

# Java Integration

# Introduction

- ❖ The Scala programming language is highly compatible with Java.
- ❖ In most cases there shouldn't be any problem combining code developed in the two languages.
- ❖ Popular frameworks such as Java Servlets & JSP, Swing and JUnit work fine with code developed in Scala.

# Using Scala from Java

- ❖ When calling code developed in Scala from within code developed in Java we first need to have our code in Scala translated into Java.
- ❖ The Scala programming language is implemented as a translation to standard Java bytecode.
- ❖ When possible, features we know from the Scala programming language are mapped directly onto their equivalent Java features.

# Using Scala from Java

- ❖ Make sure the `scala-library.jar` is available in your build path. Otherwise, the compilation will fail.

# Using Scala from Java

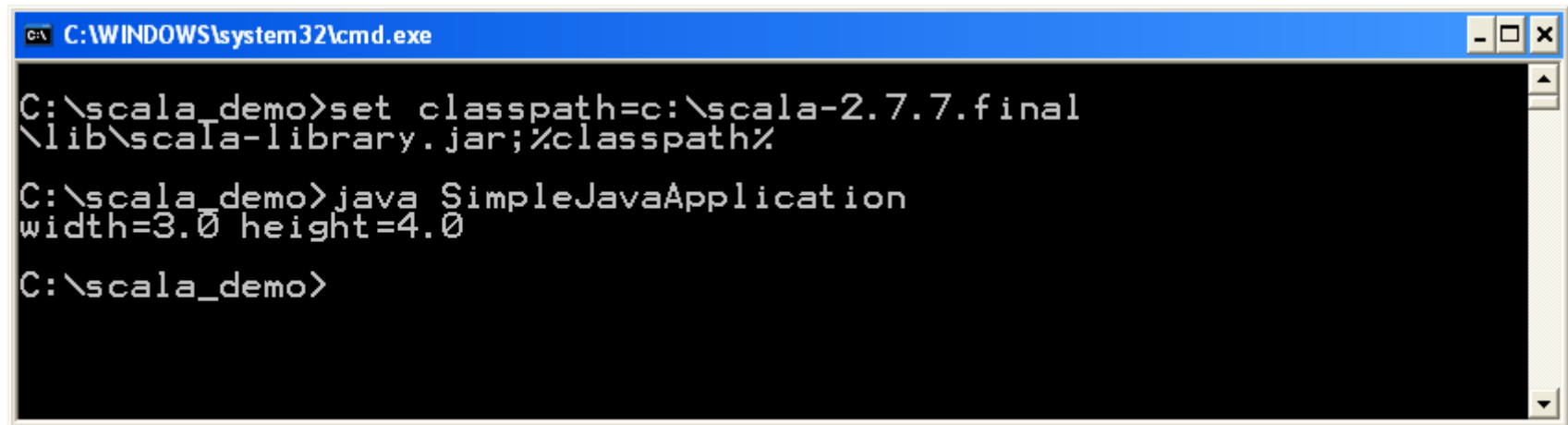
```
class ScalaRectangle(w:Double,h:Double)
{
    var width:Double = w
    var height: Double = h
    def toXML =
    <rectangle>
        <width>{width}</width>
        <height>{height}</height>
    </rectangle>
    override def toString = "width="+width+" height="+height
}
```



# Using Scala from Java

```
public class SimpleJavaApplication
{
    public static void main(String args[])
    {
        ScalaRectangle ob = new ScalaRectangle(3,4);
        System.out.println(ob);
    }
}
```

# Using Scala from Java



```
C:\WINDOWS\system32\cmd.exe

C:\scala_demo>set classpath=c:\scala-2.7.7.final
\lib\scala-library.jar;%classpath%

C:\scala_demo>java SimpleJavaApplication
width=3.0 height=4.0

C:\scala_demo>
```

# Class Variables

- ❖ When using a class that was defined in Scala from code written in Java the variables (whether `val` or `var`) inside the objects that were instantiated from that class will be accessible as if they were methods.



