Chapter 5. Implicits

Implicits are a powerful, if controversial feature in Scala. They are used to reduce boilerplate, to simulate adding new methods to existing types, and to support the creation of *domain-specific languages* (DSLs).

Implicits are controversial because they are “nonlocal” in the source code. You import implicit values and methods into the local scope, except for those that are imported automatically through `Predef`. Once in scope, an implicit might be invoked by the compiler to populate a method argument or to convert a provided argument to the expected type. However, when reading the source code, it’s not obvious when an implicit value or method is being used, which can be confusing to the reader. Fortunately, you learn from experience to be aware when implicits might be invoked, and you learn the APIs that exploit them. Nevertheless, surprises await the beginner.

Understanding how implicits work is fairly straightforward. Most of this long chapter is devoted to example design problems that implicits solve.

Implicit Arguments

In A Taste of Futures, we saw one use of the `implicit` keyword, to label method arguments that the user does not have to provide explicitly. When an implicit argument is omitted, a type-compatible value will be used from the enclosing scope, if available. Otherwise, a compiler error occurs.

Suppose we have a method to compute sales tax where the rate is implicit:

```
def calcTax(amount: Float)(implicit rate: Float): Float = amount * rate
```

In code that uses this method, an implicit value in the local scope will be used:

```
implicit val currentTaxRate = 0.08F
...
val tax = calcTax(50000F)
//
4000.0
```

For simple cases, a fixed `Float` value might be sufficient. However, an application might need to know the location where the transaction takes place, to add on city taxes, for example. Some jurisdictions might offer “tax holidays” to promote shopping during the end of the year holidays.

Fortunately, an implicit method can also be used. To function as an implicit value, it must not take arguments itself, *unless* the arguments are also implicit. Here is a complete example for calculating sales tax:

```
// src/main/scala/progscala2/implicits/implicit-args.sc

// Never use Floats for
money:
def calcTax(amount: Float)(implicit rate: Float): Float = amount * rate

object SimpleStateSalesTax {
  implicit val rate: Float = 0.05F
}

case class ComplicatedSalesTaxData(
  baseRate: Float,
  isTaxHoliday: Boolean,
  storeId: Int)

object ComplicatedSalesTax {
  private def extraTaxRateForStore(id: Int): Float = {
    // From id, determine location, then extra
    taxes...
    0.0F
  }

  implicit def rate(implicit cstd: ComplicatedSalesTaxData): Float =
    if (cstd.isTaxHoliday) 0.0F
    else cstd.baseRate + extraTaxRateForStore(cstd.storeId)
}

{
  import SimpleStateSalesTax.rate

  val amount = 100F
           "Tax on $amount =
  println(s${calcTax(amount)}"                      )
}

{
  import ComplicatedSalesTax.rate
  implicit val myStore = ComplicatedSalesTaxData(0.06F, false, 1010)

  val amount = 100F
           "Tax on $amount =
  println(s${calcTax(amount)}"                      )
}
```

It doesn’t matter that we call `calcTax` inside an interpolated string. The implicit values are still used for the `rate` argument.

For the “complicated” case, we use an implicit method, which itself takes an implicit argument with the data it needs.

Running the script produces this output:

```
Tax on 100.0 = 5.0
Tax on 100.0 = 6.0
```

Using implicitly

`Predef` defines a method called `implicitly`. Combined with a type signature addition, it provides a useful shorthand way of defining method signatures that take a single implicit argument, where that argument is a parameterized type.

Consider the following example, which wraps the `List` method `sortBy`:

```
// src/main/scala/progscala2/implicits/implicitly-
args.sc
import math.Ordering

case class MyList[A](list: List[A]) {
  def sortBy1[B](f: A => B)(implicit ord: Ordering[B]): List[A] =
    list.sortBy(f)(ord)

  def sortBy2[B : Ordering](f: A => B): List[A] =
    list.sortBy(f)(implicitly[Ordering[B]])
}

val list = MyList(List(1,3,5,2,4))

list sortBy1 (i => -i)
list sortBy2 (i => -i)
```

`List.sortBy` is one of several sorting methods available for many of the collections. It takes a function that transforms the arguments into something that satisfies `math.Ordering`, which is analogous to Java's `Comparable` abstraction. An implicit argument is required that knows how to order instances of type `B`.

`MyList` shows two alternative ways of writing a method like `sortBy`. The first implementation, `sortBy1`, uses the syntax we already know. The method takes an additional implicit value of type `Ordering[B]`. For `sortBy1` to be used, there must be an instance in scope that knows how to "order" instances of the desired type `B`. We say that `B` is bound by a "context," in this case, the ability to order instances.

This idiom is so common that Scala provides a shorthand syntax, which is used by the second implementation, `sortBy2`. The type parameter `B : Ordering` is called a *context bound*. It implies the second, implicit argument list that takes an `Ordering[B]` instance.

However, we need to access this `Ordering` instance in the method, but we no longer have a name for it, because it's no longer explicitly declared in the source code. That's what `Predef.implicitly` does for us. Whatever instance is passed to the method for the implicit argument is resolved by `implicitly`. Note the type signature that it requires, `Ordering[B]` in this case.

Note

The combination of a *context bound* and the `implicitly` method is a shorthand for the special case where we need an implicit argument of a parameterized type, where the type parameter is one of the other types in scope (for example, `[B : Ordering]` for an implicit `Ordering[B]` parameter).

Scenarios for Implicit Arguments

It’s important to use implicits wisely and sparingly. Excessive use makes it difficult for the reader to understand what the code is actually doing.

Why use implicit arguments in the first place, especially if they have drawbacks? There are several common idioms implemented with implicit arguments whose benefits fall into two broad categories. The first category is boilerplate elimination, such as providing context information implicitly rather than explicitly. The second category includes constraints that reduce bugs or limit the allowed types that can be used with certain methods with parameterized types. Let’s explore these idioms.

Execution Contexts

We saw in the `Future` example in A Taste of Futures that a second, implicit argument list was used to pass an `ExecutionContext` to the `Future.apply` method:

```
apply[T](body: => T)(implicit executor: ExecutionContext): Future[T]
```

Several other methods also have this implicit argument.

We didn’t specify an `ExecutionContext` when we called these methods, but we imported a global default that the compiler used:

```
import scala.concurrent.ExecutionContext.Implicits.global
```

Passing an “execution context” is one recommended use of implicit arguments. Other example contexts include transactions, database connections, thread pools, and user sessions. Using a method argument permits composition of behavior. Making that argument implicit creates a cleaner API for users.

Capabilities

Besides passing contexts, implicit arguments can be used to control *capabilities*.

For example, an implicit user session argument might contain authorization tokens that control whether or not certain API operations can be invoked on behalf of the user or to limit data visibility.

Suppose you are constructing a menu for a user interface and some menu items are shown only if the user is logged in, while others are shown only if the user isn’t logged in:

```
def createMenu(implicit session: Session): Menu = {
  val defaultItems = List(helpItem, searchItem)
  val accountItems =
    if (session.loggedin()) List(viewAccountItem, editAccountItem
)
    else List(loginItem)
  Menu(defaultItems ++ accountItems)
}
```

Constraining Allowed Instances

Suppose we have a method with parameterized types and we want to constrain the allowed types that can be used for the type parameters.

If the types we want to permit are all subtypes of a common supertype, we can use object-oriented techniques and avoid implicits. Let's consider that approach first.

We saw an example in Call by Name, Call by Value, where we implemented a resource manager:

```
object manage {
  def apply[R <: { def close():Unit }, T](resource: => R)(f: R => T) =
{...}
  ...
}
```

The type parameter `R` must be a subtype of any type with the `close():Unit` method. Or, if we can assume that all resources we'll manage implement a `Closable` trait (recall that traits replace and extend Java interfaces; see Traits: Interfaces and "Mixins" in Scala):

```
trait Closable {
  def close(): Unit
}
...
object manage {
  def apply[R <: Closable, T](resource: => R)(f: R => T) =
{...}
  ...
}
```

This technique doesn't help when there is no common superclass. For that situation, we can use an implicit argument to limit the allowed types. The Scala collections API does this to solve a design problem.

Many of the methods supported by the concrete collections classes are implemented by parent types. For example, `List[A].map(f: A ⇒ B): List[B]` creates a new list after applying the function `f` to each element. The `map` method is supported by most collections. Therefore, it makes sense to implement `map` once in a generic trait, then mix that trait into all the collections that need it (we discussed "mixins" using traits in Traits: Interfaces and "Mixins" in Scala). However, we want to return the same collection type we started with, so how can we tell the one `map` method to do that?

The Scala API uses a convention of passing a "builder" as an implicit argument to `map`. The builder knows how to construct a new collection of the same type. This is what the actual signature of `map` looks like in `TraversableLike`, a trait that is mixed into the collection types that are "traversable":

```
trait TraversableLike[+A, +Repr] extends ... {
  ...
  def map[B, That](f: A => B)(
    implicit bf: CanBuildFrom[Repr, B, That]): That =
{...}
  ...
}
```

Recall that `+A` means that `TraversableLike[A]` is *covariant* in `A`; if `B` is a subtype of `A`, then `TraversableLike[B]` is a subtype of `TraversableLike[A]`.

