**今天完成的事情：**

**第一部分：JAVA基础学习**

**学习预览：**

* JAVA数组
* JAVA多态
* 抽象类
* 封装

**第二部分：修真任务**

**学习预览：**

**第一部分：JAVA基础**

1. **JAVA数组**

由于之前学C语言时学习过，所以选择跳过了这一内容

但是害怕JAVA有一些数组的特性，所以决定重头在学，遂在此回头复习一遍

1. **声明数组变量**

声明数组变量有两种方法：

* 首选方法

**arrayType数组类型[ ] arrayReVar数组变量**

* 次选方法

**arrayType arrayReVar [ ]**

1. **创建数组**

JAVA语言使用new操作符来创建数组：

**arrayReVar = new datatype [ arraySize ]**

以上创建过程包含两个步骤：

* **使用 dataType[arraySize] 创建了一个数组**
* **把新创建的数组的引用（指针地址）赋值给变量 arrayRefVar**

**数组变量的声明，和创建数组可以用一条语句完成：**

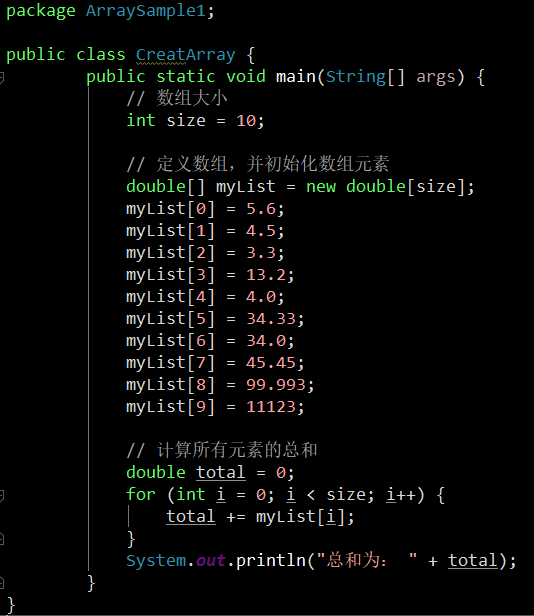
**dataType[ ] arrayRefVar = new datatype [arraySize]**

