Chapter 44
Scala & Java Interoperability

44.1 Introduction

In this chapter, we will look at the interoperation of Java and Scala. Both Java and Scala are JVM byte code languages. That is, they both compile to the byte code language that is understood by the JVM. The byte code language of the JVM was originally designed to be the *compiled* form of Java and was what the Java Virtual Machine executed. However, things have evolved such that today the JVM is a virtual environment for executing byte code languages. In fact, there are now several languages that can be compiled to JVM byte codes including Java, Groovy, Clojure, Jruby, Jython, JavaScript, Ada, Pascal as well as Scala. A common term used for these languages is that they are all byte code languages.

As such at the byte code level, there is no difference between Java and Scala—they are just different starting points for the same destination. Therefore, at runtime it is only the byte code that executes—there is no such thing as Java or Scala. Scala can thus interoperate with other byte code languages.

44.2 A Simple Example

As a simple example, consider the Scala Person shown below:

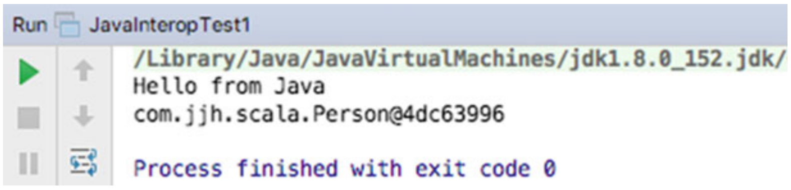
```
package com.jjh.java

class Person (name: String="John", var age:Int=47)
```

This class compiles to a `Person.class` file just as any other byte code language. This means that it can be used within Scala or Java. The following code sample illustrates the use of the Scala class Person within a Java application:

```
package com.jjh.java;

/**
 * This is a standard Java class with a main method.
 * But note that Person is a Scala class.
 *
 * This illustrates the interop between Scala and
 * Java.
 */
public class JavaInteropTest1 {

    public static void main(String [] args) {
        System.out.println("Hello from Java");
        Person p = new Person("Granny", 85);
        System.out.println(p);
    }

}
```

Notice that as far as Java is concerned that this is a class `Person` with a constructor that takes a `String` and an Integer. Also notice that both the Scala class `Person` and the Java class JavaInteropTest1 are in the same package (com. jjh.scala). This works because from a byte code point of view, there is no difference between a Scala class `Person` in the package com.jjh.scala and a Java class `Person` in the same package—they are both byte classes in the package com.jjh. scala.

The output from this application is given blow in IntelliJ IDEA:



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/
Hello from Java
com.jjh.scala.Person@4dc63996

Process finished with exit code 0
```

One thing to note is that the Scala class `Person` defines some slightly strange (from a Java perspective) methods for the property age. It defines the age() accessor method (not getAge) and the age_$eq method for setting the new value of age (as it's a var):

```
System.out.println(p.age());
p.age_$eq(55);
System.out.println(p.age());
```

The output of these lines is:

```
85
55
```

44.3 Inheritance

It is possible to inherit between Java and Scala classes. For example, in the following, Employee is a Java class while Person is the Scala class presented in the previous section. The rules for constructors are maintained, and the Java class can call the superclass constructor.

```
package com.jjh.java;

/**
 * This is a Java class that extends a Scala class!
 *
 * Note the call to super for the parent class
 * constructor.
 */
public class Employee extends Person {

   private String company;

   public Employee(String name, int age, String company) {
      super(name, age);
      this.company = company;
   }