# Class Variables

```
class Rectangle(_id:Int,_width:Double,_height:Double) {  
  val id:Int = _id  
  val width:Double = _width  
  val height:Double = _height  
}
```

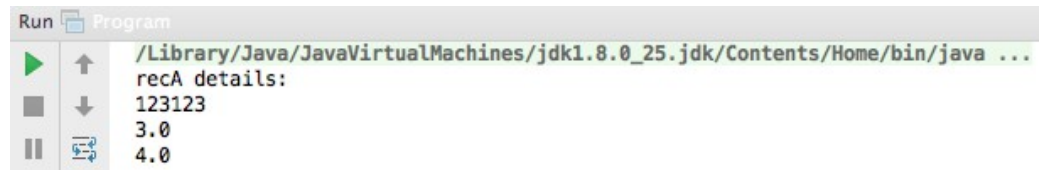
Rectangle.scala

# Class Variables

```
public class Program {  
    public static void main(String args[]) {  
        Rectangle recA = new Rectangle(123123,3,4);  
        Rectangle recB = new Rectangle(233343,5,6);  
        System.out.println("recA details:");  
        System.out.println(recA.id());  
        System.out.println(recA.width());  
        System.out.println(recA.height());  
    }  
}
```

Program.java

# Class Variables



The screenshot shows a console window titled "Run Program" with a grey header. On the left side of the console, there are four icons: a green play button, a grey square, a vertical bar, and a refresh icon. The main area of the console displays the following text:

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_25.jdk/Contents/Home/bin/java ...  
recA details:  
123123  
3.0  
4.0
```

# Beans Properties

- ❖ When adding the `@BeanProperty` annotation to a class variable defined in Scala the classic getter and setter methods will be generated (e.g. given the `width` variable we will automatically get the `getWidth` and the `setWidth` methods).

# Beans Properties

- ❖ When adding the `@BooleanBeanProperty` annotation we will get the `isFoo` variant generated (e.g. given the `visible` boolean variable we will automatically get the `isVisible` and the `setVisible` methods).

# Beans Properties

```
class Rectangle(_visible:Boolean,_width:Double,_height:Double)
{
  @BooleanBeanProperty
  val visible:Boolean = _visible
  @BeanProperty
  val width:Double = _width
  @BeanProperty
  val height:Double = _height
}
```

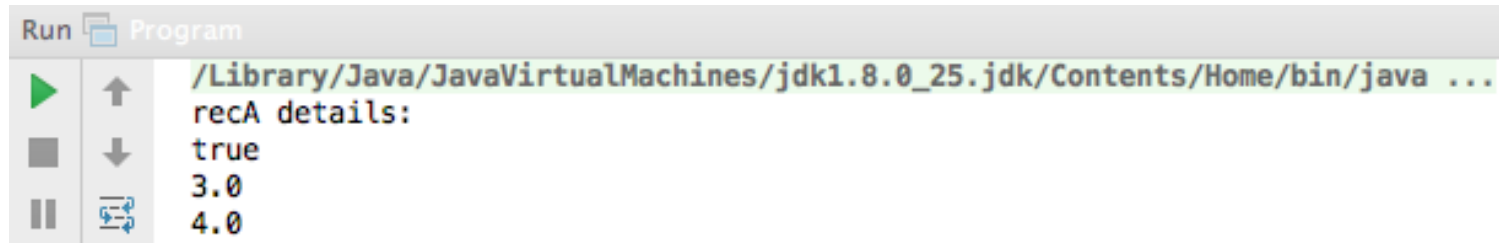
Rectangle.scala

# Beans Properties

```
public class Program {  
    public static void main(String args[]) {  
        Rectangle recA = new Rectangle(true,3,4);  
        Rectangle recB = new Rectangle(false,5,6);  
        System.out.println("recA details:");  
        System.out.println(recA.isVisible());  
        System.out.println(recA.getWidth());  
        System.out.println(recA.getHeight());  
    }  
}
```

Program.java

# Beans Properties



```
Run Program
/Library/Java/JavaVirtualMachines/jdk1.8.0_25.jdk/Contents/Home/bin/java ...
recA details:
true
3.0
4.0
```



# Using Java from Scala

- ❖ Using code written in Java from within a code written in Scala is simpler.
- ❖ There is no need in any specific jar file. We can use any class developed in Java as if it was developed in Scala.

# Using Java from Scala

