Scala核心编程

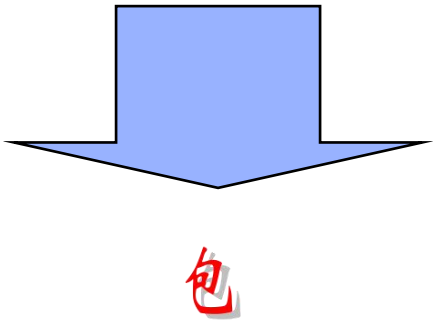
-面向对象编程（中级部分）

尚硅谷

- 包

看一个应用场景

现在有两个程序员共同开发一个项目,程序员xiaoming希望定义一个类取名Dog ,程序员xiaoqiang也想定义一个类也叫 Dog。两个程序员为此还吵了起来,怎么办?



- 包

回顾-Java包的三大作用

1) 区分相同名字的类
2) 当类很多时,可以很好的管理类
3) 控制访问范围

回顾-Java打包命令

➢ 打包基本语法

```
package com.atguigu;
```

➢ 打包的本质分析

实际上就是创建不同的**文件夹来保存类文件**，画出**示意图**。

快速入门

使用打包技术来解决上面的问题，不同包下Dog类

- 包

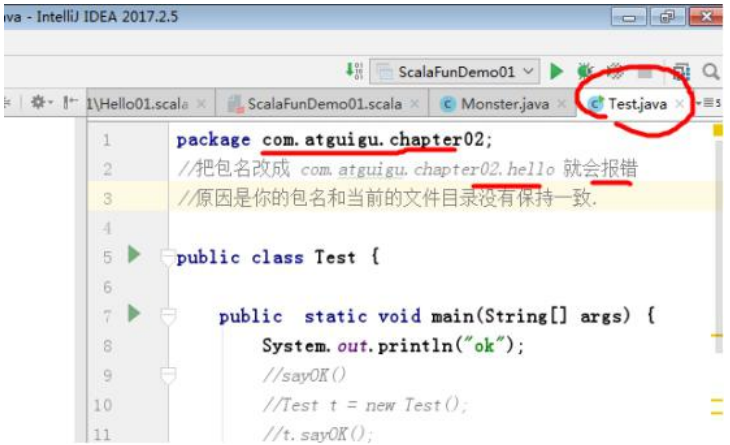
回顾-Java如何引入包

语法: import 包;

比如 import java.awt.*;
我们引入一个包的主要目的是要使用该包下的类
比如 import java.util.Scanner; 就只是引入一个类Scanner。

回顾-Java包的特点

java中包名和源码所在的系统**文件目录结构要一致**，并且编译后的字节码文件路径也和包名保持一致。[如图]



- 包

Scala包的基本介绍

和Java一样，Scala中管理项目可以使用包，但Scala中的包的功能更加强大，使用也相对复杂些，下面我们学习Scala包的使用和注意事项。

Scala包快速入门

使用打包技术来解决上面的问题，不同包下Dog类

```
package com.atguigu.chapter02.xh
class Cat {
}
```

```
package com.atguigu.chapter02.xm
class Cat {
}
```

```
var cat1 = new com.atguigu.chapter02.xh.Cat()
println("cat1" + cat1)
var cat2 = new com.atguigu.chapter02.xm.Cat()
println("cat2" + cat2)
```

- 包

Scala包的特点概述

- 基本语法
  package 包名

- Scala包的三大作用(和Java一样)
  1) 区分相同名字的类
  2) 当类很多时,可以很好的管理类
  3) 控制访问范围

- Scala中包名和源码所在的系统**文件目录结构要可以不一致**，但是编译后的字节码文件路径和包名会保持一致(这个工作由**编译器完成)**。[案例演示]

class文件路径

- 包

Scala包的命名

➢ 命名规则：

只能包含**数字**、字母、下划线、**小圆点**.,但不能用数字开头, 也不要使用关键字。
demo.**class**.exec1 //错误 , 因为class是关键字
demo.12a // 错误，因为不能以数字开头

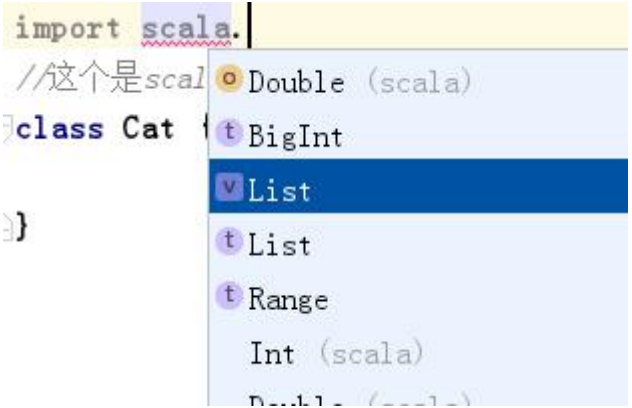
➢ 命名**规范**：

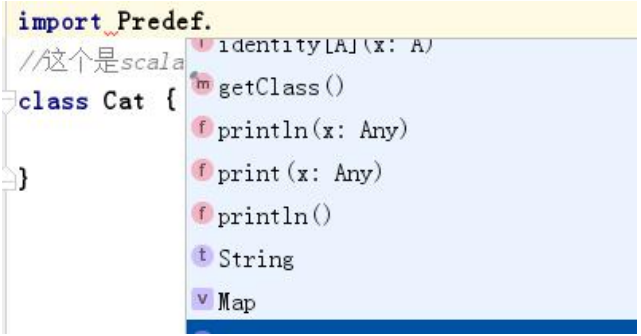
一般是小写字母+小圆点一般是
com.**公司名**.项目名.业务模块名
比如：com.atguigu.**oa**.model com.atguigu.oa.controller
com.sina.edu.user
com.sohu.bank.order