   public String toString() {
      return super.toString() + ", " + company;
   }

   }
```

44.4 Issues

There are of course some issues related to interoperating between Scala and Java. The first one of which is that Scala requires a Java 8 or newer runtime to operate within. The other is that Scala has some concepts that Java has no knowledge of such as *objects*, *traits* and *functions* as first-class language elements. This last may change with the future versions of Java. In turn Scala has no concept of an Interface. Therefore, if care is not taken, problems may arise.

In general, Scala to Java interoperability is relatively seamless as Scala builds on top of Java. However, Java to Scala can sometimes be problematic. The most common set of issues are:

- Java has no equivalent of Traits.
- Functions are object values.
- The Scala-type system is more complex.

- Scala has no notion of static, so can't access statics in the same way as Java code.
- Java doesn't understand Scala's companion module.
- Java sees Scala objects as a final class with statics.

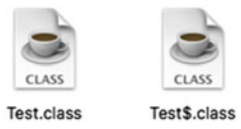
44.4.1 Scala Objects

Of course, the underlying byte code representation also lacks many of the features of Scala presented above. Scala therefore often generates more than one byte code class or type for a single Scala concept. This is illustrated below for a simple test Scala object.

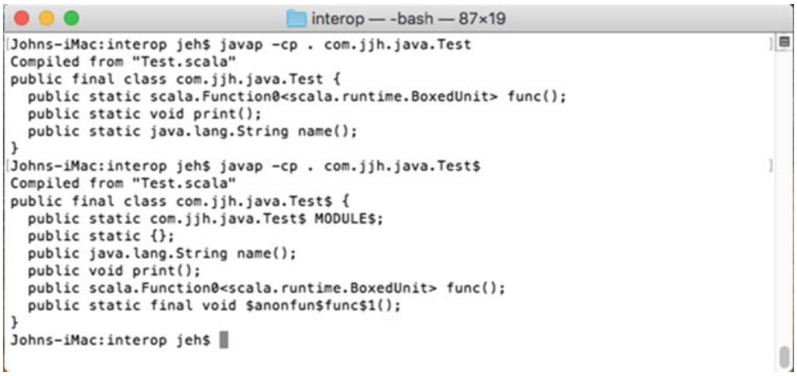
The following code generates two .class files:

```
package com.jjh.java

object Test {
  val name = "MyName"
  def print = println(s"Print $name")
  val func = () => println("Printer function $name")
}
```

The two .class files are Test.class and Test$.class:



This can make it difficult to decide how to treat a Scala concept from the Java side. It can be useful to examine what these class files contain in order to understanding the Java view of these structures. This can be done using the javap program. javap is the class file disassembler distributed with the standard Oracle SDK. The result of using the javap program with the Test examples is shown below:



```
Johns-iMac:interop jeh$ javap -cp . com.jjh.java.Test
Compiled from "Test.scala"
public final class com.jjh.java.Test {
  public static scala.Function0<scala.runtime.BoxedUnit> func();
  public static void print();
  public static java.lang.String name();
}
Johns-iMac:interop jeh$ javap -cp . com.jjh.java.Test$
Compiled from "Test.scala"
public final class com.jjh.java.Test$ {
  public static com.jjh.java.Test$ MODULE$;
  public static {};
  public java.lang.String name();
  public void print();
  public scala.Function0<scala.runtime.BoxedUnit> func();
  public static final void $anonfun$func$1();
}
Johns-iMac:interop jeh$
```

What this shows is that the Scala object *Test* is represented at the byte code level by a public final class Test and a second public final class Test$.

How would you then call the Scala object Test from java? From the javap de-compilation, you have a choice of the Test class with a static method `print` and the `Test$` class with a non-static method `print`. In fact, in this case from Java we can just treat the object `Test` as if it was a statically defined entity. This is because the static method on a final class means that we cannot extend the class Test, and then we can call the method without needing to instantiate the class. Thus, we can just write:

```
package com.jjh.test;

import com.jjh.java.Test;

public class JavaTest {

   public static void main(String[] args) {
      Test.print();
   }

}
```

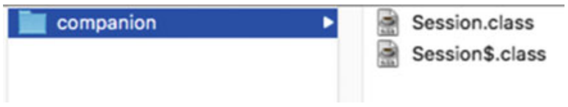
Which is semantically the closest we can get in Java to the concept embodied in the Scala Object.

44.4.2 Companion Modules

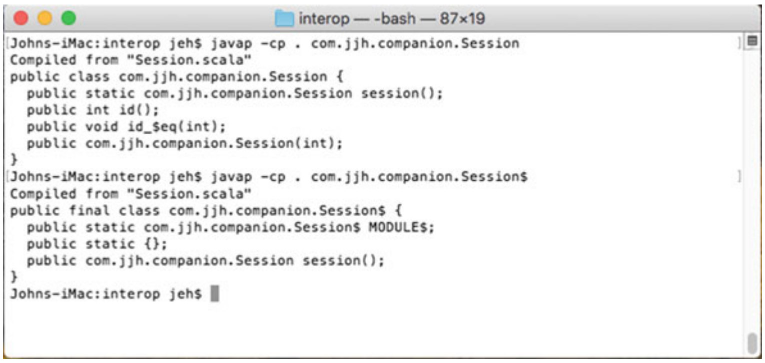
Companion modules are another point of conflict in that Scala has the concept of a class and an associated companion object (which must have the same name as the class and be defined within the same file). For example,

