Chapter 31

Combining Scala and Java

Scala code is often used in tandem with large Java programs and frameworks. Since Scala is highly compatible with Java, most of the time you can combine the languages without worrying very much. For example, standard frameworks, such as Swing, Servlets, and JUnit, are known to work just fine with Scala. Nonetheless, from time to time, you will run into some issue combining Java and Scala.

This chapter describes two aspects of combining Java and Scala. First, it discusses how Scala is translated to Java, which is especially important if you call Scala code from Java. Second, it discusses the use of Java annotations in Scala, an important feature if you want to use Scala with an existing Java framework.

31.1 USING SCALA FROM JAVA

Most of the time you can think of Scala at the source code level. However, you will have a richer understanding of how the system works if you know something about its translation. Further, if you call Scala code from Java, you will need to know what Scala code looks like from a Java point of view.

General rules

Scala is implemented as a translation to standard Java bytecodes. As much as possible, Scala features map directly onto the equivalent Java features. For example, Scala classes, methods, strings, and exceptions are all compiled to the same in Java bytecode as their Java counterparts.

To make this happen required an occasional hard choice in the design of Scala. For example, it might have been nice to resolve overloaded methods at run time, using run-time types, rather than at compile time. Such a design would break with Java's, however, making it much trickier to mesh Java and Scala. In this case, Scala stays with Java's overloading resolution, and thus Scala methods and method calls can map directly to Java methods and method calls.

Scala has its own design for other features. For example, traits have no equivalent in Java. Similarly, while both Scala and Java have generic types, the details of the two systems clash. For language features like these, Scala code cannot be mapped directly to a Java construct, so it must be encoded using some combination of the structures Java does have.

For these features that are mapped indirectly, the encoding is not fixed. There is an ongoing effort to make the translations as simple as possible so, by the time you read this, some details may be different than at the time of writing. You can find out what translation your current Scala compiler uses by examining the ".class" files with tools like javap.

Those are the general rules. Consider now some special cases.

Value types

A value type like Int can be translated in two different ways to Java. Whenever possible, the compiler translates a Scala Int to a Java int to get better performance. Sometimes this is not possible, though, because the compiler is not sure whether it is translating an Int or some other data type. For example, a particular List[Any] might hold only Ints, but the compiler has no way to be sure.

In such cases, where the compiler is unsure whether an object is a value type or not, the compiler uses objects and relies on wrapper classes. For example, wrapper classes such asjava.lang.Integer allow a value type to be wrapped inside a Java object and thereby manipulated by code that needs objects.[1]

Singleton objects

Java has no exact equivalent to a singleton object, but it does have static methods. The Scala translation of singleton objects uses a combination of static and instance methods. For every Scala singleton object, the compiler will create a Java class for the object with a dollar sign added to the end. For a singleton object named App, the compiler produces a Java class namedApp$. This class has all the methods and fields of the Scala singleton object. The Java class also has a single static field named MODULE$ to hold the one instance of the class that is created at run time.

As a full example, suppose you compile the following singleton object:

```
object App {
  def main(args: Array[String]) = {
    println("Hello, world!")
  }
}
```

Scala will generate a Java App$ class with the following fields and methods:

```
$ javap App$
public final class App$ extends java.lang.Object
implements scala.ScalaObject{
    public static final App$ MODULE$;
    public static {};
    public App$();
    public void main(java.lang.String[]);
    public int $tag();
}
```

That's the translation for the general case. An important special case is if you have a "standalone" singleton object, one which does not come with a class of the same name. For example, you might have a singleton object named App, and not have any class named App. In that case, the compiler will create a Java class named App that has a static forwarder method for each method of the Scala singleton object:

```
$ javap App
Compiled from "App.scala"
public final class App extends java.lang.Object{
    public static final int $tag();
    public static final void main(java.lang.String[]);
}
```

To contrast, if you did have a class named App, Scala would create a corresponding Java Appclass to hold the members of the App class you defined. In that case it would not add any forwarding methods for the same-named singleton object, and Java code would have to access the singleton via the MODULE$ field.

Traits as interfaces

Compiling any trait creates a Java interface of the same name. This interface is usable as a Java type, and it lets you call methods on Scala objects through variables of that type.

Implementing a trait in Java is another story. In the general case it is not practical; however, one special case is important. If you make a Scala trait that includes only abstract methods, then that trait will be translated directly to a Java interface with no other code to worry about. Essentially this means that you can write a Java interface in Scala syntax if you like.