- 包

Scala会自动引入的常用包

java.lang.*
scala包
Predef包





- 包

Scala包注意事项和使用细节

1) scala进行package 打包时，可以有如下形式。【案例+反编译】

```
ckage com.atguigu.scala
ss Person{
al name = "Nick"
ef play(message: String): Unit ={
ntln(this.name + " " + message)


码说明 传统的方式
```

等同

```
package com.atguigu
package scala
class Person{
  val name = "Nick"
  def play(message: String): Unit ={
println(this.name + " " + message)
  }
}
代码说明 ： 和第一种方式完全等价
```

```
package com.atguigu{
  package scala{
    class Person{
      val name = "Nick"
      def play(message: String
println(this.name + " " + messa
      }
    }
  }
} //代码说明
```

- 包

Scala包注意事项和使用细节

2) 包也可以像嵌套类那样嵌套使用（包中有包）,这个在前面的第三种打包方式已经讲过了，在使用第三种方式时的**好处是**：程序员可以在同一个文件中，将**类(class / object)、trait** 创建在不同的包中，这样就非常灵活了。[案例+反编译]

```
package com.atguigu{

  //这个类就是在com.atguigu包下
  class User{
  }
  //这个类对象就是在Monster$ , 也在com.atguigu包下
  object Monster {
  }

  package scala {
    //这个类就是在com.atguigu.scala包下
    class User{
```


图片6.z

- 包

Scala包注意事项和使用细节

3) 作用域原则：可以直接向上访问。即: **Scala中子包中直接访问父包中的内容,** 大括号体现作用域。(**提示：**Java中子包使用父包的类，需要import)。在子包和父包 类重名时，默认采用就近原则，如果希望指定使用某个类，则带上包名即可。【案例演示+反编译】

```
package com.atguigu{
 //这个类就是在com.atguigu包下
 class User{
 }
 //这个类对象就是在Monster$ , 也在com.atguigu包下
 object Monster {
 }
 class Dog {
 }
 package scala {
  //这个类就是在com.atguigu.scala包下
  class User{
  }
  //这个Test 类对象
  object Test {
   def main(args: Array[String]): Unit = {
      //子类可以直接访问父类的内容
      var dog = new Dog()
      println("dog=" + dog)
      //在子包和父包 类重名时，默认采用就近原则.
      var u = new User()
      println("u=" + u)
      //在子包和父包 类重名时，如果希望指定使用某个类，则
      var u2 = new com.atguigu.User()
      println("u2=" + u2)
   }
```

- 包

Scala包注意事项和使用细节

4) 父包要访问子包的内容时，需要import对应的类等

```
package com.atguigu{
 //引入在com.atguigu 包中希望使用到子包的类Tiger,因此需要引入.
 import com.atguigu.scala.Tiger
 //这个类就是在com.atguigu包下
 class User{
 }
 package scala {
  //Tiger 在 com.atguigu.scala 包中
  class Tiger {}}
 object Test2 {
  def main(args: Array[String]): Unit = {
     //如果要在父包使用到子包的类，需要import
     val tiger = new Tiger()
     println("tiger=" + tiger)
  }}}
```

5) 可以在同一个.scala文件中，声明多个并列的package(建议嵌套的pakage不要超过3层) [案例+反编译]

- 包

Scala包注意事项和使用细节

6) 包名可以相对也可以绝对，比如，访问BeanProperty的绝对路径是：_root_. scala.beans.BeanProperty，**在一般情况下**：我们使用相对路径来引入包，只有当**包名冲突时**，使用绝对路径来处理。[案例演示]

```
package com.atguigu.scala2
class Manager( var name : String ) {
  //第一种形式
  //@BeanProperty var age: Int = _
  //第二种形式, 和第一种一样，都是相对路径引入
  //@scala.beans.BeanProperty var age: Int = _
  //第三种形式, 是绝对路径引入，可以解决包名冲突
  @_root_. scala.beans.BeanProperty var age: Int = _
}
object TestBean {
  def main(args: Array[String]): Unit = {
    val m = new Manager("jack")
    println("m=" + m)
  }}
```

- 包

包对象

基本介绍：包可以包含**类、对象和特质trait**，但**不能包含函数/方法或变量的定义**。这是Java虚拟机的局限。为了弥补这一点不足，scala提供了**包对象**的概念来解决这个问题。

包对象的应用案例

案例演示

```
package com.atguigu {
 //每个包都可以有一个包对象。你需要在父包(com.atguigu)中定义它,
 package object scala {
  var name = "jack"
  def sayOk(): Unit = {
   println("package object sayOk!")
  }
 }
 package scala {
  class Test {
   def test() : Unit ={
    //这里的name就是包对象scala中声明的name
    println(name)
    sayOk()//这个sayOk 就是包对象scala中声明的sayOk
   }
  }
  object TestObj {
   def main(args: Array[String]): Unit = {
    val t  = new Test()
    t.test()
    //因为TestObje和scala这个包对象在同一包，因此也可以使用
    println("name=" + name)
   }}}}
```

- 包

包对象的底层实现机制分析(重点)

说明：看反编译代码+分析图

1) 当创建包对象后，在该包下生成 public final class package 和 public final class package$
2) 通过 package$ 的一个静态实例完成对包对象中的属性和方法的调用。[java模拟]