```
package com.jjh.companion

/**
 * The Companion class
 */
class Session(var id: Int) {

}

/**
 * Its Companion (singleton) object
 */
object Session {
  private var counter = 0
  def session() = {
    counter = counter + 1
    new Session(counter)
  }
}
```

This listing when compiles generates the classes shown below:



If we use the javap de-compiler again on these classes, then we can see that the byte code Session class combines elements of both the Scala Session class and the Scala Session object. However, note that the Session is not final and thus can be extended! This is shown below where the javap program is used to de-compile both the Session and the Session$ classes.



```
Johns-iMac:interop jeh$ javap -cp . com.jjh.companion.Session
Compiled from "Session.scala"
public class com.jjh.companion.Session {
  public static com.jjh.companion.Session session();
  public int id();
  public void id_$eq(int);
  public com.jjh.companion.Session(int);
}
Johns-iMac:interop jeh$ javap -cp . com.jjh.companion.Session$
Compiled from "Session.scala"
public final class com.jjh.companion.Session$ {
  public static com.jjh.companion.Session$ MODULE$;
  public static {};
  public com.jjh.companion.Session session();
}
Johns-iMac:interop jeh$ 
```

If you need to access the Session from within Java code, then this can be done as the Session just looks like a normal Java class with a static factory method:

```
package com.jjh.companion;

public class JavaTest {

   public static void main(String[] args) {
      Session session = Session.session();
      System.out.println(session.id());
   }

}
```

Indeed you can even be able to extend it. For example,

```
package com.jjh.companion;

public class ExtendedSession extends Session{

   public ExtendedSession(int id) {
      super(id);
   }

   }
```

44.4.3 Traits

Scala has Traits—these are a type within the Scala-type system, and they are neither abstract classes nor are they Java interfaces. Java does not have Traits although it does have interfaces and abstract classes. There cannot therefore be a direct mapping from a Scala trait to a Java concept. However, this is also true of the underlying byte code representation—it does not have a concept of a Trait. This raises the question what happens when a trait is defined at the byte code level? For example, given the trait Model shown below, how is this represented at the byte code level:

```
package com.jjh.traits

trait Model {

  def info(x: String):String

}
```

If you examine the *.class files* generated for this type, you will see that the single trait Model is represented by a single .class file Model.class (Fig. 44.1).

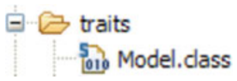
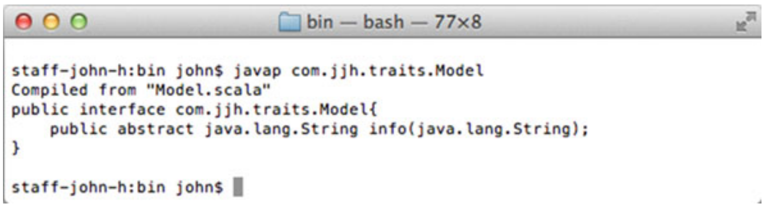


Fig. 44.1 Representing a trait via two class files



```
staff-john-h:bin john$ javap com.jjh.traits.Model
Compiled from "Model.scala"
public interface com.jjh.traits.Model{
    public abstract java.lang.String info(java.lang.String);
}

staff-john-h:bin john$ 
```

Fig. 44.2 Representing a simple trait in byte codes

The file Model.class defines an interface containing a single public abstract method that takes a string and returns a string. This can be implemented by Java class who wish to implement the Trait (Fig. 44.2).

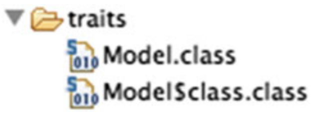
However, what happens if the trait defines actual behaviour and data. This is the biggest area of change in terms of Java interoperability from pre-Scala 2.12 to post-Scala 2.12. Prior to Scala 2.12, a trait was represented as an interface and a class that held any concrete implementations defined in the trait.

In Java 7 and older versions of Java, interfaces were only allowed to define method signatures (abstract methods) and static final constant values.

The following listing defines a modified version of the Model trait that contains behaviour (the print method) and data (the title variable):

