# 【OOP：面向对象编程\_第二天】

## 主要内容

1. static变量
2. static方法
3. static代码块
4. package和import
5. 封装性
6. 继承性
7. 方法重写

|  |  |
| --- | --- |
| 知识点 | 要求 |
| static变量 | 掌握 |
| static方法 | 掌握 |
| static代码块 | 掌握 |
| package包 | 掌握 |
| import导入 | 掌握 |
| 使用文档注释生成API | 掌握 |
| 封装的引入和作用 | 掌握 |
| 权限修饰符 | 掌握 |
| 继承的引入和作用 | 掌握 |
| 方法重写 | 掌握 |

## 学习目标

## 第一节 static关键字

static是Java中的一个关键字，单词本身是静态的含义。一个类的成员包括变量、方法、构造方法、代码块和内部类，static可以修饰除了构造方法以外的所有成员。

使用static修饰的成员成为静态成员，是属于某个类的；而不使用static修饰的成员成为实例成员，是属于类的每个对象的。

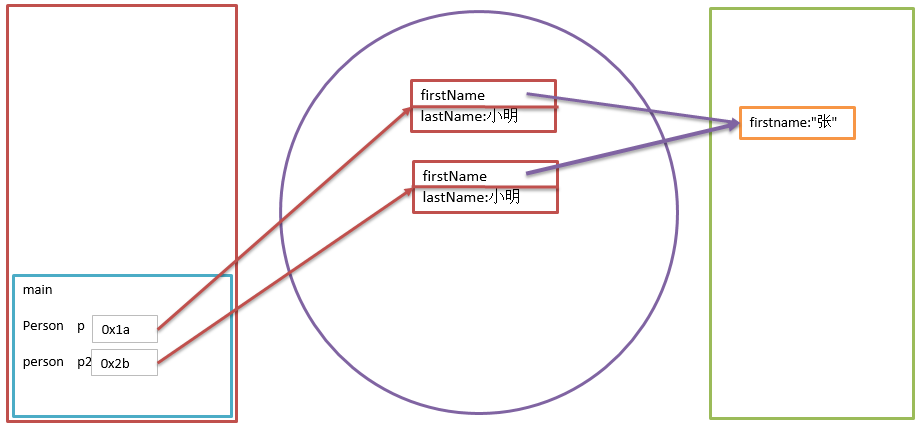
### 1.1 static变量

在类中，用static声明的成员变量为静态成员变量，也称为类变量。 类变量的生命周期和类相同，在整个应用程序执行期间都有效。它有如下特点：

* 为该类的公用变量，属于类，被该类的共享，在类被载入时被显式初始化。
* 对于该类的所有对象来说，static成员变量只有一份。被该类的所有对象共享！！
* 一般用“类名.类属性/方法”来调用。（也可以通过对象引用或类名（不需要实例化）访问静态成员。）
* 在static方法中不可直接访问非static的成员。

#### 【示例1】static变量

|  |
| --- |
| **package** com.bjsxt.staticDemo;  **public class** Person {  */\*  \* static修饰的成员变量叫做静态成员变量  \* 静态成员变量存储在方法区中  \* 静态成员变量被当前类对象共享  \* 静态成员变量可以使用对象名.属性名访问  \* 静态成员推荐使用类名.属性名的方式访问  \*  \* 没有static修饰的成员变量叫做实例成员变量  \* 实例成员变量存储在堆内存上  \* 实例成员变量是每个对象独有的  \* 实例成员变量只能有对象名.属性名的形式访问  \*  \* \*/   // 姓 名属性* **static** String *firstName*;  String **lastName**;  *// 显示姓名的方法* **public void** showName(){  System.***out***.println(*firstName*+**lastName**);  } } |



内存分配图如下：

**总结：static变量和非static变量的区别：**

1. 份数不同：静态变量：1份；非静态变量：1个对象一份
2. 存储位置不同：静态变量：方法区；非静态变量：堆中
3. 内存分配空间的时间不同：静态变量：第一次加载类的时候；非静态变量：创建对象的时候
4. 生命周期不同。静态变量和类的生命周期相同；非静态变量的生命周期和所属对象相同
5. 调用方式不同
   * 静态变量： 通过类名调用 Student.classRoom

也可以通过对象名调用stu1.classRoom ="301" 不推荐

* + 非静态变量：通过对象名调用 stu1.name ="小张";

### 1.2 static方法

* static方法的作用

访问static变量和static方法

* static方法的调用方式

通过类名调用 Student.showClassRoom(); 推荐该方式

通过对象名访问 stu1.showClassRoom();