**另还可以使用如下方式创建数组（创建数组元素）**

**dataType[ ] arrayRefVar = {value0, value1, ..., valuek}**

数组是通过索引访问的，所以数组的元素列表从0开始

1. 实例



1. **处理数组**
2. **For-Each循环**

For-Each循环能在不使用下标的情况下遍历数组

语法格式为：

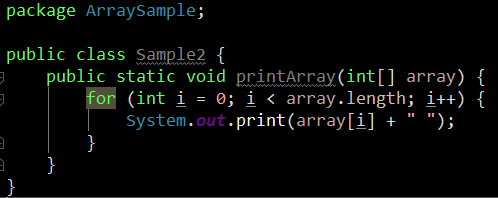
**for(type element: array){**

**System.out.println(element);**

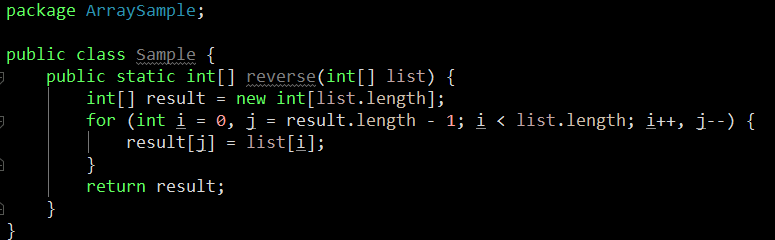
**}**

1. **数组作为函数的参数**

下例就是一个打印int数组中元素的方法：



1. **数组作为函数的返回值**



1. **二维数组**

二维数组格式如下：

**String str[ ][ ] = new String[3][4]**

1. **多维数组的动态初始化（以二维数组为例）**

* 直接为每一维分配空间

**type[ ][ ] typeName = new type[typeLength1][typeLength2]**

实例：

**int a[ ][ ] = new int[2][3]**

* 从最高维开始，分别为每一维分配空间

**String s[ ][ ] = new String[2][ ];**

**s[0] = new String[2];**

**s[1] = new String[3];**

**s[0][0] = new String("Good");**

**s[0][1] = new String("Luck");**

**s[1][0] = new String("to");**

**s[1][1] = new String("you");**

**s[1][2] = new String("!");**

1. **多维数组的引用**

对二维数组中的每个元素，引用方式为

**arrayName[index1][index2]**

例如：

**num[1][0];**

1. **Arrays类**

java.util.Arrays 类能方便地操作数组，它提供的所有方法都是静态的

具有以下功能：

* **给数组赋值：**

通过fill方法

* **对数组排序：**

通过sort方法，按升序

* **比较数组：**

通过equals方法比较数组中元素值是否相等

* **查找数组元素：**

通过binarySearch方法能对排序好的数组进行二分查找法操作

1. **JAVA多态**

多态是同一个行为具有多个不同表现形式或形态的能力

**多态存在的三个必要条件：**

* 继承
* 重写
* 父类引用指向子类对象

多态可以分为两种：

* **重载式多态（编译时多态）**

在调用这种重载的方法时，通过传入不同的参数最后得到不同的结果

* **重写式多态**

在执行时判断所引用对象的实际类型，根据其实际的类型调用其相应的方法

1. **向上转型**