`CanBuildFrom` is our builder. It's named that way to emphasize the idea that you can build any new collection you want, as long as an implicit builder object exists.

`Repr` is the type of the actual collection used internally to hold the items. `B` is the type of elements created by the function `f`.

`That` is the type parameter of the target collection we want to create. Normally, we want to construct a new collection of the same input kind, perhaps with a different type parameter. That is, `B` may or may not be the same type as `A`. The Scala API defines implicit `CanBuildFroms` for all the built-in collection types.

So, the allowed output collections of a `map` operation are constrained by the existence of corresponding instances of `CanBuildFrom`, declared in scope as `implicit`. If you implement your own collections, you'll want to reuse method implementations like `TraversableLike.map`, so you'll need to create your own `CanBuildFrom` types and import `implicit` instances of them in code that uses your collections.

Let's look at another example. Suppose you're writing a Scala wrapper for a Java database API. Here is an example inspired by Cassandra's API:

```
// src/main/scala/progscala2/implicits/java-database-api.scala

// A Java-like Database API, written in Scala for
convenience.
package progscala2.implicits {
  package database_api {

    case class InvalidColumnName(name: String)
                                "Invalid column name
      extends RuntimeException(s$name"                    )

    trait Row {
      def getInt   (colName: String): Int
      def getDouble(colName: String): Double
      def getText  (colName: String): String
    }
  }

  package javadb {
    import database_api._

    case class JRow(representation: Map[String,Any]) extends Row {
      private def get(colName: String): Any =
        representation.getOrElse(colName, throw InvalidColumnName(colName))

      def getInt   (colName: String): Int    = get(colName).asInstanceOf[Int]
      def getDouble(colName: String): Double = get(colName).asInstanceOf[Double
]
      def getText  (colName: String): String = get(colName).asInstanceOf[String
]
    }

    object JRow {
      def apply(pairs: (String,Any)*) = new JRow(Map(pairs :_*))
    }
  }
}
```

I wrote it in Scala for convenience. I used a `Map` as the representation of a row in a result set, but for efficiency, a real implementation would probably use byte arrays.

The key feature is the set of methods named `getInt`, `getDouble`, `getText`, and others that we might have implemented. They handle conversion of the “raw” data for a column into a value of the appropriate type. They will throw a `ClassCastException` if you use the wrong type method for a given column.

Wouldn’t it be nice to have a single `get[T]` method, where `T` is one of the allowed types? That would promote more uniform invocation, where we wouldn’t need a case statement to pick the correct method to call, and we could exploit type inference in many cases.

One of the distinctions Java makes between primitive types and reference types is that we can’t use primitives in parameterized methods like `get[T]`. We would have to use boxed types, like `java.lang.Integer` instead of `int`, but we don’t want the boxing overhead for a high-performance data application!

However, we can do this in Scala:

```
// src/main/scala/progscala2/implicits/scala-database-
api.scala

// A Scala wrapper for the Java-like Database
API.
package progscala2.implicits {
    package scaladb {
    object implicits {
      import javadb.JRow

      implicit class SRow(jrow: JRow) {
        def get[T](colName: String)(implicit toT: (JRow,String) => T): T =
          toT(jrow, colName)
      }

      implicit val jrowToInt: (JRow,String) => Int =
        (jrow: JRow, colName: String) => jrow.getInt(colName)
      implicit val jrowToDouble: (JRow,String) => Double =
        (jrow: JRow, colName: String) => jrow.getDouble(colName)
      implicit val jrowToString: (JRow,String) => String =
        (jrow: JRow, colName: String) => jrow.getText(colName)
    }

    object DB {
      import implicits._

      def main(args: Array[String]) = {
        val row = javadb.JRow("one" -> 1, "two" -> 2.2, "three" -> "THREE!"
)

        val oneValue1: Int      = row.get("one")
        val twoValue1: Double   = row.get("two")
        val threeValue1: String = row.get("three")
        // val fourValue1: Byte    = row.get("four")  // won't
        compile

                 "one1   ->
        println(s$oneValue1"           )
                 "two1   ->
        println(s$twoValue1"           )
                 "three1 ->
        println(s$threeValue1"           )

        val oneValue2   = row.get[Int]("one")
        val twoValue2   = row.get[Double]("two")
        val threeValue2 = row.get[String]("three")
        // val fourValue2    = row.get[Byte]("four")  // won't
        compile

                 "one2   ->
        println(s$oneValue2"           )
                 "two2   ->
        println(s$twoValue2"           )
                 "three2 ->
        println(s$threeValue2"           )
      }
      }
    }
  }
}
```

In the `Implicits` object, we add an implicit class to wrap the Java `JRow` in a type that has the `get[T]` method we want. We call these classes *implicit conversions* and we'll discuss them later in this chapter. For now, just note that our implicit conversion allows the source code to call `get[T]` on a `JRow` instance, as if that method was defined for it.

The `get[T]` method takes two argument lists. The first is the column name to retrieve from the row and the second is an implicit function argument. This function is used to extract the data for the column from a row and convert it to the correct type.

If you read `get[T]` carefully, you'll notice that it references the `jrow` instance that was passed to the constructor for `SRow`. However, that value isn't declared a `val`, so it's not a field of the class. So, how can `get[T]` reference it? Simply because `jrow` is in the scope of the class body.

Note

Sometimes a constructor argument is not declared to be a field (using `val` or `var`), because it doesn't hold state information the type should expose to clients. However, the argument can still be referenced by other members of the type, because it is in the scope of the entire body of the type.

Next we have three implicit values, functions that take a `JRow` and a `String`, where the latter is the column name, and returns a column value of the appropriate type. These functions will be used implicitly in calls to `get[T]`.

Finally, we define an object `DB` to test it. It creates a `JRow`, then calls `get[T]` on it for the three columns. It does this twice. The first time, the `T` is inferred from the types of the variables, such as `oneValue1`. The second time, we omit variable type annotations and use explicit parameter values for `T` in `get[T]`. I actually like the second style better.

To run the code, start `sbt` and type `run-main progscala2.implicits.scaladb.DB`. It will compile the code first, if necessary:

```
> run-main progscala2.implicits.scaladb.DB
[info] Running scaladb.DB
one1   -> 1
two1   -> 2.2
three1 -> THREE!
one2   -> 1
two2   -> 2.2
three2 -> THREE!
[success] Total time: 0 s, ...
```

Note that the source code has commented lines for extracting `Byte` values. If you remove the `//` characters on these lines, you'll get compilation errors. Here is the error for the first line (wrapped to fit):

```
[error] .../implicits/scala-database-api.scala:31: ambiguous implicit
values:
[error]  both value jrowToInt in object Implicits of type =>
           (javadb.JRow, String) => Int
[error]  and value jrowToDouble in object Implicits of type =>
           (javadb.JRow, String) => Double
[error]  match expected type (javadb.JRow, String) => T
[error]       val fourValue1: Byte    = row.get("four")  // won't compile
```

We get one of two possible errors. In this case, because there are implicit conversions in scope and `Byte` is a number like `Int` and `Double`, the compiler could try to use either one, but it disallows the ambiguity. It would be an error anyway, because both functions extract too many bytes!

If no implicits are in scope at all, you'll get a different error. If you temporarily comment out all three implicit values defined in object `Implicits`, you'll get an error like this for each call to `get`:

```
[error] .../implicits/scala-database-api.scala:28:
  could not find implicit value for parameter toT: (javadb.JRow, String) =>
T
[error]       val oneValue1: Int      = row.get("one")
```

To recap, we limited the allowed types that can be used for a parameterized method by passing an implicit argument and only defining corresponding implicit values that match the types we want to allow.

By the way, the API I once wrote that inspired this example made the equivalent of `JRow` and the implicit functions more generic so I could plug in "fake" and real representations of Cassandra data, where the fakes were used for testing.

Implicit Evidence

The previous section discussed using implicit objects to constrain the allowed types that don't all conform to a common supertype, and the objects were also used to help do the work of the API.

Sometimes, we just need to constrain the allowed types and not provide additional processing capability. Put another way, we need "evidence" that the proposed types satisfy our requirements. Now we'll discuss another technique called *implicit evidence* that constrains the allowed types, but doesn't require them to conform to a common supertype.

A nice example of this technique is the `toMap` method available for all traversable collections. Recall that the `Map` constructor wants key-value pairs, i.e., two-tuples, as arguments. If we have a sequence of pairs, wouldn't it be nice to create a `Map` out of them in one step? That's what `toMap` does, but we have a dilemma. We can't allow the user to call `toMap` if the sequence is *not* a sequence of pairs.

The `toMap` method is defined in `TraversableOnce`:

```
trait TraversableOnce[+A] ... {
  ...
  def toMap[T, U](implicit ev: <:<[A, (T, U)]): immutable.Map[T, U
]
  ...
}
```

The implicit argument `ev` is the "evidence" we need to enforce our constraint. It uses a type defined in `Predef` called `<:<`, named to resemble the type parameter constraint `<:`, e.g., `A <: B`.