* 不可以

静态方法中不可以访问非静态变量

静态方法中不可以访问非静态方法

静态方法中不可以访问this

理解：加载类的时候就加载静态变量和静态方法，此时可能还没有创建对象，所以非静态变量和非静态的方法还没有分配空间，无法访问

* 可以

非静态方法中可以访问静态变量

非静态方法中可以访问静态方法

理解：加载类的时候就已经加载静态变量和静态方法，创建对象后，非静态变量和非静态的方法才分配空间，此时静态变量和静态方法已经存在，可以访问

#### 【示例2】static方法

|  |
| --- |
| **package** com.bjsxt.staticDemo;  **public class** Person {  */\*  \* static修饰的方法叫做静态成员方法  \* 静态成员方法中只能直接使用静态成员变量  \* 静态成方法可以使用对象名.方法名方式调用  \* 静态成员方法推荐使用类名.方法名方式调用  \* 静态成员方法中不能使用this关键  \* 静态成员方法只能直接调用其他静态成员方法  \* 在构造方法中,往往不会对静态成员变量进行初始化  \*  \* 没有static修饰的方法叫做实例成员方法  \* 实例成员方法中既可以直接使用静态成员变量 也可以直接使用实例成员变量  \* 实例成方法只能使用对象名.方法名方式调用  \* 实例成员方法中可以使用this关键  \* 实例成员方法既能直接调用其他静态成员方法,也可以直接调用其他实例方法  \* 在构造方法中,往往只会对实例成员变量进行初始化  \* \*/   // 姓 名属性* **static** String *firstName*;  String **lastName**;  *// 显示姓名的方法* **public void** showName(){  System.***out***.println(*firstName*+**this**.**lastName**);  *methodA*();  methodB();  }   **public static void** viewName(){   System.***out***.println(*firstName/\*+lastName\*/*);  *methodA*();  *//methodB();* }   **public static void** methodA(){}  **public void** methodB(){}   **public** Person(String lastName) {  **this**.**lastName** = lastName;  }  **public** Person(){} } |

### 1.3 static代码块

* 总结1：(成员)代码块

1. 位置：类中
2. 数量：多个
3. 执行顺序：依次执行
4. 执行时间：每次创建对象的时候都执行;先执行代码块，再执行构造方法
5. 作用：实际开发中很少用； 可以将各个构造方法中公共的代码提取到代码块；**匿名内部类不能提供构造方法，此时初始化操作放到代码块中**

* 总结2：static代码块

1. 位置：类中
2. 数量：多个
3. 执行顺序：依次执行
4. 执行时间：第一次加载类的时候执行，只执行一次
5. 作用：给静态变量赋初始值。**实际开发中使用比较多，一般用于执行一些全局性的初始化操作**，比如创建工厂、加载数据库初始信息

#### 【示例3】static代码块

|  |
| --- |
| **public class** Test1 {  **public static void** main(String[] args) {  Person p =**new** Person();  p.showName();  Person p2 =**new** Person();   }   }  */\* \* 类的成员 \* 三大成员 \* 成员变量 \* 成员方法 \* 构造方法 \* 其他成员 \* 代码块 \* 内部类 \* \* 代码块 \* 普通代码块 一般用于帮助我们初始化一些实例成员变量 \* 在一个类中可以有多个普通代码块 \* 每次执行构造方法之前,普通代码块都会执行一次 每实例化对象一次,普通代码块都会执行一次 \* \* 静态代码块 一般用于初始化静态成员变量的 \* 在一个类中可以有多个静态代码块 \* 类加载进入内存的之后,静态代码块执行一次 \* \* \*/* **class** Person{  *// 成员变量* **static** String *firstName*;  String **lastName** ;  *// 成员方法* **public void** showName(){  System.***out***.println(*firstName*+**lastName**);  }   *// 构造方法 一般用于初始化当前对象本身相关实例的成员变量,一般不用于初始化静态成员变量* **public** Person (){  System.***out***.println(**"Person 无参构造方法"**);  }  **public** Person(String lastName){  System.***out***.println(**"Person 有参构造方法"**);  **this**.**lastName**=lastName;  }  *// 代码块 代码块不能被调用,代码块是在指定的环节自动执行的* {  System.***out***.println(**"代码块A"**);  **lastName**=**"小明"**;  }  {  System.***out***.println(**"代码块B"**);  }  {  System.***out***.println(**"代码块C"**);  }   **static**{  System.***out***.println(**"Person 类静态代码块A"**);  *firstName*=**"张"**;    }  **static**{  System.***out***.println(**"Person 类静态代码块B"**);  }  } |

### 1.4 static的优点和缺点

优点：不需要在进行实例化。静态变量的值，直接赋新值即可，不需要参数传递，之后可以直接进行参数引用即可；静态方法可以直接通过"类名.方法"的形式进行方法调用。通常方法被多次调用，并且方法中没有动态方法引用的时候使用比较方便。