**因为构建类的多态的过程需要类型的转换，所以在这里学习转型概念**

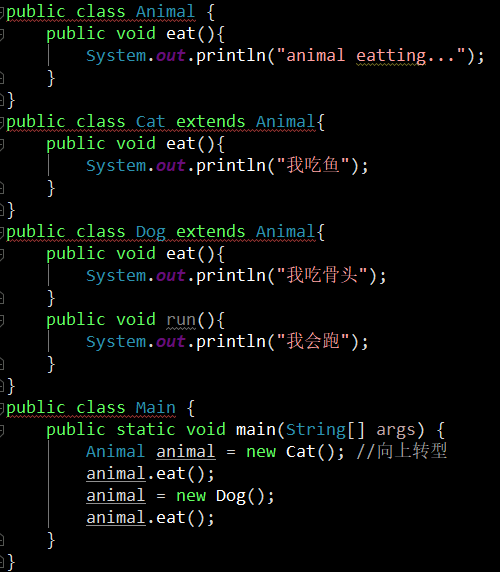
子类引用的对象类型转换为父类类型称为向上转型

既将子类对象转为父类对象，父类对象可以是接口

**格式为：**

**父类对象名 对象名 = new 子类对象名();**

代码实例如下：



**转型过程中需要注意的问题：**

* 向上转型时，子类单独定义的方法会丢失
* 子类引用不能指向父类对象

**向上转型的好处：**

* 减少重复代码，使代码变得简洁
* 提高系统扩展性

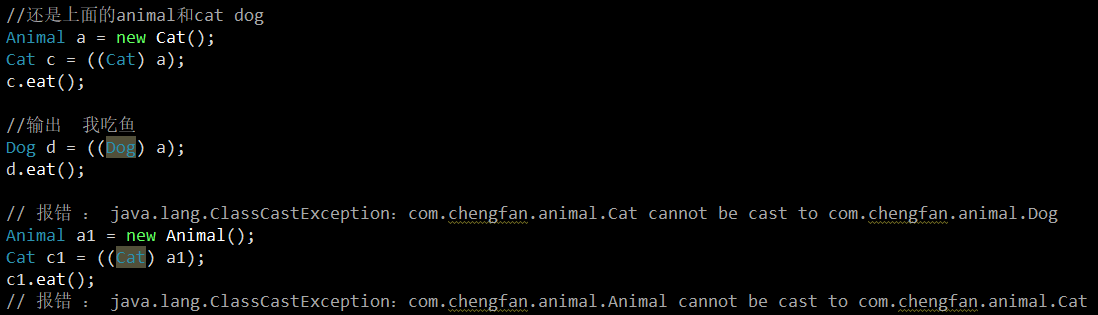
1. **向下转型**

向下转型是通过强制类型转换把父类对象转为子类对象（请注意！这里是有坑）

**强制转换格式为：( type ) value**

其中，type是要转换的类型，value是要转换的对象

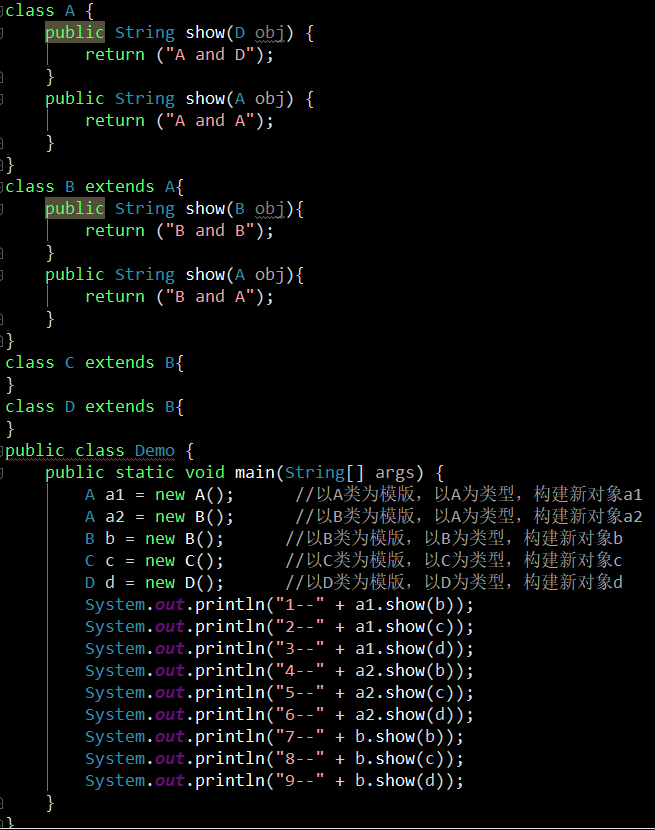
**错误实例如下：**



**向下转型注意事项**

* **向下转型的前提是父类对象指向的是子类对象（向下转型前，得先向上转型）**
* **向下转型只能转型为本类对象（不能跨类别转型，例如不能猫转狗）**

**向下转型经典案例：**



**分析：**

1. Class A中没有show（B），B转向B的父类A，执行A.show（A）
2. Class A中没有show（C），C转向C的父类B，执行A.show（B）

Class A中没有show（B），B转向B的父类A，执行A.show（A）

1. Class A中有show（D），执行A.show（D）
2. **比较特殊的例子**

**以父类声明，子类作实例**

**应当把对象a2当做子类B重写完后的父类A，既a2 = A，但方法为B**

**（以父类声明的对象，当被引用时指向父类，但方法被重写，再使用子类方法）**

**a2指向父类A，由于A中没有B，B转向父类A**

**因为父类A中有A，执行方法时使用子类方法，既执行B.show（A）**

1. 以a2为对象的过程，都如4过程来执行
2. 如上同理
3. Class B中有show（B），执行A.show（B）
4. Class B中没有show（C），C转向C的父类B，执行A.show（B）

关于以上过程4的原理说明：

* **当父类对象的声明变量引用子类对象时，被引用对象的类型决定了调用谁的成员方法，引用变量类型决定可调用的方法**
* **如果子类中没有覆盖该方法，那么会去父类中寻找**

1. **转型的必要性**

多态需要继承和重写两大特性来完成

* 因为子类不继承父类的构造器（构造方法或构造函数）

所以在继承过程中，需要用父类来构造对象，既用到了向上转型

**FatherClass a = new SonClass()**

* 因为子类重写过程中添加了新方法

所以在调用过程中，需要用强制转换来调用对象，既用到了向下转型

**SonClass b = ( SonClass ) a**

1. **继承链中对象方法的调用的优先级：**

this.show(O)==>super.show(O) ==>this.show((super)O) ==>super.show((super)O)