Recall we said that types with two type parameters can be written in "infix" notation. So, the following two expressions are equivalent:

```
<:<(A, B)
A <:< B
```

In `toMap`, the `B` is really a pair:

```
<:<(A, (T, U))
A <:< (T, U)
```

Now, when we have a traversable collection that we want to convert to a `Map`, the implicit evidence `ev` value we need will be synthesized by the compiler, but only if `A <: (T,U)`; that is, if `A` is actually a pair. If true, then `toMap` can be called and it simply passes the elements of the traversable to the `Map` constructor. However, if `A` is not a pair type, a compiler error is thrown:

```
scala> val l1 = List(1, 2, 3)
l1: List[Int] = List(1, 2, 3)

scala> l1.toMap
<console>:9: error: Cannot prove that Int <:< (T, U).
              l1.toMap
                 ^

scala> val l2 = List("one" -> 1, "two" -> 2, "three" -> 3)
l2: List[(String, Int)] = List((one,1), (two,2), (three,3
))

scala> l2.toMap
res3: scala.collection.immutable.Map[String,Int] =
  Map(one -> 1, two -> 2, three -> 3)
```

Hence, "evidence" only has to exist to enforce a type constraint. We don't have to define an implicit value ourselves to do extra, custom work.

There is a related type in `Predef` for providing evidence that two types are equivalent, called `=:=`. It is less widely used.

Working Around Erasure

With implicit evidence, we didn't use the implicit object in the computation. Rather, we only used its existence as confirmation that certain type constraints were satisfied.

Another example where the implicit object only provides evidence is a technique for working around limitations due to *type erasure*.

For historical reasons, the JVM "forgets" the type arguments for parameterized types. For example, consider the following definitions for an *overloaded* method, which is a set of methods with the same name, but unique type signatures:

```
object C {
  def m(seq: Seq[Int]): Unit = println(s"Seq[Int]: $seq")
  def m(seq: Seq[String]): Unit = println(s"Seq[String]: $seq")
}
```

Let's see what happens in a REPL session:

```
scala> :paste
// Entering paste mode (ctrl-D to
finish)

object M {
  def m(seq: Seq[Int]): Unit = println(s"Seq[Int]: $seq"
)
  def m(seq: Seq[String]): Unit = println(s"Seq[String]: $seq"
)
}
<ctrl-d>

// Exiting paste mode, now
interpreting.

<console>:8: error: double definition:
method m:(seq: Seq[String])Unit and
method m:(seq: Seq[Int])Unit at line 7
have same type after erasure: (seq: Seq)Unit
       def m(seq: Seq[String]): Unit = println(s"Seq[String]: $seq"
)
           ^
```

The `<ctrl-d>` indicates where I entered the Ctrl-D character, which is not echoed to the console, of course.

By the way, try inputting these two method definitions without the `object M` and without using `:paste` mode. You'll see no complaints. That's because the REPL lets you *redefine* types, values, and methods, for your convenience, while the compiler doesn't allow it when compiling regular code. If you forget about this "convenience," you might think you successfully defined two versions of `m`. Using `:paste` mode makes the compiler process the entire input up to the Ctrl-D as it would compile a normal file.

So, the compiler disallows the definitions because they are effectively the same in byte code.

However, we can add an implicit argument to disambiguate the methods:

```
// src/main/scala/progscala2/implicits/implicit-erasure.sc
object M {
  implicit object IntMarker
//  ❶
  implicit object StringMarker

  def m(seq: Seq[Int])(implicit i: IntMarker.type): Unit =           //
❷
    println(s"Seq[Int]: $seq")

  def m(seq: Seq[String])(implicit s: StringMarker.type): Unit =     //
❸
    println(s"Seq[String]: $seq")
}

import M._
//  ❹
m(List(1,2,3))
m(List("one", "two", "three"))
```

❶

Define two special-purpose implicit objects that will be used to disambiguate the methods affected by type erasure.

❷

Redefinition of the method that takes `Seq[Int]`. It now has a second argument list expecting an implicit `IntMarker`. Note the type, `IntMarker.type`. This is how to reference the *type* of a singleton object!

❸

The method for `Seq[String]`.

❹

Import the implicit values and the methods, then use them. The code compiles and prints the correct output.

Now the compiler considers the two `m` methods to be distinct after type erasure. You might wonder why I didn't just use implicit `Int` and `String` values, rather than invent new types. Using implicit values for such common types is not recommended. To see why, suppose that one module in the current scope defines an implicit argument of type `String`, for example, and a "default" implicit `String` value. Then another module in scope defines its own implicit `String` argument. Two things can go wrong. First, suppose the second module doesn't define a "default" implicit value, but expects the user to define an implicit value that's meaningful for the application. If the user doesn't define this value, the other module's implicit value will be used, probably causing unexpected behavior. If the user does define a second implicit value, the two values in scope will be ambiguous and the compiler will throw an error.

At least the second scenario triggers an immediate error rather than allowing unintentional behavior to occur.

The safer bet is to limit your use of implicit arguments and values to very specific, purpose-built types.

Warning

Avoid using implicit arguments and corresponding values of common types like `Int` and `String`. It's more likely that implicits of such types will be defined in multiple places and cause confusing bugs or compilation errors when they are imported into the same scope.

We'll discuss type erasure in more detail in Chapter 14.

Improving Error Messages

Let's return to the collections API and `CanBuildFrom` for a moment. What happens if we attempt to use `map` for a a custom target type that doesn't have a corresponding `CanBuildFrom` defined for it?

```
scala> case class ListWrapper(list: List[Int])
defined class ListWrapper

scala> List(1,2,3).map[Int,ListWrapper](_ * 2)
<console>:10: error: Cannot construct a collection of type ListWrapper
with elements of type Int based on a collection of type List[Int].
              List(1,2,3).map[Int,ListWrapper](_*2)
                                               ^
```

The explicit type annotations on `map`, `map[Int,ListWrapper]`, ensured that an output object of type `ListWrapper` is what we wanted, rather than the default `List[Int]`. It also triggered the error we wanted. Note the descriptive error message, "Cannot construct a collection of type ListWrapper with elements of type Int based on a collection of type List[Int]." This is *not* the usual default message the compiler emits when it can't find an implicit value for an implicit argument. Instead, `CanBuildFrom` is declared with an *annotation* (like Java annotations), called `scala.annotation.implicitNotFound`, which is used to specify a format string for these error messages (see Annotations for more on Scala annotations). The `CanBuildFrom` declaration looks like this:

```
@implicitNotFound(msg =
  "Cannot construct a collection of type ${To} with elements of type
  ${Elem}"                                                          +
  " based on a collection of type
  ${From}."                           )
trait CanBuildFrom[-From, -Elem, +To] {...}
```

You can only use this annotation on types intended for use as implicit values for satisfying implicit arguments. You can't use it to annotate methods that take implicit arguments, like our `SRow.get[T]` method.

This is a second reason for creating custom types for implicits, rather than using more common types, like `Int` or `String`, or even function types like the `(JRow,String) => T` used in our `SRow` example. With custom types, you can provide helpful error messages for your users.

Phantom Types

We saw the use of implicit arguments that add behavior, such as `CanBuildFrom`, and the use of the existence of an implicit value that allows an API call to be used, such as `toMap` on collections and implicit instances of the `<:<` type.

The next logical step in the progression is the removal of instances altogether, where just the *existence* of a type is all that's required. When such types are defined that have no instances at all, they are called *phantom types*. That's a fancy name, but it just means that we only care that the type exists. It functions as a "marker." We won't actually use any instances of it.

The use of phantom types we're about to discuss has nothing to do with implicits, but it fits nicely in the discussion of design problems that we've been solving.

Phantom types are very useful for defining work flows that must proceed in a particular order. As an example, consider a simplified payroll calculator. In US tax law, payroll deductions for insurance premiums and contributions to certain retirement savings (401k) accounts can be subtracted before calculating payroll taxes. So, a payroll calculator must process these so-called "pre-tax" deductions first, then calculate the tax deductions, then calculate post-tax deductions, if any, to determine the net pay.