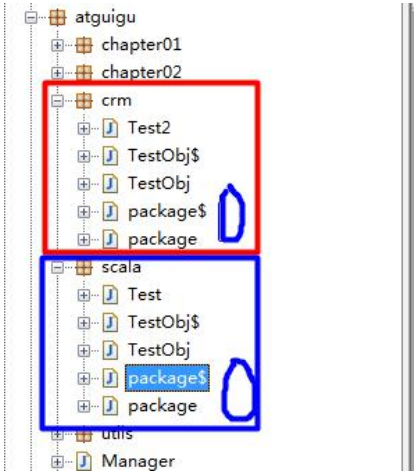
```
public class Temp100 {
    public static void main(String[] args) {
        System.out.println(package$$.MODULE$.name());
        package$$.MODULE$.sayOk();
    }}
 final class package$$
{
    public static   final package$$ MODULE$;
    private final String name = "ok";
    static {MODULE$ = new package$$(); }
    public void sayOk()
    { System.out.println("scala pkg obj sayok"); }
    public String name() { return this.name; }
}
```

包对象的底层

- 包

包对象的注意事项

1) 每个包都可以有一个包对象。你需要在父包中定义它。如图：



2) 包对象名称需要和包名一致，一般用来对包的功能补充

- 包的可见性

回顾-Java访问修饰符基本介绍

java提供**四种访问控制修饰符号**控制方法和变量的访问权限（范围）：

1) 公开级别:用public 修饰,对外公开
2) 受保护级别:用protected修饰,对子类和同一个包中的类公开
3) 默认级别:没有修饰符号,向同一个包的类公开.
4) 私有级别:用**private**修饰,只有类本身可以访问,不对外公开.

- 包的可见性

回顾-Java中4种访问修饰符的访问范围

1	访问级别	访问控制修饰符	同类	同包	子类	不同包
2	公开	public	√	√	√	√
3	受保护	protected	√	√	√	X
4	默认	没有修饰符	√	√	X	X
5	私有	private	√	X	X	X

回顾-Java访问修饰符使用注意事项

1) 修饰符可以用来修饰类中的**属性**，成员方法以及类
2) 只有**默认**的和**public**才能修饰类！，并且遵循上述访问权限的特点。

- 包的可见性

Scala中包的可见性介绍:

在Java中，访问权限分为: public，private，protected和默认。**在Scala中**，你可以通过类似的修饰符达到同样的效果。但是**使用上有区别**。

```
object Testvisit {
  def main(args: Array[String]): Unit = {
    val  c = new Clerk()
    c.showInfo()
    Clerk.test(c)
  }}
class Clerk {
  var name : String = "jack"
  private var sal : Double = 9999.9
  def showInfo(): Unit = {
    println(" name " + name + " sal= " + sal)
  }}
object Clerk{
  def test(c : Clerk): Unit = {
    //这里体现出在伴生对象中，可以访问c.sal
    println("test() name=" + c.name + " sal= " + c.sal)
```

- 包的可见性

Scala中包的可见性和访问修饰符的使用

1) 当属性访问权限为默认时，从底层看属性是private的，但是因为提供了 **xxx_$eq()**[类似setter]/**xxx()**[类似getter] 方法，因此从使用效果看是任何地方都可以访问)
2) 当方法访问权限为默认时，**默认为public访问权限**
3) private为私有权限，只在**类的内部**和**伴生对象**中可用 【案例演示】
4) **protected为受保护权限，**scala中受保护权限比Java中更严格，只能子类访问，同包无法访问 (编译器)
5) 在scala中没有public关键字,即**不能用public显式的修饰属性和方法**。 【案演】

- 包的可见性

Scala中包的可见性和访问修饰符的使用

6) 包访问权限（表示属性有了限制。同时包也有了限制），这点和Java不一样，体现出Scala包使用的灵活性。[案例演示]

```
package com.atguigu. scala
class Person {
  private[scala] val pname="hello" // 增加包访问权限后，1. private同时起作用。不仅同类可以使用 2. 同时com.atguigu.scala中包下其他类也可以使用
}
```

当然，也可以将可见度延展到上层包

```
private[atguigu] val description="zhangsan"
```

说明：private也可以变化，比如protected[atguigu], 非常的灵活。

- 包的引入

Scala引入包基本介绍

Scala引入包也是使用import, 基本的原理和机制和Java一样，但是Scala中的import功**能更加强大，也更灵活**。

因为Scala语言源自于Java，所以**java.lang**包中的类会自动引入到当前环境中，而Scala中的**scala**包和**Predef**包的类也会自动引入到当前环境中，即起其下面的类可以直接使用。

如果想要把其他包中的类引入到当前环境中，需要使用import语言

- 包的引入

Scala引入包的细节和注意事项

1) 在Scala中，import语句可以出现在任何地方，并不仅限于文件顶部，import语句的作用一直延伸到包含该语句的块末尾。这种语法的好处是：**在需要时在引入包，缩小import 包的作用范围，提高效率**。

```
class User {
    import scala.beans.BeanProperty
    @BeanProperty var  name : String = ""
}
class Dog {
    @BeanProperty var  name : String = "" //可以吗?
}
```

2) Java中如果想要导入包中所有的类，可以通过通配符*，Scala中采用下 _ [案例演示]

- 包的引入

Scala引入包的细节和注意事项

3) 如果不想要某个包中全部的类，而是其中的几个类，可以采用**选取器**(大括号)

```
def test(): Unit = {
    import scala.collection.mutable.{HashMap, HashSet}
    var map = new HashMap()
    var set = new HashSet()
}
```