1. **封装**

封装（Encapsulation）是指将抽象性函数接口的实现细节部份包装、隐藏起来的方法

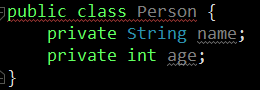
封装防止该类的代码和数据被外部类定义的代码随机访问

封装的优点：

* 良好的封装能减少耦合
* 类内部的结构可以自由修改
* 可以对成员变量进行更精确的控制
* 隐藏信息，实现细节

1. **实现封装的步骤**
2. **修改属性的可见性来限制对属性的访问（一般限制为private）**

例如：



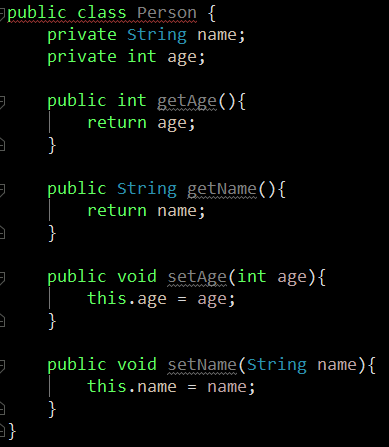
name和age属性设置为私有的，只能本类才能访问，其他类都访问不了

1. **对每个值属性提供对外的公共方法访问，既创建一对赋取值方法**

**（通过建立getter函数的方法，调用返回值来访问私有属性）**

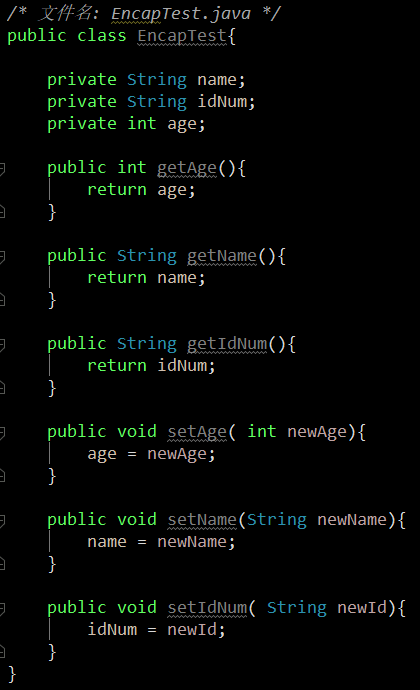
**用于对私有属性的访问**

例如：



采用this关键字是为了解决实例变量和局部变量之间发生的同名的冲突

1. **封装类实例**

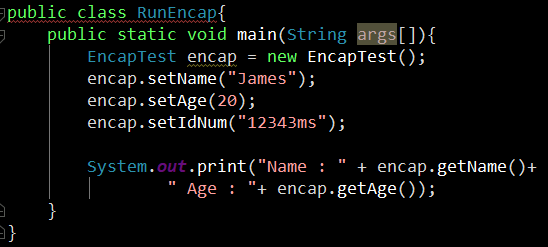


**以上实例中public方法是外部类访问该类成员变量的入口**

**通常情况下，这些方法被称为getter和setter方法**

因此，任何**要访问类中私有成员变量的类都要通过getter和setter方法**

如下为访问实例：



1. **抽象类**

在学习接口之前，需要掌握抽象类的概念，在此进行学习

在面向对象的概念中：

* 所有的对象都是通过类来描绘的
* 并不是所有的类都是用来描绘对象的

如果一个类中**没有包含足够的信息来描绘一个具体的对象**，这样的类就是抽象类

**抽象类特点：**

* 抽象类除了不能实例化对象之外，类的其它功能依然存在

**成员变量、成员方法和构造方法的访问方式和普通类一样**

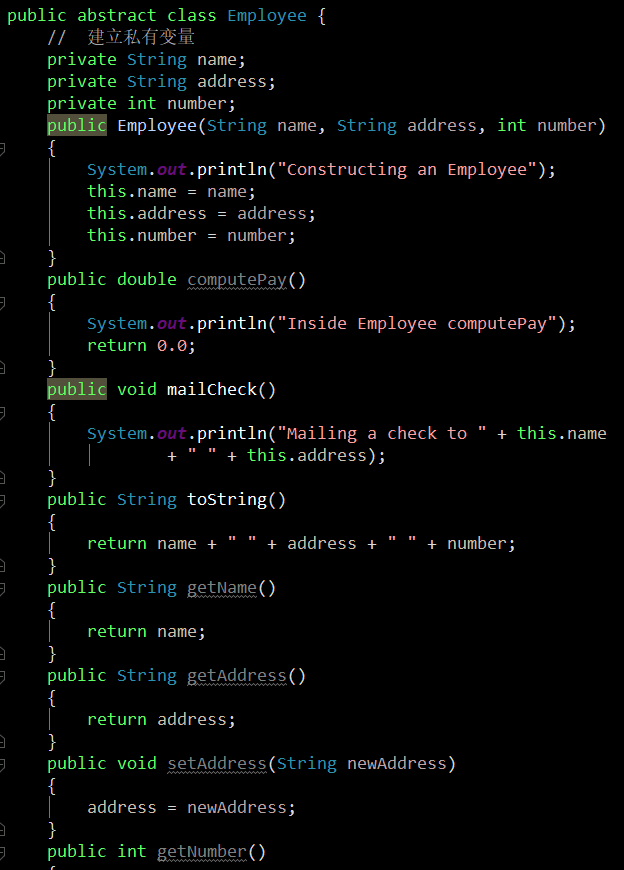
* 由于抽象类不能实例化对象，所以抽象类必须被继承，才能被使用
* **在Java中抽象类表示的是一种继承关系**