Here is one possible implementation:

```
// src/main/scala/progscala2/implicits/phantom-
types.scala

// A workflow for payroll
calculations.

package progscala.implicits.payroll

sealed trait PreTaxDeductions
sealed trait PostTaxDeductions
sealed trait Final

// For simplicity, use Float for money. Not
recommended...
case class Employee(
  name: String,
  annualSalary: Float,
                   // For simplicity, just 1 rate covering all
  taxRate: Float,  taxes.
  insurancePremiumsPerPayPeriod: Float,
  _401kDeductionRate: Float,
// A pretax, retirement savings plan in the
USA.
  postTaxDeductions: Float)

case class Pay[Step](employee: Employee, netPay: Float)

object Payroll {
  // Biweekly paychecks. Assume exactly 52 weeks/year for
  simplicity.
  def start(employee: Employee): Pay[PreTaxDeductions] =
    Pay[PreTaxDeductions](employee, employee.annualSalary / 26.0F)

  def minusInsurance(pay: Pay[PreTaxDeductions]): Pay[PreTaxDeductions] = {
    val newNet = pay.netPay - pay.employee.insurancePremiumsPerPayPeriod
    pay copy (netPay = newNet)
  }

  def minus401k(pay: Pay[PreTaxDeductions]): Pay[PreTaxDeductions] = {
    val newNet = pay.netPay - (pay.employee._401kDeductionRate * pay.netPay)
    pay copy (netPay = newNet)
  }

  def minusTax(pay: Pay[PreTaxDeductions]): Pay[PostTaxDeductions] = {
    val newNet = pay.netPay - (pay.employee.taxRate * pay.netPay)
    pay copy (netPay = newNet)
  }

  def minusFinalDeductions(pay: Pay[PostTaxDeductions]): Pay[Final] = {
    val newNet = pay.netPay - pay.employee.postTaxDeductions
    pay copy (netPay = newNet)
  }
}

object CalculatePayroll {
  def main(args: Array[String]) = {
    val e = Employee("Buck Trends", 100000.0F, 0.25F, 200F, 0.10F, 0.05F)
    val pay1 = Payroll start e
    // 401K and insurance can be calculated in either
    order.
    val pay2 = Payroll minus401k pay1
    val pay3 = Payroll minusInsurance pay2
    val pay4 = Payroll minusTax pay3
    val pay  = Payroll minusFinalDeductions pay4
    val twoWeekGross = e.annualSalary / 26.0F
    val twoWeekNet   = pay.netPay
    val percent      = (twoWeekNet / twoWeekGross) * 100
             "For ${e.name}, the gross vs. net pay every 2 weeks
    println(sis:"                                                    )
    println(
       "  $$${twoWeekGross}%.2f vs. $$${twoWeekNet}%.2f or
      f${percent}%.1f%%"                                                   )
  }
}
```

This code is already compiled by `sbt`, so we can run it at the `sbt` prompt:

```
> run-main progscala.implicits.payroll.CalculatePayroll
[info] Running progscala.implicits.payroll.CalculatePayroll
For Buck Trends, the gross vs. net pay every 2 weeks is:
  $3846.15 vs. $2446.10 or 63.6%
```

`sealed`
Note the `trait` s with no data and no classes that implement them. Because they are "sealed," they can't be implemented in other files. Hence, they only serve as "markers."

These markers are used as type parameters for the `Pay` type, which is a token passed through the `Payroll` calculator object. Each method in `Payroll` takes a `Pay[Step]` object with a particular type for the `Step` parameter. So, we can't call `minus401k` with a `Pay[PostTaxDeductions]` object. Hence, the tax rules are enforced by the API.

The `CalculatePayroll` object demonstrates the use of the API. If you try reordering some of the steps, for example, moving the `Payroll.minusFinalDeductions` call before the `Payroll.minusTax` call (and fixing the variable names), you get a compiler error.

Note that the example ends with a *printf* interpolated string very similar to the example we discussed in Interpolated Strings.

Actually, this `main` routine is not very elegant, undermining the virtue of this approach. Let's fix that, borrowing a "pipelining" operator in `F#`, Microsoft's functional programming language. This example is adapted from James Iry's blog:

```
// src/main/scala/progscala2/implicits/phantom-types-
pipeline.scala
package progscala.implicits.payroll
import scala.language.implicitConversions

object Pipeline {
  implicit class toPiped[V](value:V) {
    def |>[R] (f : V => R) = f(value)
  }
}

object CalculatePayroll2 {
  def main(args: Array[String]) = {
    import Pipeline._
    import Payroll._

    val e = Employee("Buck Trends", 100000.0F, 0.25F, 200F, 0.10F, 0.05F)
    val pay = start(e) |>
      minus401k        |>
      minusInsurance   |>
      minusTax         |>
      minusFinalDeductions
    val twoWeekGross = e.annualSalary / 26.0F
    val twoWeekNet   = pay.netPay
    val percent      = (twoWeekNet / twoWeekGross) * 100
             "For ${e.name}, the gross vs. net pay every 2 weeks
    println(sis:"                                                    )
    println(
       "  $$${twoWeekGross}%.2f vs. $$${twoWeekNet}%.2f or
      f${percent}%.1f%%"
)
  }
}
```

Now the `main` routine is a more elegant sequencing of steps, the calls to `Payroll` methods. Note that the pipeline *operator* `|>` looks fancy, but all it really does is reorder tokens. For example, it turns an expression like this:

```
pay1 |> Payroll.minus401k
```

into this expression:

```
Payroll.minus401k(pay1)
```

Rules for Implicit Arguments

Returning to implicit arguments, the following sidebar lists the general rules for implicit arguments.

Note the errors reported in these examples, which break the rules:

```
scala> class Bad1 {
     |   def m(i: Int, implicit s: String) =
"boo"
<console>:2: error: identifier expected but 'implicit' found.
         def m(i: Int, implicit s: String) = "boo"
                       ^

scala> }
<console>:1: error: eof expected but '}' found.
       }
       ^

scala> class Bad2 {
     |   def m(i: Int)(implicit s: String)(implicit d: Double) =
"boo"
<console>:2: error: '=' expected but '(' found.
         def m(i: Int)(implicit s: String)(implicit d: Double) =
"boo"
                                            ^

scala> }
<console>:1: error: eof expected but '}' found.
       }
       ^

scala> class Good1 {
     |   def m(i: Int)(implicit s: String, d: Double) =
"boo"
     |
}
defined class Good1

scala> class Good2 {
     |   def m(implicit i: Int, s: String, d: Double) =
"boo"
     |
}
defined class Good2
```

Implicit Conversions

We saw in Tuple Literals that there are several ways to create a pair:

```
(1, "one")
1 -> "one"
                   // Using → instead of -
1 → "one"          >
Tuple2(1, "one")
Pair(1, "one")
// Deprecated as of Scala
2.11
```

It seems a bit wasteful to bake into the Scala grammar two literal forms: `(a, b)` and `a -> b`.

The `a -> b` (or equivalently, `a → b`) idiom for pair creation is popular for initializing a `Map`:

```
scala> Map("one" -> 1, "two" -> 2)
res0: scala.collection.immutable.Map[String,Int] = Map(one -> 1, two -> 2)
```

The `Map.apply` method being called expects a variable argument list of pairs:

```
def apply[A, B](elems: (A, B)*): Map[A, B
]
```

In fact, Scala knows nothing about `a -> b`, so it's not "wasteful." This "literal" form is actually a method `->` and a special Scala feature, *implicit conversions*, which allows us to use this method between two values of any type. Also, because `a -> b` is not a literal syntax for tuples, we must somehow convert that expression to `(a, b)`.

The obvious first step is to define a method `->`, but where? We want the ability to call this method for all possible objects that might be the first element in a pair. Even if we could edit the code for all the types where we want this method, it wouldn't be practical nor wise to do so.

The trick is to use a "wrapper" object that has `->` defined. Scala has one in `Predef` already:

```
implicit final class ArrowAssoc[A](val self: A) {
  def -> [B](y: B): Tuple2[A, B] = Tuple2(self, y
)
}
```

(I omitted a few unimportant details of the actual declaration for clarity.) Like for Java, the `final` keyword prevents subclasses of `ArrowAssoc` from being declared.

We could use it like this to create a `Map`:

```
scala> Map(new ArrowAssoc("one") -> 1, new ArrowAssoc("two") -> 2)
res0: scala.collection.immutable.Map[String,Int] = Map(one -> 1, two -> 2)
```

That's way worse than just using `Map( ("one", 1), ("two", 2) )`. However, it does take us partly there. `ArrowAssoc` accepts an object of any type, then when `->` is called, a pair is returned.

This is where the `implicit` keyword comes into play again. Because `ArrowAssoc` is declared implicit, the compiler goes through the following logical steps:

1. It sees that we're trying to call a `->` method on `String` (e.g., `"one" -> 1`).
2. Because `String` has no `->` method, it looks for an *implicit conversion* in scope to a type that has this method.

3. It finds `ArrowAssoc`.
4. It constructs an `ArrowAssoc`, passing it the `"one"` string.
5. It then resolves the `-> 1` part of the expression and confirms the whole expression's type matches the expectation of `Map.apply`, which is a pair instance.

For something to be considered an *implicit conversion*, it must be declared with the `implicit` keyword and it must either be a class that takes a single constructor argument or it must be a method that takes a single argument.

Before Scala 2.10, it was necessary to define a wrapper class without the `implicit` keyword, and an `implicit` method that performed the conversion to the wrapper class. That "two-step" seemed pointless, so you can now mark the class `implicit` and eliminate the method. So, for example, `ArrowAssoc` looked like this before Scala 2.10 (`any2ArrowAssoc` still exists, but it's deprecated):

```
final class ArrowAssoc[A](val self: A) {
  def -> [B](y: B): Tuple2[A, B] = Tuple2(self, y)
}
...
implicit def any2ArrowAssoc[A](x: A): ArrowAssoc[A] = new ArrowAssoc(x)
```

Implicit methods can still be used, but now they are only necessary when converting to a type that already exists for other purposes and it isn't declared implicit.

Another example is the pipeline operator we defined earlier in Phantom Types:

```
... pay1 |> Payroll.minus401k ...
```

Indiscriminate use of implicit methods can lead to mysterious behavior that is hard to debug. For this reason, implicit methods are considered an optional feature as of Scala 2.10, so you should enable the feature explicitly with the import statement, `import scala.language.implicitConversions`, or with the global `-language:implicitConversions` compiler option.