```
package com.jjh.traits

trait Model {

  var title = "CS123-10"

  def info(x: String):String

  def print = println("Hello World")

}
```

This is used to result in two class files being generated as shown in here:



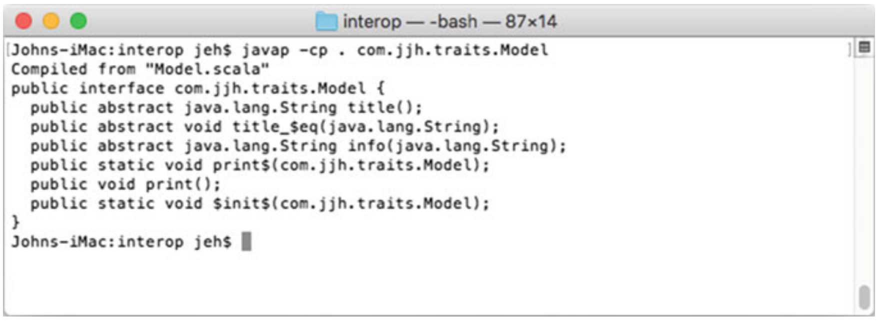
```
▼ traits
      Model.class
      Model$class.class
```

In this older version of Scala, the Model.class file contained an interface definition `Model` with a single public abstract method `info`. The Model$class.class file contained an abstract class called Model$class that extends the java.lang.Object type and defined two static methods one `info` and the other `$init$` both of which take a Model as their parameters.

This made interoperability of such traits with Java difficult!

However, Java 8 introduced new features to *interfaces* that allow interfaces to have concrete methods defined. This means that Scala 2.12 (and newer) is able to compile a trait to a single interface class file.

This is shown below:



```
Johns-iMac:interop jeh$ javap -cp . com.jjh.traits.Model
Compiled from "Model.scala"
public interface com.jjh.traits.Model {
  public abstract java.lang.String title();
  public abstract void title_$eq(java.lang.String);
  public abstract java.lang.String info(java.lang.String);
  public static void print$(com.jjh.traits.Model);
  public void print();
  public static void $init$(com.jjh.traits.Model);
}
Johns-iMac:interop jeh$ 
```

This means that from the Java world, a Trait appears as an *interface* to implement.

However, there are still a couple of things to bear in mind. The first is that the interface that has been generated has three abstract methods that must be provided by any Java class that wish to *implement* this Interface. The first of these is fairly obvious; the method `info(x: String)` is an abstract method in the Scala trait is therefore an abstract method in the Java interface.

However, the other two methods are

```
String title() and
void title_eq(String)
```

Neither of these methods I listed in the Scala trait—instead there is a var property called title. What has happened here is that the Scala compiler converts a `val` or a `var` into 1 or two methods (one for reading and one for writing) the value. As this is a `var,` there are two methods required. However, no default implementation is given and thus the subclass must provide appropriate implementations.

In this case, however, the `var` is initialised to the string “CS123-10”. It is therefore also necessary to provide a way to initialise such properties. This is provided by the slightly strangely named method:

public static void $init$(com.jjh.traits.Model)

This method can be invoked in the subclass (e.g. in the constructor) to initialise any properties defined in the Trait. This is done via reference to the interface name:

```
Model.$init$(this);
```

The end result is that the class implementing the Model *interface* in Java looks like this:

```
package com.jjh.traits;

public class Foo implements Model {

   private String _title;

   public Foo() {
      Model.$init$(this);
   }

   public String info(String x) {
      return "title: " + this.title() + " with " + x;
   }

   public String title() {
      return this._title;
   }

   public void title_$eq(String s) {
      System.out.println("Setting title: " + s);
      this._title = s;
   }