一个类只能继承一个抽象类，而一个类却可以实现多个接口

1. **抽象类**

在Java语言中使用abstract class来定义抽象类

实例如下



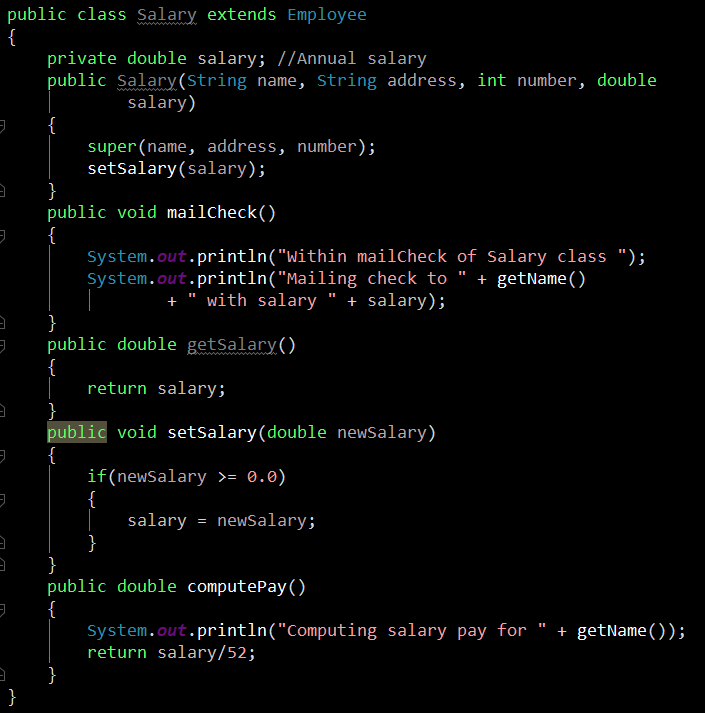
声明了一个抽象类，带有三个变量，一个构造函数

但是该抽象类Employee不能构建对象

1. **构建抽象类**

通过一般的方法继承Employee抽象类

实例如下：

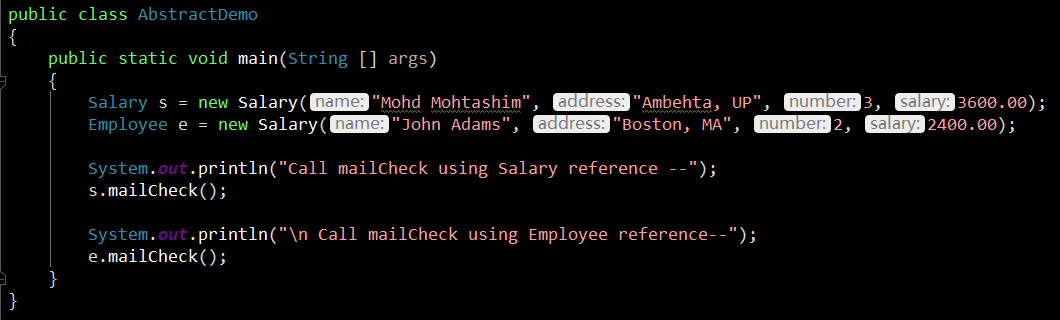


**尽管不能用父类Employee实例化一个对象**

**但是可以用子类Salary实例化一个对象**

* 该对象将从Employee类继承7个成员方法
* 且通过该方法可以设置或获取三个成员变量

实例如下：

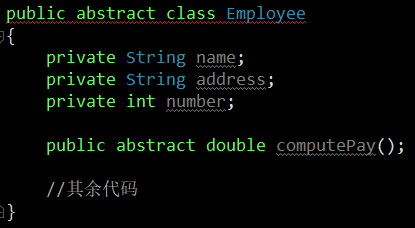


1. **抽象方法**

Abstract 关键字同样可以用来声明抽象方法

**抽象方法只包含一个方法名，而没有方法体，以分号结尾，没有花括号**

实例如下：



**声明抽象方法会造成以下两个结果：**

* 如果一个类包含抽象方法，那么该类必须是抽象类
* 任何子类必须重写父类的抽象方法，或者声明自身为抽象类

**抽象方法的要求：**

* 首先，继承抽象方法的子类必须重写该方法

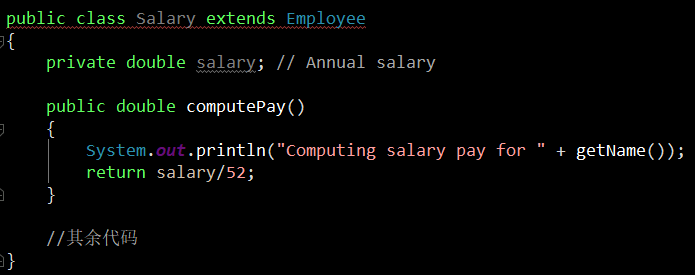
否则，该子类也必须声明为抽象类

* 最终，必须有子类实现该抽象方法，