Here is a summary of the lookup rules used by the compiler to find and apply conversions methods:

1. No conversion will be attempted if the object and method combination type check successfully.
2. Only classes and methods with the `implicit` keyword are considered.
3. Only implicit classes and methods in the current scope are considered, as well as implicit methods defined in the *companion object* of the *target* type (see the following discussion).
4. Implicit methods aren't chained to get from the available type, through intermediate types, to the target type. Only a method that takes a single available type instance and returns a target type instance will be considered.
5. No conversion is attempted if more than one possible conversion method could be applied. There must be one and only one, unambiguous possibility.

The third point that mentions implicit methods in companion objects has the following meaning. First, no implicit conversion is used unless it's already in the current scope, either because it's declared in an enclosing scope or it was imported from another scope, such as a separate object that defines some implicit conversions.

However, another scope automatically searched *last* is the companion object of the type to which a conversion is needed, if there is one. Consider the following example:

```
// src/main/scala/progscala2/implicits/implicit-conversions-resolution.sc
import scala.language.implicitConversions

case class Foo(s: String)
object Foo {
  implicit def fromString(s: String): Foo = Foo(s)
}

class O {
  def m1(foo: Foo) = println(foo)
  def m(s: String) = m1(s)
}
```

The Foo companion object defines a conversion from String. The O.m method attempts to call O.m1 with a String, but it expects a Foo. The compiler finds the Foo.fromString conversion method, even though we didn't import it explicitly into the scope of O.

However, if there is another Foo conversion in scope, it will override the Foo.fromString conversion:

```
// src/main/scala/progscala2/implicits/implicit-conversions-resolution.sc
import scala.language.implicitConversions

case class Foo(s: String)
object Foo {
  implicit def fromString(s: String): Foo = Foo(s)
}

class O {
  def m1(foo: Foo) = println(foo)
  def m(s: String) = m1(s)
}
```

Now, overridingConversion will be used instead.

We mentioned before that a recommended convention is to put implicit values and conversions into a special package named implicits or an object named Implicits, except for those defined in companion objects, as we just discussed. The resulting import statements make it clear to the reader that custom implicits are being used.

To finish, note that Scala has several implicit wrapper types for Java types like String and Array. For example, the following methods appear to be String methods, but they are actually implemented by WrappedString:

```
               "Programming
scala> val s = Scala"
s: String = Programming Scala

scala> s.reverse
res0: String = alacS gnimmargorP

scala> s.capitalize
res1: String = Programming Scala

scala> s.foreach(c => print(s"$c-"))
P-r-o-g-r-a-m-m-i-n-g- -S-c-a-l-a-
```

The implicit conversions for the built-in “Wrapped” types are always in scope. They are defined in Predef.

Another example is the Range type we saw earlier. For example, 1 to 100 by 3 represents every third integer from 1 to 100. You can now guess that the words to, by, and the exclusive range until are actually methods on wrapper types, not language keywords. For example, scala.runtime.RichInt wraps Int and it has these methods. Similar wrapper types exist for the other numeric types in the same package: RichLong, RichFloat, RichDouble, and RichChar. The types scala.math.BigInt and scala.math.BigDecimal are already wrapper types for the Java equivalents, so they don’t have their own wrapper types. They just implement the methods to, until, and by themselves.

Build Your Own String Interpolator

Let’s look at a final example of an implicit conversion, one that lets us define our own string interpolation capability. Recall from Interpolated Strings that Scala has several built-in ways to format strings through interpolation. For example:

```
val name = ("Buck", "Trends")
         "Hello, ${name._1}
println(s${name._2}"                   )
```

When the compiler sees an expression like x"foo bar", it looks for an x method in scala.StringContext. The last line in the previous example is translated to this:

```
              "Hello,    "
StringContext("        , "   , "").s(name._1, name._2)
```

The arguments passed to StringContext.apply are the “parts” around the ${…} expressions that were extracted. The arguments passed to s are the extracted expressions. (Try experimenting with variations of this example.) There are also StringContext methods named f and raw for those forms of interpolation.

We can define our own interpolator using an implicit conversion in the usual way to “extend” StringContext with new methods. Let’s flesh out the example shown on the StringContext Scaladoc page, an interpolator that converts a simple JSON string to a scala.util.parsing.json.JSONObject object.

We’ll make a few simplifying assumptions. First, we won’t handle arrays or nested objects, just “flat” JSON expressions like `{"a": "A", "b": 123, "c": 3.14159}`. Second, we'll require that the keys are hardcoded in the string and the values are all specified as interpolation parameters, e.g., `{"a": $a, "b": $b, "c": $c}`. This second restriction would be reasonable if we were using the implementation as a *template* mechanism for generating similar JSON objects with different values. Here it is:

```
// src/main/scala/progscala2/implicits/custom-string-interpolator.sc
import scala.util.parsing.json._

object Interpolators {
  implicit class jsonForStringContext(val sc: StringContext) {       //
❶
    def json(values: Any*): JSONObject = {
// ❷
      val keyRE = """^[\s{,]*(\S+):\s*""".r
// ❸
      val keys = sc.parts map {
// ❹
        case keyRE(key) => key
        case str => str
      }
      val kvs = keys zip values
// ❺
      JSONObject(kvs.toMap)
// ❻
    }
  }
}

import Interpolators._

val name = "Dean Wampler"
val book = "Programming Scala, Second Edition"

val jsonobj = json"{name: $name, book: $book}"                       //
❼
println(jsonobj)
```

❶

Implicit classes must be defined inside objects to limit their scope. (This is a "safety measure.") The import statement after the class brings the implementation class into the scope of the code that needs it.

❷

A `json` method. It takes a variable argument list, the parameters embedded in the string, and it returns a constructed `scala.util.parsing.json.JSONObject`.

❸

Regular expression to extract the key name from a string fragment.

❹

Extract the key names from the `parts` (string fragments), using the regular expression. If the regular expression doesn't match, just use the whole string, but it might be better to throw an exception to avoid using an invalid key string.

❺

"Zip" the keys and values together into a collection of key-value pairs. We'll discuss the `zip` method further after this list.

❻

Construct a `Map` using the key-value pairs and use it to construct the `JSONObject`.

❼

Use our string interpolator, just like the built-in versions.

Custom string interpolators don't have to return a `String`, like `s`, `f`, and `raw` return. We return a `JSONObject` instead. Hence, they can function as instance factories that are driven by data encapsulated in strings.

The `zip` method on collections is a handy way to line up the values between two collections, like a zipper. Here's an example:

```
scala> val keys = List("a", "b", "c", "d")
keys: List[String] = List(a, b, c, d)

scala> val values = List("A", 123, 3.14159)
values: List[Any] = List(A, 123, 3.14159)

scala> val keysValues = keys zip values
keysValues: List[(String, Any)] = List((a,A), (b,123), (c,3.14159
))
```

The elements of the zipped collection are two-element tuples, `(key1, value1)`, etc. Note that because one list is longer than the other, the extra elements at the end are simply dropped. That's actually what we want in `json`, because there is one more string fragment than there are value parameters. It's the trailing fragment at the end of the string and we don't need it.

Here's the output from the last two lines in a REPL session:

```
scala> val jsonobj = json"{name: $name, book: $book}"
jsonobj: scala.util.parsing.json.JSONObject = \
  {"name" : "Dean Wampler", "book" : "Programming Scala, Second Edition"}

scala> println(jsonobj)
{"name" : "Dean Wampler", "book" : "Programming Scala, Second Edition"}
```

The Expression Problem

Let's step back for a moment and ponder what we've just accomplished: we've effectively added a new method to all types without editing the source code for any of them!

This desire to extend modules without modifying their source code is called the *Expression Problem*, a term coined by Philip Wadler.

Object-oriented programming solves this problem with subtyping, more precisely called *subtype polymorphism*. We program to abstractions and use derived classes when we need changed behavior. Bertrand Meyer coined the term *Open/Closed Principle* to describe the OOP approach, where base types declare behavior in an abstraction and subtypes implement appropriate variations of the behavior, without modifying the base types.

Scala certainly supports this technique, but it has drawbacks. What if it's questionable that we should have that behavior defined in the type hierarchy in the first place? What if the behavior is only needed in a few contexts, while for most contexts, it's just a burden that the client code carries around?

It can be a burden for several reasons. First, the extra, unused code takes up system resources. A few cases don't matter much, but inevitably a mature code base will have a lot of this baggage. Second, it's also inevitable that most defined behaviors will be refined over time. Every change to an unused behavior forces unwanted changes on client code that doesn't use that behavior.

This problem led to the *Single Responsibility Principle*, a classic design principle that encourages us to define abstractions and implementing types with just a single behavior.

Still, in realistic scenarios, it's sometimes necessary for an object to combine several behaviors. For example, a service often needs to "mix in" the ability to log messages. Scala makes these *mixin* features relatively easy to implement, as we saw in Traits: Interfaces and "Mixins" in Scala. We can even declare objects that combine traits on the fly.

Dynamically typed languages typically provide *metaprogramming* facilities that allow you to modify classes in the runtime environment without modifying source code. This approach can partially solve the problem of types with rarely used behavior. Unfortunately, for most dynamic languages, any runtime modifications to a type are global, so all users are affected.