}
```

As this example illustrates, some further work is still involved, so care must be taken if a trait is meant to be implemented in Java. Briefly, if a trait does any of the following, its subclasses require synthetic code:

- defining fields (val or var, but a constant is ok—final val without result type)
- calling super
- initialiser statements in the body
- extending a class
- relying on linearisation to find implementations in the right supertrait

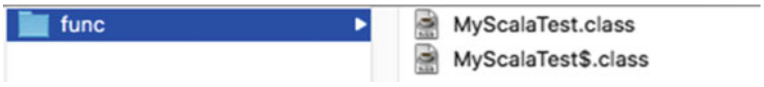
44.5 Functions

Scala is a hybrid Object-Oriented and functional language. However, Java (at least up until Java 8) is an Object-Oriented language and thus has no concept of a Function. Scala of course treats Functions as top-level entities, or first-class elements in the language with functions making up part of the type system of the language. However, the underlying byte code representation to which Scala compiles also does not have a concept of a function thus there must be some from of mapping from the Scala world into the byte code world. This thus means that from the Java side of things that mapping can be exploited.

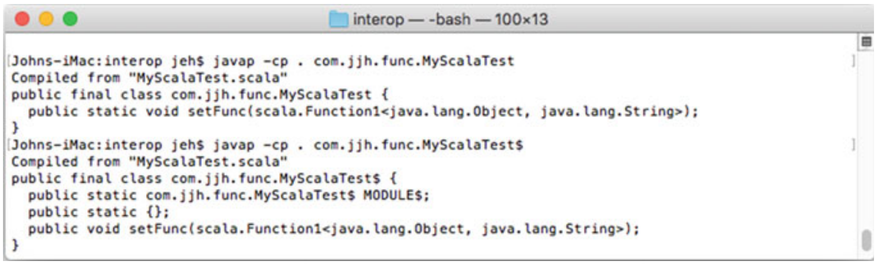
Functions are actually represented at the byte code level by the various Function types that model a function. The following code which defines an object called `MyScalaTest` containing a method which takes a single parameter of type `Int => String`. This method is represented at the byte code level as being a method that takes a single parameter of type Function1 which is parameterised to use an Object and a String as the types involved.

```
package com.jjh.func

object MyScalaTest {

  def setFunc(func: Int => String) {
    println(func(10))
  }

}
```

The object MyScalaTest is represented by two *.class* files at the byte code level as illustrated below.



This is shown when we use javap to de-compile the compiled version of the MyScalaTest. The MyScalaTest.class contains a final class that defines a single public static method setFunc that takes a parameter of type `scala.Function1`. The associated `MyScalaTest$` class defines a non-static (instance side) method `setFunc` that also takes a `scala.Function1` parameter (Fig. 44.3).



```
Johns-iMac:interop jeh$ javap -cp . com.jjh.func.MyScalaTest
Compiled from "MyScalaTest.scala"
public final class com.jjh.func.MyScalaTest {
  public static void setFunc(scala.Function1<java.lang.Object, java.lang.String>);
}
Johns-iMac:interop jeh$ javap -cp . com.jjh.func.MyScalaTest$
Compiled from "MyScalaTest.scala"
public final class com.jjh.func.MyScalaTest$ {
  public static com.jjh.func.MyScalaTest$ MODULE$;
  public static {};
  public void setFunc(scala.Function1<java.lang.Object, java.lang.String>);
}
```

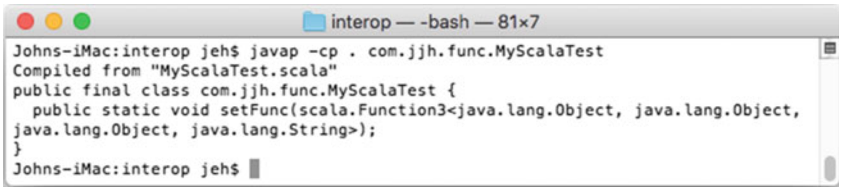
Fig. 44.3 De-compiling the MyScalaTest class files

In fact, within Scala there are a range of types (actually Traits) that are used to represent functions from Function1, Function2 and Function3 through to `Function22`. Therefore functions can have up to 22 parameters. This appears to be an arbitrary choice and is limited only because the underlying types are only written from Function1 through to Function22. If you find yourself hitting this limit, then you probably need to rethink your design!

As an example, if we changed the function type taken by setFunc to have three input parameters and one result, as follows:

```
package com.jjh.func

object MyScalaTest {

  def setFunc(func: (Int, Int, Int) => String) {
    println(func(10, 1, 2))
  }

}
```

Then when we de-compile the resulting .class byte codes, we would find that this is now as shown below.



