import Wildcard._
  println(x)
}
```

Once again, the `Explicit` object is used to create a new namespace for another `x` entity. The `testExplicitImport` method first imports this entity directly and then uses the wildcard import against the `Wildcard` object. Although the wildcard import is after the explicit import, the binding precedence rules kick in and the method will use the `x` binding from the `Explicit` object. Let's take a look:

```
scala> test.Test.testExplicitImport()
Explicit Import x
```

As expected, the returned string is the one from `Explicit.x`. This precedence rule is important when dealing with implicit resolution, but we'll get to that in section 5.1.3.

The final precedence rule to test is for local declarations. Let's modify the `testExplicitImport` method to define a local binding for the name `x`:

Listing 5.5 Inline definitions

```
def testInlineDefinition() {
    val x = "Inline definition x"
    import Explicit.x
    import Wildcard._
    println(x)
  }
```

The first line in the `testInlineDefinition` method declares a local variable named `x`. The next two lines explicitly import and implicitly import `x` bindings from the `Explicit` and `Wildcard` objects, as we saw earlier. Finally, we call `println(x)` and see which binding is selected.

```
scala> test.Test.testInlineDefinition()
Inline definition x
```

Again, even though the import statements come after the `val x` statement, the local variable is chosen based on the binding priorities.

Non-shadowing bindings

It's possible to have two bindings available for the same name. In this case, the compiler will warn you that the name is ambiguous. Here's an example directly from the Scala Language Specification:

```
scala> {
 | val x = 1;
 | {
 | import test.x;
 | x
 | }
 | }
 <console>:11: error: reference to x is ambiguous; it is both defined in
             value res7 and imported subsequently by import test.x
 x
 ^
```

In this example, the name `x` is bound in an outer scope. The name `x` is also imported from the test package in a nested scope. Neither of these bindings shadows the other. The value `x` from the outer scope isn't eligible to shadow within the nested scope, and the imported value `x` doesn't have high enough precedence to shadow.

Why all the emphasis on name resolution within the compiler? Implicit resolution is intimately tied to name resolution, so these intricate rules become important when using implicits. Let's look at the compiler's implicit resolution scheme.

5.1.3 Implicit resolution

The Scala Language Specification declares two rules for looking up entities marked as implicit:

- The implicit entity binding is available at the lookup site with no prefix—that is, not as `foo.x` but only `x`.
- If there are no available entities from this rule, then all `implicit` members on objects belong to the implicit scope of an implicit parameter's type.

The first rule is intimately tied to the binding rules of the previous section. The second rule is a bit more complex and we'll look into it in section 5.1.4.

First, let's look at our earlier example of implicit resolution:

```
scala> def findAnInt(implicit x : Int) = x
findAnInt: (implicit x: Int)Int

scala> implicit val test = 5
test: Int = 5
```

The `findAnInt` method is declared with an implicit parameter list of a single integer. The next line defines a `val test` with the `implicit` marker. This makes the identifier, `test`, available on the local scope with no prefix. If we were to write `test` in the REPL, it would return the value `5`. When we write this method call, `findAnInt`, the compiler will rewrite it as `findAnInt(test)`. This lookup uses the binding rules we examined earlier.

The second rule for implicit lookup is used when the compiler can't find any available implicits using the first rule. In this case, the compiler will look for implicits defined within any object in the *implicit scope* of the type it's looking for. The implicit scope of a type is defined as all companion modules that are associated with that type. This means that if the compiler is looking for a parameter to the method `def foo(implicit param : Foo)`, that parameter will need to conform to the type `Foo`. If no value of type `Foo` is found using the first rule, then the compiler will use the *implicit scope* of `Foo`. The implicit scope of `Foo` would consist of the companion object to `Foo`.

Let's look at the following listing:

Listing 5.6 Companion object and implicit lookup

```
scala> object holder {
     | trait Foo
     | object Foo {
     |   implicit val x = new Foo {
     |     override def toString = "Companion Foo"
     |   }
     | }
     | }
defined module holder

scala> import holder.Foo
import holder.Foo

scala> def method(implicit foo : Foo) = println(foo)
method: (implicit foo: holder.Foo)Unit

scala> method
Companion Foo
```

The `holder` object is used so we can define a trait and companion object within the REPL, as described in section 2.1.2. Inside, we define a trait `Foo` and companion object `Foo`. The companion object `Foo` defines a member `x` of type `Foo` that's available for implicit resolution. Next we import the `Foo` type from the `holder` object into the current scope. This step isn't necessary, it's done to simplify the method definition. Next is the definition of `method`. The method takes an implicit parameter of type `Foo`.

When called with no argument lists, the compiler will use the `implicit val x` defined on the companion.

Because the implicit scope is looked at second, we can use the implicit scope to store default implicits while allowing users to import their own overrides as necessary. We'll investigate this a bit further in section 7.2.

As stated previously, the implicit scope of a type `T` is the set of companion objects for all types associated with the type `T`—that is, there's a set of types that are associated with `T`. All of the companion objects for these types are searched during implicit resolution. The Scala Language Specification defines association as any class that's a base class of some *part* of type `T`. The parts of type `T` are:

- The subtypes of `T` are all parts of `T`. If type `T` is defined as `A` with `B` with `C`, then `A`, `B`, and `C` are all parts of the type `T` and their companion objects will be searched during implicit resolution for type `T`.
- If `T` is parameterized, then all type parameters and their parts are included in the parts of type `T`. For example, an implicit search for the type `List[String]` would look in `List`'s companion object and `String`'s companion object.
- If `T` is a singleton type `T`, then the parts of the type `p` are included in the parts of type `T`. This means that if the type `T` lives inside an object, then the object itself is inspected for implicits. Singleton types are covered in more detail in section 6.1.1.
- If T is a type projection `S#T`, then the parts of `S` are included in the parts of type `T`. This means that if type `T` lives in a class or trait, then the class or trait's companion objects are inspected for implicits. Type projections are covered in more detail in section 6.1.1.

The implicit scope of a type includes many different locations and grants a lot of flexibility in providing handy implicit resolution.

Let's look at a few of the more interesting cases of implicit scope.

IMPLICIT SCOPE VIA TYPE PARAMETERS

The Scala language defines the implicit scope of a type to include the companion objects of all types or subtypes included in the type's parameters. This means, for example, that we can provide an implicit value for `List[Foo]` by including it in the type `Foo`'s companion object. Here's an example:

```
scala> object holder {
     |   trait Foo
     |   object Foo {
     |     implicit val list = List(new Foo{})
     |   }
     | }
defined module holder

scala> implicitly[List[holder.Foo]]
res0: List[holder.Foo] = List(holder$Foo$$anon$1@2ed4a1d3)
```

The `holder` object is used, again, to create companion objects in the REPL. The `holder` object contains a trait `Foo` and its companion object. The companion object contains an implicit definition of a `List[Foo]` type. The next line calls Scala's `implicitly` function. We can use this function to look up a type using the current implicit scope. The `implicitly` function is defined as `def implicitly[T](implicit arg : T) = arg`. It uses the type parameter `T` to allow us to reuse it for every type we're looking for. We'll cover type parameters in more detail in section 6.2. The call to `implicitly` for the type `List[holder.Foo]` returns the implicit list defined within `Foo`'s companion object.

This mechanism is used to implement *type traits* sometimes called *type classes*. Type traits describe generic interfaces using type parameters such that implementations can be created for any type. For example, we can define a `BinaryFormat[T]` type trait. This trait can be implemented for a given type to describe how it should be serialized into a binary format. Here's an example interface:

```
trait BinaryFormat[T] {
  def asBinary(entity: T) : Array[Bytes]
}
```

The `BinaryFormat` trait defines one method, `asBinary`. This method takes in an instance of the type parameter and returns an array of bytes representing that parameter. Code that needs to serialize objects to disk can now attempt to find a `BinaryFormat` type trait via implicits. We can provide an implementation for our type `Foo` by providing an `implicit` in `Foo`'s companion object, as follows:

```
trait Foo {}
object Foo {
  implicit lazy val binaryFormat = new BinaryFormat[Foo] {
    def asBinary(entity: Foo) = "serializedFoo".toBytes
  }
}
```

The `Foo` trait is defined as an empty trait. Its companion object is defined with an `implicit val` that holds the implementation of the `BinaryFormat`. Now, when code that requires a `BinaryFormat` sees the type `Foo`, it will be able to find the `BinaryFormat` implicitly. The details of this mechanism and design techniques are discussed in detail in section 7.2.

Implicit lookup from type parameters enables elegant type trait programming. Nested types provides another great means to supply implicit arguments.

IMPLICIT SCOPE VIA NESTING

Implicit scope also includes companion objects from outer scopes if a type is defined in an inner scope. This allows us to provide a set of handy implicits for a type in the outer scope. Let's look at an example.

```
scala> object Foo {
     | trait Bar
     | implicit def newBar = new Bar {
     |   override def toString = "Implicit Bar"
```

```
     | }
     | }
defined module Foo

scala> implicitly[Foo.Bar]
res0: Foo.Bar = Implicit Bar
```

The object `Foo` is the outer type. Inside is defined the trait `Bar`. The `Foo` object also defines an implicit method that creates an instance of the `Bar` trait. When calling `implicitly[Foo.Bar]`, the implicit value is found from a search of the `Foo` outer class. This technique is similar to placing implicits directly in a companion object. Defining implicits for nested types is convenient when the outer scope contains several subtypes. We can use this technique in situations where we can't create an implicit on a companion object.

Scala objects can't have companion objects for implicits. Because of this, implicits associated with the object's type, that are desired on the implicit scope of that object's type, must be provided from an outer scope. Here's an example:

```
scala> object Foo {
     |   object Bar { override def toString = "Bar" }
     |   implicit def b : Bar.type = Bar
     | }
defined module Foo

scala> implicitly[Foo.Bar.type]
res1: Foo.Bar.type = Bar
```

The object `Bar` is nested inside the object `Foo`. The object `Foo` also defines an implicit that returns `Bar.type`. Now, when calling `implicitly[Foo.Bar.type]`, the object `Bar` is returned. This mechanism allows defining an implicit for objects.

An additional case of nesting that may surprise those not used to it is the case of package objects. As of Scala 2.8, objects can be defined as package objects. A package object is an object defined using the `package` keyword. It's convention in Scala to locate all package objects in a file called `package.scala` in a directory corresponding to the package name.

Any class that's defined within a package is nested inside the package. Any implicits defined on a package object will be on the implicit scope for all types defined inside the package. This provides a handy location to store implicits rather than defining companion objects for every type in a package, as shown in the following example:

```
package object foo {
  implicit def foo = new Foo
}

package foo {
  class Foo {
    override def toString = "FOO!"
  }
}
```

The package object foo is declared with a single implicit that returns a new instance of the Foo class. Next, the class Foo is defined within the package foo. In Scala, packages can be defined in multiple files and the types defined in each source file is aggregated to create the complete package. There can only be one package object defined in all source files for any given package. The Foo class has an overridden toString method that will print the string "Foo!". Let's compile the foo package and use it in the REPL, as follows:

```
scala> implicitly[foo.Foo]
res0: foo.Foo = FOO!
```

Without importing the package object or its members, the compiler can find the implicit for the foo.Foo object. It's common in Scala to find a set of implicit definitions within the package object for a library. Usually this package object also contains implicit views, a mechanism for converting between types.

5.2 Enhancing existing classes with implicit views

An implicit view is an automatic conversion of one type to another to satisfy an expression. An implicit view definition takes the general form: implicit def <myConversionName>(<argumentName> : OriginalType) : ViewType. The previous conversion would implicitly convert a value of OriginalType to a value of ViewType if available on the implicit scope.

Let's look at a simple example attempting to convert an integer to a string:

```
scala> def foo(msg : String) = println(msg)
foo: (msg: String)Unit

scala> foo(5)
<console>:7: error: type mismatch;
 found   : Int(5)
 required: String
       foo(5)
```

The foo method is defined to take a String and print it to the console. The call to foo using the value 5 fails, as there's a type mismatch. An implicit view can make this succeed. Let's define one:

```
scala> implicit def intToString(x : Int) = x.toString
intToString: (x: Int)java.lang.String

scala> foo(5)
5
```

The method intToString is defined using the implicit keyword. It takes a single value of type Int and returns a String. This method is the implicit view, and is commonly referred to as the view Int => String. Now, when calling the foo method with the value 5, it prints the string 5. The compiler detected that the types did not conform and that there was a single implicit view that could correct the situation.

Implicit views are used in two situations:

- If an expression doesn't meet the type expected by the compiler, the compiler will look for an implicit view that would make it meet the expected type. An example of this would be passing a value of type `Int` to a function that expects a `String` would require an implicit view of `String => Int` in scope.
- Given a selection `e.t`, where selection means a member access, such that `e`'s type doesn't have a member `t`, the compiler will look for an implicit view that will apply to `e` and whose resulting type contains a member `t`. If we try to call method `foo` on a `String`, then the compiler will look for an implicit view from `String` that can be used to make the expression compile. The expression `"foo".foo()` would require an implicit view like the following: `implicit def stringToFoo(x : String) = new { def foo() : Unit = println("foo") }`.

The implicit scope used for implicit views is the same as for implicit parameters. But when the compiler is looking for type associations, it uses the type it's attempting to convert from, not the type it's attempting to convert to. Let's look at an example:

```
scala> object test {
     |   trait Foo
     |   trait Bar
     |   object Foo {
     |     implicit def fooToBar(foo : Foo) = new Bar {}
     |   }
     | }
defined module test

scala> import test._
import test._
```

The `test` object is a scoping object used so we can create a companion object in the REPL. This contains the `Foo` and `Bar` traits as well as a companion object to `Foo`. The companion object to `Foo` contains an implicit view from `Foo` to `Bar`. Remember that when the compiler is looking for implicit views, the type it's converting *from* defines the implicit scope. This means the implicit views defined in `Foo`'s companion object will be inspected only when attempting to convert an expression of type `Foo` to some other expression. Let's try this out by defining a method that expects the type `Bar`.

```
scala> def bar(x : Bar) = println("bar")
bar: (x: test.Bar)Unit
```

The `bar` method takes a `bar` and prints the string `bar`. Let's try to call it with a value of `foo` and see what happens:

```
scala> val x = new Foo {}
x: java.lang.Object with test.Foo = $anon$1@15e565bd

scala> bar(x)
bar
```

The `x` value is of type `Foo`. The expression `bar(x)` triggers the compiler to look for an implicit view. Because the type of `x` is `Foo`, the compiler look in associated types of `Foo` for implicit views. Finding the `fooToBar` view, the compiler inserts the necessary transformation and the method compiles successfully.

This style of implicits allows us to adapt libraries to other libraries, or add our own convenience methods to types. It's a common practice in Scala to adapt Java libraries so that they work well with the Scala standard library. For example, the standard library defines a `scala.collection.JavaConversions` module that helps the Java collections library interoperate with the Scala collections library. This module is a set of implicit views that can be imported into the current scope to allow automatic conversion between Java collections and Scala collections and to "add" methods to the Java collections. Adapting Java libraries, or third party libraries, into your project using implicit views is a common idiom in Scala. Let's look at an example.

We'd like to write a wrapper around the `java.security` package for easier usage from Scala. Specifically, we want to simplify the task of running privileged code using `java.security.AccessController`. The `AccessController` class provides the static method `doPrivileged`, which allows us to run code in a privileged permission state. The `doPrivileged` method has two variants, one that grants the current context's permissions to the privileged code and one that takes an `AccessControlContext` containing the privileges to grant the privileged code. The `doPrivileged` method takes an argument of type `PrivilegedExceptionAction` which is a trait that defines one method: `run`. The trait is similar to Scala's `Function0` trait, and we'd like to be able to use an anonymous function when calling the `doPrivileged` method.

Let's create an implicit view from a `Function0` type to a `doPrivileged` method:

```
object ScalaSecurityImplicits {
  implicit def functionToPrivilegedAction[A](func : Function0[A]) =
    new PrivilegedAction[A] {
      override def run() = func()
    }
}
```

This defines an object `ScalaSecurityImplicits` which contains the implicit view. The implicit view `functionToPrivilegedAction` takes a `Function0` and returns a new `PrivilegedAction` object such that the `run` method calls the function. Let's use this implicit:

```
scala> import ScalaSecurityImplicits._
import ScalaSecurityImplicits._

scala> AccessController.doPrivileged( () =>
     | println("This is privileged"))
This is privileged
```

The first statement imports the implicit view into scope. Next, the call to `doPrivileged` passed the anonymous function `() => println("this is privileged")`. Again, the compiler sees that the anonymous function doesn't match the expected type. The compiler then looks and finds the implicit view defined and imported from `ScalaSecurityImplicits`. This technique can also be used when wrapping Java objects with Scala objects

It's common to write a wrapper class for existing Java libraries that add more advanced Scala idioms. Scala implicits can be used to convert from the original type into the wrapped type and vice versa. For example, let's look at adding some convenience methods onto the `java.io.File` class.

We'd like to provide a convenience notation for `java.io.File` so that the `/` operator can be used to create new file objects. Let's create the wrapper class that will provide the `/` operator:

```
class FileWrapper(val file: java.io.File) {
    def /(next : String) = new FileWrapper(new java.io.File(file, next))
    override def toString = file.getCanonicalPath
}
```

The class `FileWrapper` takes a `java.io.File` in its constructor. It provides one new method `/` that takes a string and returns a new `FileWrapper` object. The newly returned `FileWrapper` object points to a file with the name specified to the `/` method inside the directory of the original file. For example, if the original `FileWrapper`, called `file`, pointed at the `/tmp` directory, then expression `file / "mylog.txt"` will return a `FileWrapper` object that points at the `/tmp/mylog.txt` file. We'd like to use implicits to automatically convert between `java.io.File` and `FileWrapper`, so let's add an implicit view to `FileWrapper`'s companion object:

```
object FileWrapper {
  implicit def wrap(file : java.io.File)  = new FileWrapper(file)
}
```

The `FileWrapper` companion object defines one method, `wrap`, which takes a `java.io.File` and returns a new `FileWrapper`. Let's look at an example usage in the REPL:

```
scala> import FileWrapper.wrap
import FileWrapper.wrap

scala> val cur = new java.io.File(".")
cur: java.io.File = .