31.2 ANNOTATIONS

Scala's general annotations system is discussed in Chapter 27. This section discusses Java-specific aspects of annotations.

Additional effects from standard annotations

Several annotations cause the compiler to emit extra information when targeting the Java platform. When the compiler sees such an annotation, it first processes it according to the general Scala rules, and then it does something extra for Java.

Deprecation For any method or class marked @deprecated, the compiler will add Java's own deprecation annotation to the emitted code. Because of this, Java compilers can issue deprecation warnings when Java code accesses deprecated Scala methods.

Volatile fields Likewise, any field marked @volatile in Scala is given the Java volatilemodifier in the emitted code. Thus, volatile fields in Scala behave exactly according to Java's semantics, and accesses to volatile fields are sequenced precisely according to the rules specified for volatile fields in the Java memory model.

Serialization

Scala's three standard serialization annotations are all translated to Java equivalents.
A@serializable class has Java's Serializable interface added to it.
A @SerialVersionUID(1234L)annotation is converted to the following Java field definition:

```
// Java serial version marker
private final static long SerialVersionUID = 1234L
```

Any variable marked @transient is given the Java transient modifier.

Exceptions thrown

Scala does not check that thrown exceptions are caught. That is, Scala has no equivalent to Java's throws declarations on methods. All Scala methods are translated to Java methods that declare no thrown exceptions.[2]

The reason this feature is omitted from Scala is that the Java experience with it has not been purely positive. Because annotating methods with throws clauses is a heavy burden, too many developers write code that swallows and drops exceptions, just to get the code to compile without adding all those throws clauses. They may intend to improve the exception handling later, but experience shows that all too often time-pressed programmers will never come back and add proper exception handling. The twisted result is that this well-intentioned feature often ends up making code *less* reliable. A large amount of production Java code swallows and hides runtime exceptions, and the reason it does so is to satisfy the compiler.

Sometimes when interfacing to Java, however, you may need to write Scala code that has Java-friendly annotations describing which exceptions your methods may throw. For example, each method in an RMI remote interface is required to mentionjava.io.RemoteException in its throws clause. Thus, if you wish to write an RMI remote interface as a Scala trait with abstract methods, you would need to list RemoteException in the throwsclauses for those methods. To accomplish this, all you have to do is mark your methods with@throws annotations. For example, the Scala class shown in Listing 31.1 has a method marked as throwing IOException.

```
import java.io._
class Reader(fname: String) {
  private val in =
    new BufferedReader(new FileReader(fname))

  @throws(classOf[IOException])
  def read() = in.read()
}
```

Listing 31.1 - A Scala method that declares a Java throws clause.

Here is how it looks from Java:

```
$ javap Reader
Compiled from "Reader.scala"
public class Reader extends java.lang.Object implements
scala.ScalaObject{
    public Reader(java.lang.String);
    public int read()       throws java.io.IOException;
    public int $tag();
}
$
```

Note that the read method indicates with a Java throws clause that it may throw an IOException.

Java annotations

Existing annotations from Java frameworks can be used directly in Scala code. Any Java framework will see the annotations you write just as if you were writing in Java.

A wide variety of Java packages use annotations. As an example, consider JUnit 4. JUnit is a framework for writing and running automated tests. The latest version, JUnit 4, uses annotations to indicate which parts of your code are tests. The idea is that you write a lot of tests for your code, and then you run those tests whenever you change the source code. That way, if your changes add a new bug, one of the tests will fail and you will find out immediately.

Writing a test is easy. You simply write a method in a top-level class that exercises your code, and you use an annotation to mark the method as a test. It looks like this:

```
import org.junit.Test
import org.junit.Assert.assertEquals

class SetTest {

  @Test
  def testMultiAdd = {
    val set = Set() + 1 + 2 + 3 + 1 + 2 + 3
    assertEquals(3, set.size)
  }
}
```

The testMultiAdd method is a test. This test adds multiple items to a set and makes sure that each is added only once. The assertEquals method, which comes as part of the JUnit API, checks that its two arguments are equal. If they are different, then the test fails. In this case, the test verifies that repeatedly adding the same numbers does not increase the size of a set.

The test is marked using the annotation org.junit.Test. Note that this annotation has been imported, so it can be referred to as simply @Test instead of the more cumbersome@org.junit.Test.

