

9 继承

9.1 扩展如下的 **BankAccount** 类，新类 **CheckingAccount** 对每次存款和取款都收取 1 美元的手续费 **class**

```
BankAccount(initialBalance:Double){  
    private var balance = initialBalance  
    def deposit(amount:Double) = { balance += amount; balance}  
    def withdraw(amount:Double) = {balance -= amount; balance}  
}
```

继承语法的使用。代码如下

Scala 代码 

```
1. class CheckingAccount(initialBalance:Double) extends BankAccount(initialBalance){  
2.     override def deposit(amount: Double): Double = super.deposit(amount - 1)  
3.  
4.     override def withdraw(amount: Double): Double = super.withdraw(amount + 1)  
5. }
```


9.2 扩展前一个练习的 **BankAccount** 类，新类 **SavingsAccount** 每个月都有利息产生(**earnMonthlyInterest** 方法被调用)，并且有每月三次免手续费的存款或取款。在 **earnMonthlyInterest** 方法中重置交易计数。

Scala 代码 

```
1. class SavingsAccount(initialBalance:Double) extends BankAccount(initialBalance){  
2.     private var num:Int = _  
3.  
4.     def earnMonthlyInterest()={  
5.         num = 3  
6.         super.deposit(1)  
7.     }  
8.  
9.     override def deposit(amount: Double): Double = {  
10.         num -= 1  
11.         if(num < 0) super.deposit(amount - 1) else super.deposit(amount)  
12.     }  
13.  
14.     override def withdraw(amount: Double): Double = {  
15.         num -= 1  
16.         if (num < 0) super.withdraw(amount + 1) else super.withdraw(amount)  
17.     }  
18. }
```

9.3 翻开你喜欢的 **Java** 或 **C++** 教科书，一定会找到用来讲解继承层级的实例，可能是员工，宠物，图形或类似的东西。用 **Scala** 来实现这个示例。

Thinking in Java 中的代码

Java 代码 

```

1.  class Art{
2.      Art(){System.out.println("Art constructor");}
3.  }
4.
5.  class Drawing extends Art{
6.      Drawing() {System.out.println("Drawing constructor");}
7.  }
8.
9.  public class Cartoon extends Drawing{
10.     public Cartoon() { System.out.println("Cartoon constructor");}
11. }

```

使用 Scala 改写如下

Scala 代码 

```

1.  class Art{
2.      println("Art constructor")
3.  }
4.
5.  class Drawing extends Art{
6.      println("Drawing constructor")
7.  }
8.
9.  class Cartoon extends Drawing{
10.     println("Cartoon constructor")
11. }

```

9.4 定义一个抽象类 **Item**,加入方法 **price** 和 **description**。**SimpleItem** 是一个在构造器中给出价格和描述的物件。利用 **val** 可以重写 **def** 这个事实。**Bundle** 是一个可以包含其他物件的物件。其价格是打包中所有物件的价格之和。同时提供一个将物件添加到打包当中的机制, 以及一个适合的 **description** 方法

Scala 代码 

```

1.  import collection.mutable.ArrayBuffer
2.
3.
4.  abstract class Item{
5.      def price():Double
6.      def description():String
7.
8.      override def toString():String={
9.          "description:" + description() + " price:" + price()
10.     }
11. }
12.
13. class SimpleItem(val price:Double, val description:String) extends Item{


```

```

14.
15. }
16.
17. class Bundle extends Item{
18.
19.     val items = new ArrayBuffer[Item]()
20.
21.     def addItem(item:Item){
22.         items += item
23.     }
24.
25.     def price(): Double = {
26.         var total = 0d
27.         items.foreach(total += _.price())
28.         total
29.     }
30.
31.     def description(): String = {
32.         items.mkString(" ")
33.     }
34. }

```

9.5 设计一个 **Point** 类，其 **x** 和 **y** 坐标可以通过构造器提供。提供一个子类 **LabeledPoint**，其构造器接受一个标签值和 **x,y** 坐标,比如:`new LabeledPoint("Black Thursday",1929,230.07)`

Scala 代码 

```

1.     class Point(x:Int,y:Int){
2.     }
3.
4.     class LabeledPoint(label:String,x:Int,y:Int) extends Point(x,y){
5.     }

```

9.6 定义一个抽象类 **Shape**，一个抽象方法 **centerPoint**，以及该抽象类的子类 **Rectangle** 和 **Circle**。为子类提供合适的构造器，并重写 **centerPoint** 方法

Scala 代码 

```

1.     abstract class Shape{
2.         def centerPoint()
3.     }
4.
5.     class Rectangle(startX:Int,startY:Int,endX:Int,endY:Int) extends Shape{
6.         def centerPoint() {}
7.     }
8.

```

```

9.    class Circle(x:Int,y:Int,radius:Double) extends Shape{
10.        def centerPoint() {}
11.    }

```

9.7 提供一个 **Square** 类，扩展自 **java.awt.Rectangle** 并且是三个构造器：一个以给定的端点和宽度构造正方形，一个以 **(0,0)** 为端点和给定的宽度构造正方形，一个以 **(0,0)** 为端点, **0** 为宽度构造正方形

Scala 代码 

```

1.    import java.awt.{Point, Rectangle}
2.
3.
4.    class Square(point:Point,width:Int) extends Rectangle(point.x,point.y,width,width){
5.
6.        def this(){
7.            this(new Point(0,0),0)
8.        }
9.
10.       def this(width:Int){
11.           this(new Point(0,0),width)
12.       }
13.    }

```

9.8 编译 8.6 节中的 **Person** 和 **SecretAgent** 类并使用 **javap** 分析类文件。总共有多少 **name** 的 **getter** 方法？它们分别取什么值？(提示：可以使用 **-c** 和 **-private** 选项)

总共两个。**Person** 中取得的是传入的 **name**,而 **SecretAgent** 中取得的是默认的"secret"

9.9 在 8.10 节的 **Creature** 类中，将 **val range** 替换成一个 **def**。如果你在 **Ant** 子类中也用 **def** 的话会有什么效果？如果在子类中使用 **val** 又会有什么效果？为什么？

在 **Ant** 中使用 **def** 没有问题。但是如果使用 **val** 则无法编译。因为 **val** 只能重写不带参数的 **def**。这里的 **def** 是带参数的

9.10 文件 **scala/collection/immutable/Stack.scala** 包含如下定义：

```
class Stack[A] protected (protected val elems: List[A])
```

请解释 **protected** 关键字的含义。(提示：回顾我们在第 5 章中关于私有构造器的讨论)

此构造方法只能被其子类来调用,而不能被外界直接调用