缺点：初始化加载到内存，如果后续没被引用，加大了内存负担和程序运行负担，影响程序运行效率（一般很小），并且静态变量如果多处被引用赋值，很可能导致参数值混乱

如果从线程安全、性能、兼容性上来看  也是选用实例化方法为宜。

### 本节作业

1. static变量和非static变量的区别和联系
2. static方法为什么不可以访问非static的变量和方法
3. static代码块的执行时机和执行次数

## 第二节 package和import

### 2.1 package包

* **为什么使用包**

文件太多，并且会有同名文件，计算机的硬盘需要不同级别的文件夹来存储；

包机制是Java中管理类的重要手段。开发中，我们会遇到大量同名的类，通过包我们很容易对解决类重名的问题，也可以实现对类的有效管理。**除了以上考虑外，还和访问权限有密切关系**。

* **如何定义包**

我们通过package实现对类的管理，package的使用有两个要点：

1. 包名：域名倒着写即可，再加上模块名，便于内部管理类。
2. 包名字母一律小写。

|  |
| --- |
| com.bjsxt.oop.object  cn.com.sina.mapper....  com.bjsxt.stumgr.dao  com.bjsxt.stumgr.dao.impl |

* **如何使用包**

通常是类的第一句非注释性语句。

必须以；结尾。

### 2.2 import导入

如果我们要使用其他包的类，需要使用import导入，从而可以在本类中直接通过类名来调用，否则就需要书写类的完整包名和类名。import后，便于编写代码，提高可维护性。

|  |
| --- |
| **注意要点**   * 默认是当前包的类和接口 * Java会默认导入java.lang包下所有的类，因此这些类我们可以直接使用。 * 可以使用通配符，比如import com.bjsxt.oop.object.\*; 会导入该包下所有类和接口（但不包括下级包） * 如果导入两个同名的类，只能用包名+类名来显示调用相关类：   java.util.Date date = new java.util.Date(); |

静态导入(static import)是在JDK1.5新增加的功能，其作用是用于导入指定类的静态属性和静态方法，这样我们可以直接使用静态属性和静态方法。

#### 【示例4】导入和静态导入

|  |
| --- |
| // 导入另一个类中所有的静态成员 **import static** java.lang.Math.\*;  **public class** Test3 {  **public static void** main(String[] args) {  **double** r=*random*();  System.***out***.println(***PI***);  System.***out***.println(*pow*(2,3));  System.***out***.println(*sqrt*(4));  System.***out***.println(*abs*(-4));  System.***out***.println(*max*(-4,5));  System.***out***.println(*min*(-4,5));   } } |

* **Java常用包**

|  |  |
| --- | --- |
|  | |
| **Java中的常用包** | **说明** |
| java.lang | 包含一些Java语言的核心类，如String、Math、Integer、System和Thread，提供常用功能。是JAVA中,唯一不需要import就自动导入的包 |
| java.awt | 包含了构成抽象窗口工具集（abstract window toolkits）的多个类，这些类被用来构建和管理应用程序的图形用户界面(GUI)。 |
| java.net | 包含执行与网络相关的操作的类。 |
| java.io | 包含能提供多种输入/输出功能的类。 |
| java.util | 包含一些实用工具类，如定义系统特性、使用与日期日历相关的函数。 |

|  |
| --- |
| **注意事项**   * 写项目时都要加包，不要使用默认包。 * com.gao和com.gao.car，这两个包没有包含关系，是两个完全独立的包。只是逻辑上看起来后者是前者的一部分。 |

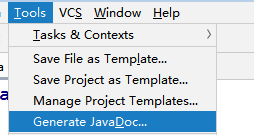
### 2.3 使用文档注释生成API文档

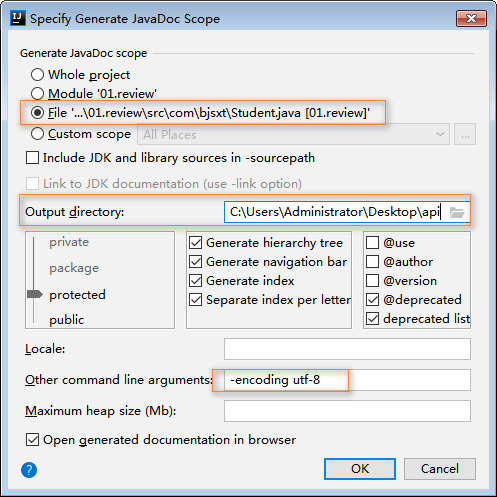
IntelliJ IDEA 本身提供了很好的 JavaDoc 生成功能，以及标准 JavaDoc 注释转换功能，其实质是在代码编写过程中，按照标准 JavaDoc 的注释要求，为需要暴露给使用者的类、方法以及其他成员编写注释。然后使用 IDEA 的功能自动调用 javadoc.exe（JDK 自带的工具）根据源代码中的注释内容自动生成 JavaDoc 文档（超文本格式）。