That's all there is to it. The test can be run using any JUnit test runner. Here it is being run with the command-line test runner:

```
$ scala -cp junit-4.3.1.jar:. org.junit.runner.JUnitCore SetTest
JUnit version 4.3.1
.
Time: 0.023

OK (1 test)
```

Writing your own annotations

To make an annotation that is visible to Java reflection, you must use Java notation and compile it with javac. For this use case, writing the annotation in Scala does not seem helpful, so the standard compiler does not support it. The reasoning is that the Scala support would inevitably fall short of the full possibilities of Java annotations, and further, Scala will probably one day have its own reflection, in which case you would want to access Scala annotations with Scala reflection.

Here is an example annotation:

```
import java.lang.annotation.*; // This is Java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Ignore { }
```

After compiling the above with javac, you can use the annotation as follows:

```
object Tests {
  @Ignore
  def testData = List(0, 1, -1, 5, -5)

  def test1 = {
    assert(testData == (testData.head :: testData.tail))
  }

  def test2 = {
    assert(testData.contains(testData.head))
  }
}
```

In this example, test1 and test2 are supposed to be test methods, but testData should be ignored even though its name starts with "test".

To see when these annotations are present, you can use the Java reflection APIs. Here is sample code to show how it works:

```
for {
  method <- Tests.getClass.getMethods
  if method.getName.startsWith("test")
  if method.getAnnotation(classOf[Ignore]) == null
} {
  println("found a test method: " + method)
}
```

Here, the reflective methods getClass and getMethods are used to inspect all the fields of the input object's class. These are normal reflection methods. The annotation-specific part is the use of method getAnnotation. Many reflection objects have a getAnnotation method for searching for annotations of a specific type. In this case, the code looks for an annotation of our newIgnore type. Since this is a Java API, success is indicated by whether the result is null or an actual annotation object.

Here is the code in action:

```
$ javac Ignore.java
$ scalac Tests.scala
$ scalac FindTests.scala
$ scala FindTests
found a test method: public void Tests$.test2()
found a test method: public void Tests$.test1()
```

As an aside, notice that the methods are in class Tests$ instead of class Tests when viewed with Java reflection. As described at the beginning of the chapter, the implementation of a Scala singleton object

is placed in a Java class with a dollar sign added to the end of its name. In this case, the implementation of Tests is in the Java class Tests$.

Be aware that when you use Java annotations you have to work within their limitations. For example, you can only use constants, not expressions, in the arguments to annotations. You can support @serial(1234) but not @serial(x * 2), because x * 2 is not a constant.

31.3 WILDCARD TYPES

All Java types have a Scala equivalent. This is necessary so that Scala code can access any legal Java class. Most of the time the translation is straightforward. Pattern in Java is Patternin Scala, and Iterator<Component> in Java is Iterator[Component] in Scala. For some cases, though, the Scala types you have seen so far are not enough. What can be done with Java wildcard types such as Iterator<?> or Iterator<? extends Component>? What can be done about raw types like Iterator, where the type parameter is omitted? For Java wildcard types and raw types, Scala uses an extra kind of type also called a *wildcard type*.

Wildcard types are written using *placeholder syntax*, just like the short-hand function literals described in Section 8.5. In the short hand for function literals, you can use an underscore (_) in place of an expression; for example, (_ + 1) is the same as (x => x + 1). Wildcard types use the same idea, only for types instead of expressions. If you write Iterator[_], then the underscore is replacing a type. Such a type represents an Iterator where the element type is not known.

You can also insert upper and lower bounds when using this placeholder syntax. Simply add the bound after the underscore, using the same <: syntax used with type parameters (Section 19.8 and Section 19.5). For example, the type Iterator[_ <: Component] is an iterator where the element type is not known, but whatever type it is, it must be a subtype of Component.