```
Johns-iMac:interop jeh$ javap -cp . com.jjh.func.MyScalaTest
Compiled from "MyScalaTest.scala"
public final class com.jjh.func.MyScalaTest {
  public static void setFunc(scala.Function3<java.lang.Object, java.lang.Object,
java.lang.Object, java.lang.String>);
}
Johns-iMac:interop jeh$
```

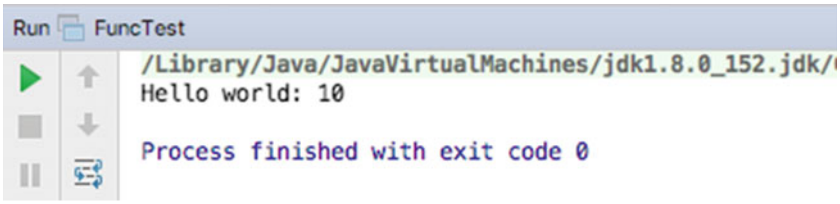
To exploit this within the Java world, we can create Java code that implements the *Function1* type (which appears as an Interface in the Java world). This can be done by using one of the abstract runtime classes that implement the appropriate interface. For example, if we have a single parameter function, then the interface for it in the Java world is Function1 and the abstract class that provides the basic infrastructure for that interface is AbstractFunction1 (where as for a three parameter function we would use the Function3 interface and the AbstractFunction1 class). The following listing illustrates how we can create a Scala function in Java and use it with a Scala *object* that expects to receive a function:

```
package com.jjh.func;

import scala.Function1;
import scala.runtime.AbstractFunction1;

public class FuncTest {

   public static void main(String[] args) {
      Function1<Object, String> f =
            new AbstractFunction1<Object, String>() {
               public String apply(Object someInt) {
                  return "Hello world: " + someInt;
               }
            };
      MyScalaTest.setFunc(f);
   }

}
```

In the above example, a new (anonymous) inner class is created on the fly based on the *AbstractFunction1* class. It defines a method apply that takes a parameter of type *Object* (anything in the Java world) and returns a *String*. In this case, the string is constructed by prefixing whatever was passed in with the string “Hello World”.

Using this new anonymous class, a new instance is created and a reference to that instance stored in the variable ‘f’ which is of type Function1. Note that the parameters to Function1 indicate that the function being defined takes one parameter (the first type in the angle brackets ‘<>’ and the return type is String the second type in the angle brackets.

The instance referenced by f is then passed on the *setFunc* method of *MyScalaTest*, which is the Java representation of the *MyScalaTest* object. The end result of executing this Java program is the output:



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/
Hello world: 10

Process finished with exit code 0
```

It is worth noting that Scala does not use Java’s representation of functions. This is primarily for two reasons, firstly Scala had functions before Java did and thus developed its own representation but secondly (and more importantly) Scala’s representation of functions is considerably richer than that in Java 8.

44.6 Collection Classes

Both Java and Scala have libraries of collection classes, and confusingly many of the names are similar.

As an example, there is a `Set` collection in both Java and Scala and there is a `List` in both Java and Scala. However, the Scala collections are *not* just wrappers around the Java collection classes and thus you cannot just assign from one to the other.

For example, you cannot assign a Java `Set` type to a Scala `Set` type. The following does not work:

```
package com.jeh.scala.interop

object SetTest1 extends App {

  val jSet: java.util.Set[String] = new

java.util.HashSet[String]()

  jSet.add("Adam")
  jSet.add("Phoebe")

  val sSet: scala.collection.mutable.Set[String] = jSet

}
```

If you do try this, you will find that the assignment of the jSet to the Scala Set will result in a compilation error. There is no relationship between these two types of Sets to allow a cast to occur.

You might think that you can create a new instance of the Scala set using the Java Set as a parameter to the constructor:

```
val sSet: Set[String] = Set(jSet.toArray(): _*)
```

For an appropriate Scala set, this could work but the array type generated by Java is an Array of Objects, but we are using a Set with an array of String (which is what both types of sets hold).

It is therefore necessary to look at the Java Converter facilities provided in the scala.collections package. For example, the JavaConverters object (previously known as JavaConverters but that name is now deprecated and introduced in Scala 2.8) provides numerous conversions.