The implicit conversion feature of Scala lets us implement a statically typed alternative approach, called *type classes*, which was pioneered in Haskell—see, for example, *A Gentle Introduction to Haskell*. The name comes from Haskell and it shouldn't be confused with Scala's usual notion of classes.

Type Class Pattern

Type classes help us avoid the temptation of creating "kitchen-sink" abstractions, like Java's `Object`, because we can add behavior on an ad hoc basis. Scala's `->` pair-construction idiom is one example. Recall that we aren't modifying these types; we are using the implicit mechanism to wrap objects with types that provide the behaviors we need. It only appears that we are modifying the types, as seen in the source code.

Consider the ubiquitous `Object.toString` in Java, which Scala inherits as a JVM language. Note that Java's default `toString` is of little value, because it just shows the type name and its address in the JVM's heap. The syntax used by Scala for case classes is more useful and human readable. There are times when a machine-readable format like JSON or XML would also be very useful. With implicit conversions, it's feasible to "add" `toJSON` and `toXML` methods to any type. If `toString` weren't already present, we could use implicit conversions for it, as well.

Type classes in Haskell define the equivalent of an interface, then concrete implementations of it for specific types. The *Type Class Pattern* in Scala adds the missing interface piece that our example implementations of implicit conversions so far didn't provide.

Let’s look at a possible `toJSON` type class:

```
// src/main/scala/progscala2/implicits/toJSON-type-class.sc

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

trait ToJSON {
  def toJSON(level: Int = 0): String

                    "
  val INDENTATION = "
  def indentation(level: Int = 0): (String,String) =
    (INDENTATION * level, INDENTATION * (level+1))
}

implicit class AddressToJSON(address: Address) extends ToJSON {
  def toJSON(level: Int = 0): String = {
    val (outdent, indent) = indentation(level)
    s"""{
      |${indent}"street":
"${address.street}",
      |${indent}"city":
"${address.city}"

|$outdent}"""      .stripMargin
  }
}

implicit class PersonToJSON(person: Person) extends ToJSON {
  def toJSON(level: Int = 0): String = {
    val (outdent, indent) = indentation(level)
    s"""{
      |${indent}"name":
"${person.name}",
      |${indent}"address": ${person.address.toJSON(level +
1)}

|$outdent}"""      .stripMargin
  }
}

                "1 Scala
val a = Address(Lane"         , "Anytown")
val p = Person("Buck Trends", a)

println(a.toJSON())
println()
println(p.toJSON())
```

For simplicity, our `Person` and `Address` types have just a few fields and we’ll format multiline JSON strings rather than objects in the `scala.util.parsing.json` package (see Examples: XML and JSON DSLs for Scala for details).

We define a default indentation string in the `ToJSON` trait and a method that calculates the actual indentation for fields and for the closing brace of a JSON object `{…}`. The `toJSON` method takes an argument that specifies the indentation level; that is, how many units of `INDENTATION` to indent. Because of this argument for `toJSON`, clients must provide empty parentheses or an alternative indentation level. Note that we put double quotes around string values, but not integer values.

Running this script produces the following output:

```
{
            "1 Scala
  "street": Lane"         ,
  "city":   "Anytown"
}

{
  "name":    "Buck Trends",
  "address": {
              "1 Scala
    "street": Lane"
,
    "city":   "Anytown"
  }
}
```

Scala doesn't allow the `implicit` and `case` keywords together. That is, an `implicit` class can't also be a case class. It wouldn't make much sense anyway, because the extra, auto-generated code for the case class would never be used. Implicit classes have a very narrow purpose.

Note that we used this mechanism to add methods to existing classes. Hence, another term for this capability is *extension methods*. Languages like C# and F# support a different mechanism for extension methods (see, for example, the Extension Methods page on the Microsoft Developer Network). This term is perhaps more obvious than *type classes*, but the latter term is more widely used.

From another perspective, this capability is called *ad hoc polymorphism*, because the polymorphic behavior of `toJSON` is not tied to the type system, as in *subtype polymorphism*, the conventional object-oriented inheritance. Recall that we discussed a third kind, *paremetric polymorphism*, in Abstract Types Versus Parameterized Types, where containers like `Seq[A]` behave uniformly for any `A` type.

The Type Class Pattern is ideal for situations where certain clients will benefit from the "illusion" that a set of classes provide a particular behavior that isn't useful for the majority of clients. Used wisely, it helps balance the needs of various clients while maintaining the Single Responsibility Principle.

Technical Issues with Implicits

So what's not to like? Why not define types with very little behavior that are little more than "bags" of fields (sometimes called *anemic* types), then add all behaviors using type classes?

First, the extra code involved in defining implicits is extra work you have to do and the compiler must work harder to process implicits. Therefore, a project that uses implicits heavily is a project that is slow to build.

Implicit conversions also incur extra runtime overhead, due to the extra layers of indirection from the wrapper types. It's true that the compiler will inline some method calls and the JVM will do additional optimizations, but the extra overhead needs to be justified. In Chapter 14 we’ll discuss *value types* where the extra runtime overhead is eliminated during compilation.

There are some technical issues involving the intersection with other Scala features, specifically subtyping (for a more complete discussion, see the thread from the *scala-debate* email group).

Let’s use a simple example to demonstrate the point:

```
// src/main/scala/progscala2/implicits/type-classes-subtyping.sc

trait Stringizer[+T] {
  def stringize: String
}

implicit class AnyStringizer(a: Any) extends Stringizer[Any] {
  def stringize: String = a match {
    case s: String => s
    case i: Int => (i*10).toString
    case f: Float => (f*10.1).toString
    case other =>
                                               "Can't stringize
      throw new UnsupportedOperationException(s$other"
)
  }
}

val list: List[Any] = List(1, 2.2F, "three", 'symbol)

list foreach { (x:Any) =>
  try {
    println(s"$x: ${x.stringize}")
  } catch {
    case e: java.lang.UnsupportedOperationException => println(e)
  }
}
```

We define a contrived `Stringizer` abstraction. If we followed the example of `ToJSON` previously, we would create implicit classes for all the types we want to “stringize.” There’s a problem, though. If we want to process a list of instances with heterogeneous types, we can only pass *one* `Stringizer` instance implicitly in the `list map` … code. Therefore, we’re forced to define an `AnyStringerize` class that embeds all knowledge of all types we know about. It even includes a default clause to throw an exception.

This is quite ugly and it violates a core rule in object-oriented programming that you should not write *switch* statements that make decisions based on types that might change. Instead, you’re supposed to use polymorphic dispatch, such as the way `toString` works in both Scala and Java.

For a more involved attempt to make `ToJSON` work with a list of objects, see the example in the code distribution, *Implicits/type-classes-subtyping2.sc*.

To finish, here are a few other tips that can help you avoid potential problems.

Always specify the return type of an implicit conversion method. Allowing type inference to determine the return type sometimes yields unexpected results.

Also, the compiler does a few “convenient” conversions for you that are now considered more troublesome than beneficial (future releases of Scala will probably change these behaviors).

First, if you define a `+` method on a type and attempt to use it on an instance that actually isn’t of the correct type, the compiler will instead call `toString` on the instance so it can then call the `String` `+` (concatenation) operation. So, if you get a mysterious error about a `String` being the wrong type in the particular context, it’s probably for this reason.

Also, the compiler will “auto-tuple” arguments to a method when needed. Sometimes this is very confusing. Fortunately, Scala 2.11 now warns you:

```
scala> def m(pair:Tuple2[Int,String]) = println(pair)

scala> m(1, "two")
<console>:9: warning: Adapting argument list by creating a 2-tuple:
  this may not be what you want.
        signature: m(pair: (Int, String)): Unit
  given arguments: 1, "two"
 after adaptation: m((1, "two"): (Int, String))
              m(1,"two")
               ^
(1,two)
```

It’s okay if your head is spinning a bit at this point, but hopefully it’s clear that implicits are a powerful tool in Scala, but they have to be used wisely.

Implicit Resolution Rules

When Scala looks for implicits, it follows a sequence of fairly sophisticated rules for the search, some of which are designed to resolve potential ambiguities.[]

I’ll just use the term “value” in the following discussion, although methods, values, or classes can be used, depending on the implicit scenario:

- Any type-compatible implicit value that doesn’t require a prefix path. In other words, it is defined in the same scope, such as within the same block of code, within the same type, within its companion object (if any), and within a parent type.
- An implicit value that was imported into the current scope. (It also doesn’t require a prefix path to use it.)

Imported values, the second bullet point, take precedence over the already-in-scope values.

In some cases, several possible matches are type compatible. The most specific match wins. For example, if the type of an implicit argument is `Foo`, then an implicit value in scope of type `Foo` will be chosen over an implicit value of type `AnyRef`, if both are in scope.

If two or more implicit values are ambiguous, such as they have the same specific type, it triggers a compiler error.

The compiler always puts some library implicits in scope, while other library implicits require an import statement. Let’s discuss these implicits now.

Scala’s Built-in Implicits

The source code for the Scala 2.11 library has more than 300 implicit methods, values, and types. Most of them are methods, and most of those are used to convert from one type to another. Because it's important to know what implicits might affect your code, let's discuss them as groups, without listing every single one of them. It's definitely not important to learn every detail, but developing a "sense" of what implicits exist will be beneficial. So, skim this section as you see fit.