That's how you write a wildcard type, but how do you use it? In simple cases, you can ignore the wildcard and call methods on the base type. For example, suppose you had the following Java class:

```
// This is a Java class with wildcards
public class Wild {
  public Collection<?> contents() {
    Collection<String> stuff = new Vector<String>();
    stuff.add("a");
    stuff.add("b");
    stuff.add("see");
    return stuff;
  }
}
```

If you access this in Scala code you will see that it has a wildcard type:

```
scala> val contents = (new Wild).contents
contents: java.util.Collection[_] = [a, b, see]
```

If you want to find out how many elements are in this collection, you can simply ignore the wildcard part and call the size method as normal:

```
scala> contents.size()
res0: Int = 3
```

In more complicated cases, wildcard types can be more awkward. Since the wildcard type has no name, there is no way to use it in two separate places. For example, suppose you wanted to create a mutable Scala set and initialize it with the elements of contents:

```
import scala.collection.mutable
val iter = (new Wild).contents.iterator
val set = mutable.Set.empty[???]     // what type goes here?
while (iter.hasMore)
  set += iter.next()
```

A problem occurs on the third line. There is no way to name the type of elements in the Java collection, so you cannot write down a satisfactory type for set. To work around this kind of problem, here are two tricks you should consider:

1. When passing a wildcard type into a method, give a parameter to the method for the placeholder. You now have a name for the type that you can use as many times as you like.
2. Instead of returning wildcard type from a method, return an object that has abstract members for each of the placeholder types. (See Chapter 20 for information on abstract members.)

Using these two tricks together, the previous code can be written as follows:

```
import scala.collection.mutable
import java.util.Collection

abstract class SetAndType {
  type Elem
  val set: mutable.Set[Elem]
}

def javaSet2ScalaSet[T](jset: Collection[T]): SetAndType = {
  val sset = mutable.Set.empty[T]  // now T can be named!

  val iter = jset.iterator
  while (iter.hasNext)
    sset += iter.next()

  return new SetAndType {
    type Elem = T
    val set = sset
  }
}
```

You can see why Scala code normally does not use wildcard types. To do anything sophisticated with them, you tend to convert them to use abstract members. So you may as well use abstract members to begin with.

31.4 COMPILING SCALA AND JAVA TOGETHER

Usually when you compile Scala code that depends on Java code, you first build the Java code to class files. You then build the Scala code, putting the Java code's class files on the classpath. However, this

approach doesn't work if the Java code has references back into the Scala code. In such a case, no matter which order you compile the code, one side or the other will have unsatisfied external references. These situations are not uncommon; all it takes is a mostly Java project where you replace one Java source file with a Scala source file.

To support such builds, Scala allows compiling against Java source code as well as Java class files. All you have to do is put the Java source files on the command line as if they were Scala files. The Scala compiler won't compile those Java files, but it will scan them to see what they contain. To use this facility, you first compile the Scala code using Java source files, and then compile the Java code using Scala class files.

Here is a typical sequence of commands:

```
$ scalac -d bin InventoryAnalysis.scala InventoryItem.java \
       Inventory.java
$ javac -cp bin -d bin Inventory.java InventoryItem.java \
       InventoryManagement.java
$ scala -cp bin InventoryManagement
Most expensive item = sprocket($4.99)
```

31.5 JAVA 8 INTEGRATION IN SCALA 2.12

Java 8 added a few improvements to the Java language and bytecodes that Scala takes advantage of in its 2.12 release.[3] By exploiting new features of Java 8, the Scala 2.12 compiler can generate smaller class and jar files and improve the binary compatibility of traits.

Lambda expressions and "SAM" types

From the Scala programmer's perspective, the most visible Java 8-related enhancement in Scala 2.12 is that Scala function literals can be used like Java 8 lambda expressions as a more concise form for anonymous class instance expressions. To pass behavior into a method prior to Java 8, Java programmers often defined anonymous inner class instances, like this:

```
JButton button = new JButton(); // This is Java
button.addActionListener(
  new ActionListener() {
    public void actionPerformed(ActionEvent event) {
      System.out.println("pressed!");
    }
  }
);
```

In this example, an anonymous instance of ActionListener is created and passed to theaddActionListener of a Swing JButton. When a user clicks on the button, Swing will invoke theactionPerformed method on this instance, which will print "pressed!".

In Java 8, a lambda expression can be used anywhere an instance of a class or interface that contains just a single abstract method (SAM) is required. ActionListener is such an interface, because it contains a single abstract method, actionPerformed. Thus a lambda expression can be used to register an action listener on a Swing button. Here's an example:

```
JButton button = new JButton(); // This is Java 8
button.addActionListener(
  event -> System.out.println("pressed!")
);
```

In Scala, you could also use an anonymous inner class instance in the same situation, but you might prefer to use a function literal, like this:

```
val button = new JButton
button.addActionListener(
  _ => println("pressed!")
)
```

As you have already seen in Section 21.1, you could support such a coding style by defining an implicit conversion from the ActionEvent => Unit function type to ActionListener.