scala> cur / "temp.txt"
res0: FileWrapper = .../temp.txt
```

The first line imports the implicit view into scope. The next line creates a new `java.io.File` object with the string `"."`. This string denotes that the file object should point to the current directory. The last line calls the `/` method against a `java.io.File`. The compiler doesn't find this method on a standard `java.io.File` and looks for an implicit view that would enable this line to compile. Finding the `wrap` method in scope, the compiler wraps the `java.io.File` into a `FileWrapper` and calls the `/` method. The resulting `FileWrapper` object is returned.

This mechanism is a great way to append methods onto existing Java classes, or any library. We have the performance overhead of the wrapper object instantiation, but the `HotSpot` optimizer may mitigate this. I say "may" here because there's no guarantee that the `HotSpot` optimizer will remove the wrapper allocation, but in some microbenchmarks this will occur. Again, it's best to profile an application to determine critical regions rather than assuming HotSpot will take care of allocations.

One issue with the FileWrapper is that calling its / method will return another FileWrapper object This means we can't pass the result directly into a method that expects a vanilla java.io.File. The / method could change to instead return a java.io.File object, but Scala also provides another solution. When passing a FileWrapper to a method that expects a java.io.File type, the compiler will begin a search for a valid implicit view. As stated earlier, this search will include the companion object for the FileWrapper type itself. Let's add an unwrap implicit view to the companion object and see if this works:

```
object FileWrapper {
  implicit def wrap(file : java.io.File) = new FileWrapper(file)
  implicit def unwrap(wrapper : FileWrapper) = wrapper.file
}
```

The FileWrapper companion object now contains two methods: wrap and unwrap. The unwrap method takes an instance of FileWrapper and returns the wrapped java.io.File type. We'll test this out in the REPL:

```
scala> import test.FileWrapper.wrap
import test.FileWrapper.wrap

scala> val cur = new java.io.File(".")
cur: java.io.File = .

scala> def useFile(file : java.io.File) = println(file.getCanonicalPath)
useFile: (file: java.io.File)Unit

scala> useFile(cur / "temp.txt")
/home/jsuereth/projects/book/scala-in-depth/chapter5/wrappers/temp.txt
```

The first line imports the wrap implicit view. The next line construct a java.io.File object pointing to the current directory. The third line defines a useFile method. This method expects an input of type java.io.File and will print the path to the file. The last line calls the useFile method with the expression: cur / "temp.txt". Again, the compile sees the / method call and looks for an implicit view to resolve the expression. The resulting type of the expression is a FileWrapper, but the useFile method requires a java.io.File. The compiler performs another implicit lookup using the type Function1[java.io.File, FileWrapper]. This search finds the unwrap implicit view on FileWrapper's companion object. The types are now satisfied and the compiler has completed the expression. The runtime evaluation yields the correct string.

Notice that utilizing the unwrap implicit view doesn't require an import, as needed for the wrap method. This is because the wrap implicit view was used when the compile did not know the required type to satisfy the cur / "temp.txt" expression; therefore it looked for only local implicits, as java.io.File has no companion object. This feature allows us to provide a wrapper object with additional functionality and near-invisible conversions to and from the wrapper.

Take care when providing additional functionality to existing classes using implicit views. This mechanism makes it much harder to determine if there's a name conflict across differing implicit views of a type. It also has a performance penalty that may not be mitigated by the `HotSpot` optimizer. Finally, for folks not using a modern Scala IDE, it can be difficult to determine which implicit views are providing methods used in a block of code.

Rule 13

Avoid implicit views

Implicit views are the most abused feature in Scala. While they seem like a good idea in a lot of situations, Scala provides better alternatives in most cases. Using too many implicit views can greatly increase the ramp-up time of new developers on a code base. While useful, they should be limited to situations where they are the right solution.

Scala implicit views provide users with the flexibility to adapt an API to their needs. Using wrappers and companion object implicit views can drastically ease the pain of integrating libraries with varied but similar interfaces or can allow developers to add functionality to older libraries. Implicit views are a key component in writing expressive Scala code, and should be handled with care.

Implicits also have an interesting interaction with another Scala feature—default parameters.

5.3 Utilize implicit parameters with defaults

Implicit arguments provide a great mechanism to ensure that users don't have to specify redundant arguments. They also work well with default parameters. In the event that no parameter is specified and no implicit value is found using implicit resolution, the default parameter is used. This allows us to create default parameters that remove redundant ones while still allowing users to provide different parameters.

For example, let's implement a set of methods designed to perform matrix calculations. These methods will utilize threads to parallelize work when performing calculations on matrices. But as a library designer, we don't know where these methods will be called. They may be operating within a context where threading isn't allowed, or they may already have their own work queue set up. We want to allow users to tell us how to use threads in their context but provide a default for everyone else.

Let's start by defining the `Matrix` class:

Listing 5.7 Simple `Matrix` class

```
class Matrix(private val repr : Array[Array[Double]]) {
  def row(idx : Int) : Seq[Double] = {
    repr(idx)
  }
  def col(idx : Int) : Seq[Double] = {
    repr.foldLeft(ArrayBuffer[Double]()) {
      (buffer, currentRow) =>
        buffer.append(currentRow(idx))
        buffer
    } toArray
```

```
  }

  lazy val rowRank = repr.size
  lazy val colRank = if(rowRank > 0) repr(0).size else 0
  override def toString = "Matrix" + repr.foldLeft(") {
    (msg, row) => msg + row.mkString("\n|", " | ", "|")
  }
}
```

The `Matrix` class takes an array of double values and provides two similar methods: `row` and `col`. These methods take an index and return an array of the values for a given matrix row or column respectively. The `Matrix` class also provides `rowRank` and `colRank` values which return the number of rows and columns in the matrix respectively. Finally the `toString` method is overridden to create a prettier output of the matrix.

The `Matrix` class is complete and ready for a parallel multiplication algorithm. Let's start by creating an interface we can use in our library for threading:

```
trait ThreadStrategy {
  def execute[A](func : Function0[A]) : Function0[A]
}
```

The `ThreadStrategy` interface defines one method, `execute`. This method takes a function that returns a value of type `A`. It also returns a function that returns a value of type `A`. The returned function should return the same value as the passed-in function, but could block the current thread until the function is calculated on its desired thread. Let's implement our matrix calculation service using this `ThreadStrategy` interface:

```
object MatrixUtils {
  def  multiply(a: Matrix,
                b: Matrix)(
                implicit threading: ThreadStrategy): MatrixN = {
      ...
  }
}
```

The `MatrixUtils` object contains the method `multiply`. The method takes two `Matrix` classes, assumed to have the correct dimensions, and will return a new matrix that's the multiplication of the passed-in matrices. `Matrix` multiplication involves multiplying the elements in `Matrix` `a`'s rows by the elements in `Matrix` `b`'s columns and adding the results. This multiplication and summation is done for every element in the resulting matrix. A simple way to parallelize this is to compute each element of the result matrix on a separate thread. The algorithm for the `MatrixUtils.multiply` method is simple:

- Create a buffer to hold results.
- Create a closure that will compute a single value for a row/column pair and place it in the buffer.
- Send the closures created to the `ThreadStrategy` provided.

- Call the functions returned from ThreadStrategy to ensure they have completed.
- Wrap the buffer in a Matrix class and return it.

Let's start with creating the buffer:

```
def  multiply(a: Matrix,
              b: Matrix)(
              implicit threading : ThreadStrategy): Matrix = {
    assert(a.colRank == b.rowRank)
    val buffer = new Array[Array[Double]](a.rowRank)
    for ( i <- 0 until a.rowRank ) {
      buffer(i) = new Array[Double](b.colRank)
    }
    ...
 }
```

The initial assert statement is used to ensure that the Matrix objects passed in are compatible for multiplication. By definition, the number of columns in Matrix a must equal the number of rows in Matrix b. We then construct an array of arrays to use as the buffer. The resulting matrix will have the same number of rows as Matrix a and the same number of columns as Matrix b. Now that the buffer is ready, let's create a set of closures in the following listing that will compute the values and place them in the buffer:

Listing 5.8 Matrix multiplication

```
def  multiply(a: Matrix,
              b: Matrix)(
              implicit threading : ThreadStrategy) : Matrix = {
   ...
    def computeValue(row : Int, col : Int) : Unit = {
       val pairwiseElements =
         a.row(row).zip(b.col(col))
       val products =
         for((x,y) <- pairwiseElements)
         yield x*y
       val result = products.sum
       buffer(row)(col) = result
    }
...
```

The computeValue helper method takes a row and a column attribute and computes the value in the buffer at that row and column. The first step is matching the elements of the row of a with the elements of the column of b in a pairwise fashion. Scala provides the zip function which, given two collections, will match their elements. Next, the paired elements are multiplied to create a list of the products of each element. The final calculation takes a sum of all the products. This final value is placed into the correct row and column in the buffer. The next step is to take this method and construct a function for every row and column in the resulting matrix and pass these functions to the threading strategy, as follows:

```
val computations = for {
   i <- 0 until a.rowRank
   j <- 0 until b.colRank
 } yield threading.execute { () => computeValue(i,j) }
```

This `for` expression loops every row and column in the resulting matrix and passes a function into the `ThreadStrategy` parameter threading. The `() =>` syntax is used when creating anonymous function objects that take no arguments, required by the type `Function0`. After farming out the work to threads, the multiply method must ensure that all work is complete before returning results. We do this by calling each method returned from the `ThreadStrategy`.

```
def  multiply(a: Matrix,
              b: Matrix)(
              implicit threading : ThreadStrategy) : Matrix = {
  ...
  computations.foreach(_())
  new Matrix(buffer)
}
```

The last portion of the multiple method ensures all work is completed and returns the result `Matrix` built from the buffer object. Let's test this in the REPL, but first we need to implement the `ThreadStrategy` interface. Let's create a simple version that executes all work on the current thread:

```
object SameThreadStrategy extends ThreadStrategy {
   def execute[A](func : Function0[A]) = func
 }
```

The `SameThreadStrategy` ensures that all passed-in work operates on the calling thread by returning the original function. Let's test out the `multiply` method in the REPL, as follows:

```
scala> implicit val ts = sameThreadStrategy
ts: ThreadStrategy.sameThreadStrategy.type = ...

scala> val x = new Matrix(Array(Array(1,2,3), Array(4,5,6)))
x: library.Matrix =
Matrix
|1.0 | 2.0 | 3.0|
|4.0 | 5.0 | 6.0|

scala> val y = new Matrix(Array(Array(1), Array(1), Array(1)))
y: library.Matrix =
Matrix
|1.0|
|1.0|
|1.0|

scala> MatrixService.multiply(x,y)
res0: library.Matrix =
Matrix
|6.0|
|15.0|
```

The first line is creating an implicit `ThreadStrategy` that will be used for all remaining calculations. We then construct two matrices and multiply the results. The 2 x 3 matrix is multiplied by a 3 x 1 matrix to product a 2 x 1 matrix, as expected. Everything appears to be working correctly with a single thread, so let's create a multithreaded service, as in the following listing:

Listing 5.9 Concurrent strategey

```
import java.util.concurrent.{Callable, Executors}

  object ThreadPoolStrategy extends ThreadStrategy {
    val pool = Executors.newFixedThreadPool(
                 java.lang.Runtime.getRuntime.availableProcessors)
    def execute[A](func : Function0[A] ) = {
      val future = pool.submit(new Callable[A] {
        def call() : A = {
          Console.println("Executing function on thread: " +
                          Thread.currentThread.getName)
          func()
        }
      })
      () => future.get()
    }
  }
```

The first thing the `ThreadPoolStrategy` implementation does is create a pool of threads using Java's `java.util.concurrent.Executors` library. The thread pool is constructed with the number of threads equal to the number of available processors. The `execute` method takes the passed-in function and creates an anonymous `Callable` instance. The `Callable` interface is used in Java's concurrent library to pass work into the thread pool. This returns a `Future` that can be used to determine when the passed-in work is completed. The last line of `execute` returns an anonymous closure that will call `get` on `future`. This call blocks until the original function executes and returns the value returned by the function. Also, every time a function is executed inside the `Callable`, it will print a message informing which thread it's executing on. Let's try this out in the REPL:

```
scala> implicit val ts = ThreadPoolStrategy
ts: ThreadStrategy.ThreadPoolStrategy.type = ...