```
public class JavaRectangle
{
    private double width;
    private double height;
    public JavaRectangle(double w, double h)
    {
        setWidth(w);
        setHeight(h);
    }
    public void setWidth(double val)
    {
        if(val>0)
        {
            width = val;
        }
    }
}
```



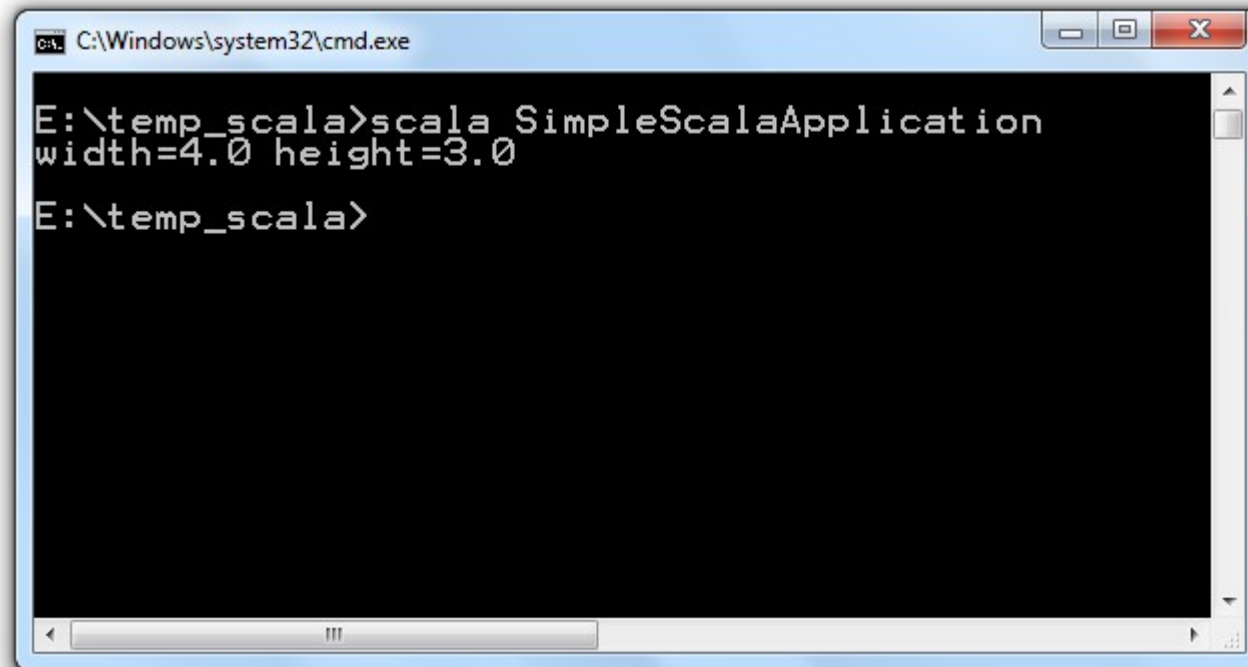
# Using Java from Scala

```
public void setHeight(double val)
{
    if(val>0)
    {
        height = val;
    }
}
public String toString()
{
    return "width="+width+" height="+height;
}
}
```

# Using Java from Scala

```
object SimpleScalaApplication
{
  def main(args: Array[String])
  {
    var ob = new JavaRectangle(4,3);
    println(ob);
  }
}
```

# Using Java from Scala



```
C:\Windows\system32\cmd.exe
E:\temp_scala>scala SimpleScalaApplication
width=4.0 height=3.0
E:\temp_scala>
```

# Traits are Interfaces

- ❖ Unlike interfaces in Java (up to version 1.7 included), when we define a trait in Scala we can include methods definitions as part of the trait.
- ❖ Although this difference, the Scala compiler compiles a trait into an interface.

# Traits are Interfaces

- ❖ When a trait includes an implemented method the implementation (until Java version 1.7 included) is taken out of the trait into a new class that its name starts with the name of the trait together with '\$class'.
- ❖ We can find the implemented method defined as a static method with an additional parameter for getting the reference for the concrete object on which it should be executed.

# Traits are Interfaces

```
package com.abelski.scala.samples
```

```
trait PowerfulMan
```

```
{
```

```
  def doAbstractStuff: Unit
```

```
  def doConcreteStuff: Unit = println("do the concrete stuff")
```

```
}
```

PowerfulMan.scala





# Traits are Interfaces

```
package com.abelski.scala.samples;

