Java进阶1 第2天

**【学习目标】理解、了解、应用、记忆**

通过今天的学习，参训学员能够：（解释的时候说出二级目标的掌握程度）

1. **【应用】抽象类与抽象方法**
2. 【应用】能够独立定义一个抽象方法
3. 【应用】能够独立定义一个抽象类并在抽象类中定义抽象方法,实现该抽象类中抽象方法
4. 【应用】能够利用子类来使用抽象父类中的构造方法
5. **【应用】接口**
6. 【应用】能够自定义一个接口,并实现该接口
7. 【理解】能够阐述接口中成员的默认修饰符
8. 【理解】能够阐述接口的注意事项
9. 【理解】能够阐述接口和抽象类的区别
10. **【应用】多态**
11. 【理解】能够阐述什么是多态的向上类型提升和向下转型
12. 【应用】能够使用instanceof关键字判断对象类型
13. 【应用】能够阐述多态的优点和使用场景
14. **【应用】能够使用static关键字与final关键字**
15. 【理解】阐述static修饰符的作用及使用场景
16. 【理解】阐述静态的访问方式及注意事项
17. 【理解】阐述final修饰类、方法、变量有什么特点
18. **【理解】能够定义并使用包，释义四种权限修饰符使用特点**
19. 【理解】阐述包的概念及定义包的格式
20. 【了解】知道包的全名访问格式
21. 【理解】能够使用import导入其他包中的类
22. 【理解】阐述四种权限修饰符在同包的访问情况，跨包的访问情况，不同包子父类的访问情况
23. **【应用】定义内部类并使用内部类**
24. 【理解】阐述阐述内部类的概念及内部类的分类
25. 【理解】阐述成员内部类定义的格式及成员内部类的使用方式
26. 【理解】阐述局部内部类定义的格式及局部内部类的使用方式
27. 【应用】独立编码创建匿名内部类对象并使用匿名内部类

# 抽象类与抽象方法

## 抽象类与抽象方法引入

抽象类用来描述一种类型应该具备的基本特征与功能，具体如何去完成这些行为由子类通过方法重写来完成，如:

犬科均会吼叫，但属于犬科的狼与狗其吼叫内容不同。所以犬科规定了有吼叫功能，但并不明确吼叫的细节。

吼叫的细节应该由狼与狗这样的犬科子类重写吼叫的方法具体实现。

类似上边犬科中的吼叫功能，并不明确实现细节但又需要声明的方法可以使用抽象方法的方式完成。即抽象方法指只有功能声明，没有功能主体实现的方法。

具有抽象方法的类一定为抽象类。

那么犬科就可以定义为抽象类，吼叫方法为抽象方法，没有方法体。

### 案例代码一

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: AbstractDemo  \* **@Description**: 抽象类的分析  \* **@date** 2017年11月14日 上午10:18:06  \* Company www.igeekhome.com  \*  \* 酒店中员工、经理、服务员、厨师之间的关系是怎样的？你会与一个员工沟通还是会与一个具体职位的服务员或经理等沟通？  \*  \* 员工父类,定义了其他类的共性内容  \* 成员变量:工号,姓名,年龄,工资  \* 成员方法:工作方法(work),父类简单地完成工作方法,声明只要是员工就应该有工作方法的逻辑  \*  \* 经理,服务员,厨师  \*  \* 真正创建对象,使用对象时,我们往往只使用其子类.  \* 对于上边父类中work的这个方法,每个子类都会重写该方法.  \*  \* 通过以上分析发现:  \* 员工的work方法一定会被子类重写为具体的逻辑,此时可以将该方法定义为抽象方法,仅仅声明有该方法,但是没有具体的方法体,该方法称为抽象方法.  \* 员工父类通常不应该创建对象,又包含了抽象方法,则该员工类应该定义为抽象类.  \*/  **public** **class** AbstractDemo {  **public** **static** **void** main(String[] args) {  }  } |

## 抽象类的定义格式

抽象类定义的格式：

abstract在class前修饰类：

abstract class 类名 {

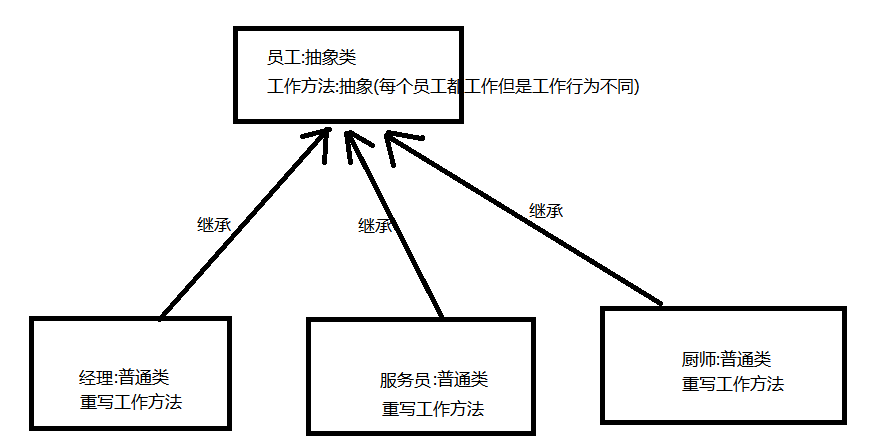
}

抽象方法定义的格式：

abstract在访问权限后，返回值类型前修饰方法，方法没有方法体:

public abstract 返回值类型 方法名(参数);

