* Simple Classes and Parameterless Methods

Demo:

class Counter {  
 private var value = 0 // You *must* initialize the field  
 def increment() { value += 1 } // Methods are public by default  
 def current() = value  
}

在scala里面，一个scala源文件，可以包含多个类，他们之间可以相互访问。

* Properties with Getters and Setters

在Java中

public class Person { // This is Java

private int age;

public int getAge() { return age; }

public void setAge(int age) { this.age = age; }

}

我们创建一个类Person，他有一个属性age，有自己的getter和setter，任何人都可以调用setAge（21），那么他就永远21岁了，我们可以通过改变setAge方法，

public void setAge(int newValue) { if (newValue > age) age = newValue; }那么他就不能变年轻了。Getter 和 setter比pulic要好，因为他能根据具体情况来get和set。

Scala为每个属性提供getter和setter方法，这里是一个public field:

class Person {  
var age = 0  
}

Scala在JVM中生成这样一个类，它带有一个private的age field，它具有getter和setter方法。这两个方法都是public，因为我们没有将age设为private。如果设为private字段，那么getter和setter方法都为private。

在scala中，getter和setter方法是age和age\_=，举个例子：

println(fred.age) // Calls the *method* fred.age()  
fred.age = 21 // Calls fred.age\_=(21)

在scala中虽然getter和setter不叫getXxx和setXxx，但是，他们实现了相同的功能。

class Person {  
 private var privateAge = 0 // Make private and rename  
 def age = privateAge  
 def age\_=(newValue: Int) {  
 if (newValue > privateAge) privateAge = newValue; // Can’t get younger  
 }  
}

当我们用的时候，仍然是fred.age,但是fred不能变年轻了：

Val fred = new Person

fred.age = 30

fred.age = 21

println(fred.age) //30

你可能会感觉奇怪，scala为每个field生成getter和setter方法，但是我们也可以进行控制：

1. 如果field是private，getter 和setter就是private
2. 如果field是val，只会生成getter
3. 如果你不想要getter或者setter，可以将field声明为private[this]

* Properties with Only Getters

如果你只想要一个只读的field，只需要getter，不需要setter。那么使用val：

class Message {  
val timeStamp = java.time.Instant.now  
...  
}

scala编译器会产生一个private final field，同时带有一个public getter方法没有setter方法的Java类。

总结一下，你可以有四种方法：

1. Var foo： Scala会生成getter和setter
2. Val foo： scala 只会生成 getter
3. 你可以自己定义foo和foo\_=方法
4. 你可以只定义foo方法

在scala中没有write-only的field。Scala类中的field，和java中的类的field有所不同，它是一个private field，带有getter（val field）方法或者带有getter和setter方法（var field）

* Object-Private Field

和java一样，在scala中，class的所有对象都可以访问类的私有field。

class Counter {  
 private var value = 0  
 def increment() { value += 1 }  
 def isLess(other : Counter) = value < **other.**value  
 // Can access private field of other object  
}

Scala可以有更严格的访问限制符：private[this]

private[this] var value = 0 // Accessing *someObject*.value is not allowed

现在，只能在当前的object中访问value，而不能在Couter的其他object中访问。Class-privatescala会生成private getter和setter方法，而object-private field，不会生成getter和setter方法。

NOTE：

Scala 允许我们对特定的类授予访问权限，private[ClassName]指明只有指定类的方法可以访问给定的field。这些方法将被公开，因为JVM没有细粒度的访问控制系统。

