Annotations

Introduction

- The annotations in Scala, as in Java, are additional pieces of data we can add to our code.
- The syntax for using annotations is the same syntax been used in Java.

```
@deprecated def doSomething():Unit =
{
    ...
}
```

Annotations

- We can use annotations available in Java 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.
- Developing our own annotations isn't possible in Scala. We can develop our own annotations in Java only.

The `@deprecated` Annotation

- Using this annotation we can annotate a class member as deprecated one. Whenever been used in code a compilation warning will be generated.

```
object Utils {
  @deprecated("Use MathUtils.calc method for better accurace","1.9")
  def calc(num:Int,fileName:String): Unit =     {
    //...
    //...
  }
}
```

The `@throws` Annotation

- Using this annotation we can specify a specific exception type might be thrown when calling the annotated method. This annotation replaces the throws statement we know in Java.

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

The `@SerialVersionUID` Annotation

- Using this annotation we can specify the `static SerialVersionUID` field of a serializable class.

```
@SerialVersionUID(1000L)
object Utils {
    def calc(num:Int,fileName:String): Unit = {
    //...
    //...
    }
}
```

The `@Cloneable` Annotation

- Using this annotation we can specify the class is Cloneable just as if we were implementing Cloneable in a class we define in Java.

```
@Cloneable
class Point {
    def calc(x:Int,y:Int): Unit = {
    //...
    //...
    }
}
```

The `@Native` Annotation

- Using this annotation we can specify a method as a native one. It replaces the `native` keyword we know in Java.

```
class Point {
    @Native
    def calc(x:Int,y:Int): Unit = {
    //...
    //...
    }
}
```

The `@Serializable` Annotation

- Using this annotation is a replacement for implementing the `Serializable` interface in Java.

```
@Serializable
class Point {
    def calc(x:Int,y:Int): Unit = {
    //...
    //...
    }
}
```

The `@Transient` Annotation

- Using this annotation is a replacement for using the `transient` keyword in Java.

```
@Serializable
class Point {
    @Transient
    var x:Int = 0
    var y:Int = 0
    def calc(x:Int,y:Int): Unit = {
    //...
    //...
    }
}
```

The `@Volatile` Annotation

- Using this annotation is a replacement for using the `volatile` keyword in Java.

```
@Serializable
class Point {
    @Volatile
    var x:Int = 0
    @Volatile
    var y:Int = 0
    def calc(x:Int,y:Int): Unit = {
    //...
    //...
    }
}
```

The `@BeanProperty` Annotation

- 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).

The `@BooleanBeanProperty` Annotation

- 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).

The `@BooleanBeanProperty` Annotation

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

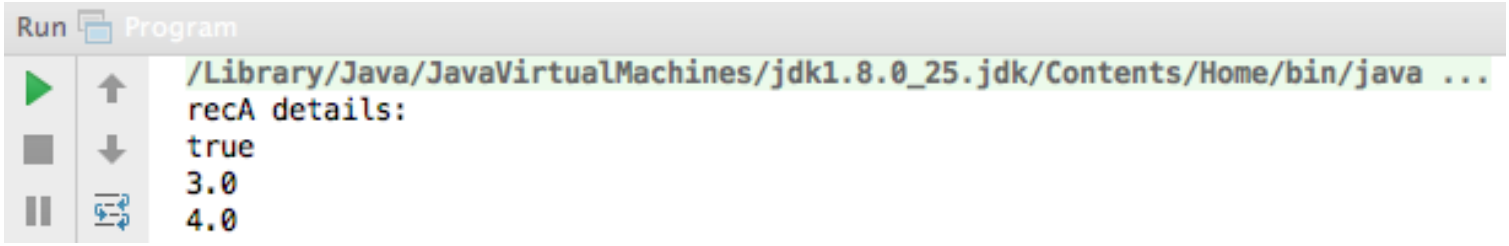
Rectangle.scala

The `@BooleanBeanProperty` Annotation

```
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

The `@BooleanBeanProperty` Annotation



The `@Uncheck` Annotation

- Using this annotation we can instruct the compiler to show less warnings for match statement selector we marked. This annotation actually tells the compiler not to worry for our avoidance of few cases.

```
object Utils {
  def calc(e:Any): Unit =     {
    (e: @unchecked)
    e match {
        case 1 => println("one")
        case 2 => println("two")
    }
  }
}
```

The `@varargs` Annotation

- Using this 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
  }
}
```

The `@varargs` Annotation

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