### 抽象类和抽象方法案例

已知员工,经理,服务员,厨师,他们都要工作,分析他们的继承关系以及谁作为抽象类

### 案例代码二(只实现服务员分支)

Employee抽象父类：

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: Employee  \* **@Description**: 抽象的父类Employee员工类  \* **@date** 2017年11月14日 上午10:43:20  \* Company www.igeekhome.com  \*  \* 抽象的父类  \*/  **public** **abstract** **class** Employee {  //抽象方法。需要abstract修饰，并分号;结束  /\*\*  \* **@Title**: work  \* **@Description**: 抽象工作方法  \*/  **public** **abstract** **void** work();  } |

Waiter具体子类：

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: Waiter  \* **@Description**: 具体子类Waiter服务员类  \* **@date** 2017年11月14日 上午10:44:32  \* Company www.igeekhome.com  \*  \* 定义抽象父类的具体子类  \*/  **public** **class** Waiter **extends** Employee {  /\*  \* 重写了父类的抽象方法,加入了方法体,描述出具体逻辑  \*/  /\*\*  \* **@Title**: work  \* **@Description**: 服务员的工作类  \* **@see** com.igeek\_03.Employee#work()  \*/  @Override  **public** **void** work() {  System.***out***.println("等着顾客叫餐!");  }  } |

测试类AbstractDemo：

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: AbstractDemo  \* **@Description**: 抽象类的测试类  \* **@date** 2017年11月14日 上午10:48:06  \* Company www.igeekhome.com  \*  \* 酒店中员工、经理、服务员、厨师之间的关系是怎样的？你会与一个员工沟通还是会与一个具体职位的服务员或经理等沟通？  \*  \* 员工父类,定义了其他类的共性内容  \* 成员变量:工号,姓名,年龄,工资  \* 成员方法:工作方法(work),父类简单地完成工作方法,声明只要是员工就应该有工作方法的逻辑  \*  \* 经理,服务员,厨师  \*  \* 真正创建对象,使用对象时,我们往往只使用其子类.  \* 对于上边父类中work的这个方法,每个子类都会重写该方法.  \*  \* 通过以上分析发现:  \* 员工的work方法一定会被子类重写为具体的逻辑,此时可以将该方法定义为抽象方法,仅仅声明有该方法,但是没有具体的方法体,该方法称为抽象方法.  \* 员工父类通常不应该创建对象,又包含了抽象方法,则该员工类可以定义为抽象类.  \*  \* 抽象相关的具体定义:  \* 抽象类用来描述一种类型应该具备的基本特征与功能，具体如何去完成这些行为由子类通过方法重写来完成  \* 抽象方法指只有功能声明，没有功能主体实现的方法  \* 具有抽象方法的类一定为抽象类  \*  \* 抽象定义关键字:abstract  \* 抽象类不能创建对象.  \*  \* 抽象类也有构造方法  \*/  **public** **class** AbstractDemo {  **public** **static** **void** main(String[] args) {  //创建一个父类对象不可以,因为该类为抽象类  //Employee e = new Employee();    //创建具体的子类对象  Waiter waiter = **new** Waiter();  //调用子类重写的父类抽象方法  waiter.work();  }  } |

## 抽象类的构造方法

抽象类的构造方法存在的意义：

子类构造方法中通过super语句调用抽象父类的构造方法,为抽象父类中的成员变量赋值初始化；

而赋好值的成员变量可以被子类的对象使用。

### 案例代码三

Employee抽象父类：

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Employee  \* **@Description**: 抽象的父类Employee员工类  \* **@date** 2017年11月14日 上午10:43:20  \* Company www.igeekhome.com  \*  \* 抽象的父类  \*/  **public** **abstract** **class** Employee {    //抽象类中定义正常的成员变量  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;    //构造方法  /\*\*  \* **@Title**: Employee  \*/  **public** Employee() {  **super**();  }    //该构造方法,不能直接被程序员调用,因为该类为抽象类,不能直接创建对象.  //但是在创建子类对象时,子类的构造方法,可以调用父类的构造方法,为子类对象中的父类存储空间赋值  /\*\*  \* **@Title**: Employee  \* **@param** name  \*/  **public** Employee(String name) {  **super**();  **this**.name = name;  }    //抽象方法。需要abstract修饰，并分号;结束  /\*\*  \* **@Title**: work  \* **@Description**: 抽象工作方法  \*/  **public** **abstract** **void** work();  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  } |

