Java + Scala

This lesson covers Java interoperability.

Javap

javap is a tool that ships with the JDK. Not the JRE. There's a difference. Javap decompiles class definitions and shows you what's inside. Usage is pretty simple

```
[local ~/projects/interop/target/scala_2.8.1/classes/com/twitter/interop]$ javap MyTrait
Compiled from "Scalaisms.scala"
public interface com.twitter.interop.MyTrait extends scala.ScalaObject{
    public abstract java.lang.String traitName();
    public abstract java.lang.String upperTraitName();
}
```

If you're hardcore you can look at byte code

```
[local ~/projects/interop/target/scala_2.8.1/classes/com/twitter/interop]$ javap -c MyTrait\$class
Compiled from "Scalaisms.scala"
public abstract class com.twitter.interop.MyTrait$class extends java.lang.Object{
public static java.lang.String upperTraitName(com.twitter.interop.MyTrait);
  Code:
   0:aload_0
   1:invokeinterface#12,  1; //InterfaceMethod com/twitter/interop/MyTrait.traitName:()Ljava/lang/String;
   6:invokevirtual#17; //Method java/lang/String.toUpperCase:()Ljava/lang/String;
   9:areturn

public static void $init$(com.twitter.interop.MyTrait);
  Code:
   0:return

}
```

If you start wondering why stuff doesn't work in Java land, reach for javap!

Classes

The four major items to consider when using a Scala *class* from Java are

- Class parameters
- Class vals
- Class vars
- Exceptions

We'll construct a simple scala class to show the full range of entities

```
package com.twitter.interop

import java.io.IOException
import scala.throws
import scala.reflect.{BeanProperty, BooleanBeanProperty}

class SimpleClass(name: String, val acc: String, @BeanProperty var mutable: String) {
  val foo = "foo"
  var bar = "bar"
  @BeanProperty
  val fooBean = "foobean"
  @BeanProperty
  var barBean = "barbean"
  @BooleanBeanProperty
  var awesome = true

  def dangerFoo() = {
    throw new IOException("SURPRISE!")
  }

  @throws(classOf[IOException])
  def dangerBar() = {
    throw new IOException("NO SURPRISE!")
  }
}
```

Class parameters

- by default, class parameters are effectively constructor args in Java land. This means you can't access them outside the class.
- declaring a class parameter as a val/var is the same as this code

```
class SimpleClass(acc_: String) {
  val acc = acc_
}
```

which makes it accessible from Java code just like other vals

Vals

- vals get a method defined for access from Java. You can access the value of the val "foo" via the method "foo()"

Vars

- vars get a method _$eq defined. You can call it like so

```
foo$_eq("newfoo");
```

BeanProperty

You can annotate vals and vars with the @BeanProperty annotation. This generates getters/setters that look like POJO getter/setter definitions. If you want the isFoo variant, use the BooleanBeanProperty annotation. The ugly foo$_eq becomes

```
setFoo("newfoo");
getFoo();
```

Exceptions

Scala doesn't have checked exceptions. Java does. This is a philosophical debate we won't get into, but it **does** matter when you want to catch an exception in Java. The definitions of dangerFoo and dangerBar demonstrate this. In Java I can't do this

```
        // exception erasure!
        try {
            s.dangerFoo();
        } catch (IOException e) {
            // UGLY
        }
```

Java complains that the body of s.dangerFoo never throws IOException. We can hack around this by catching Throwable, but that's lame.

Instead, as a good Scala citizen it's a decent idea to use the throws annotation like we did on dangerBar. This allows us to continue using checked exceptions in Java land.

Further Reading

A full list of Scala annotations for supporting Java interop can be found here http://www.scala-lang.org/node/106.

Traits

How do you get an interface + implementation? Let's take a simple trait definition and look

```
trait MyTrait {
  def traitName:String
  def upperTraitName = traitName.toUpperCase
}
```

This trait has one abstract method (traitName) and one implemented method (upperTraitName). What does Scala generate for us? An interface named MyTrait, and a companion implementation named MyTrait$class.

The implementation of MyTrait is what you'd expect

```
[local ~/projects/interop/target/scala_2.8.1/classes/com/twitter/interop]$ javap MyTrait
Compiled from "Scalaisms.scala"
public interface com.twitter.interop.MyTrait extends scala.ScalaObject{
    public abstract java.lang.String traitName();
    public abstract java.lang.String upperTraitName();
}
```

The implementation of MyTrait$class is more interesting

```
[local ~/projects/interop/target/scala_2.8.1/classes/com/twitter/interop]$ javap MyTrait\$class
Compiled from "Scalaisms.scala"
public abstract class com.twitter.interop.MyTrait$class extends java.lang.Object{
    public static java.lang.String upperTraitName(com.twitter.interop.MyTrait);
    public static void $init$(com.twitter.interop.MyTrait);
}
```

MyTrait$class has only static methods that take an instance of MyTrait. This gives us a clue as to how to extend a Trait in Java.

