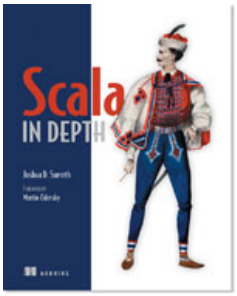
MANNING PUBLICATIONS



Scala in Depth
By Joshua D. Suereth

Scala is working on language features to integrate directly with dynamic languages, but even with the 2.9.0 release, these features are considered experimental. This article from chapter 10 of Scala in Depth, focuses on an area of concern when integrating with Java—the overuse of implicit conversions to adapt Java libraries into Scala idioms.

You may also be interested in…

Scala/Java Interaction: Be Wary of Implicit Conversions

One common mechanism of supporting the Scala/Java interaction is to create implicit conversions within Scala that promote Java types into a more Scala-friendly form. This can help ease the pain of using classes not designed for Scala but comes at a cost. Implicit conversions carry a few dangers that developers need to be aware of:

- Object identity and equality
- Chaining implicits.

The most common example of using implicit conversions to ease integration between Java and Scala are found in the Scala object `scala.collection.JavaConverters`. This object contains a set of implicit conversions to convert collections from Java to their Scala equivalents and vice versa. These implicit conversions are immensely handy but also suffer from all the issues associated with this design. Let's look into how object identity and equality can become a problem when using `JavaConversions`.

Object identity and equality

One of the dangers of using implicits to wrap Scala or Java objects for interoperability is that it can alter object identity. This breaks equality in any code that might require equality. Let's look at a simple example of converting a Java collection into a Scala one:

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

scala> val x = new java.util.ArrayList[String]
x: java.util.ArrayList[String] = []

scala> x.add("Hi"); x.add("You")

scala> val y : Iterable[String] = x
y: Iterable[String] = Buffer(Hi, You)

scala> x == y
res1: Boolean = false
```

The first line imports the `JavaConversions` implicit conversions. The next line creates the Java collection `ArrayList`. The values `"Hi"` and `"You"` are added to the array list.

The `val y` is constructed with the type of `scala.Iterable`. This invokes an implicit conversion to adapt the Java `ArrayList` into a Scala `Iterable`. Finally, when testing equality of the two collections, the value is `false`. When wrapping a Java collection, the wrapped collection isn't equal to the original.

AVOID IMPLICIT VIEWS Implicit views, when interfacing with Java, can cause silent object identity issues and other problems. It's best to be explicit.

The nuance of this issue can be subtle. For example, the implicit conversion from a Java collection to a Scala collection isn't as obvious as in the previous example. Imagine there's a Java class that looks as follows:

```
import java.util.ArrayList;

class JavaClass {
  public static ArrayList<String> CreateArray() {
    ArrayList<String> x = new ArrayList<String>();
    x.add("HI");
    return x;
  }
}
```

The class `JavaClass` has one method called `CreateArray` that returns an `ArrayList` containing the value `"HI"`. Now imagine the following Scala class:

```
object ScalaClass {
  def areEqual(x : Iterable[String], y : AnyRef) = x == y
}
```

The object `ScalaClass` is defined with one method, `areEqual`. This method takes a `scala.Iterable` and an `AnyRef` and checks the equality. Now let's use these two classes together.

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

scala> val x = JavaClass.CreateArray()
x: java.util.ArrayList[String] = [HI]

scala> ScalaClass.areEqual(x,x)
res3: Boolean = false
```

The first line imports the implicit conversions for `Collection`. The next line calls the Java class and constructs the new `ArrayList`. Finally, the same variable is placed into both sides of the `areEqual` method. Because the compiler is running the implicit conversions behind the scenes, the fact that `x` is being wrapped is less apparent in this code. The result of `areEqual` is false.

Although this example is contrived, it demonstrates how the issue can become hidden behind method calls. In real-world programming, this issue can be difficult to track down when it occurs, as the method call chains are often more complex.

Chaining implicits

The second issue facing implicits as a means to ease Java integration is that of chaining implicits. Scala and Java both support generic types. Collections in both languages have one generic parameter. The implicits that convert from Java to Scala and back again will alter the collection type, but usually not the underlying generic parameter.

This means that if the generic parameter type also needs to be converted for smooth Java/Scala integration, then it's possible the implicit won't be triggered.

Let's look at a common example: boxed types and Java collections.

```
scala> val x = new java.util.ArrayList[java.lang.Integer]

x: java.util.ArrayList[java.lang.Integer] = []
scala> val y : Iterable[Int] = x
<console>:17: error: type mismatch;
 found : java.util.ArrayList[java.lang.Integer]
 required: Iterable[Int]
       val y : Iterable[Int] = x
```

The first line constructs a new Java `ArrayList` collection with generic parameter set to `java.lang.Integer`. In Scala, because the compiler doesn't differentiate between primitives and objects, the type `scala.Int` can be safely used for generic parameters.