scala> val x = new Matrix(Array(Array(1,2,3), Array(4,5,6)))
x: library.Matrix =
Matrix
|1.0 | 2.0 | 3.0|
|4.0 | 5.0 | 6.0|

scala> val y = new Matrix(Array(Array(1), Array(1), Array(1)))
y: library.Matrix =
Matrix
|1.0|
|1.0|
|1.0|

scala> MatrixUtils.multiply(x,y)
```

```
Executing function on thread: pool-2-thread-1
Executing function on thread: pool-2-thread-2
res0: library.Matrix =
Matrix
|6.0|
|15.0|
```

The first line creates an implicit `ThreadPoolStrategy` that will be used for all remaining calculations within the REPL session. Again, the `x` and `y` variables are created as 2 x 3 and 3 x 1 matrices, respectively. But the `MatrixService.multiply` now outputs two lines indicating that the calculations for the result matrix are occurring on different threads. The resulting matrix displays the correct values, as before.

Now what if we wanted to provide a default threading strategy for users of the library, and still allow them to override if desired? We can use the default parameter mechanism to provide a default. This will be used if no value is available in the implicit scope, meaning that our users can override the default in a scope by importing or creating their own implicit `ThreadStrategy`. Users can also override the behavior for a single method call by explicitly passing the `ThreadStrategy`. Let's modify the signature of `MatrixService.multiply`:

```
def  multiply(a: Matrix, b: Matrix)(
              implicit threading: ThreadStrategy = SameThreadStrategy
             ) : Matrix = {
  ...
}
```

The `multiply` method now defines the `SameThreadStrategy` as the default strategy. Now when we use this library, we don't have to provide our own implicit `ThreadStrategy`:

```
scala> val x = new Matrix(Array(Array(1,2,3), Array(4,5,6)))
x: library.Matrix =
Matrix
|1.0 | 2.0 | 3.0|
|4.0 | 5.0 | 6.0|

scala> val y = new Matrix(Array(Array(1), Array(1), Array(1)))
y: library.Matrix =
Matrix
|1.0|
|1.0|
|1.0|

scala> MatrixService.multiply(x,y)
res0: library.Matrix =
Matrix
|6.0|
|15.0|
```

Unlike normal default parameters, an implicit parameter list with defaults doesn't need to be specified in the method call with an additional `()`. This means we get the elegance of implicit parameters with the utility of default parameters. We can still utilize implicits as normal:

```
scala> implicit val ts = ThreadPoolStrategy
ts: ThreadStrategy.ThreadPoolStrategy.type = ...

scala> MatrixUtils.multiply(x,y)
Executing function on thread: pool-2-thread-1
Executing function on thread: pool-2-thread-2
res1: library.Matrix =
Matrix
|6.0|
|15.0|
```

The first line creates an implicitly available thread strategy. Now when calling the `MatrixService.multiply` call, the method is using the `ThreadPoolStrategy`. This allows users of the `MatrixService` to decide when to parallelize computations performed with the library. They can do this for a particular scope by providing an implicit or for a single method call by explicitly passing the `ThreadStrategy`.

This technique of creating an implicit value for a scope of computations is a powerful, flexible means of using the strategy pattern. The *strategy pattern* is an idiom where a piece of code needs to perform some operation, but certain behaviors, or execution "strategy," can be swapped into the method. The `ThreadPoolStrategy` is such a behavior that we're passing into our `MatrixUtils` library methods. This same `ThreadPoolStrategy` could be used across different subsections of components in our system. It provides an alternative means of composing behavior than using inheritance, as discussed in section 4.3.

Another good example of implicits with default parameters is reading the lines of a file. In the general case, users don't care if the line endings are `\r`, `\n`, or `\r\n`. However, a complete library would handle all situations. This can be done by providing an implicit argument for the line ending strategy and providing a default value of "don't care."

Implicits provide a great way to reduce boilerplate in code, such as repeated parameters. The most important thing to remember when using them is be careful, which is the topic of the next section.

5.4 Limiting the scope of implicits

The most important aspect of dealing with implicits is ensuring that programmers can understand what's happening in a block of code. Programmers can do this by limiting the places they must check to discover available implicits. Let's look at the possible locations of implicits:

- The companion objects of any associated types, including package objects
- The `scala.Predef` object
- Any imports that are in scope.

As seen in section 1.1.3, Scala will look in the companion objects of associated types for implicits. This behavior is core to the Scala language. Companion and package objects should be considered part of the API of a class. When investigating how to use a new library, check the companion and package objects for implicit conversions that you may use.

Rule 14

Limit the scope of implicits

Because implicit conflicts require explicit passing of arguments and conversions, it's best to avoid them. This can be accomplished by limiting the number of implicits that are in scope and providing implicits in a way that they can overridden or hidden.

At the beginning of every compiled Scala file there's an implicit `import scala.Predef._`. The `Predef` object contains many useful transformations, in particular the implicits used to add methods to the `java.lang.String` type so that it can support the methods required by the Scala Language Specification. It also contains implicits that will convert between Java's boxed types and Scala's unified types for primitives. For example, there's an implicit conversion in `scala.Predef` for `java.lang.Integer => scala.Int`. When coding in Scala, it's a good idea to know the implicits are available in the `scala.Predef` object.

The last possible location for implicits are explicit `import` statements within the source code. Imported implicits can be difficult to track down. They're also hard to document when designing a library. Because these are the only form of implicits that require an explicit `import` statement in every source file they're used, they require the most amount of care.

5.4.1 Creating implicits for import

When defining a new implicit view or parameter that's intended to be explicitly imported, you should ensure the following:

- The implicit view or parameter doesn't conflict with any other implicit.
- The implicit view or parameter's name doesn't conflict with anything in the `scala.Predef` object.
- The implicit view or parameter is *discoverable*, which means that users of the library or module should be able to find the location of the implicit and determine its use.

Because Scala uses scope resolution to look up implicits, if there's a naming conflict between two implicit definitions it can cause issues. These conflicts are hard to detect because implicit views and parameters can be defined in any scope and imported. The `scala.Predef` object has its contents implicitly imported into every Scala file so that conflicts become immediately apparent. Let's look at what happens when there's a conflict:

```
object Time {
  case class TimeRange(start : Long, end : Long)
  implicit def longWrapper(start : Long) = new {
    def to(end : Long) = TimeRange(start, end)
  }
}
```

This defines a `Time` object that contains a `TimeRange` class. An implicit conversion on `Long` provides a `to` method. You can use this method to construct time range objects.

This implicit conflicts with `scala.Predef.longWrapper` which, among other things, provides an implicit view that also has a `to` method. This `to` method returns a `Range` object that can be used in `for` expressions. Imagine a scenario where someone is using this `TimeRange` implicit to construct time ranges, and then desires the original implicit defined in `Predef` for a `for` expression. One way to solve this is to import the `Predef` implicit at a higher precedence level in a lower scope where it's needed. This can be confusing, as shown in the following example:

Listing 5.10 Scoped precedence