Our first try is the following

```
package com.twitter.interop;

public class JTraitImpl implements MyTrait {
    private String name = null;

    public JTraitImpl(String name) {
        this.name = name;
    }

    public String traitName() {
        return name;
    }
}
```

And we get the following error

```
[info] Compiling main sources...
[error] /Users/mmcbride/projects/interop/src/main/java/com/twitter/interop/JTraitImpl.java:3: com.twitter.interop.JTraitImpl
is not abstract and does not override abstract method upperTraitName() in com.twitter.interop.MyTrait
[error] public class JTraitImpl implements MyTrait {
[error]        ^
```

We *could* just implement this ourselves. But there's a sneakier way.

```
package com.twitter.interop;

    public String upperTraitName() {
        return MyTrait$class.upperTraitName(this);
    }
```

We can just delegate this call to the generated Scala implementation. We can also override it if we want.

Objects

Objects are the way Scala implements static methods/singletons. Using them from Java is a bit odd. There isn't a stylistically perfect way to use them, but in

Scala 2.8 it's not terrible

A Scala object is compiled to a class that has a trailing "$". Let's set up a class and a companion object

```
class TraitImpl(name: String) extends MyTrait {
  def traitName = name
}

object TraitImpl {
  def apply = new TraitImpl("foo")
  def apply(name: String) = new TraitImpl(name)
}
```

We can naïvely access this in Java like so

```
MyTrait foo = TraitImpl$.MODULE$.apply("foo");
```

Now you may be asking yourself, WTF? This is a valid response. Let's look at what's actually inside TraitImpl$

```
local ~/projects/interop/target/scala_2.8.1/classes/com/twitter/interop]$ javap TraitImpl\$
Compiled from "Scalaisms.scala"
public final class com.twitter.interop.TraitImpl$ extends java.lang.Object implements scala.ScalaObject{
    public static final com.twitter.interop.TraitImpl$ MODULE$;
    public static {};
    public com.twitter.interop.TraitImpl apply();
    public com.twitter.interop.TraitImpl apply(java.lang.String);
}
```

There actually aren't any static methods. Instead it has a static member named MODULE$. The method implementations delegate to this member. This makes access ugly, but workable if you know to use MODULE$.

Forwarding Methods

In Scala 2.8 dealing with Objects got quite a bit easier. If you have a class with a companion object, the 2.8 compiler generates forwarding methods on the companion class. So if you built with 2.8, you can access methods in the TraitImpl Object like so

```
MyTrait foo = TraitImpl.apply("foo");
```

Closures Functions

One of Scala's most important features is the treatment of functions as first class citizens. Let's define a class that defines some methods that take functions as arguments.

```
class ClosureClass {
  def printResult[T](f: => T) = {
    println(f)
  }

  def printResult[T](f: String => T) = {
    println(f("HI THERE"))
  }
}
```

In Scala I can call this like so

```
val cc = new ClosureClass
cc.printResult { "HI MOM" }
```

In Java it's not so easy, but it's not terrible either. Let's see what ClosureClass actually compiled to:

```
[local ~/projects/interop/target/scala_2.8.1/classes/com/twitter/interop]$ javap ClosureClass
Compiled from "Scalaisms.scala"
public class com.twitter.interop.ClosureClass extends java.lang.Object implements scala.ScalaObject{
    public void printResult(scala.Function0);
    public void printResult(scala.Function1);
    public com.twitter.interop.ClosureClass();
}
```

This isn’t so scary. “f: => T” translates to “Function0”, and “f: String => T” translates to “Function1”. Scala actually defines Function0 through Function22, supporting this stuff up to 22 arguments. Which really should be enough.

Now we just need to figure out how to get those things going in Java. Turns out Scala provides an AbstractFunction0 and an AbstractFunction1 we can pass in like so

```
    @Test public void closureTest() {
        ClosureClass c = new ClosureClass();
        c.printResult(new AbstractFunction0() {
                public String apply() {
                    return "foo";
                }
            });
        c.printResult(new AbstractFunction1<String, String>() {
                public String apply(String arg) {
                    return arg + "foo";
                }
            });
    }
```

Note that we can use generics to parameterize arguments.