- 包的引入

Scala引入包的细节和注意事项

4) 如果引入的多个包中含有相同的类，那么可以将不需要的类进行重命名进行区分，这个就是**重命名**。【案例演示+小结】

```
import java.util.{ HashMap=>JavaHashMap, List}
import scala.collection.mutable._
var map = new HashMap() // 此时的HashMap指向的是scala中的HashMap
var map1 = new JavaHashMap(); // 此时使用的java中hashMap的别名
```

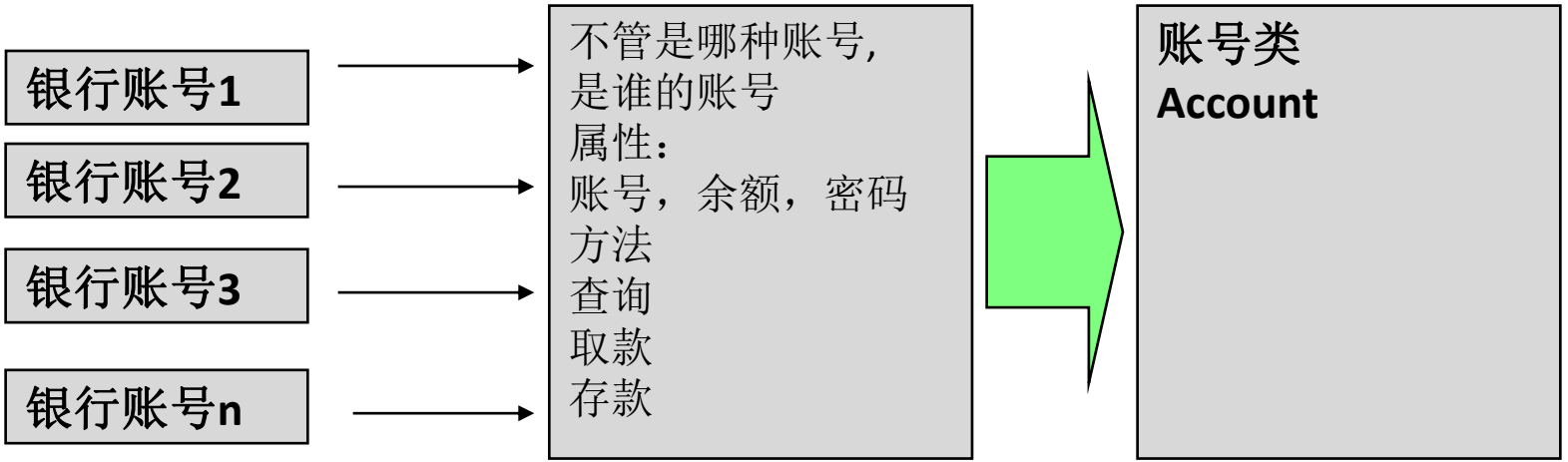
5) 如果某个冲突的类根本就不会用到，那么这个类可以直接**隐藏**掉。[案例演示]

```
import java.util.{ HashMap=>_, _} // 含义为 引入java.util包的所有类，但是忽略HahsMap类.
var map = new HashMap() // 此时的HashMap指向的是scala中的HashMap, 而且idea工具，的提示也不会显示java.util的HashMaple
```

- 面向对象编程方法-抽象

如何理解抽象

我们在前面去定义一个类时候，实际上就是把一类事物的共有的**属性和行为提取出来**，形成一个物理模型(模板)。这种**研究问题的方法称为抽象**。[见后面ppt]



- **面向对象编程三大特征**

基本介绍

面向对象编程有三大特征：**封装**、**继承**和**多态**。
下面我们一一为同学们进行详细的讲解。

- 面向对象编程-封装

封装介绍

封装(encapsulation)就是把抽象出的**数据和对数据的操作**封装在一起,数据被保护在内部,程序的其它部分只有通过**被授权的操作**(成员方法),才能对数据进行操作。

对电视机的操作就是典型封装

- 面向对象编程-封装

封装的理解和好处

1) 隐藏实现细节
2) 提可以对**数据进行验证**，保证**安全**合理

如何体现封装

1) 对类中的属性进行封装
2) 通过成员方法，包实现封装

- 面向对象编程-封装

封装的实现步骤

1) 将属性进行私有化
2) 提供一个公共的set方法，用于对属性判断并赋值

```
def  setXxx(参数名 : 类型) : Unit = {
        //加入数据验证的业务逻辑
        属性 = 参数名
}
```

3) 提供一个公共的get方法，用于获取属性的值

```
def getXxx() [: 返回类型] = {
        return 属性
 }
```

面向对象编程-封装

快速入门案例

➢ 看一个案例

那么在Scala中如何实现这种类似的控制呢?
请大家看一个小程序(TestEncap.scala),不能随便查看人的年龄,工资等隐私，
并对输入的年龄进行合理的验证[要求1-120之间]。

```
class Person {
  var name: String = _
  //var age ; //当是public时， 可以随意的进行修改，不安全
  private var age: Int = _
  private var salary: Float = _
  private var job: String = _
} //案例演示
```

```
def setAge(age: Int): Unit = {
    if (age >= 0 && age <= 120) {
      this.age = age
    } else {
      println("输入的数据不合理");
      //可考虑给一个默认值
      this.age = 20
    }}
```

- 面向对象编程-封装

课堂练习(学员做, 较简单)

创建程序,在其中定义两个类：Account和AccountTest类体会Scala的封装性。

1) Account类要求具有属性：姓名（长度为2位3位或4位）、余额(必须>20)、密码（必须是六位）
2) 通过setXxx的方法给Account 的属性赋值。
3) 完成转账的功能.
4) 在AccountTest中测试