#### 【示例5】使用文档注释

|  |
| --- |
| **package** com.bjsxt.testc;  */\*\*  \* Person 人类,定义了一些人类通常有的属性和功能  \*/* **public class** Person {  */\*\*  \* 姓名属性  \*/* String **name**;  */\*\*  \* 性别属性  \*/* String **gender**;  */\*\*  \* 年龄属性  \*/* **int age**;   */\*\*  \* 模拟人类吃饭的方法  \** ***@param food*** *食物名称  \*/* **public void** eat(String food){   }   */\*\*  \* 模拟人类进行加法运算的功能  \** ***@param a*** *加法中的第一个整数  \** ***@param b*** *加法中的第二个整数  \** ***@return*** *两个整数的和  \*/* **public static int** getSum(**int** a,**int** b){  **return** a+b;  }   */\*\*  \* 无参数构造器  \*/* **public** Person(){   }   */\*\*  \* 该构造器可以帮助初始化 name属性和gender属性  \** ***@param name*** *姓名  \** ***@param gender*** *性别  \*/* **public** Person(String name,String gender){  this.name=name;  this.gender=gender;  } } |

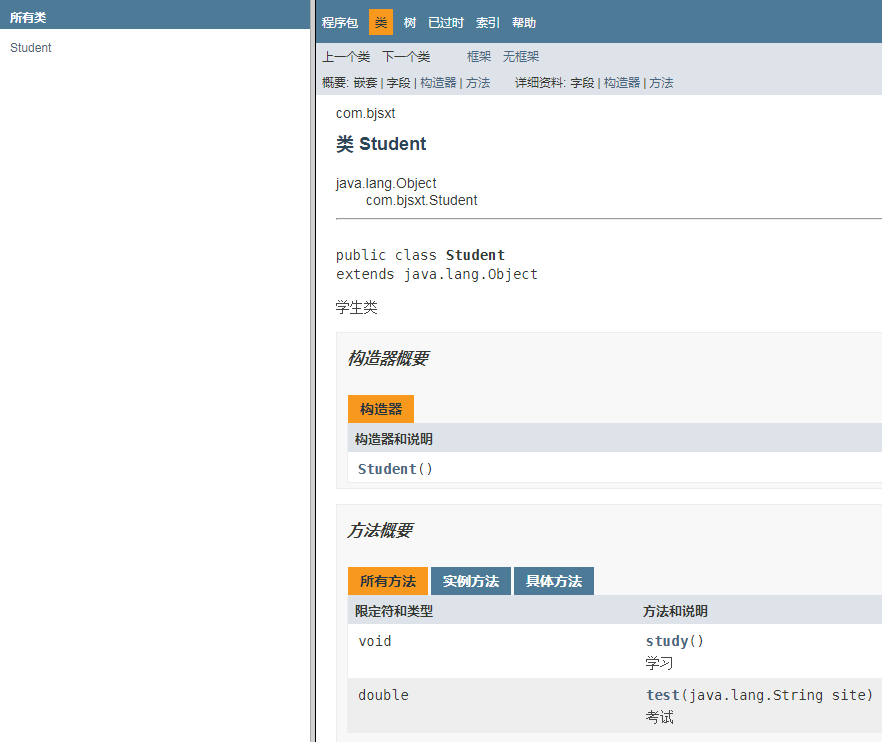
在idea中主要操作步骤如下：





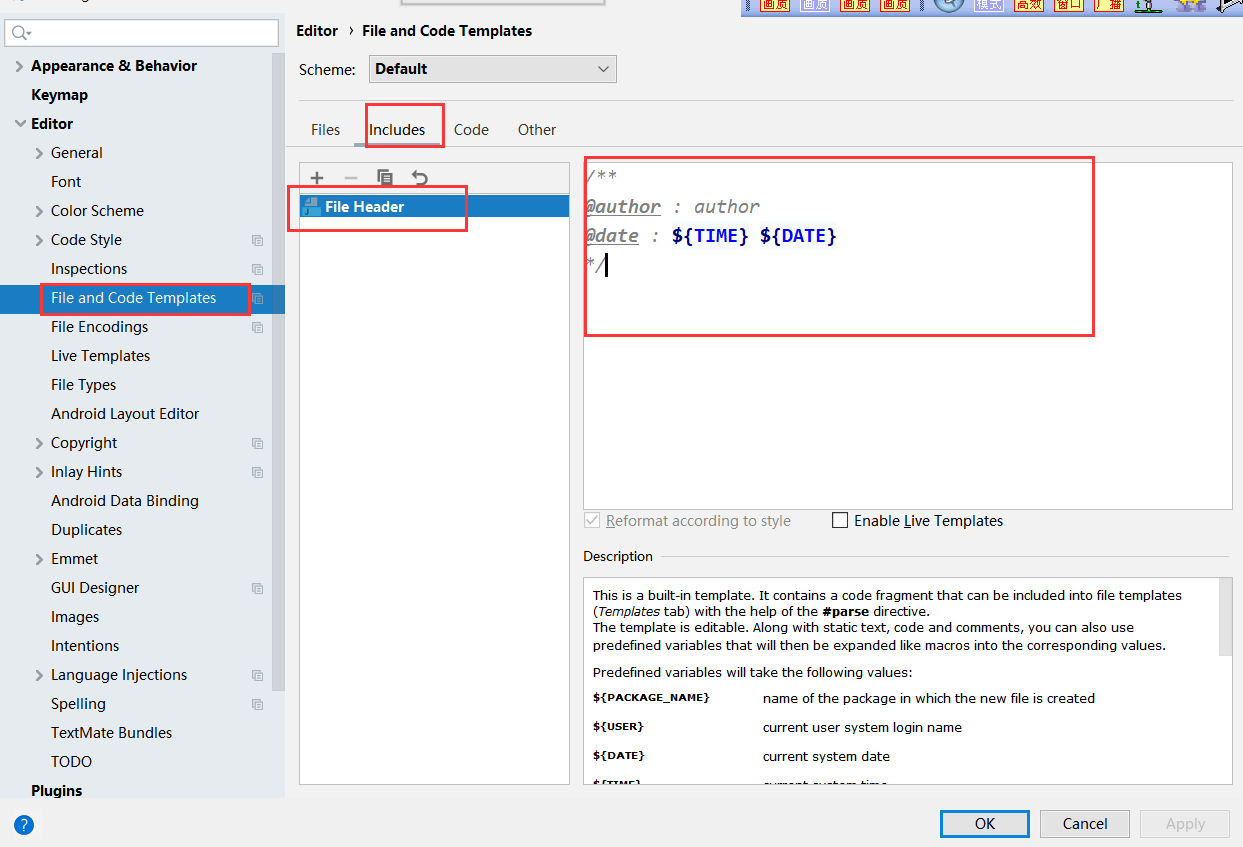
指定生成文档的编码:-encoding utf-8

最终可以生成如图所示的API帮助文档。



新建类的时候添加日期时间

|  |
| --- |
| /\*\* @author : 尚学堂 @date : **${TIME} ${DATE}** \*/ |



### 本节作业

1. 理解package包的作用
2. Java中常用包及其作用
3. import导入的注意事项

## 第三节 封装性

封装（encapsulation）是面向对象三大特征之一。对于程序合理的封装让外部调用更加方便，更加利于写作。同时，对于实现者来说也更加容易修正和改版代码。

面向对象编程,语言的三大特征 封装 继承 多态

面向对象编程,语言的四大特征 封装 继承 多态 抽象

### 3.1 引入封装



需要让用户知道的才暴露出来，不需要让用户知道的全部隐藏起来，这就是封装。说的专业一点，封装就是把对象的属性和操作结合为一个独立的整体，并尽可能隐藏对象的内部实现细节,仅仅对外公开使用的接口/方法。

我们程序设计要追求“高内聚，低耦合”。高内聚就是类的内部数据操作细节自己完成，不允许外部干涉；低耦合是仅暴露少量的方法给外部使用，尽量方便外部调用。

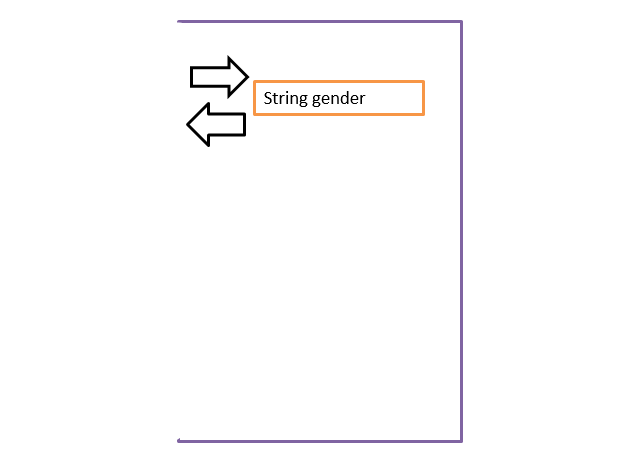
**编程中封装的具体优点：**

* 提高代码的安全性。
* 提高代码的复用性。
* “高内聚”：封装细节，便于修改内部代码，提高可维护性。
* “低耦合”：简化外部调用，便于调用者使用，便于扩展和协作。