Waiter具体子类：

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Waiter  \* **@Description**: 具体子类Waiter服务员类  \* **@date** 2017年11月14日 上午10:44:32  \* Company www.igeekhome.com  \*  \* 定义抽象父类的具体子类  \*/  **public** **class** Waiter **extends** Employee {    /\*\*  \* **@Title**: Waiter  \*/  **public** Waiter() {  **super**();  }  //子类的构造方法可以调用父类的构造方法  //这里,一个参数的子类构造,调用了父类一个参数的构造,为父类的成员变量赋值  //但是最终还是子类对象自己使用这个成员变量  /\*\*  \* **@Title**: Waiter  \* **@param** name  \*/  **public** Waiter(String name) {  **super**(name);  }  /\*  \* 重写了父类的抽象方法,加入了方法体,描述出具体逻辑  \*/  /\*\*  \* **@Title**: work  \* **@Description**: 服务员的工作类  \* **@see** com.igeek\_03.Employee#work()  \*/  @Override  **public** **void** work() {  System.***out***.println("等着顾客叫餐!");  }  } |

测试类AbstractDemo：

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: AbstractDemo  \* **@Description**: 抽象类的测试类  \* **@date** 2017年11月14日 上午10:48:06  \* Company www.igeekhome.com  \*  \* 抽象类不能创建对象,抽象有构造方法  \*  \* 抽象父类的构造方法,可以完成类似为成员变量赋值的动作,从而这些成员变量可以被子类对象使用.  \*/  **public** **class** AbstractDemo {  **public** **static** **void** main(String[] args) {  //创建具体的子类对象时,子类的构造会调用父类的构造  //为成员变量赋值  Waiter waiter = **new** Waiter("Rose");  //调用子类重写的父类抽象方法  waiter.work();  System.***out***.println(waiter.getName());  }  } |

## 抽象类的特点及常见疑惑

### 特点：

A：抽象类和抽象方法都需要被abstract修饰。抽象方法一定要定义在抽象类中。

B：抽象类不可以直接创建对象，原因：调用抽象方法没有意义。

C：只有覆盖了抽象类中所有的抽象方法后，其子类才可以创建对象。否则该子类还是一个抽象类。

之所以继承抽象类，更多的是在思想，是面对共性类型操作会更简单。

### 疑虑：

A：抽象类一定是个父类，因为抽象类是不断抽取共性需求而来的。

B：抽象类中是可以不定义抽象方法的，此时仅仅是不让该类创建对象，用于某些特殊的设计需要。

C：设计时由具体类抽取出抽象类，而开发阶段应该先定义抽象父类，再根据不同需求由父类定义子类。

## 抽象类的练习(暂空)

# 接口

## 接口概念

接口是功能的集合，同样可看作是一种数据类型，是比抽象类更为抽象的”类”。

接口只描述所应该具备的方法，并没有具体实现，具体的实现由接口的实现类(相当于接口的子类)来完成。这样将功能的定义与实现分离，优化了程序设计。

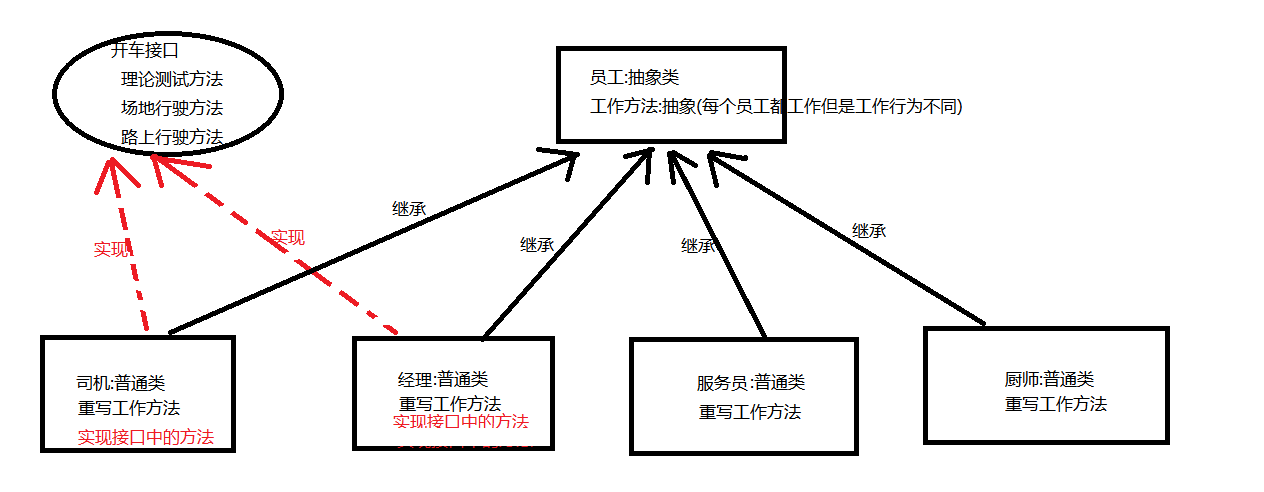
请记住：一切事物均有功能，即一切事物均有接口。

### 接口案例分析

在员工,经理,服务员,厨师案例的基础上

添加新需求:

经理和司机有开车理论测试,场地行驶,路上行驶三个方法,而厨师和服务员都没有。



## 接口的定义

与定义类的class不同，接口定义时需要使用interface关键字。

定义接口所在的仍为.java文件，虽然声明时使用的为interface关键字的编译后仍然会产生.class文件。这点可以让我们将接口看作是一种只包含了功能声明的特殊类。

定义格式：

public interface 接口名 {

抽象方法1;

抽象方法2;

抽象方法3;

}

使用interface代替了原来的class，其他步骤与定义类相同：

接口中的方法均为公共访问的抽象方法

接口中无法定义普通的成员变量

### 案例代码四

开车接口：

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: Driveable  \* **@Description**: 开车接口  \* **@date** 2017年11月14日 下午2:06:39  \* Company www.igeekhome.com  \*  \* 具备开车功能的类,具备三个方法:理论,场地,路上行驶  \*  \*/  **public** **interface** Driveable {    /\*\*  \* **@Title**: theoryTest  \* **@Description**: 理论  \*/  **public** **abstract** **void** theoryTest();    /\*\*  \* **@Title**: fieldDrive  \* **@Description**: 场地  \*/  **public** **abstract** **void** fieldDrive();    /\*\*  \* **@Title**: roadDrive  \* **@Description**: 路上行驶  \*/  **public** **abstract** **void** roadDrive();  } |

## 接口的使用方式

接口不可以创建对象。

我们使用实现来表示一个类与一个接口之间的关系，这是接口最常用的使用方法。

实现的关键字为implements。

格式:

class 类 implements 接口 {

重写接口中方法

}

在实现后，该类就会将接口中的抽象方法”继承”过来，此时该类需要重写该抽象方法，完成具体的逻辑。

则最终总结为：

接口中定义功能，只声明了应该具备该方法，是功能的声明，一切类需要具有该功能时均可以实现该接口。

具体实现类中重写方法，实现功能，是方法的具体实现。

### 案例代码五

员工抽象父类Employee:

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Employee  \* **@Description**: 抽象的父类Employee员工类  \* **@date** 2017年11月14日 上午10:43:20  \* Company www.igeekhome.com  \*  \* 抽象的父类  \*/  **public** **abstract** **class** Employee {    //抽象类中定义正常的成员变量  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;    //构造方法  /\*\*  \* **@Title**: Employee  \*/  **public** Employee() {  **super**();  }    //该构造方法,不能直接被程序员调用,因为该类为抽象类,不能直接创建对象.  //但是在创建子类对象时,子类的构造方法,可以调用父类的构造方法,为子类对象中的父类对象空间赋值  /\*\*  \* **@Title**: Employee  \* **@param** name  \*/  **public** Employee(String name) {  **super**();  **this**.name = name;  }    //抽象方法。需要abstract修饰，并分号;结束  /\*\*  \* **@Title**: work  \* **@Description**: 抽象工作方法  \*/  **public** **abstract** **void** work();  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  } |

开车接口:

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Driveable  \* **@Description**: 开车接口  \* **@date** 2017年11月14日 下午2:06:39  \* Company www.igeekhome.com  \*  \* 具备开车功能的类,具备三个方法:理论,场地,路上行驶  \*  \*/  **public** **interface** Driveable {    /\*\*  \* **@Title**: theoryTest  \* **@Description**: 理论  \*/  **public** **abstract** **void** theoryTest();    /\*\*  \* **@Title**: fieldDrive  \* **@Description**: 场地  \*/  **public** **abstract** **void** fieldDrive();    /\*\*  \* **@Title**: roadDrive  \* **@Description**: 路上行驶  \*/  **public** **abstract** **void** roadDrive();  } |

服务员具体子类Waiter:

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Waiter  \* **@Description**: 具体子类Waiter服务员类  \* **@date** 2017年11月14日 上午10:44:32  \* Company www.igeekhome.com  \*  \* 定义抽象父类的具体子类  \*/  **public** **class** Waiter **extends** Employee {    /\*\*  \* **@Title**: Waiter  \*/  **public** Waiter() {  **super**();  }  //子类的构造方法可以调用父类的构造方法  //这里,一个参数的子类构造,调用了父类一个参数的构造,为父类的成员变量赋值  //但是最终还是子类对象自己使用这个成员变量  /\*\*  \* **@Title**: Waiter  \* **@param** name  \*/  **public** Waiter(String name) {  **super**(name);  }  /\*  \* 重写了父类的抽象方法,加入了方法体,描述出具体逻辑  \*/  /\*\*  \* **@Title**: work  \* **@Description**: 服务员的工作类  \* **@see** com.igeek\_03.Employee#work()  \*/  @Override  **public** **void** work() {  System.***out***.println("等着顾客叫餐!");  }  } |