We've already discussed `CanBuildFrom` for collections. There is a similar `CanCombineFrom` used by various operations for combining instances, as the name implies. We won't list these definitions.

All of the companion objects for `AnyVal` types have widening conversions, such as converting an `Int` to a `Long`. Most actually just call `toX` methods on the type. For example:

```
object Int {
  ...
  implicit def int2long(x: Int): Long = x.toLong
  ...
}
```

The following code snippits list these `AnyVal` implicit conversions. Note that because of the implicit conversion feature, the Scala grammar doesn't need to implement the most common type conversions that other language grammars have to implement.

From the `Byte` companion object:

```
implicit def byte2short(x: Byte): Short = x.toShort
implicit def byte2int(x: Byte): Int = x.toInt
implicit def byte2long(x: Byte): Long = x.toLong
implicit def byte2float(x: Byte): Float = x.toFloat
implicit def byte2double(x: Byte): Double = x.toDouble
```

From the `Char` companion object:

```
// From the scala.Char companion
object:
implicit def char2int(x: Char): Int = x.toInt
implicit def char2long(x: Char): Long = x.toLong
implicit def char2float(x: Char): Float = x.toFloat
implicit def char2double(x: Char): Double = x.toDouble
```

From the `Short` companion object:

```
implicit def short2int(x: Short): Int = x.toInt
implicit def short2long(x: Short): Long = x.toLong
implicit def short2float(x: Short): Float = x.toFloat
implicit def short2double(x: Short): Double = x.toDouble
```

From the `Int` companion object:

```
implicit def int2long(x: Int): Long = x.toLong
implicit def int2float(x: Int): Float = x.toFloat
implicit def int2double(x: Int): Double = x.toDouble
```

From the `Long` companion object:

```
implicit def long2float(x: Long): Float = x.toFloat
implicit def long2double(x: Long): Double = x.toDouble
```

From the `Float` companion object:

```
implicit def float2double(x: Float): Double = x.toDouble
```

The `scala.math` types `BigInt` and `BigDecimal` have converters from many of the `AnyVal` types and from the corresponding Java implementations. From the `BigDecimal` companion object:

```
implicit def int2bigDecimal(i: Int): BigDecimal = apply(i)
implicit def long2bigDecimal(l: Long): BigDecimal = apply(l)
implicit def double2bigDecimal(d: Double): BigDecimal = ...
implicit def javaBigDecimal2bigDecimal(x: BigDec): BigDecimal = apply(x)
```

The calls to `apply` are invocations of the `BigDecimal.apply` factory methods. These implicits are a convenient, alternative way of invoking these methods.

From the `BigInt` companion object:

```
implicit def int2bigInt(i: Int): BigInt = apply(i)
implicit def long2bigInt(l: Long): BigInt = apply(l)
implicit def javaBigInteger2bigInt(x: BigInteger): BigInt = apply(x)
```

`Option` can be converted to a list of zero or one items:

```
implicit def option2Iterable[A](xo: Option[A]): Iterable[A] = xo.toList
```

Scala uses many of Java's types, including `Array[T]` and `String`. There are corresponding `ArrayOps[T]` and `StringOps` types that provide the operations commonly defined for all Scala collections. So, implicit conversions to and from these wrapper types are useful. Other operations are defined on types with the word `Wrapper` in their name.

Most are defined in `Predef`. Some of these definitions have the `@inline` annotation, which encourages the compiler to try especially hard to inline the method call, eliminating the stack frame overhead. There is a corresponding `@noinline` annotation that prevents the compiler from attempting to inline the method call, even if it can.

Several methods convert one type to another, such as wrapping a type in a new type that provides additional methods:

```
@inline implicit def augmentString(x: String): StringOps = new StringOps(x)
@inline implicit def unaugmentString(x: StringOps): String = x.repr
implicit def tuple2ToZippedOps[T1, T2](x: (T1, T2))
  = new runtime.Tuple2Zipped.Ops(x)
implicit def tuple3ToZippedOps[T1, T2, T3](x: (T1, T2, T3))
  = new runtime.Tuple3Zipped.Ops(x)
implicit def genericArrayOps[T](xs: Array[T]): ArrayOps[T] = ...

implicit def booleanArrayOps(xs: Array[Boolean]): ArrayOps[Boolean] =
  = new ArrayOps.ofBoolean(xs)
          // Similar functions for the other AnyVal
...       types.
implicit def refArrayOps[T <: AnyRef](xs: Array[T]): ArrayOps[T]
  = new ArrayOps.ofRef[T](xs)

@inline implicit def byteWrapper(x: Byte) = new runtime.RichByte(x)
          // Similar functions for the other AnyVal
...       types.

implicit def genericWrapArray[T](xs: Array[T]): WrappedArray[T] = ...
implicit def wrapRefArray[T <: AnyRef](xs: Array[T]): WrappedArray[T] = ...
implicit def wrapIntArray(xs: Array[Int]): WrappedArray[Int] = ...
          // Similar functions for the other AnyVal
...       types.

implicit def wrapString(s: String): WrappedString = ...
implicit def unwrapString(ws: WrappedString): String = ...
```

To understand the purpose of `runtime.Tuple2Zipped.Ops`, first note that most of the collections have a `zip` method for joining two collections paired by elements, like closing a zipper:

```
scala> val zipped = List(1,2,3) zip List(4,5,6)
zipped: List[(Int, Int)] = List((1,4), (2,5), (3,6
))
```

Such zipped collections are useful for pair-wise operations, for example:

```
scala> val products = zipped map { case (x,y) => x * y }
products: List[Int] = List(4, 10, 18)
```

Note that we used the pattern-matching trick for anonymous functions that take tuple arguments, because each `Int` pair is passed to the anonymous function that multiplies the pair elements.

`Tuple2Zipper.Ops` and `Tuple3Zipper.Ops` provide an `invert` method to convert a two- and three-element tuple of collections, respectively, into collections of tuples. In other words, they zip collections that are already held in a tuple. For example:

```
scala> val pair = (List(1,2,3), List(4,5,6))
pair: (List[Int], List[Int]) = (List(1, 2, 3),List(4, 5, 6))

scala> val unpair = pair.invert
unpair: List[(Int, Int)] = List((1,4), (2,5), (3,6))
```

```
val pair = (List(1,2,3), List("one", "two", "three"))
tuple2ToZippedOps(pair) map {case (int, string) => (int*2, string.toUpperCase)}

val pair = (List(1,2,3), List(4,5,6))
pair map { case (int1, int2) => int1 + int2 }
```

Next, there are a lot of conversions to and from other Java types. Again, from `Predef`:

```
implicit def byte2Byte(x: Byte) = java.lang.Byte.valueOf(x)
implicit def Byte2byte(x: java.lang.Byte): Byte = x.byteValue
         // Similar functions for the other AnyVal
...      types.
```

For completeness, recall these definitions from `Predef` that we used earlier in the chapter:

```
implicit def conforms[A]: A <:< A = ...
implicit def tpEquals[A]: A =:= A = ...
```

To convert between `java.util.Random` and `scala.util.Random`:

```
implicit def javaRandomToRandom(r: java.util.Random): Random = new Random(r)
```

The `scala.collection.convert` package has several traits that add conversion methods between Java and Scala collections, which are very handy for Java interopability. Actually, where possible, no conversion is done for efficiency, but the abstractions of the target collection are layered on top of the underlying collection.

Here are the “decorations” of Scala collections as Java collections defined in `DecorateAsJava`. The return type annotations are wrapped to fit the page and the `ju` and `jl` are import aliases for `java.lang` and `java.util` in the actual source code for `DecorateAsJava`. The `AsJava*` types are helpers for exposing operations:

```
implicit def asJavaIteratorConverter[A](i : Iterator[A]):
  AsJava[ju.Iterator[A]] = ...
implicit def asJavaEnumerationConverter[A](i : Iterator[A]):
  AsJavaEnumeration[A] = ...
implicit def asJavaIterableConverter[A](i : Iterable[A]):
  AsJava[jl.Iterable[A]] = ...
implicit def asJavaCollectionConverter[A](i : Iterable[A]):
  AsJavaCollection[A] = ...
implicit def bufferAsJavaListConverter[A](b : mutable.Buffer[A]):
  AsJava[ju.List[A]] = ...
implicit def mutableSeqAsJavaListConverter[A](b : mutable.Seq[A]):
  AsJava[ju.List[A]] = ...
implicit def seqAsJavaListConverter[A](b : Seq[A]):
  AsJava[ju.List[A]] = ...
implicit def mutableSetAsJavaSetConverter[A](s : mutable.Set[A]):
  AsJava[ju.Set[A]] = ...
implicit def setAsJavaSetConverter[A](s : Set[A]):
  AsJava[ju.Set[A]] = ...
implicit def mutableMapAsJavaMapConverter[A, B](m : mutable.Map[A, B]):
  AsJava[ju.Map[A, B]] = ...
implicit def asJavaDictionaryConverter[A, B](m : mutable.Map[A, B]):
  AsJavaDictionary[A, B] = ...
implicit def mapAsJavaMapConverter[A, B](m : Map[A, B]):
  AsJava[ju.Map[A, B]] = ...
implicit def mapAsJavaConcurrentMapConverter[A, B](m: concurrent.Map[A, B])
:
  AsJava[juc.ConcurrentMap[A, B]] = ...
```