提示知识点：

```
var name : String = “”
var len = name.length() //按照字符的个数统计的
```

- 面向对象编程-封装

Scala封装的注意事项和细节

前面讲的Scala的封装特性，大家发现和Java是一样的，下面我们看看Scala封装还有哪些特点。

1) **Scala中为了简化代码的开发**，当声明属性时，本身就自动提供了对应setter/getter方法，如果属性声明为private的，那么自动生成的setter/getter方法也是private的，如果属性省略访问权限修饰符，那么自动生成的setter/getter方法是public的[案例+反编译+说明]

```
class Cat(inAge:Int) {
  private var age: Int = 0
  var name: String = ""
  private val age2: Int = 0
  val name2: String = ""
}
```

- 面向对象编程-封装

Scala封装的注意事项和细节

2） 因此我们如果只是对一个属性进行简单的set和get，只要声明一下该属性(属性使用默认访问修饰符) 不用写专门的getset，默认会创建，访问时，直接对象. 变量。这样也是为了保持访问一致性［案例］

3) 从形式上看 dog.food 直接访问属性，其实底层仍然是**访问的方法**, 看一下反编译的代码就明白

4) 有了上面的特性，目前很多新的框架，在进行反射时，也支持对属性的直接反射

- 面向对象编程-继承

Java继承的简单回顾

class 子类名 extends 父类名 { 类体 }
子类继承父类的属性和方法

继承基本介绍和示意图

继承可以**解决代码复用**,让我们的编程更加靠近人类思维.当**多个类存在相同的属性**(变量)和**方法时**,可以从这些类中抽象出父类(比如Student),在父类中定义这些相同的属性和方法，所有的子类不需要重新定义这些属性和方法，只需要通过extends语句来声明继承父类即可。

和Java一样，Scala也支持类的单继承

画出继承的示意图

- 面向对象编程-继承

Scala继承的基本语法

class 子类名 extends 父类名 { 类体 }

Scala继承快速入门

- 编写一个Student 继承 Person的案例，体验一下Scala继承的特点

```
class Person {
  var name : String = _
  var age : Int = _
  def showInfo(): Unit = {
    println("学生信息如下：")
    println("名字：" + this.name)
  }
}

class Student extends Person {
  def studying(): Unit = {
    println(this.name + "学习 scala中....")
  }
}
```

- 面向对象编程-继承

Scala继承给编程带来的便利

1) 代码的复用性提高了
2) 代码的扩展性和维护性提高了【面试官问:当我们修改父类时，对应的子类就会继承相应的方法和属性】

scala子类继承了什么,怎么继承了?

子类继承了**所有的属性**，只是私有的属性不能直接访问，需要**通过公共的方法**去访问【**debug**代码验证可以看到】


继承的属

```
object Extends02 {
  def main(args: Array[String]): Unit =
    val sub = new Sub()
    sub.sayOk()
  }}

class Base {
  var n1: Int = 1
  protected var n2: Int = 2
  private var n3: Int = 3
  def test100(): Unit = {
    println("base 100")
  }
  protected def test200(): Unit = {
    println("base 200")
  }
  private def test300(): Unit = {
    println("base 300")
  }
}
class Sub extends Base {
  def sayOk(): Unit = {
    this.n1 = 20
    this.n2 = 40
    println("范围" + this.n1 + this.n2)
  }}
```

- 面向对象编程-继承

重写方法

说明: scala明确规定，重写一个非抽象方法需要用**override**修饰符，调用超类的方法使用**super**关键字 【案例演示+反编译+注释】

```
class Person {
  var name : String = "tom"
  def printName() {
    println("Person printName() " + name)
  }
}
```

```
class Emp extends Person {
  //这里需要显式的使用override
  override def printName() {
    println("Emp printName() " + name)
    super.printName()
  }
}
```

- 面向对象编程-继承

Scala中类型检查和转换

➢ 基本介绍

要测试某个对象是否属于某个给定的类，可以用**isInstanceOf**方法。用**asInstanceOf**方法将引用转换为子类的引用。classOf获取对象的类名。

1) classOf[String]就如同Java的 String.class 。
2) obj.isInstanceOf[T]就如同Java的obj instanceof T **判断obj是不是T类型**。
3) obj.asInstanceOf[T]就如同Java的(T)obj **将obj强转成T类型**。

前面案例基础演示+文档

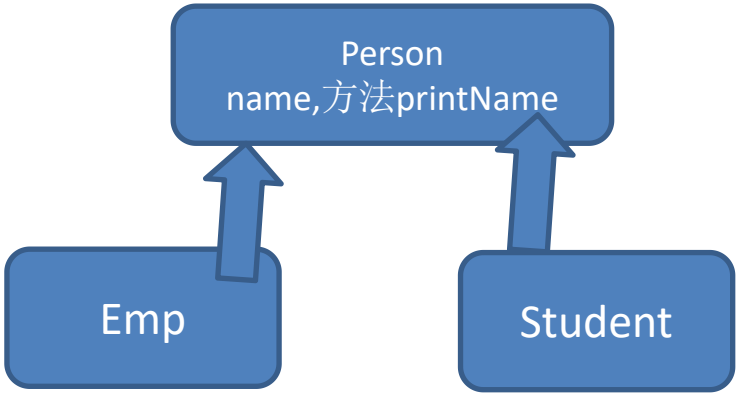
```
// 获取对象类型
println(classOf[String])
val s = "zhangsan"
println(s.getClass.getName) //这种是Java中反射方式得到类型
println(s.isInstanceOf[String])
println(s.asInstanceOf[String]) //将s 显示转换成String

var p = new Person2
val e = new Emp
p = e //将子类对象赋给父类。
p.name = "xxx"
println(e.name)
p.asInstanceOf[Emp].sayHi()
```