#### 【示例6】未进行封装的代码演示

|  |
| --- |
| 对Person类进行封装处理  **public class** Student {  */\*  \* 如何对一个类进行封装处理  \* 1属性私有化  \* 使用private访问修饰符修饰成员变量  \* private 修饰的类的成员只能在当前类内部使用  \* 2给属性提供公有的外界访问的方法 get set  \* \*/* **private static** String *firstName*;  **private** String **name**;  **private** String **gender**;  **private int age**;  **private double score**;  **private boolean pass**;  *// 静态的属性,其对应的get和set方法也应该是静态的方法* **public static void** setFirstName(String firstName){  Student.*firstName*=firstName;  }  **public static** String getFirstName(){  **return** *firstName*;  }  *// 如果属性类型为boolean类型,其对应的get方法叫做 is... set方法仍然是set... :* **public void** setPass(**boolean** pass){  **this**.**pass**=pass;  }  **public boolean** isPass(){  **return pass**;  }   *// 定义一个专门给name属性赋值的方法* **public void** setName(String name){  **this**.**name**=name;  }  **public** String getName(){  **return name**;  }   **public void** setGender(String gender){  *// 如果是第一次则允许赋值* **if**(**null** == **this**.**gender**){  **this**.**gender**=gender;  }  }  **public** String getGender(){  **return gender**;  }  **public void** setAge(**int** age){  **if**(age >=1 && age <=100){  **this**.**age**=age;  }  }  **public int** getAge(){  **return age**;  }  **public void** setScore(**double** score){  **this**.**score**=score;  }  **public double** getScore(){  **return score**;  }  **public void** showInfo(){  System.***out***.println(**"姓名:"**+**name**+**" 性别:"**+**gender**+**" 年龄:"**+**age**+**" 分数:"**+**score**);  }  } |

我们都知道，性别不是男就是女,不可能是其他值，但是如果没有使用封装的话，便可以给性别赋为任何值，这显然不符合我们的正常逻辑思维。



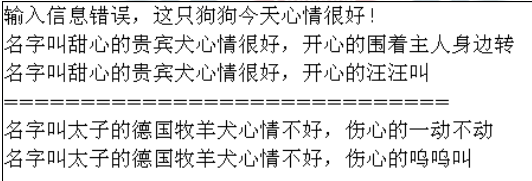
### 3.3 封装练习

以面向对象的思想，编写自定义类描述图书信息。设定属性包括：书名，作者，出版社名，价格；方法包括：信息介绍  
要求：  
1) 设置属性的私有访问权限，通过公有的get,set方法实现对属性的访问  
2) 限定价格必须大于10，如果无效进行提示  
3) 设计构造方法实现对属性赋值  
4) 信息介绍方法描述图书所有信息  
5) 编写测试类，测试图书类的对象及相关方法（测试数据信息自定）

### 本节作业

1. 使用面向对象的思想，编写自定义描述狗的信息。设定属性包括：品种，年龄，心情，名字；方法包括：叫，跑。要求：
2. 设置属性的私有访问权限，通过公有的get,set方法实现对属性的访问
3. 限定心情只能有“心情好”和“心情不好”两种情况，如果无效输入进行提示，默认设置“心情好”。
4. 设置构造函数实现对属性赋值
5. 叫和跑的方法，需要根据心情好坏，描述不同的行为方式。
6. 编写测试类，测试狗类的对象及相关方法（测试数据信息自定义）

运行效果图：



## 第四节 继承性

继承（Inheritance）是面向对象编程的三大特征之一，它让我们更加容易实现对于已有类的扩展、更加容易实现对于现实世界的建模。

### 4.1 继承及其作用

继承让我们更加容易实现类的扩展。 比如，我们定义了人类，再定义Boy类就只需要扩展人类即可。实现了代码的重用，不用再重新发明轮子(don’t reinvent wheels)。

从英文字面意思理解， **extends**的意思是“扩展”。子类是父类的扩展。现实世界中的继承无处不在。比如：

|  |
| --- |
| 图5-1 |
| **图5-1 现实世界中的继承** |

上图中，哺乳动物继承了动物。意味着，动物的特性，哺乳动物都有；在我们编程中，如果新定义一个Student类，发现已经有Person类包含了我们需要的属性和方法，那么Student类只需要继承Person类即可拥有Person类的属性和方法。