司机具体子类Driver：

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Driver  \* **@Description**: 司机类  \* **@date** 2017年11月14日 下午2:09:34  \* Company www.igeekhome.com  \*  \* 定义司机类  \* 是员工的子类,继承员工  \* 会开车,实现开车接口  \*/  **public** **class** Driver **extends** Employee **implements** Driveable {  //如果需要的话,可以加入带参的构造方法    //重写接口方法  /\*\*  \* **@Title**: theoryTest  \* **@Description**: 司机的理论测试  \* **@see** com.igeek\_03.Driveable#theoryTest()  \*/  @Override  **public** **void** theoryTest() {  System.***out***.println("驾照理论习题1000道");  }  /\*\*  \* **@Title**: fieldDrive  \* **@Description**: 司机的场地测试  \* **@see** com.igeek\_03.Driveable#fieldDrive()  \*/  @Override  **public** **void** fieldDrive() {  System.***out***.println("会倒车入库,会侧方停车");  System.***out***.println("坡道起步");  }  /\*\*  \* **@Title**: roadDrive  \* **@Description**: 司机的路上行驶测试  \* **@see** com.igeek\_03.Driveable#roadDrive()  \*/  @Override  **public** **void** roadDrive() {  System.***out***.println("不撞人,不闯红灯,不超员");  System.***out***.println("以前开公交,现在开大巴");  }  //重写父类方法  /\*\*  \* **@Title**: work  \* **@Description**: 司机的工作方法  \* **@see** com.igeek\_03.Employee#work()  \*/  @Override  **public** **void** work() {  System.***out***.println("上班接人,下班送人");  }  } |

经理具体子类Manager:

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Manager  \* **@Description**: 经理类  \* **@date** 2017年11月14日 下午2:14:12  \* Company www.igeekhome.com  \*  \* 经理需要具备开车的功能,则实现开车接口  \*/  **public** **class** Manager **extends** Employee **implements** Driveable {  /\*\*  \* **@Title**: Manager  \*/  **public** Manager() {  **super**();  }    /\*\*  \* **@Title**: Manager  \* **@param** name  \*/  **public** Manager(String name) {  **super**(name);  }  //让经理类实现开车功能接口,重写三个方法,具备开车功能  /\*\*  \* **@Title**: theoryTest  \* **@Description**: 经理的理论测试  \* **@see** com.igeek\_03.Driveable#theoryTest()  \*/  @Override  **public** **void** theoryTest() {  System.***out***.println("驾照理论习题1000道");  }  /\*\*  \* **@Title**: fieldDrive  \* **@Description**: 经理的场地驾驶  \* **@see** com.igeek\_03.Driveable#fieldDrive()  \*/  @Override  **public** **void** fieldDrive() {  System.***out***.println("会倒车入库,会侧方停车");  }  /\*\*  \* **@Title**: roadDrive  \* **@Description**: 经理的路上驾驶  \* **@see** com.igeek\_03.Driveable#roadDrive()  \*/  @Override  **public** **void** roadDrive() {  System.***out***.println("不撞人,不闯红灯");  System.***out***.println("经理开的是高级车奇瑞qq~");  }  /\*\*  \* **@Title**: work  \* **@Description**: 经理的工作  \* **@see** com.igeek\_03.Employee#work()  \*/  @Override  **public** **void** work() {  System.***out***.println("管理餐厅人员及设备!");  }  } |

接口使用的测试类：

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: InterfaceDemo  \* **@Description**: 接口使用的测试类  \* **@date** 2018年1月18日 下午4:29:13  \* Company www.igeekhome.com  \*  \* 接口的定义格式，使用格式  \*  \* 接口关键字：interface  \* 使用接口，就是使用类实现接口，实现的关键字是implements，重写方法  \*  \*/  **public** **class** InterfaceDemo {  **public** **static** **void** main(String[] args) {  //创建实现了接口的经理类，调用接口方法  Manager m = **new** Manager("Jack");  m.theoryTest();  m.fieldDrive();  m.roadDrive();    System.***out***.println("================");    //创建新的扩展类，调用方法  Driver driver = **new** Driver();  driver.theoryTest();  driver.fieldDrive();  driver.roadDrive();  }  } |

## 接口的注意事项

A：类单继承类，接口多继承接口，类多实现接口

B：类可以在继承一个类的同时，实现多个接口

C：接口与父类的功能可以重复，均代表要具备某种功能，并不冲突

class A {

public void method() {

}

}

interface B {

public abstract void method();

}

class C extends A implements B {

//如果不重写method()方法,使用的是A中的method()方法

//如果重写method()方法将同时重写A和接口B中的method()方法

}

## 接口中成员特点

A：接口中可以定义变量，但是变量必须有固定的修饰符修饰，public static final 所以接口中的变量也称之为常量，其值不能改变。

后面我们会讲解static与final关键字

B：接口中可以定义方法，方法也有固定的修饰符，public abstract

C：接口不可以创建对象。

D：子类必须覆盖掉接口中所有的抽象方法后，子类才可以实例化。否则子类是一个抽象类。

### 案例代码六

|  |
| --- |
| **package** com.igeek\_05;  /\*\*  \* **@ClassName**: MyInterface  \* **@Description**: 测试接口的固定成员  \* **@date** 2017年11月14日 下午2:43:21  \* Company www.igeekhome.com  \*  \*/  **public** **interface** MyInterface {  //接口中都使用public static final修饰  /\*\*  \* **@Fields** name : 名字的常量  \*/  **public** **static** **final** String ***name*** = "你好";    //所有的成员方法均为public abstract修饰  /\*\*  \* **@Title**: method  \* **@Description**: 抽象方法  \*/  **public** **abstract** **void** method();  } |