```
object Test {
  println(1L to 10L)
  import Time._
  println(1L to 10L)
  def x() = {
    import scala.Predef.longWrapper
    println(1L to 10L)
    def y() = {
      import Time.longWrapper
      println(1L to 10L)
    }
    y()
  }
  x()
}
```

The `Test` object is defined and immediately prints the expression `(1L to 10L)`. The `Time` implicits are imported and the expression is again printed. Next, in a lower scope, the `Predef longWrapper` is imported and the expression is printed. Finally, in yet a lower scope, the `Time longWrapper` is imported and the expression is again printed. The result of this objects construction is:

```
scala> Test
NumericRange(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
TimeRange(1,10)
NumericRange(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
TimeRange(1,10)
res0: Test.type = Test$@2d34ab9b
```

The first `NumericRange` result is the expression `(1L to 10L)` before any import statements. The second `TimeRange` result is after the `Time` implicit conversion is imported. The next `NumericRange` result is from the nested scope in method `x()` and the final `TimeRange` result is the result of the statement in the deeply nested `y()` method. If the `Test` object contained a lot of code such that all these scopes were not visible within a single window, it would be hard to figure out what the result of the expression `(1L to 10L)` would return at any particular point. Avoid this kind of confusion. The best way is to avoid conflicts across implicit views, but sometimes this is difficult. In those cases, it's better to pick one conversion to be implicit and use the other explicitly.

Making implicits discoverable also helps make code readable, as it helps a new developer determine what is and should be happening in a block of code. Making implicits discoverable is important when working on a team. Within the Scala community, it's common practice to limit importable implicits into one of two places:

- Package objects
- Singleton objects that have the `postfix Implicits`

Package objects make a great place to store implicits because they're already on the implicit scope for types defined within the package. Users need to investigate the package object for implicits relating to the package. Placing implicit definitions that need explicit import on the package object means that there's a greater chance a user will find the implicits and be aware of them. When providing implicits via package object, make sure to document if they require explicit imports for usage.

A better option to documenting explicit import of implicits is to avoid import statements altogether.

5.4.2 Implicits without the import tax

Implicits work well without requiring any sort of import. Their secondary lookup rules, which inspect companion objects of associated types, allow the definition of implicit conversions and values that don't require explicit `import` statements for these implicit values. With some creative definitions, expressive libraries can be defined that make the full use of implicits without requiring any imports. Let's look at an example of this: a library for expressing complex numbers.

Complex numbers are numbers that have a rational and imaginary part to them. The imaginary part is the part multiplied by the square root of -1, also known as *i* (or *j* for electrical engineers). This is simple to model using a case class in Scala:

```
package complexmath
case class ComplexNumber(real : Double, imaginary : Double)
```

The `ComplexNumber` class defines a real component of type `Double` called `real`. The `ComplexNumber` class also defines an imaginary component of type `Double` called `imaginary`. This class represents complex numbers using floating point arithmetic for the component parts. Complex numbers allow addition and multiplication. Let's take a look at those methods:

Listing 5.11 `ComplexNumber` class

```
package complexmath

case class ComplexNumber(real : Double, imaginary : Double) {
  def *(other : ComplexNumber) =
    ComplexNumber( (real*other.real) + (imaginary * other.imaginary),
                   (real*other.imaginary) + (imaginary * other.real) )
  def +(other : ComplexNumber) =
    ComplexNumber( real + other.real, imaginary + other.imaginary )
}
```

Addition,`+`, is defined such that the real/imaginary component of the sum of two complex numbers is the sum of the real/imaginary components of two numbers. Multiplication,`*`, is more complicated and defined as follows:

- The real component of the product of two complex numbers is the product of their real components added to the product of their imaginary components: `(real*other.real) + (imaginary * other.imaginary)`.
- The imaginary component of the product of two complex numbers is the sum of the product of the real component of one number with the imaginary component of the other number: `(real*other.imaginary) + (imaginary * other.real)`.

The complex number class now supports addition and multiplication. Let's look at the class in action:

```
scala> ComplexNumber(1,0) * ComplexNumber(0,1)
res0: imath.ComplexNumber = ComplexNumber(0.0,1.0)

scala> ComplexNumber(1,0) + ComplexNumber(0,1)
res1: imath.ComplexNumber = ComplexNumber(1.0,1.0)
```

The first line multiplies a real component by an imaginary component and the resulting complex number is imaginary. The second line adds a real component to an imaginary component, resulting in a complex number with both real and imaginary parts. The operators * and + work as desired, but calling the `ComplexNumber` factory method is a bit verbose. This can be simplified using a new notation for complex numbers.

In mathematics, complex numbers are usually represented as a sum of the real and imaginary parts. An example representation of `ComplexNumber(1.0,1.0)` would be `1.0 + 1.0*i`, where `i` is the symbol for the imaginary number, the square root of –1. This notation would make an ideal syntax for the complex number library. Let's define the symbol `i` to refer to the square root of –1.

```
package object complexmath {
  val i = ComplexNumber(0.0,1.0)
}
```

This defines the `val i` on the package object for `complexmath`. This places the name `i` available within the `complexmath` package and allows it to be imported directly. This name can be used to construct complex numbers from their component parts. But a piece is missing, as shown in the following REPL session:

```
scala> i * 1.0
<console>:9: error: type mismatch;
 found   : Double(1.0)
 required: ComplexNumber
       i * 1.0
```

Attempting to multiply the imaginary number `i` by a `Double` fails because the `ComplexNumber` type only defines multiplication on `ComplexNumber` types. In mathematics, real numbers can be multiplied by complex numbers because a real number can be considered a complex number that has no imaginary component. This property can be emulated in Scala using an implicit conversion from `Double` to `ComplexNumber`:

```
package object complexmath {
  implicit def realToComplex(r : Double) = new ComplexNumber(r, 0.0)
  val i = ComplexNumber(0.0, 1.0)
}
```

The `complexmath` package object now contains the definition for the value `i` as well as an implicit conversion from `Double` to `ComplexNumber` called `realToComplex`. We'd like to limit the usage of this implicit conversion so that it's only used when absolutely needed. Let's try using the `complexmath` package without explicitly importing any implicit conversions:

```
scala> import complexmath.i
import complexmath.i

scala> val x = i*5.0 + 1.0
x: complexmath.ComplexNumber = ComplexNumber(1.0,5.0)
```

The `val x` is declared using the expression `i*5 + 1` and has the type `ComplexNumber` with a real component of `1.0` and an imaginary component of `5.0`. The important thing to note here is that only the name `i` is imported from `complexmath`. The rest of the implicit conversions are all trigged from the `i` object when the compiler first sees the expression `i*5`. The value `i` is known to be a `ComplexNumber` and defines a `*` method that takes another `ComplexNumber`. The literal `5.0` isn't of the type `ComplexNumber`, but `Double`. The compiler issues an implicit search for the type `Double => complexmath.ComplexNumber`. This search finds the `realToComplex` conversion on the package object and applies it. Next the compiler sees the expression `(... : ComplexNumber) + 1.0`. The compiler finds a `+` method defined on `ComplexNumber` that accepts a `ComplexNumber`. The value `1.0` is of type `Double`, not `ComplexNumber` so the compiler issues another implicit search for the type `Double => ComplexNumber`. Again this is found and applied, resulting in the final value for the expression of `ComplexNumber(1.0, 5.0)`.

Notice how the value `i` is used to trigger complex arithmetic. Once a complex number is seen, the compiler can accurately find implicits to ensure that expressions are compiled. The syntax is elegant and concise, and no implicit conversions were needed to make this syntax work. The downside is that the value `i` must be used to begin a `ComplexNumber` expression. Let's look at what happens when `i` appears at the end of the expression:

```
scala> val x = 1.0 + 5.0*i
<console>:6: error: overloaded method value * with alternatives:
  (Double)Double <and>
  (Float)Float <and>
  (Long)Long <and>
  (Int)Int <and>
  (Char)Int <and>
  (Short)Int <and>
```

```
  (Byte)Int
 cannot be applied to (complexmath.ComplexNumber)
       val x = 1 + 5*i
```

The compiler complains about the expression because it can't find a + method defined for the type `Double` that takes a `ComplexNumber`. This issue could be solved by importing the implicit view of `Double => ComplexNumber` into scope:

```
scala> import complexmath.realToComplex
import complexmath.realToComplex

scala> val x = 1.0 + 5.0*i
x: complexmath.ComplexNumber = ComplexNumber(1.0,5.0)
```

The `realToComplex` implicit view is imported first. Now the expression `1 + 5*i` evaluates correctly to a `ComplexNumber(1.0, 5.0)`. The downside is that there's now an additional implicit view in scope for the type `Double`. This can cause issues if other implicit views are defined that provide similar methods to `ComplexNumber`. Let's define a new implicit conversion that adds an `imaginary` method to `Double`.

```
scala> implicit def doubleToReal(x : Double) = new {
     |   def real = "For Reals(" + x + ")"
     | }
doubleToReal: (x: Double)java.lang.Object{def real: java.lang.String}

scala> 5.0 real
<console>:10: error: type mismatch;
 found   : Double
 required: ?{val real: ?}
Note that implicit conversions are not applicable
 because they are ambiguous:
 both method doubleToReal in object $iw of type
   (x: Double)java.lang.Object{def real: java.lang.String}
 and method realToComplex in package complexmath of type
   (r: Double)complexmath.ComplexNumber
 are possible conversion functions from
   Double to ?{val real: ?}
       5.0 real
```

The first statement defines an implicit view on the `Double` type that adds a new type containing a `real` method. The `real` method returns a string version of the `Double`. The next statement attempts to call the `real` method and is unable to do so. The compiler complains about finding ambiguous implicit conversions. The issue here is the `ComplexNumber` type also defines a method `real`, and so the implicit conversion from `Double => ComplexNumber` is getting in the way of our `doubleToReal` implicit conversion. This conflict can be avoided by not importing the `Double => ComplexNumber` conversion:

```
scala> import complexmath.i
import complexmath.i

scala> implicit def doubleToReal(x : Double) = new {
     |   def real = "For Reals(" + x + ")"
     | }
```

```
doubleToReal: (x: Double)java.lang.Object{def real: java.lang.String}

scala> 5.0 real
res0: java.lang.String = For Reals(5.0)
```

The example starts a new REPL session that only imports `complexmath.i`. The next statement redefines the `doubleToReal` conversion. Now the expression `5.0 real` successfully compiles because there's no conflict.

You can use this idiom to successfully create expressive code without all the dangers of implicit conflicts. The pattern takes the following form:

- Define the core abstractions for a library, such as the `ComplexNumber` class.
- Define the implicit conversions needed for expressive code in one of the associated types of the conversion. The `Double => ComplexNumber` conversion was created in the `complexmath` package object which is associated with the `ComplexNumber` type and therefore discovered in any implicit lookup involving the `ComplexNumber` type.
- Define an *entry point* into the library such that implicit conversions are disambiguated after the entry point. In the `complexmath` library, the value `i` is the entry point.
- Some situations require an explicit import. In the `complexmath` library, the entry point `i` allows certain types of expressions but not others that intuition would suggest should be there. For example, `(i * 5.0 + 1.0)` is accepted and `(1.0 + 5.0*i)` is rejected. In this situation, it's acceptable to provide implicit conversions that can be imported from a well-known location. In `complexmath`, this location is the package object.

Following these guidelines helps create expressive APIs that are also discoverable.

5.5 Summary

In this chapter, we discussed the implicit lookup mechanism of Scala. Scala supports two types of implicits: implicit value and implicit views. Implicit values can be used to provide arguments to method calls. Implicit views can be used to convert between types or to allow method calls against a type to succeed. Both implicit values and implicit views use the same implicit resolution mechanism. Implicit resolution uses a two stage process. The first stage looks for implicits that have no prefix in the current scope. The second stage looks in companion objects of associated types. Implicits provide a powerful way to enhance existing classes. They can also be used with default parameters to reduce the noise for method calls and tie behavior to the scope of an implicit value.

Most importantly, implicits provide a lot of power and should be used responsibly. Limiting the scope of implicits and defining them in well-known or easily discoverable locations is key to success. You can do this by providing unambiguous entry points into implicit conversions and expressive APIs. Implicits also interact with Scala's type system in interesting ways. We'll discuss these in chapter 7, but first let's look at Scala's type system.