#### 【示例9】实现继承

**继承下的封装处理**

定义父类

|  |
| --- |
| **package** com.bjsxt.extendDemo;  **public class** Car {  */\*  private 修饰的成员不可以被继承,仅能在当前类内部使用  \*/* **private** String **brand**;  **private** String **type**;  **private** String **color**;  **private int doornum**;   **public** String getBrand() {  **return brand**;  }   **public void** setBrand(String brand) {  **this**.**brand** = brand;  }   **public** String getType() {  **return type**;  }   **public void** setType(String type) {  **this**.**type** = type;  }   **public** String getColor() {  **return color**;  }   **public void** setColor(String color) {  **this**.**color** = color;  }   **public int** getDoornum() {  **return doornum**;  }   **public void** setDoornum(**int** doornum) {  **this**.**doornum** = doornum;  }   *// 具有 启动 加速 停止的功能* **public void** start(){  System.***out***.println(**"一辆"**+**doornum**+**"开门的,"**+**color**+**"颜色的"**+**brand**+**type**+**"正在启动...."**);  }  **public void** speedUp(){  System.***out***.println(**"一辆"**+**doornum**+**"开门的,"**+**color**+**"颜色的"**+**brand**+**type**+**"正在加速...."**);  }  **public void** speedDown(){  System.***out***.println(**"一辆"**+**doornum**+**"开门的,"**+**color**+**"颜色的"**+**brand**+**type**+**"正在停止...."**);  }   **public** Car(){   }  **public** Car(String brand, String type, String color, **int** doornum){  **this**.**brand**=brand;  **this**.**type** =type;  **this**.**color**=color;  **this**.**doornum**=doornum;  } } |

定义子类

|  |
| --- |
| **package** com.bjsxt.extendDemo;  **public class** Audi **extends** Car {  **private** String **level**;   **public** String getLevel() {  **return level**;  }   **public void** setLevel(String level) {  **this**.**level** = level;  }   **public** Audi(String level) {  **this**.**level** = level;  }   **public** Audi() {  }   **public** Audi(String brand, String type, String color, **int** doornum, String level) {  **super**(brand, type, color, doornum);  **this**.**level** = level;  }   **public void** gps(){  System.***out***.println(**"一辆"**+getDoornum()+**"开门的,"**+getColor()+**"颜色的"**+getBrand()+getType()+**level**+**"级的轿车开启了导航"**);  } }  **package** com.bjsxt.extendDemo;  **public class** BMW **extends** Car {  **private double price**;   **public double** getPrice() {  **return price**;  }   **public void** setPrice(**double** price) {  **this**.**price** = price;  }   **public** BMW(**double** price) {  **this**.**price** = price;  }   **public** BMW() {  }   **public void** startAC(){  System.***out***.println(**"一辆"**+**price**+**"万元的"**+getDoornum()+**"开门的,"**+getColor()+**"颜色的"**+getBrand()+getType()+**"的轿车开启了空调:38.6"**);  } } |

测试代码

|  |
| --- |
| **package** com.bjsxt.extendDemo;  **public class** Test1 {  **public static void** main(String[] args) {  *// 创建父类对象* Car car=**new** Car();  car.setBrand(**"大众"**);  car.setColor(**"银灰"**);  car.setDoornum(4);  car.setType(**"捷达"**);   car.start();  car.speedUp();  car.speedDown();   *// 实例化子类对象* Audi audi=**new** Audi();  audi.setBrand(**"奥迪"**);  audi.setColor(**"黑"**);  audi.setDoornum(4);  audi.setType(**"A6"**);  audi.setLevel(**"C"**);   audi.start();  audi.speedUp();  audi.speedDown();  audi.gps();   BMW bmw=**new** BMW();  bmw.setBrand(**"宝马"**);  bmw.setColor(**"白"**);  bmw.setDoornum(4);  bmw.setType(**"X5"**);  bmw.setPrice(50.36);;   bmw.start();  bmw.speedUp();  bmw.speedDown();  bmw.startAC();   } } |

**继承使用要点**

1. 父类也称作超类、基类。子类：派生类等。
2. Java中只有单继承，没有像C++那样的多继承。多继承会引起混乱，使得继承链过于复杂，系统难于维护。
3. **子类继承父类，可以得到父类的全部属性和方法 (除了父类的构造方法)，但不见得可以直接访问(比如，父类私有的属性和方法)。**
4. 如果定义一个类时，没有调用extends，则它的父类是：java.lang.Object。

### 4.2 方法重写

父类的方法introduce()已经无法满足子类的需求，怎么办？同理，Object类的toString()已经无法满足Animal类、Dog类的需求，怎么办？可通过方法重写（override）解决，或者称为方法覆盖。