|  |
| --- |
| **package** com.igeek\_05;  /\*\*  \* **@ClassName**: MyClass4Interface  \* **@Description**: 实现了接口的具体类  \* **@date** 2017年11月14日 下午2:47:41  \* Company www.igeekhome.com  \*  \*/  **public** **class** MyClass4Interface **implements** MyInterface{  //使用方法重写时,访问权限子类与父类相同,或者子类比父类权限大  /\*\*  \* **@Title**: method  \* **@Description**: 实现具体的方法  \* **@see** com.igeek\_05.MyInterface#method()  \*/  @Override  **public** **void** method() {  System.***out***.println("子类重写的方法");  }  } |

|  |
| --- |
| **package** com.igeek\_05;  /\*\*  \* **@ClassName**: InterfaceMemberDemo  \* **@Description**: 接口成员的测试类  \* **@date** 2017年11月14日 下午2:49:38  \* Company www.igeekhome.com  \*  \* 接口中固定修饰符  \* 没有构造方法  \* 所有的成员变量都不是成员变量,而是静态常量(讲static关键字时,详细介绍),都使用public static final修饰  \* 所有的成员方法均为public abstract修饰  \*/  **public** **class** InterfaceMemberDemo {  **public** **static** **void** main(String[] args) {  //创建实现类的对象  MyClass4Interface m4i = **new** MyClass4Interface();  m4i.method();  }  } |

## 接口与抽象类概念辨析

A：当"我是你的一种时"，子类继承父类，即代表了类与类之间的关系，该体系的类都应该具备父类的成员。

例如:厨师,经理,服务员 都属于员工 的一种,都应该继承员工

B：当"我应该像你一样具备这些功能时"，类实现接口，即代表类与功能的关系，将功能的声明与实现分离。并不是该体系内所有类都需要的额外功能。

例如:只有经理 和司机 具备开车的技能,需要把开车的技能单独定义在接口中,让经理和司机去实现

## 接口与抽象类使用辨析

A: 类继承类extends，只能单继承

接口继承接口extends，可以多继承

类实现接口implements，可以多实现

接口不可以继承类

B: 抽象类中可以有非抽象方法

接口中全部为抽象方法

C: 抽象类具有成员变量

接口没有普通的成员变量

D: 抽象类中的成员无固定修饰符

接口中的成员有固定修饰符

## 接口练习(暂空)

# 多态

## 多态概述

A：多态是继封装、继承之后，面向对象的第三大特性。

现实事物经常会体现出多种形态，如学生，学生是人的一种，则一个具体的同学张三既是学生也是人，即出现两种形态。

B：Java作为面向对象的语言，同样可以描述一个事物的多种形态。如Student类继承了Person类，一个Student的对象便既是Student，又是Person。

Java中多态的代码体现在：一个子类(实现类)对象既可以赋值给这个子类(实现类)引用变量，又可以赋值给这个子类(实现类)的父类(接口)引用变量。

如Student类可以为Person类的子类。那么一个Student对象既可以赋值给一个Student类型的引用，也可以赋值给一个Person类型的引用。

最终多态体现为父类引用变量可以指向子类对象。

C：多态的前提是必须有子父类关系或者类实现接口关系，否则无法完成多态。

在使用多态后的父类引用变量调用方法时，会调用子类重写后的方法。

## 多态的定义与使用格式

多态的定义格式：就是父类的引用变量指向子类对象

A：普通类多态定义的格式

父类 变量名 = new 子类();

如： class Fu {}

class Zi extends Fu {}

//类的多态使用

Fu f = new Zi();

B：抽象类多态定义的格式

抽象类 变量名 = new 抽象类子类();

如： abstract class Fu {

public abstract void method();

}

class Zi extends Fu {

public void method(){

System.out.println(“重写父类抽象方法”);

}

}

//类的多态使用

Fu fu= new Zi();

C：接口多态定义的格式

接口 变量名 = new 接口实现类();

如： interface Fu {

public abstract void method();

}

class Zi implements Fu {

public void method(){

System.out.println(“重写接口抽象方法”);

}

}

//接口的多态使用

Fu fu = new Zi();

## 多态案例:

已知动物类:

动物都有吃饭和睡觉方法

人类是动物的一种:

人不但吃饭,睡觉,而且会学习和工作

### 案例代码七

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Animal  \* **@Description**: 抽象父类Animal  \* **@date** 2017年11月14日 下午3:15:53  \* Company www.igeekhome.com  \*  \* 用于展示多态的抽象父类  \*  \* 吃饭,睡觉功能  \*/  **public** **abstract** **class** Animal {    /\*\*  \* **@Title**: eat  \* **@Description**: 普通的吃的方法  \*/  **public** **void** eat() {  System.***out***.println("动物吃了");  }    /\*\*  \* **@Title**: sleep  \* **@Description**: 抽象睡觉方法  \*/  **public** **abstract** **void** sleep();  } |