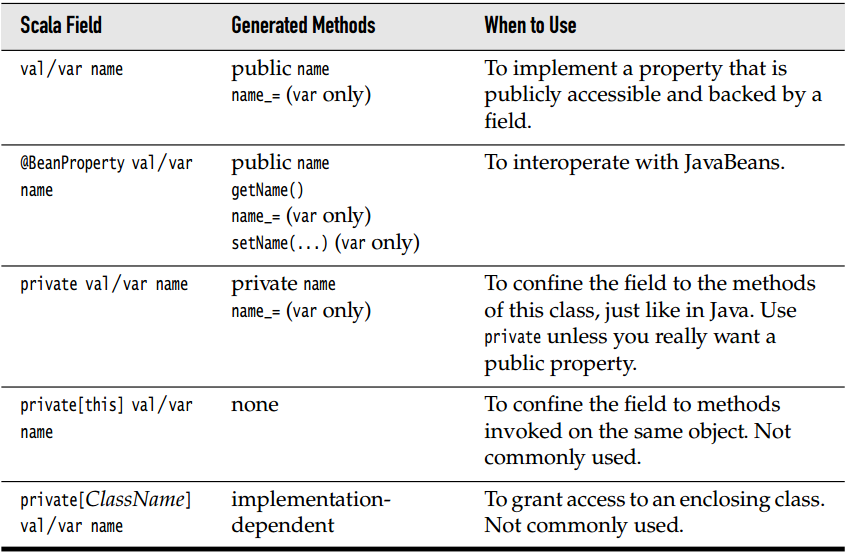
* Bean Peoperties

Scala为我们定义的field提供getter和setter方法，然而这些方法名并不是Java工具所期望的。当用@BeanProperty注解一个Scala字段时，这些方法是自动生成的。例如：

import scala.beans.BeanProperty  
class Person {  
@BeanProperty var name: String = \_  
}

会生成以下四个方法：

generates *four* methods:  
1. name: String  
2. name\_=(newValue: String): Unit  
3. getName(): String  
4. setName(newValue: String): Unit



Val var 都是生成的getter和setter都是public

Private val var生成的getter和setter是private，这个field仅限于这个类及这个类的对象。

Private[this] var val那么这个field仅限于这个object

Private[ClassName] 授权某个类对这个field有访问权限

* Auxiliary Constructors

与java类似，在scala中你可以有很多个构造函数，但是，scala的类中有一个构造函数比其他构造函数都重要，它就是primary constructor，所以一个类会有很多auxiliary constructors

我们先讲auxiliary constructor是因为它比较简单。它们与java中的构造函数类似，但是每一个auxiliary constructor必须调用前一个auxiliary constructor或者primary constructor。

DEMO：

class Person {  
 private var name = ""  
 private var age = 0  
 def this(name: String) { // An auxiliary constructor  
 this() // Calls primary constructor  
 this.name = name  
 }  
 def this(name: String, age: Int) { // Another auxiliary constructor  
 this(name) // Calls previous auxiliary constructor  
 this.age = age  
 }  
}

接下来我会讲解primary constructor，现在只需要知道Person类中有一个没有参数的primary constructor。

我们可以这样构造Person 的实例：

val p1 = new Person // Primary constructor  
val p2 = new Person("Fred") // First auxiliary constructor  
val p3 = new Person("Fred", 42) // Second auxiliary constructor

* The Primary Constructor

在scala中，每一个类都有一个primary constructor. 这个primary constructor不是用this定义的。他是穿插在类定义中的。

Primary constructor的参数放在类名后面：

class Person**(val name: String, val age: Int)** {  
 // Parameters of primary constructor in (...)  
 ...  
}

primary constructor的参数变成类的field，他们通过constructor的参数来初始化。在我们的demo中，name和age都是Person类的field。 New Person（”Fred”, 42）将会创建一个实例，并且将初始化name和age。

上面的一行code，等于在java中：

public class Person { // This is Java  
 **private String name;**

**private int age;**

**public Person(String name, int age)** {  
 **this.name = name; this.age = age;**  
 }  
 **public String name() {**

**return this.name;**

**}**

**public int age() {**

**return this.age;**

**}**  
 ...  
}

Primary constructor执行类定义中的所有语句。

DEMO：

class Person(val name: String, val age: Int) {  
 **println("Just constructed another person")**  
 def description = s"$name is $age years old"  
}