- 面向对象编程-继承

Scala中类型检查和转换

➢ 最佳实践

类型检查和转换的最大价值在于：可以判断传入对象的类型，然后转成对应的子类对象，进行相关操作，这里也体现出多态的特点。【案例演示】



- 面向对象编程-继承

Scala中超类的构造

➢ 回顾-Java中超类的构造

说明：

从代码可以看出：在Java中，创建子类对象时，子类的构造器总是去调用一个父类的构造器(显式或者隐式调用)。

```
class A {
    public A() {
        System.out.println("A()");
    }
    public A(String name) {
        System.out.println("A(String name)" + name);
    }
}
class B extends A{
    public B() {
        //这里会隐式调用super(); 就是无参的父类构造器A()
        System.out.println("B()");
    }
    public B(String name) {
        super(name);
        System.out.println("B(String name)" + name);
    }
}
```

- 面向对象编程-继承

Scala中超类的构造

➢ Scala超类的构造说明

1) 类有一个主构器和任意数量的辅助构造器，而每个辅助构造器都必须先调用主构造器(也可以是间接调用.)，这点在前面我们说过了。

```
class Person {
    var name = "zhangsan"
    println("Person...")}
class Emp extends Person {
      println("Emp ....")
      def this(name : String) {
          this // 必须调用主构造器
          this.name = name
           println("Emp 辅助构造器~")
      }}
```

- 面向对象编程-继承

Scala中超类的构造

2) 只有**主构造器可以调用父类的构造器**。辅助构造器不能直接调用父类的构造器。在Scala的构造器中，你不能调用super(params) 【案例演示+反编译】

```
class Person(name: String) { //父类的构造器
}
class Emp (name: String) extends Person(name) {// 将子类参数传递给父类构造器,这种写法√

  // super(name)  (×) 没有这种语法
  def  this() {
          super("abc") // (×)不能在辅助构造器中调用父类的构造器
  }
}
```

- 面向对象编程-继承

Scala中超类的构造

➢ 应用实例
编写程序，创建一个学生对象。体会Scala中超类的构造流程。【案例+反编译】

```
class Person( pname : String )  {
  var name : String = pname
  def printName() {
    println("Person printName() " + name)
  }
}
```

```
class Student( studentname : String ) extends  Person(studentname)

  var sno : Int = 20

  override def printName() {
    println("Student printName() " + name)
    super.printName()
  }
}
```


图片1.z


图片2.z

- 面向对象编程-继承

覆写字段

➢ 基本介绍

在Scala中，子类改写父类的字段，我们称为**覆写/重写**字段。覆写字段需使用override修饰。

回顾：在Java中只有方法的重写，没有属性/字段的重写，准确的讲，是隐藏字段代替了重写。参考：**Java中为什么字段不能被重写.doc** [字段隐藏案例]。

Java中为什么字段不

- 面向对象编程-继承

覆写字段

Java的动态绑定机制简

- 回顾-Java另一重要特性: 动态绑定机制

```
class A {
    public int i = 10;
    public int sum() {
        return getI() + 10;
    }
    public int sum1() {
        return i + 10;
    }
    public int getI() {
        return i;
    }
}
```

```
class B extends A {
    public int i = 20;
    public int sum() {
        return i + 20;
    }
    public int getI() {
        return i;
    }
    public int sum1() {
        return i + 10;
    }
}
```

```
A a = new B();
System.out.println(a.sum());  //?
System.out.println(a.sum1()); //?
```

- 面向对象编程-继承

覆写字段

➢ **Scala覆写字段快速入门**

我们看一个关于覆写字段的案例。【案例演示+分析+反编译】

```
class A {
  val age : Int = 10
}
```

```
class B extends A {
  override val age : Int = 20
}
```

```
val obj : A = new B()
val obj2 : B = new B()
//看看各个对象的age值?
```

- 面向对象编程-继承

覆写字段

➢ 覆写字段的注意事项和细节

1) def只能重写另一个def(即：方法只能重写另一个方法)
2) val只能重写另一个val 属性 或 **重写不带参数**的def [案例+分析]

```
//代码正确吗?
class AAAA {
  var name: String = ""
}
class BBBB extends AAAA {
  override  val name: String = "jj"
}
//如果真的ok?
```

```
class A {
   def sal(): Int = {
      return 10
   }}

class B extends A {
 override val sal : Int = 0
}
```

- 面向对象编程-继承

覆写字段

➢ 覆写字段的注意事项和细节

3) var只能重写另一个**抽象的var属性** [案例+反编译+小结]

```
abstract class A03{
  val age:Int = 10
  var name:String
}
```

```
class B03 extends A03{
  override val age: Int = 20
  var name:String = _
}
```


图片3.z

➢ 抽象属性：声明未初始化的变量就是抽象的属性,抽象属性在抽象类

➢ var重写抽象的var属性小结

1. 一个属性没有初始化，那么这个属性就是抽象属性
2. 抽象属性在编译成字节码文件时，属性并不会声明，但是会自动生成抽象方法，所以类必须声明为抽象类
3. 如果是覆写一个父类的抽象属性，那么override 关键字可省略 [原因：父类的抽象属性，生成的是抽象方法，因此就不涉及到方法重写的概念，因此override可省略]

- 面向对象编程-继承

抽象类

➢ 基本介绍

在Scala中，通过abstract关键字标记不能被实例化的类。方法不用标记abstract，只要省掉方法体即可。抽象类可以拥有抽象字段，抽象字段/属性就是没有初始值的字段