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Person  \* **@Description**: 具体子类Person  \* **@date** 2017年11月14日 下午3:23:23  \* Company www.igeekhome.com  \*  \* 用于展示多态的子类  \*  \* 在父类的共性功能之外,有自己的学习工作方法  \*/  **public** **class** Person **extends** Animal{  //重写父类方法  /\*\*  \* **@Title**: eat  \* **@Description**: 重写父类的吃方法  \* **@see** com.igeek\_03.Animal#eat()  \*/  @Override  **public** **void** eat() {  System.***out***.println("用工具吃饭");  }    /\*\*  \* **@Title**: sleep  \* **@Description**: 实现父类的睡觉方法  \* **@see** com.igeek\_03.Animal#sleep()  \*/  @Override  **public** **void** sleep() {  System.***out***.println("在床上躺着睡觉");  }  //定义自身特有方法  /\*\*  \* **@Title**: study  \* **@Description**: 学习方法  \*/  **public** **void** study() {  System.***out***.println("学习书本知识与实践科学");  }    /\*\*  \* **@Title**: work  \* **@Description**: 工作方法  \*/  **public** **void** work() {  System.***out***.println("努力,用够时间");  }  } |

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: SubClassTypeDemo  \* **@Description**: 子类多态的测试类  \* **@date** 2017年11月14日 下午3:26:43  \* Company www.igeekhome.com  \*  \* 多态的前提必须有子父类关系,或者实现接口关系  \*  \* 举例:  \* 父类:动物类  \* 共性的吃饭,睡觉方法  \* 子类:  \* 猪类:共性功能  \* 拱地功能  \* 狗类:共性功能  \* 看门功能  \* 人类:共性功能  \* 学习,工作...功能  \*  \* 多态:  \* 父类(接口)引用变量可以指向子类对象,展现出父类(接口)的功能  \* 子类引用变量可以指向子类对象,展现出子类的功能  \* 子类对象可以表现出多种形态,这多种形态叫做多态  \*  \* 多态代码格式:  \* 父类类型变量名 = new 子类类型();  \* 变量名.方法名();  \*  \* 当出现多态后,其调用的方法是类重写后的方法  \*  \*/  **public** **class** SubClassTypeDemo {  **public** **static** **void** main(String[] args) {    //创建子类对象,让其展示不同的状态(多态)  //不产生多态  Person p = **new** Person();  p.eat();  p.sleep();  p.work();  p.study();  System.***out***.println("===================");    //使用多态,只表现出父类(等号左边的数据类型)的功能  Animal a = **new** Person();  a.eat();  a.sleep();  //a.work(); 多态后不能再调用子类特有方法了  }  } |

## 多态使用场景

我们一般在以下场景当中使用多态：

A: 使用多态为成员变量赋值,局部变量赋值

**class** Father{

}

**class** Son **extends** Father{

Father f=**new** Son();//使用多态形式为成员变量f赋值

**public void** method(){

Father f2=**new** Son();//使用多态的形式为局部变量f2赋值

}

}

B: 方法传参使用多态形式(最常用最能体现出多态优点的应用)

**class** Father{

}

**class** Son **extends** Father{

}

**class** Demo{

**public static void** main(String[] args) {

Son s=**new** Son();

*method*(s);

}

**public static void** method(Father f){//f接收父类对象或者子类对象

}

}

C: 方法返回返回值使用多态形式

**class** Father{

}

**class** Son **extends** Father{

}

**class** Demo{

**public static void** main(String[] args) {

Son s=**new** Son();

Father f=*method*();

}

**public static** Father method(){

Son s = **new** Son();

//由于返回值类型为Father,所以既可以返回父类对象又可以返回子类对象

**return** s;

}

}

### 案例代码八

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: Animal  \* **@Description**: 抽象父类Animal  \* **@date** 2017年11月14日 下午3:15:53  \* Company www.igeekhome.com  \*  \* 用于展示多态的抽象父类  \*  \* 吃饭,睡觉功能  \*/  **public** **abstract** **class** Animal {    /\*\*  \* **@Title**: eat  \* **@Description**: 普通的吃的方法  \*/  **public** **void** eat() {  System.***out***.println("动物吃了");  }    /\*\*  \* **@Title**: sleep  \* **@Description**: 抽象睡觉方法  \*/  **public** **abstract** **void** sleep();  } |