public class JavaMan implements PowerfulMan
{
    public void doAbstractStuff()
    {
        System.out.println("do the abstract stuff");
    }
    public void doConcreteStuff()
    {
        PowerfulMan$class.doConcreteStuff(this);
    }
}
```

PowerfulMan.java

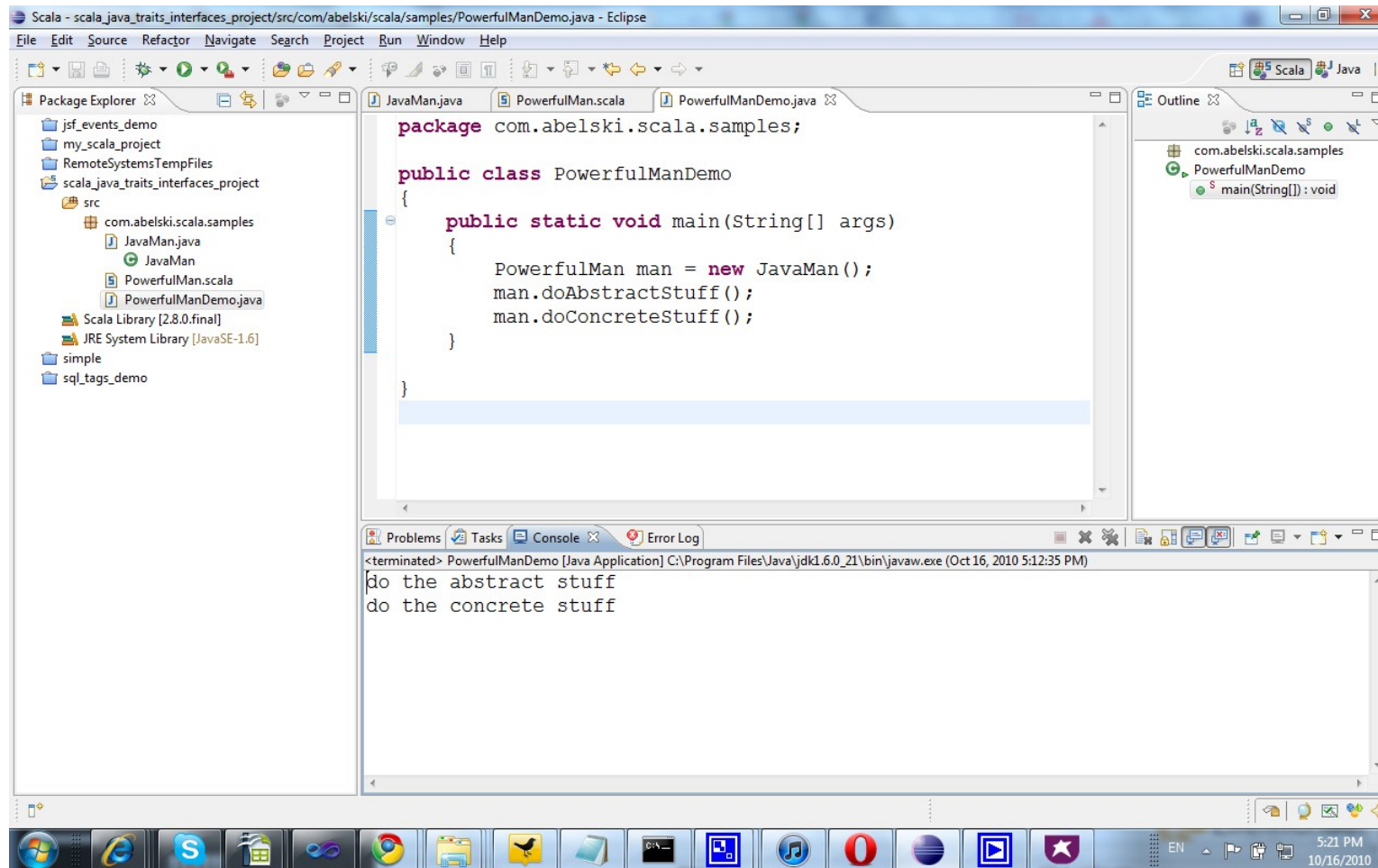
# Traits are Interfaces

```
package com.abelski.scala.samples;

public class PowerfulManDemo
{
    public static void main(String[] args)
    {
        PowerfulMan man = new JavaMan();
        man.doAbstractStuff();
        man.doConcreteStuff();
    }
}
```

PowerfulManDemo.java

# Traits are Interfaces



# Exceptions Handling

- ❖ Scala doesn't have checked exceptions and it doesn't support the throws statement. As a result, whenever we define a function in Scala that might throw an exception one way for catching it in Java would be by placing a catch statement for Throwable. Alternatively, we can use the @throws annotation.

# Exceptions Handling

```
public class Program {  
    public static void main(String args[]) {  
        try {  
            Utils.doSomething(123, "temp.txt");  
        } catch(Throwable e) {  
            //...  
        }  
    }  
}
```

Program.java

# Exceptions Handling

```
object Utils {  
  def doSomething(num:Int, fileName:String): Unit =  
  {  
    //...  
    //...  
  }  
}
```

Utils.scala

# Exceptions Handling

- ❖ We can alternatively use the `@throws` annotation in order to mark the function we define in Scala as one that might throw an exception of a specific type.

# Exceptions Handling

```
import java.io._

object Utils {
  @throws(classOf[IOException])
  def doSomething(num:Int, fileName:String): Unit = {
    //...
    //...
  }
}
```

Utils.scala



# Exceptions Handling