The following conversions are supported via asJava, asScala

- scala.collection.Iterable <=> java.lang.Iterable
- scala.collection.Iterator <=> java.util.Iterator
- scala.collection.mutable.Buffer <=> java.util.List

- scala.collection.mutable.Set <=> java.util.Set
- scala.collection.mutable.Map <=> java.util.Map
- scala.collection.mutable.ConcurrentMap <=> java.util.concurrent.ConcurrentMap

The following conversions are supported via `asScala` and through specially named extension methods to convert to Java collections, as shown:

- scala.collection.Iterable <=> java.util.Collection (via asJavaCollection)
- scala.collection.Iterator <=> java.util.Enumeration (via asJavaEnumeration)
- scala.collection.mutable.Map <=> java.util.Dictionary (via asJavaDictionary)

In addition, the following one-way conversions are provided via `asJava`:

- scala.collection.Seq => java.util.List
- scala.collection.mutable.Seq => java.util.List
- scala.collection.Set => java.util.Set
- scala.collection.Map => java.util.Map

The following one-way conversion is provided via asScala:

- java.util.Properties => scala.collection.mutable.Map

Thus, the earlier example should be written as follows.

```
package com.jeh.scala.interop

import scala.collection.JavaConverters._

object SetTest1 extends App {

  val jSet: java.util.Set[String] = new java.util.HashSet[String]()

  jSet.add("Adam")
  jSet.add("Phoebe")

  val sSet: scala.collection.mutable.Set[String] = jSet.asScala

}
```

However, note that the *asScala* returns a mutable set not an immutable set. This is because Java collections are all mutable whereas Scala has mutable and immutable collections.

It should also be noted that the JavaConverters classes use the Adapter pattern to wrap the original Java collection (the underlier) within a Scala interface that resembles the Scala collection types. Thus, both converting and accessing converted collections is a constant time (O(1)) operation introducing only a minor overhead. Due to this Design Pattern, it is also worth noting that converting Java collection to Scala and then back to Java yields the original collection, not double-wrapper.

44.7 Implementing a Java Interface

Java makes extensive use of interfaces, which are used to define abstract definitions of method signatures (and static constant values). Scala has no concept of an interface; however; it does have traits. One of viewing an interface is as a very restricted type of trait. Thus, a Scala class can *implement* a Java interface by treating it as a *Trait* which it is mixing into that class.

However, unlike Scala traits, interfaces can only have abstract methods. Thus, the class mixing in the interface must implement the method (or methods) specified by the interface.

For example, given the following Java interface:

```
package com.jjh.java;

import java.util.List;

public interface Processor {
   double calc(List<String> l);
}
```

Any Scala class must provide an implementation for the *abstract* method `calc`. Note also that the method `calc` takes a *List*, which in the Java world is an interface itself. Thus whatever is passed into the `calc` method will be a class or an object that implements that interface.

The following listing provides a simple Scala class that implements the Processor interface. The

```
package com.jjh.interop

import java.util.List

import com.jjh.java.Processor
class MyProcessor extends Processor {
  def calc(l: List[String]): Double = {
    import scala.collection.JavaConverters._
    val list = l.asScala
    return list.foldLeft(0)
    { (total, element) => total + element.toInt }
  }
}
```

The simple class *MyProcessor* converts the Java list into a Scala list to make it easier to work with. It then processes all the elements within the list in order to generate a total (it is assumed that all the string sin the list passed in will contain integer values allowing the *toInt* operation to convert the string into an integer). The result returned is a Double (which is treated as a raw value double in the Java world).

The interesting thing is that the Scala class *MyProcessor* could be used from the Scala world or from the Java world. Thus the following listing implemented in Java creates a new instance of the *MyProcessor* and stores it into a variable of type *Processor*. It then creates a list of strings and passes them into the *MyProcessor* object, etc.

```
package com.jjh.java;

import java.util.ArrayList;
import java.util.List;

import com.jjh.interop.MyProcessor;

public class TestApp {
   public static void main(String[] args) {
      Processor proc = new MyProcessor();
      List<String> l = new ArrayList<String>();
      l.add("32");
      l.add("5");
      double x = proc.calc(l);
      System.out.println(x);
   }
}
```

The output of this program is 37.0.