否则，从最初的父类到最终的子类都不能用来实例化对象

实例如下：

**如果Salary类继承了Employee类，那么Salary类必须实现computePay()方法**



**抽象类总结：**

* 抽象类不能被实例化，只有抽象类的非抽象子类可以创建对象
* 抽象类中不一定包含抽象方法，但是有抽象方法的类必定是抽象类
* 抽象类中的抽象方法只是声明，不包含方法体
* 构造方法，类方法（用static修饰的方法）不能声明为抽象方法
* 抽象类的子类必须给出抽象类中的抽象方法的具体实现，除非该子类也是抽象类

**明天计划完成的事情：**

今天完成了抽象类的学习，明天计划完成接口的知识学习。

至此JAVA基本只是概念就将学习完毕，明天开始完成修真任务，并且开始了解Spring框架，部署Spring框架的构件。

**遇到的问题：**

学到JAVA的多态时，转型的概念很难理解，特别是通过代码来理解时，牵扯到一些知识点需要重新回顾。

转型的概念：

* **向上转型包含了继承和重写的相关细节：**

**父类声明的对象在引用时指向父类，但是执行方法使用子类对象的执行方法**

* **向下转型包含了强制类型转换的概念：**

**使父类声明的对象类型转换成子类类型，但要注意数据类型的范围**

**收获：**

通过两天对面向对象各概念的学习，了解了继承多态抽象的概念，明天完成接口和包的学习后，JAVA基础知识即可告一段落，开始整合Spring框架的知识，重新阅读框架代码，理解框架作用，并对框架构建进行部署。