We can decorate Java collections as Scala collections using implicits defined in `DecorateAsScala`:

```
implicit def asScalaIteratorConverter[A](i : ju.Iterator[A]):
  AsScala[Iterator[A]] = ...
implicit def enumerationAsScalaIteratorConverter[A](i : ju.Enumeration[A]):
  AsScala[Iterator[A]] = ...
implicit def iterableAsScalaIterableConverter[A](i : jl.Iterable[A]):
  AsScala[Iterable[A]] = ...
implicit def collectionAsScalaIterableConverter[A](i : ju.Collection[A]):
  AsScala[Iterable[A]] = ...
implicit def asScalaBufferConverter[A](l : ju.List[A]):
  AsScala[mutable.Buffer[A]] = ...
implicit def asScalaSetConverter[A](s : ju.Set[A]):
  AsScala[mutable.Set[A]] = ...
implicit def mapAsScalaMapConverter[A, B](m : ju.Map[A, B]):
  AsScala[mutable.Map[A, B]] = ...
implicit def mapAsScalaConcurrentMapConverter[A, B](m: juc.ConcurrentMap[A, B])
:
  AsScala[concurrent.Map[A, B]] = ...
implicit def dictionaryAsScalaMapConverter[A, B](p: ju.Dictionary[A, B]):
  AsScala[mutable.Map[A, B]] = ...
implicit def propertiesAsScalaMapConverter(p: ju.Properties):
  AsScala[mutable.Map[String, String]] = ...
```

While the methods are defined in these `trait`s, the object `JavaConverters` gives you the hook to bring them in scope with an important statement:

```
import scala.collection.JavaConverters._
```

The purpose of these converters is to allow you to call `asJava` on a Scala collection to create a corresponding Java collection, and call `asScala` on a Java collection to go the other way. These methods effectively define a one-to-one correspondence between one Scala collection type and one Java collection type.

For the more general case, where you want to choose the output collection type, rather than accept the one choice `asScala` or `asJava` offer, there are additional conversion methods defined in `WrapAsJava` and `WrapAsScala`.

Here are the `WrapAsJava` methods:

```
implicit def asJavaIterator[A](it: Iterator[A]): ju.Iterator[A] = ...
implicit def asJavaEnumeration[A](it: Iterator[A]): ju.Enumeration[A] = ...
implicit def asJavaIterable[A](i: Iterable[A]): jl.Iterable[A] = ...
implicit def asJavaCollection[A](it: Iterable[A]): ju.Collection[A] = ...
implicit def bufferAsJavaList[A](b: mutable.Buffer[A]): ju.List[A] = ...
implicit def mutableSeqAsJavaList[A](seq: mutable.Seq[A]): ju.List[A] = ...
implicit def seqAsJavaList[A](seq: Seq[A]): ju.List[A] = ...
implicit def mutableSetAsJavaSet[A](s: mutable.Set[A]): ju.Set[A] = ...
implicit def setAsJavaSet[A](s: Set[A]): ju.Set[A] = ...
implicit def mutableMapAsJavaMap[A, B](m: mutable.Map[A, B]): ju.Map[A, B]
=...
implicit def asJavaDictionary[A, B](m: mutable.Map[A, B]): ju.Dictionary[A, B]
implicit def mapAsJavaMap[A, B](m: Map[A, B]): ju.Map[A, B] = ...
implicit def mapAsJavaConcurrentMap[A, B](m: concurrent.Map[A, B]):
  juc.ConcurrentMap[A, B] = ...
```

Here are the `WrapAsScala` methods:

```
implicit def asScalaIterator[A](it: ju.Iterator[A]): Iterator[A] = ...
implicit def enumerationAsScalaIterator[A](i: ju.Enumeration[A]):
  Iterator[A] = ...
implicit def iterableAsScalaIterable[A](i: jl.Iterable[A]): Iterable[A] = ...
implicit def collectionAsScalaIterable[A](i: ju.Collection[A]): Iterable[A]=...
implicit def asScalaBuffer[A](l: ju.List[A]): mutable.Buffer[A] = ...
implicit def asScalaSet[A](s: ju.Set[A]): mutable.Set[A] = ...
implicit def mapAsScalaMap[A, B](m: ju.Map[A, B]): mutable.Map[A, B] = ...
implicit def mapAsScalaConcurrentMap[A, B](m: juc.ConcurrentMap[A, B]):
  concurrent.Map[A, B] = ...
implicit def dictionaryAsScalaMap[A, B](p: ju.Dictionary[A, B]):
  mutable.Map[A, B] = ...
implicit def propertiesAsScalaMap(p: ju.Properties):
  mutable.Map[String, String] = ...
[source,scala]
```

Similar to `JavaConverters`, the methods defined in these `trait`s can be imported into the current scope using the object `JavaConversions`:

```
import scala.collection.JavaConversions._
```

Because sorting of collections is so commonly done, there are many implicits for `Ordering[T]`, where `T` is a `String`, one of the `AnyVal` types that can be converted to `Numeric`, or a user-defined ordering. (`Numeric` is an abstraction for typical operations on numbers.) See also `Ordered[T]`.

We won't list the implicit `Orderings` defined in many collection types, but here are the definitions in `Ordering[T]`:

```
implicit def ordered[A <% Comparable[A]]: Ordering[A] = ...
implicit def comparatorToOrdering[A](implicit c: Comparator[A]):Ordering[A]=...
implicit def seqDerivedOrdering[CC[X] <: scala.collection.Seq[X], T](
  implicit ord: Ordering[T]): Ordering[CC[T]] = ...
implicit def infixOrderingOps[T](x: T)(implicit ord: Ordering[T]):
  Ordering[T]#Ops = ...
implicit def Option[T](implicit ord: Ordering[T]): Ordering[Option[T]] = ...
implicit def Iterable[T](implicit ord: Ordering[T]): Ordering[Iterable[T]] =...
implicit def Tuple2[T1, T2](implicit ord1: Ordering[T1], ord2: Ordering[T2]):
   Ordering[(T1, T2)] = ...
         // Similar functions for Tuple3 through
...      Tuple9.
```

Finally, there are several conversions that support "mini-DSLs" for concurrency and process management.

First, the `scala.concurrent.duration` package provides useful ways of defining time durations (we'll see them and the other types mentioned next for concurrency in use in Chapter 17):

```
implicit def pairIntToDuration(p: (Int, TimeUnit)): Duration = ...
implicit def pairLongToDuration(p: (Long, TimeUnit)): FiniteDuration = ...
implicit def durationToPair(d: Duration): (Long, TimeUnit) = ...
```

Here are miscellaneous conversions in several files in the `scala.concurrent` package:

```
//
scala.concurrent.FutureTaskRunner:
implicit def futureAsFunction[S](x: Future[S]): () => S

// scala.concurrent.JavaConversions:
implicit def asExecutionContext(exec: ExecutorService):
  ExecutionContextExecutorService = ...
implicit def asExecutionContext(exec: Executor): ExecutionContextExecutor = ...

// scala.concurrent.Promise:
private implicit def internalExecutor: ExecutionContext = ...

// scala.concurrent.TaskRunner:
implicit def functionAsTask[S](fun: () => S): Task[S] = ...

//
scala.concurrent.ThreadPoolRunner:
implicit def functionAsTask[S](fun: () => S): Task[S] = ...
implicit def futureAsFunction[S](x: Future[S]): () => S = ...
```

Lastly, `Process` supports operating systems processes, analogous to running UNIX shell commands:

```
implicit def buildersToProcess[T](builders: Seq[T])(
  implicit convert: T => Source): Seq[Source] = ...
implicit def builderToProcess(builder: JProcessBuilder): ProcessBuilder = ...
implicit def fileToProcess(file: File): FileBuilder = ...
implicit def urlToProcess(url: URL): URLBuilder = ...
implicit def stringToProcess(command: String): ProcessBuilder = ...
implicit def stringSeqToProcess(command: Seq[String]): ProcessBuilder = ...
```

Whew! This is a long list, but hopefully skimming the list gives you a sense of how the Scala library supports and uses implicits.

Wise Use of Implicits

The implicit argument mechanism is quite powerful for building DSLs (domain-specific languages) and other APIs that minimize boilerplate. However, because the implicit arguments and values passed for them are almost invisible, code comprehension is harder. So, they should be used wisely.

One way to improve their visibility is to adopt the practice of putting implicit values in a special package named `implicits` or an object named `Implicits`. That way, readers of code see the word “implicit” in the imports and know to be aware of their presence, in addition to Scala’s ubiquitous, built-in implicits. Fortunately, IDEs can now show when implicits are being invoked, too.

Recap and What’s Next

We dove into the details of implicits in Scala. I hope you can appreciate their power and utility, but also their drawbacks.

Now we’re ready to dive into the principles of functional programming. We’ll start with a discussion of the core concepts and why they are important. Then we’ll look at the powerful functions provided by most container types in the library. We’ll see how we can use those functions to construct concise, yet powerful programs.