讲述完class的基本知识，接下来会讲解scala中object的相关内容。主要有一下几个方面：

1. Singletons
2. Companion Objects
3. Objects Extends a Class or Trait
4. The apple Method
5. Application Objects
6. Enumerations

* Singletons

http://blog.csdn.net/shenxiaoming77/article/details/51491531

在Scala中没有静态method和field。取而代之的是你可以用object结构去实现。一个object可以定义一个类的单例。举个例子：

object Accounts {  
 private var lastNumber = 0  
 def newUniqueNumber() = { lastNumber += 1; lastNumber }  
}

object的constructor只会在object第一次用到的时候执行，即将JVM将class文件加载。在我们的例子中，Accounts的constructor只会在第一次调用Accounts.newUniqueNumber()时执行。如果这个object没有被用过，那么它的constructor就不会执行。

一个object基本上可以拥有一个类的所有功能，甚至可以继承其他的class或trait。有一个例外：也就是constructor不能有参数。因为object的构建从jvm的角度来看，就是类加载，类加载也就不需要有参数了。

当你在Java中使用单例时，就可以在Scala中使用object：

1. 作为工具函数和常量的地方
2. 作为一个不可变的instance被共享
3. 单例模式

NOTE:

很多人对单例模式有偏见，其实scala给我们提供这种工具，至于怎么用就看我们自己了。

* Companion Objects

在java中你可以在一个类中包含instance的方法和static方法。但是在scala中，你需要有一个class和一个Companion object（伴生对象），他们的名字必须相同。

DEMO：

class Account {  
 val id = Account.newUniqueNumber()  
 private var balance = 0.0  
 def deposit(amount: Double) { balance += amount }  
 ...  
}  
object Account { // The companion object  
 private var lastNumber = 0  
 private def newUniqueNumber() = { lastNumber += 1; lastNumber }  
}

Class和它的companion object可以相互访问彼此的private feature。他们必须在同一个源文件中。但是有一点要注意：companion object的feature和它的companion class并不在同一范围中，举个例子来讲：如果Accounts class只能用Accounts.newUniqueNumber()来调用newUniqueNumber方法，而不能用newUniqueNumber。

* Objects Extending a Class or Trait
* The apply Method

Apply 方法讲过多次了，如果还有什么不明白，可以来问我。

* Application Objects

每一个Scala程序必须开始于object的main方法

DEMO：

object Hello {  
 def main(args: Array[String]) {  
 println("Hello, World!")  
 }  
}

除了提供main方法，你也可以继承App trait（trait在后面的会讲到，暂时和java中的interface类比，其实他们有很大的不同）：

object Hello **extends App** {  
 println("Hello, World!")  
}

如果你需要输入命令行参数，可以这样：

object Hello **extends App** {  
 if (args.length > 0)  
 println(f"Hello ${args(0)}")  
 else  
 println("Hello, World!")  
}

Note：

老版本的Scala有一个Application trait功能和App trait一样。因为Application在静态初始化时执行程序的行为，而不能被just-in-time编译器优化，所以用App来替代。我觉得为什么继承App trait可以实现和定义main函数相同的功能并不是很重要，如果感兴趣可以自行Google，或者问我。

* Enumerations

与java不同的是，scala并没有枚举类型。但是标准库中提供了Enumeration辅助类帮助我们创建枚举类型。

定义一个object，继承Enumeration class，然后用value方法初始化每个值。

DEMO：

object TrafficLightColor extends Enumeration {  
 val Red, Yellow, Green = Value  
}

这里我们定义了三个fields， Red，Yellow和Green，并且调用Value函数进行了初始化，上面是下面的简写：

val Red = Value  
val Yellow = Value  
val Green = Value

*/\*\* Creates a fresh value, part of this enumeration. \*/***protected final def** Value: Value = Value(*nextId*)

**protected final def** Value(i: Int): Value = Value(i, nextNameOrNull)

**protected final def** Value(name: String): Value = Value(*nextId*, name)

**protected final def** Value(i: Int, name: String): Value = **new** Val(i, name)

每次对Value方法的调用，返回一个内部类的新instance，它的类型也是Value

或者，你也可以传递ID，name，或者两者都传到Value方法

val Red = Value(0, "Stop")  
val Yellow = Value(10) // Name "Yellow"  
val Green = Value("Go") // ID 11

如果没有指定，则ID比先前分配的ID多一个，开始于零。 默认名称是字段名称。

现在你可以引用枚举值，像：TrafficLightColor.Red，如果你觉得麻烦，可以：

import TrafficLightColor.\_

Note：

枚举的类型是TranfficLightColor.Value,不是TranfficLightColor,它只是一个object，用来存放values。

获取枚举值的ID，可以用id 方法，获取name，可以用toString方法，不再演示。

获取所有值可以用values方法：

for (c <- TrafficLightColor.values) println(s"${c.id}: $c")

最后我们还可以通过ID或者name来查找Enumeration，像这样：

TrafficLightColor(0) // Calls Enumeration.apply  
TrafficLightColor.withName("Red")

**object** TrafficLightColor **extends** Enumeration{  
 **val** *Red*, *Yellow*, *Green* = Value  
  
 **def** main(args: Array[String]): Unit = {  
 **for**(c <- TrafficLightColor.values)  
 *println*(**s"$**{c.id}**: $**c**"**)  
  
 *println*(TrafficLightColor(0).toString)  
 *println*(TrafficLightColor.withName(**"Yellow"**).toString)  
 }  
}

输出：

0: Red

1: Yellow

2: Green

Red

Yellow