Chapter 22. Java Interoperability

Of all the alternative JVM languages, Scala's interoperability with Java source code is among the most seamless. This chapter begins with a discussion of interoperability with code written in Java.

Because Scala syntax is primarily a superset of Java syntax, invoking Java code from Scala is usually straightforward. Going the other direction requires that you understand how some Scala features are encoded in byte code while still satisfying the JVM specification. We discuss several of the interoperability issues here.

Using Java Names in Scala Code

Java's rules for type, method, field, and variable names are more restrictive than Scala's rules. So, in *almost* all cases, you can just use the Java names in Scala code. You can create new instances of Java types, call Java methods, and use Java variables and instance fields.

The exception is when a Java name is actually a Scala keyword. As we saw in Reserved Words, "escape" the name with single back ticks. For example, consider the `match` keyword in Scala and the `match` method on `java.util.Scanner`. Call the latter with ``myScanner.`match` ``.

Java and Scala Generics

All along, we've been using Java types in Scala code, like `java.lang.String`. You can even use Java *generic* classes, such as Java collections in Scala.

What about using Scala parameterized types in Java? Consider the following JUnit 4 test, which uses `scala.collection.mutable.LinkedHashMap` and `scala.Option`. It shows some of the idiosyncrasies you might encounter:

```
// src/test/java/progscala2/javainterop/SMapTest.java
import org.junit.*;
import org.junit.runner.RunWith;
import org.junit.runners.JUnit4;
import static org.junit.Assert.*;
import scala.*;
import scala.collection.mutable.LinkedHashMap;

public class SMapTest extends org.scalatest.junit.JUnitSuite {        //
❶
  static class Name {
    public String firstName;
    public String lastName;

    public Name(String firstName, String lastName) {
      this.firstName = firstName;
      this.lastName  = lastName;
    }
  }

  LinkedHashMap<Integer, Name> map;

  @Before
  public void setup() {
    map = new LinkedHashMap<Integer, Name>();
    map.update(1, new Name("Dean", "Wampler"));
  }

  @Test
  public void usingMapGetWithOptionName() {
//  ❷
    assertEquals(1, map.size());
                                              // Note:
    Option<Name> n1 = map.get(1);   Option<Name>
    assertTrue(n1.isDefined());
    assertEquals("Dean", n1.get().firstName);
  }

  @Test
  public void usingMapGetWithOptionExistential() {                        //
❸
    assertEquals(1, map.size());
                                             // Note: Option<?
    Option<?> n1 = map.get(1);     >
    assertTrue(n1.isDefined());
    assertEquals("Dean", ((Name) n1.get()).firstName);
 }
}
```

❶

This JUnit test will be executed by ScalaTest if `JUnitSuite` is mixed in.

❷

A test using typed values.

❸

A test using existential types for the values.

You can also use Scala's tuple types, although you can't exploit Scala's syntactic sugar, e.g., `("someString", 101)`:

```
// src/test/java/progscala2/javainterop/ScalaTuples.java
package progscala2.javainterop;
import scala.Tuple2;

public class ScalaTuples {
  public static void main(String[] args) {
    Tuple2 stringInteger = new Tuple2<String,Integer>("one", 2
);

    System.out.println(stringInteger);
  }
}
```

However, the `FunctionN` types are very difficult to use from Java due to "hidden" members that the compiler synthesizes automatically. For example, attempting to compile the following code will fail:

```
//
src/test/java/progscala2/javainterop/ScalaFunctions.javaX
package progscala2.javainterop;
import scala.Function1;

public class ScalaFunctions {
  public static void main(String[] args) {

// Fails to compile, due to missing methods the Scala compiler would
add.
    Function1 stringToInteger = new Function1<String,Integer>() {
      public Integer apply(String s) {
        Integer.parseInt(s);
      }
    };

    System.out.println(stringToInteger("101"));
  }
}
```

The compiler will complain that the abstract method `apply$mcVJ$sp(long)` is undefined. The Scala compiler would generate this for us.

This severely limits the ability to call the higher-order functions in Scala's collections from Java code. You might try to pass a Java 8 *lambda* where a `scala.FunctionN` is expected, but they are incompatible. (The plan for Scala 2.12 is to unify Scala Functions and Java lambdas, eliminating this incompatibility.)

Hence, if you want to call a Scala API from Java, you can't call *higher-order* methods, those that take functions arguments or return functions.

JavaBean Properties

We saw in Chapter 8 that Scala does not follow the *JavaBeans* conventions for field reader and writer methods, in order to support the more useful *Uniform Access Principle*.

However, there are times when you need JavaBeans accessor methods. For example, some *dependency injection* frameworks exploit them. Also, JavaBeans accessor methods are used by IDEs that support bean "introspection."

Scala solves this problem with an annotation that you can apply to a field, `@scala.beans.BeanProperty`, which tells the compiler to generate JavaBeans-style getter and setter methods. The `scala.beans` package also contains other annotations for configuring bean properties, etc.

For Scala 2.10 and earlier, the package name is actually `scala.reflect` for the JavaBeans annotations.

For example, we can annotate the fields of the `Complex` class we saw previously:

```
//
src/main/scala/progscala2/javainterop/ComplexBean.scala
package progscala2.javainterop

// Scala 2.11. For Scala 2.10 and earlier, use
scala.reflect.BeanProperty.
case class ComplexBean(
  @scala.beans.BeanProperty real: Double,
  @scala.beans.BeanProperty imaginary: Double) {

  def +(that: ComplexBean) =
    new ComplexBean(real + that.real, imaginary + that.imaginary)
  def -(that: ComplexBean) =
    new ComplexBean(real - that.real, imaginary - that.imaginary)
}
```

This class has already been compiled by SBT. If you decompile the *ComplexBean.class* file, you'll see the following methods in the output:

```
$ javap -cp target/scala-2.11/classes javainterop.ComplexBean
...
  public double real();
  public double imaginary();
  ...
  public double getReal();
  public double getImaginary();
  ...
}
```

No setters are shown, because the fields are immutable. In contrast, decompiling the original `Complex` reveals only the `real()` and `imaginary()` methods. Hence, even when you use the `BeanProperty` annotation, you still get the normal field reader and optionally writer methods.

AnyVal Types and Java Primitives

Notice also in the previous `Complex` example that the `Double` fields are compiled to Java primitive `doubles`. All the `AnyVal` types are converted to their corresponding Java primitives. In particular, `Unit` is mapped to `void`.

Scala Names in Java Code

Scala allows more flexible identifiers, e.g., *operator characters* like `*`, `<`, etc., which aren't allowed in byte-code identifiers. Hence, these characters are encoded (or "mangled") to satisfy the JVM constraints. They are translated as shown in Table 22-1.

Table 22-1. Encoding of operator characters

Operator	Encoding	Operator	Encoding	Operator	Encoding	Operator	Encoding
`=`	`$eq`	`>`	`$greater`	`<`	`$less`		
`+`	`$plus`	`-`	`$minus`	`*`	`$times`	`/`	`$div`
`\`	`$bslash`	`|`	`$bar`	`!`	`$bang`	`?`	`$qmark`
`:`	`$colon`	`%`	`$percent`	`^`	`$up`	`&`	`$amp`

Recap and What's Next

An important benefit of Scala is that you can continue using existing Java code. Calling Java from Scala is easy (with a few exceptions).

Our next chapter covers application design considerations essential for truly succeeding with Scala.