But Java's boxed integer, `java.lang.Integer`, isn't the same type as `scala.Int`, but the two can be converted seamlessly. Scala provides an implicit conversion from java.lang.Integer to scala.Int:

```
scala> val x : Int = new java.lang.Integer(1)
x: Int = 1
```

This line constructs a `java.lang.Integer` with the value `1` and assigns it to the value `x` with the type `scala.Int`. The implicit in `scala.Predef` kicks in here and automatically converts from the `java.lang.Integer` type into `scala.Int`. This implicit doesn't kick in when looking for implicit conversions from Java to Scala.

Let's naively try to construct an implicit that can convert from a collection type and modify its nested element all in one go.

```
implicit def naiveWrap[A,B](
  col: java.util.Collection[A])(implicit conv: A => B) =
    new Iterable[B] { ... }
```

The `naiveWrap` method is defined with two type parameters: one for the original type in the Java collection, `A`, and another for the Scala version of that type, `B`. The `naive-Wrap` method takes another implicit conversion from the Java type `A` to the Scala type `B`. The hope is that an implicit view will bind the type parameter `A` to `java.lang.Integer` and `B` to `scala.Int` and the conversion from `java.util.ArrayList [java.lang.Integet]` to `scala.Iterable[Int]` will succeed.

Let's try this out in the REPL:

```
scala> val x = new java.util.ArrayList[java.lang.Integer]
x: java.util.ArrayList[java.lang.Integer] = []

scala> val y : Iterable[Int] = x
<console>:17: error: type mismatch;
 found : java.util.ArrayList[java.lang.Integer]
 required: Iterable[Int]
       val y : Iterable[Int] = x
```

This is the same error as before. The Java list `x` isn't able to be converted to an `Iterable[Int]` directly. This is the same problem we saw before where the type inference doesn't like inferring the `A` and `B` types from the `naiveWrap` method.

The solution to this problem is to defer the type inference of the parameters. Let's try to implement the wrap method again.

```
trait CollectionConverter[A] {
  val col: java.util.Collection[A]
  def asScala[B](implicit fun: A => B) =
    new Iterable[B] { ... }
}
object Test {
  implicit def wrap[A](i: ju.Collection[A]) =
    new CollectionConverter[A] {
      override val col = i
    }
}
```

The `CollectionConverter` type is implemented to capture the original `A` type from the `naiveWrap` method. The `Converter` trait holds the Java collection that needs to be converted. The `asScala` method is defined to capture the `B` type from the `naiveWrap` method. This method takes an implicit argument that captures the conversion from `A` to `B`. The `asScala` method is what constructs the Scala `Iterable`. The `Test` object is defined with a new implicit `wrap` method. This method captures the original `A` type and constructs a new `CollectionConverter`.

The new implicit conversions requires the `asScala` method to be called directly. Let's take a look:

```
scala> import Test.wrap
import Test.wrap

scala> val x = new java.util.ArrayList[java.lang.Integer]
x: java.util.ArrayList[java.lang.Integer] = []

scala> x.add(1); x.add(2);

scala> val y: Iterable[Int] = x.asScala
y : Iterable[Int] = CollectionConverter(1, 2)
```

First, the new implicit wrap method is imported. Next, a Java `ArrayList[java.lang.Integer]` is constructed and values are added to it. Finally, the conversion is attempted using the `asScala` method, and this time it succeeds.

The downside to this approach is the requirement of the additional method call to ensure the types are inferred correctly. But, as a general solution, this is more ideal. The explicit `asScala` method call denotes a transformation to a new object. This makes it easy to know when a collection is being converted between the Scala and Java libraries.

> **SCALAJ-COLLECTIONS** The scalaj-collections library from Jorge Ortiz provides collection conversions to and from Scala and Java collections. The library uses the same technique of having an `asScala` and `asJava` method implicitly added to collections of the respected types. The scalaj library offers a more robust solution than what's available in the standard library.

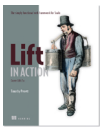
Although using implicits to wrap Java libraries into Scala libraries can be dangerous, it's still a helpful technique and is used throughout the standard library. It's important to know when only simple implicit conversions won't be enough and how to solve these issues. Chaining implicit conversions can solve a lot of the remaining issues.

The important point here is that implicits aren't magic and can't automatically convert between Scala and Java types for all situations. Implicits can and should be used to *reduce* the overhead of these interaction points.

Summary

Using Java from Scala is usually a painless process. This article covered one of the areas of concern and offered the solution. That area of concern is when there exists a solution to a problem in both Scala and Java. The canonical example is the differing collections libraries. The Scala collections API isn't friendly to use from Java, and the Java collections API lacks many of the functional features found in the Scala version. To ease integration between Java portions of code and Scala portions, providing implicit conversions on the Scala side can be beneficial. It's important to be careful here to ensure you don't make assumptions about equality. Using explicit conversion functions can help highlight where object identities are changing. They can also be used to perform more than one implicit coercion.

Here are some other Manning titles you might be interested in:



Lift in Action
Timothy Perrett

Scala in Action
Nilanjan Raychaudhuri

DSLs in Action
Debasish Ghosh

Last updated: June 8, 2012