#### 【示例8】实现方法重写

|  |
| --- |
| **package** com.bjsxt.extendsDemo3;  **public class** Test1 {  **public static void** main(String[] args) {  B b=**new** B();  b.showName();   } }  **class** A{  String **name**=**"父类中的name属性"**;  **public void** showName(){  System.***out***.println(**name**);  }  *// 升序排列一个数组的方法* **public void** sort(**int**[] arr){  *//冒泡* }  }  **class** B **extends** A{  */\*  \* 子类中如果定义的和父类中同名的属性,则父类中的属性默认隐藏  \* 如果要在子类中继续使用父类中被隐藏的同名属性,需要用super关键字  \* \*/* String **name**=**"子类中的name属性"**;  **public void** viewName(){  String name=**"子类方法中的name值"**;  System.***out***.println(**super**.**name**);  System.***out***.println(**this**.**name**);  System.***out***.println(name);  }   */\*  \* 子类中对父类同名方法的再次定义就是方法的重写  \* 当子类重写父类方法时,从父类继承的同名方法就会被隐藏  \* 如果向调用父类中被隐藏的同名方法 需要super关键字  \* super关键字代表父类对象  \* 方法重写的必要性  \* 1子类和父类完成同样的功能,但是可能采取不同的算法/方式  \* 2子类要在父类原有的功能之上进行功能的扩展,子类的方法中要做到更多的事  \*   \*  \* 方法重写的要求  \* 1方法名必须相同  \* 2参数列表必须相同, 如果不同就不是重写*  *\* 3返回值类型小于等于  \* 4访问修饰符子类重写的方法不能小于父类中的方法  \* \*/* **public void** showName(){  System.***out***.println(**"子类中定义的showName方法"**);  */\* super.showName();\*/* }   **public void** sort(**int**[] arr){  *// 快排* }  } |

面试题:对比方法重载和方法重写

|  |  |
| --- | --- |
| 总 | 方法重载和方法重写（覆盖）是面向对象中两个重要概念，其实这两个概念之间没有什么关系，但是毕竟都是关于方法的，毕竟容易引起混淆。对此我也做了一些归纳，感觉能够把这两个概念很好的区分开。我打算从总体区别、细节区别两个方面来说明。 |
| 分 | 总体的区别：最主要的区别，是解决的问题不同，即作用不同。   |  |  |  |  | | --- | --- | --- | --- | |  | 英文 | 位置不同 | 作用不同 | | 重载 | overload | 同一个类中 | 在一个类里面为一种行为提供多种实现方式并提高可读性 | | 重写 | override | 子类和父类间 | 父类方法无法满足子类的要求，子类通过方法重写满足要求 |   细节的区别：一个方法的声明自左向右包括权限修饰符、方法返回值、方法名、参数列表、抛出的异常类型等。下面从这几方面说明区别   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | 修饰符 | 返回值 | 方法名 | 参数 | 抛出异常 | | 重载 | 无关 | 无关 | 相同 | 不同 | 无关 | | 重写 | 大于等于 | 小于等于 | 相同 | 相同 | 小于等于 | |
| 总 | 重载实例：构造方法重载、println()方法重载  重写实例：Object类的toString()、equals()、hashCode()等都可以被子类重写 |
| 可选 | 1. 某些方法使用final修饰，将无法被重写。比如Object类的wait()、notify()等 2. 静态方法无法进行方法重写。在Java中，如果父类中含有一个静态方法，且在子类中也含有一个返回类型、方法名、参数列表均与之相同的静态方法，那么该子类实际上只是将父类中的该同名方法进行了隐藏，而非重写。添加@override注解将会报错。 |

### 4.3 权限修饰符

Java是使用“访问控制符”来控制哪些细节需要封装，哪些细节需要暴露的。 Java中4种“访问控制符”分别为private、默认、protected、public，它们说明了面向对象的封装性，所以我们要利用它们尽可能的让访问权限降到最低，从而提高安全性。

下面详细讲述它们的访问权限问题。其访问权限范围如表所示。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **访问权限修饰符** | | | | |
| **修饰符** | **同一个类** | **同一个包中(同包可继承)** | **子类** | **所有包的所有类** |
| private | **\*** |  |  |  |
| 默认/defalut | **\*** | **\*** |  |  |
| protected | **\*** | **\*** | **\*** |  |
| public | **\*** | **\*** | **\*** | **\*** |

1. private 表示私有，只有自己类能访问
2. default(friendly)表示没有修饰符修饰，只有同一个包的类能访问
3. protected表示可以被同一个包的类以及其他包中的子类访问
4. public表示可以被该项目的所有包中的所有类访问

**类的成员变量的处理:**

* 一般使用private访问权限修饰成员变量。
* 提供相应的get/set方法来访问相关属性，这些方法通常是public修饰的，以提供对属性的赋值与读取操作（注意：boolean变量的get方法是is开头!）。
* 一些只用于本类的辅助性方法可以用private修饰，希望其他类调用的方法用public修饰。

**类的处理:**

* 类只能使用public和默认来修饰
* 默认：当前包
* public：当前项目的所有包
* public类要求类名和文件名相同，一个java文件中至多一个public类

方法

### 本节作业

1. 继承的作用
2. 定义父类Animal、定义Dog、Cat类继承Animal类
3. 方法重写和方法重载的比较
4. 重写Animal、Dog、Cat的toString()