➢ 快速入门案例

我们看看如何把Animal做成抽象类，包含一个抽象的方法cry()

```
abstract class Animal{
  var name : String //抽象的字段
  var age : Int // 抽象的字段
  var color : String = "black"
  def cry()
}
```

- 面向对象编程-继承

抽象类

➢ 抽象类基本语法

```
abstract class Person() { // 抽象类
  var name: String // 抽象字段, 没有初始化
  def printName // 抽象方法, 没有方法体

}
```

说明：抽象类的价值更多是在于设计，是设计者设计好后，让子类继承并实现抽象类(即：实现抽象类的抽象方法)

- 面向对象编程-继承

抽象类

➢ Scala抽象类使用的注意事项和细节讨论

1) 抽象类不能被实例
2) 抽象类不一定要包含abstract方法。也就是说,抽象类可以没有abstract方法
3) 一旦类包含了抽象方法或者抽象属性,则这个类必须声明为abstract

4) 抽象方法不能有主体，不允许使用abstract修饰。
5) 如果一个类继承了抽象类，则它必须实现抽象类的所有抽象方法和**抽象属性**，除非它自己也声明为abstract类。【案例演示+反编译】
6) **抽象方法和抽象属性**不能使用private、final 来修饰，因为这些关键字都是和**重写/实现**相违背的。
7) 抽象类中可以有实现的方法.
8) 子类重写抽象方法不需要override，写上也不会错.

- 面向对象编程-继承

匿名子类

➢ 基本介绍

和Java一样，可以**通过包含带有定义或重写的代码块的方式**创建一个匿名的子类.

➢ 回顾-java匿名子类使用

```
abstract class A2{
   abstract public   void cry();
}
```

```
A2 obj = new A2() {
        @Override
        public void cry() {
           System.out.println("okook!");
        }
};
```

➢ scala匿名子类案例

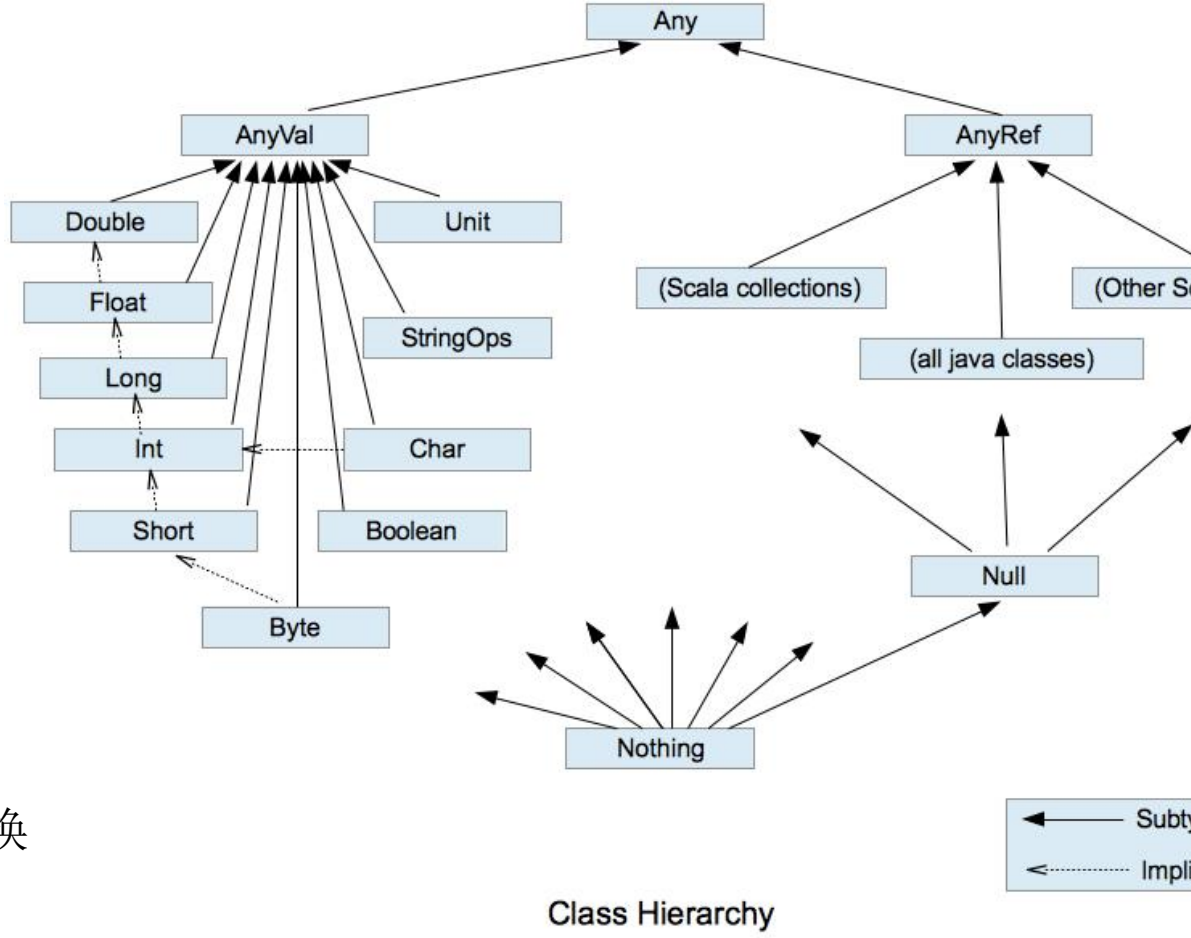
```
abstract class Monster{
  var name : String
  def cry()}
```

```
var monster = new Monster {
     override var name: String = "牛魔王"
     override def cry(): Unit = {
       println("牛魔王哼哼叫唤..")
     }
   }
```