Scala 2.12 enables a function literal to be used in this case even in the absence of such an implicit conversion. As with Java 8, Scala 2.12 will allow a function type to be used where an instance of a class or trait declaring a single abstract method (SAM) is required. This will work with any SAM in Scala 2.12. For example, you might define a trait, Increaser, with a single abstract method, increase:

```
scala> trait Increaser {
         def increase(i: Int): Int
       }
defined trait Increaser
```

You could then define a method that takes an Increaser:

```
scala> def increaseOne(increaser: Increaser): Int =
         increaser.increase(1)
increaseOne: (increaser: Increaser)Int
```

To invoke your new method, you could pass in an anonymous instance of trait Increaser, like this:

```
scala> increaseOne(
         new Increaser {
           def increase(i: Int): Int = i + 7
         }
       )
res0: Int = 8
```

In Scala 2.12, however, you could alternatively just use a function literal, because Increaser is a SAM type:

```
scala> increaseOne(i => i + 7) // Scala 2.12
res1: Int = 8
```

Using Java 8 Streams from Scala 2.12

Java's Stream is a functional data structure that offers a map method taking ajava.util.function.IntUnaryOperator. From Scala you could invoke Stream.map to increment each element of an Array, like this:

```
scala> import java.util.function.IntUnaryOperator
import java.util.function.IntUnaryOperator

scala> import java.util.Arrays
import java.util.Arrays

scala> val stream = Arrays.stream(Array(1, 2, 3))
stream: java.util.stream.IntStream = ...

scala> stream.map(
         new IntUnaryOperator {
           def applyAsInt(i: Int): Int = i + 1
         }
       ).toArray
res3: Array[Int] = Array(2, 3, 4)
```

Because IntUnaryOperator is a SAM type, however, you could in Scala 2.12 invoke it more concisely with a function literal:

```
scala> val stream = Arrays.stream(Array(1, 2, 3))
stream: java.util.stream.IntStream = ...

scala> stream.map(i => i + 1).toArray // Scala 2.12
res4: Array[Int] = Array(2, 3, 4)
```

Note that only function literals will be adapted to SAM types, not arbitrary expressions that have a function type. For example, consider the following val, f, which has type Int => Int:

```
scala> val f = (i: Int) => i + 1
f: Int => Int = ...
```

Although f has the same type as the function literal passed to stream.map previously, you can't use f where an IntUnaryOperator is required:

```
scala> val stream = Arrays.stream(Array(1, 2, 3))
stream: java.util.stream.IntStream = ...

scala> stream.map(f).toArray
<console>:16: error: type mismatch;
 found   : Int => Int
 required: java.util.function.IntUnaryOperator
       stream.map(f).toArray
                  ^
```

To use f, you can explicitly call it using a function literal, like this:

```
scala> stream.map(i => f(i)).toArray
res5: Array[Int] = Array(2, 3, 4)
```

Or, you could annotate f with IntUnaryOperator, the type expected by Stream.map, when you define f:

```
scala> val f: IntUnaryOperator = i => i + 1
f: java.util.function.IntUnaryOperator = ...

scala> val stream = Arrays.stream(Array(1, 2, 3))
stream: java.util.stream.IntStream = ...
```

```
scala> stream.map(f).toArray
res6: Array[Int] = Array(2, 3, 4)
```

With Scala 2.12 and Java 8, you can also invoke methods compiled with Scala from Java, passing Scala function types using Java lambda expressions. Although Scala function types are defined as traits that include concrete methods, Scala 2.12 compiles traits to Java interfaces with default methods, a new feature of Java 8. As a result, Scala function types appear to Java as SAMs.

31.6 CONCLUSION

Most of the time, you can ignore how Scala is implemented, and simply write and run your code. But sometimes it is nice to "look under the hood," so this chapter has gone into three aspects of Scala's implementation on the Java platform: What the translation looks like, how Scala and Java annotations work together, and how Scala's wildcard types let you access Java wildcard types. It also covered using Java's concurrency primitives from Scala and compiling combined Scala and Java projects. These topics are important whenever you use Scala and Java together.

Footnotes for Chapter 31:

[1] The implementation of value types was discussed in detail in Section 11.2.

[2] The reason it all works is that the Java bytecode verifier does not check the declarations anyway! The Java compiler checks, but not the verifier.

[3] Scala 2.12 requires Java 8 so that it can take advantage of Java 8 features.