println语句是primary constructor的一部分。当一个object被构建时，它将会执行。

NOTE：

当类名后面没有参数时，这个类具有一个没有参数的primary constructor，这个constructor将会执行类定义中的所有语句。

构造器的参数也可以不用val或者var来定义：

1. 如果一个参数没有val或者var，但是在类内部有至少一个函数用到了它，它将变成一个field。例如：

class Person(name: String, age: Int) {

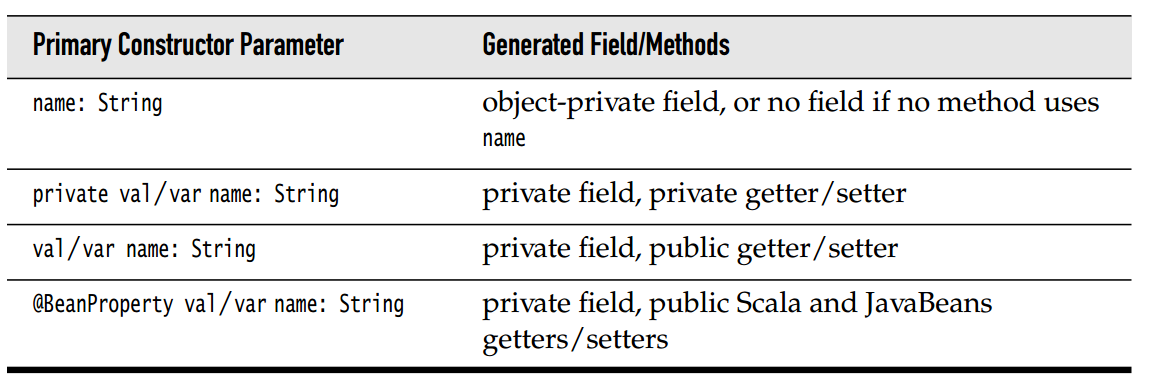
def description = s"$name is $age years old" // this method uses name and age

}

这样就会在类内部声明和初始化不可变的field： name和age，它们是object-private（private[this]）

1. 如果类内部没有方法用到，name参数将不会转化成field，此时它仅仅是一个参数，可以在primary constructor中访问，严格来讲，只是一个特定的实现优化。

下面是一个表，他展示primary constructor参数对应生成方法和field。



如果你感觉primary constructor概念有点混乱，你可以暂时不用它，通过使用auxiliary constructor来解决，但如果不调用另一个auxiliary constructor，请记住调用this（）。

但是，许多程序员喜欢简洁的语法。所以为了看懂别人的code，我们尽可能的去用这种方法，如果有什么不理解，可以来问我。在Scala中，类就像方法一样带参数。

Note：

当你把主要构造函数的参数看作是类参数时，没有val或var的参数就变得更容易理解了。这样一个参数的范围就是整个类。 因此，您可以在方法中使用该参数。 如果这样做，编译器的工作就是将其保存在一个字段中。

Scala的设计者认为每次敲击键盘都是珍贵的，所以他把类定义和primary constructor整合到一起。但是，当我们阅读scala类的时候，你需要把他们分开。举个例子：

class Person(val name: String) {  
 var age = 0  
 def description = s"$name is $age years old"  
}

类定义：

class Person(val name: String){

var age : Int = \_

def description = s"$name is $age years old"

}

Primary constructor:

Person(name: String){

Age = 0}

Note：

如果你想将primary constructor设为private，你可以这样：

class Person private(val id: Int) { ... }

这样类的使用者必须用auxiliary constructor来创建Person的实例。

* Nested Classes

在scala中，你可以在函数中定义函数，在类中定义类，下面是一个例子：

import scala.collection.mutable.ArrayBuffer  
class Network {  
 class Member(val name: String) {  
 val contacts = new ArrayBuffer[Member]  
 }

private val members = new ArrayBuffer[Member]  
 def join(name: String) = {  
 val m = new Member(name)  
 members += m

m  
 }  
}

我们可以构建两个network：

Val chatter = new Network

Val myFace = new Network

在scala 中，每个instance都有一个自己的class Member（nested class），就想每个instance都有一个自己的field members。也就是说chatter.Member和myFace.Member是不同的类。

NOTE：

这一点与java不同，在java中内部类属于外部类

Scala的用法看起来更符合习惯。你想构建一个内部类instance，你可以用new加类名：

new chatter.Member

在Java中，你需要用一个看起来有点怪的语法：chatter. New Member()

在我们network的例子中，你可以添加一个同一network的member，但是不能跨network访问。

val fred = chatter.join("Fred")  
val wilma = chatter.join("Wilma")  
fred.contacts += wilma // OK  
val barney = myFace.join("Barney") // Has type myFace.Member  
fred.contacts += barney // No—can’t add a myFace.Member to a buffer of chatter.Member elements

对于人的社交网络，这种行为可能是有道理的。 如果你不想要，有两个解决方案。

1. 你可以将Member的定义移到object模块（object后面我会讲到）

object Network {  
 class Member(val name: String) {  
 val contacts = new ArrayBuffer[Member]  
 }  
}  
class Network {  
 private val members = new ArrayBuffer[Network.Member]  
 ...  
}

1. 可以用反射来处理，Network#Member，这相当于任何Network中的Member：

class Network {

class Member(val name: String) {

val contacts = new ArrayBuffer[Network#Member]

}

...

}

现在不太懂反射没关系，用到的机会不是很多，我后面会专门找时间讲解。