- 面向对象编程-继承

继承层级

- **Scala继承层级一览图**



subtype : 子类型
implicit Conversion 隐式转换
class hierarchy : 类层次

- 面向对象编程-继承

继承层级

➢ 继承层级图小结

1) 在scala中，所有其他类都是AnyRef的子类，类似Java的Object。
2) AnyVal和AnyRef都扩展自Any类。Any类是根节点
3) Any中定义了isInstanceOf、asInstanceOf方法，以及哈希方法等。

4) Null类型的唯一实例就是null对象。可以将null赋值给任何引用，但不能赋值给值类型的变量[案例演示]。
5) Nothing类型没有实例。它对于泛型结构是有用处的，举例：空列表Nil的类型是List[Nothing]，它是List[T]的子类型，T可以是任何类。

- 面向对象编程作业

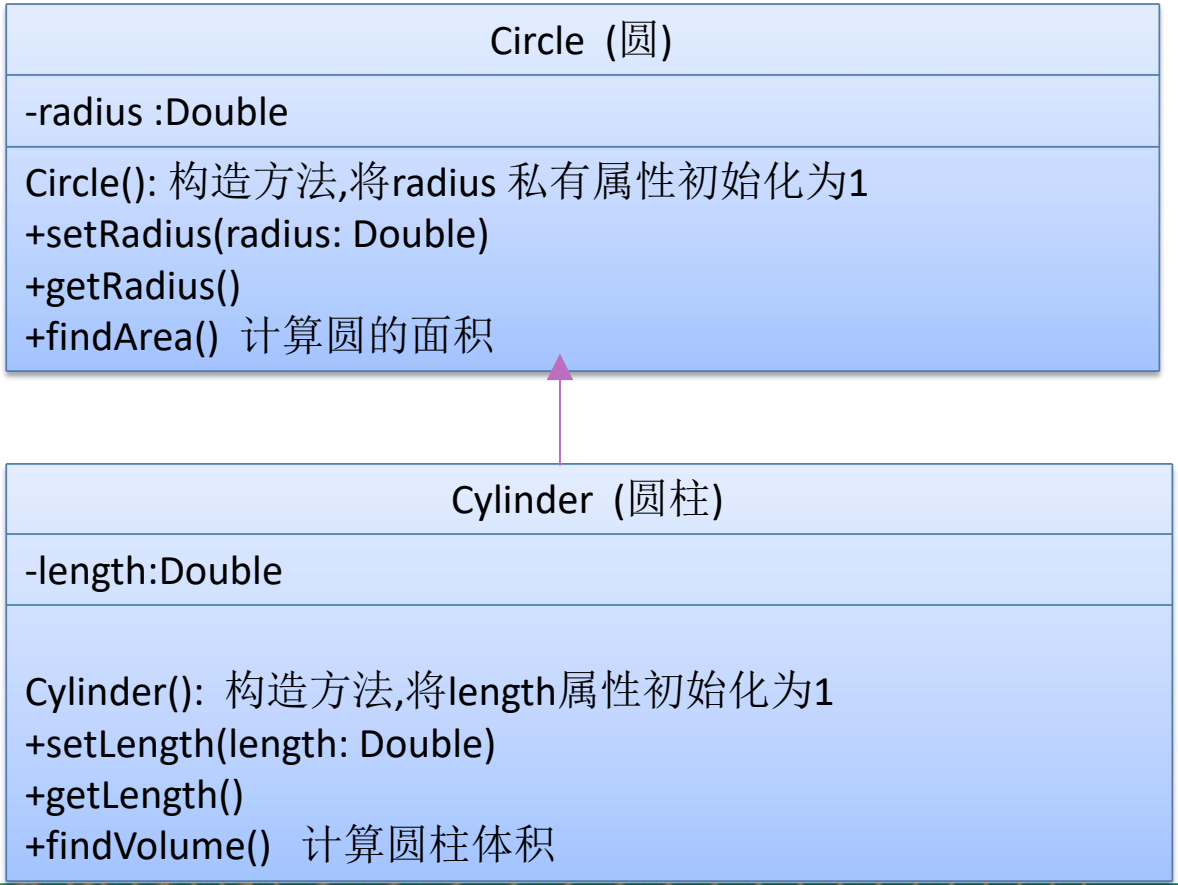
课堂练习【学生先做】

- 练习1
  编写Computer类，包含CPU、内存、硬盘等属性，getDetails方法用于返回Computer的详细信息
  编写PC子类，继承Computer类，添加特有属性【品牌brand】
  编写NotePad子类，继承Computer类，添加特有属性【颜色color】
  编写Test Object，在main方法中创建PC和NotePad对象，分别对象中特有的属性赋值，以及从Computer类继承的属性赋值，并使用方法并打印输出信息。

- 面向对象编程作业

课堂练习

➢ 练习2
根据下图实现类。在TestCylinder类中创建Cylinder类的对象，设置圆柱的底面半径和高，并输出圆柱的体积



- 面向对象编程作业

➢ **练习3(多态应用)**

定义员工类Employee，包含姓名和月工资，以及计算年工资getAnnual的方法。普通员工和经理继承了员工，经理类多了奖金bonus属性和管理manage方法，普通员工类多了work方法，普通员工和经理类要求分别重写getAnnual方法

测试类中添加一个方法showEmpAnnal，实现获取任何员工对象的年工资,并在main方法中调用该方法

测试类中添加一个方法，testWork,如果是普通员工，则调用work方法，如果是经理，则调用manage方法 【10min】

谢谢！ 欢迎收看！