```
public class Program {  
    public static void main(String args[]) {  
        try {  
            Utils.doSomething(123, "temp.txt");  
        } catch(IOException e) {  
            //...  
        }  
    }  
}
```

Program.java

# Static Members

- ❖ Scala doesn't allow us to define static members in our class. The closest possibility is to define an Object with methods.
- ❖ Invoking those methods in code written in Java will be very similar to invoking static methods.
- ❖ Static methods we define in Java will be available in Scala as methods we invoke on object. The class with those static methods in Java will be available in Scala as object.

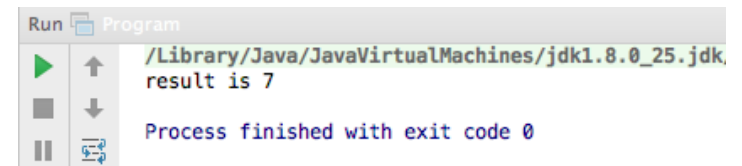
# Static Members

```
public class Program {  
    public static void main(String args[]) {  
        System.out.println("result is "+Utils.sum(3,4));  
    }  
}
```

Program.java

```
object Utils {  
    def sum(numA:Int,numB:Int): Int = numA+numB  
}
```

Utils.scala



# Closures

- ❖ Scala allows us to define functions as first class citizens and to define functions within the scope of other functions.
- ❖ In Java, each and every function that was defined in Scala is an object from an anonymous inner class that extends the `Function1/Function2/Function3... abstract class`.
- ❖ **The** `Function1/Function2/Function3...Function22` types support up to 22 parameters.

# Closures

- ❖ The function code is placed within the implementation for the `apply` method, we should define when extending any of these abstract classes.

```
func = new AbstractFunction1<String, String>() {  
    public String apply(String arg) {  
        return arg + "foo";  
    }  
};
```

# Annotations

- ❖ We can use annotations available in Java frameworks directly from within the code we developed in Scala. In most cases the Java framework will identify the annotations as if our code was written in Java.
- ❖ Nevertheless, inventing our own annotations won't be feasible in Scala. We must write our own annotations in Java and compile the code using the javac utility.

# Annotations

- ❖ Scala already supports various annotations that assist us with developing code in Scala and later use it in Java.
- ❖ The `@SerialVersionUID` annotation allows us to specify the static `SerialVersionUID` field of a serializable class.
- ❖ The `@Cloneable` annotation allows us to mark a class as if we were marking that class as cloneable by implementing the `Cloneable` interface.

# Annotations

- ❖ The `@Deprecated` annotation allows us to mark a class member as a deprecated one.
- ❖ The `@Native` annotation allows us to mark a method as a native one.
- ❖ The `@Serializable` annotation allows us to mark a class as one that implements the `Serializable` interface.



# Annotations

- ❖ The `@Transient` annotation allows us to mark a class variable as a transient one.
- ❖ The `@Volatile` annotation allows us to mark a class variable as if it was marked with `volatile` keyword.

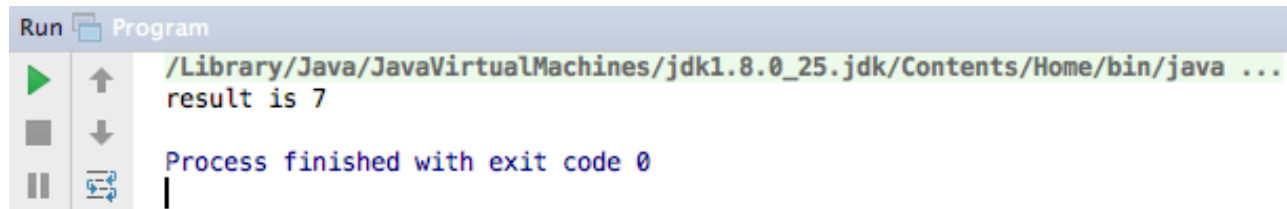
# Varied Number of Arguments

- ❖ Using the `@varargs` annotation we can mark a method we define in Scala as one that receives a variable number of arguments so we could invoke it that way from code in Java.

```
object Utils {  
  
  @varargs  
  def calc(number:Int*): Int = {  
    var sum:Int = 0  
    number.foreach(num => sum = sum+num)  
    sum  
  }  
}
```

# Varied Number of Arguments

```
public class Program {  
    public static void main(String args[]) {  
        System.out.println(Utils.calc(2,3,5));  
    }  
}
```



```
Run Program  
/Library/Java/JavaVirtualMachines/jdk1.8.0_25.jdk/Contents/Home/bin/java ...  
result is 7  
Process finished with exit code 0
```