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: Person  \* **@Description**: 具体子类Person  \* **@date** 2017年11月14日 下午3:23:23  \* Company www.igeekhome.com  \*  \* 用于展示多态的子类  \*  \* 在父类的共性功能之外,有自己的学习工作方法  \*/  **public** **class** Person **extends** Animal{  //重写父类方法  /\*\*  \* **@Title**: eat  \* **@Description**: 重写父类的吃方法  \* **@see** com.igeek\_04.Animal#eat()  \*/  @Override  **public** **void** eat() {  System.***out***.println("用工具吃饭");  }    /\*\*  \* **@Title**: sleep  \* **@Description**: 实现父类的睡觉方法  \* **@see** com.igeek\_04.Animal#sleep()  \*/  @Override  **public** **void** sleep() {  System.***out***.println("在床上躺着睡觉");  }  //定义自身特有方法  /\*\*  \* **@Title**: study  \* **@Description**: 学习方法  \*/  **public** **void** study() {  System.***out***.println("学习书本知识与实践科学");  }    /\*\*  \* **@Title**: work  \* **@Description**: 工作方法  \*/  **public** **void** work() {  System.***out***.println("努力,用够时间");  }  } |

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: SubClassTypeDemo  \* **@Description**: 子类多态的测试类  \* **@date** 2017年11月14日 下午3:26:43  \* Company www.igeekhome.com  \*  \* 多态的前提必须有子父类关系,或者实现接口关系  \*  \* 举例:  \* 父类:动物类  \* 共性的吃饭,睡觉方法  \* 子类:  \* 猪类:共性功能  \* 拱地功能  \* 狗类:共性功能  \* 看门功能  \* 人类:共性功能  \* 学习,工作...功能  \*  \* 多态:  \* 父类(接口)引用变量可以指向子类对象,展现出父类(接口)的功能  \* 子类引用变量可以指向子类对象,展现出子类的功能  \* 子类对象可以表现出多种形态,这多种形态叫做多态  \*  \* 多态代码格式:  \* 父类类型变量名 = new 子类类型();  \* 变量名.方法名();  \*  \* 当出现多态后,其调用的方法是类重写后的方法  \*  \* 多态的使用场景:  \* 为变量赋值时使用多态,达到父类应用指向子类对象的赋值  \* 直接为变量赋值  \* 调用方法时,为方法参数赋值  \* 方法返回值,可以定义父类类型,但是返回子类对象.当该方法被调用时,用父类类型接收.  \*  \* 多态的好处:  \* 大大提高了程序的扩展性  \* 提高了程序的复用性  \*/  **public** **class** SubClassTypeDemo {  **public** **static** **void** main(String[] args) {    //创建子类对象,让其展示不同的状态(多态)  //不产生多态  Person p = **new** Person();  p.eat();  p.sleep();  p.work();  p.study();  System.***out***.println("===================");    //使用多态,只表现出父类(等号左边的数据类型)的功能  Animal a = **new** Person();  a.eat();  a.sleep();  //a.work(); 多态后不能再调用子类特有方法了    //调用feed方法  //使用多态的形式,创建子类对象,赋值给父类引用  //Animal a2=new Person();  //feed(a2);    //Person p2=new Person();  //feed(p2)    *feed*(**new** Person());  }    /\*\*  \* **@Title**: feed  \* **@Description**: 接受一个动物,调用该动物吃饭的方法  \* **@param** x  \*/  **public** **static** **void** feed(Animal x){//x可以接收Animal的任意子类对象  //调用该动物的吃饭方法  System.***out***.println("feed方法内的动物吃饭方法被调用了");  x.eat();  }  } |

## 多态转型

多态的转型分为向上转型与向下转型两种：

A：向上转型：当有子类对象赋值给一个父类引用时，便是向上转型，多态本身就是向上转型的过程。

使用格式：

父类类型 变量名 = new 子类类型();

如：Person p = new Student();

B：向下转型：一个已经向上转型的子类对象可以使用强制类型转换的格式，将父类引用转为子类引用，这个过程是向下转型。如果是直接创建父类对象，是无法向下转型的

使用格式：

子类类型 变量名 = (子类类型) 父类类型的变量;

如:Student stu = (Student) p; //变量p 实际上指向Student对象

### 案例代码九

|  |
| --- |
| **package** com.igeek\_05;  /\*\*  \* **@ClassName**: Animal  \* **@Description**: 抽象父类Animal  \* **@date** 2017年11月14日 下午3:15:53  \* Company www.igeekhome.com  \*  \* 用于展示多态的抽象父类  \*  \* 吃饭,睡觉功能  \*/  **public** **abstract** **class** Animal {    /\*\*  \* **@Title**: eat  \* **@Description**: 普通的吃的方法  \*/  **public** **void** eat() {  System.***out***.println("动物吃了");  }    /\*\*  \* **@Title**: sleep  \* **@Description**: 抽象睡觉方法  \*/  **public** **abstract** **void** sleep();  } |

|  |
| --- |
| **package** com.igeek\_05;  /\*\*  \* **@ClassName**: Person  \* **@Description**: 具体子类Person  \* **@date** 2017年11月14日 下午3:23:23  \* Company www.igeekhome.com  \*  \* 用于展示多态的子类  \*  \* 在父类的共性功能之外,有自己的学习工作方法  \*/  **public** **class** Person **extends** Animal{  //重写父类方法  /\*\*  \* **@Title**: eat  \* **@Description**: 重写父类的吃方法  \* **@see** com.igeek\_05.Animal#eat()  \*/  @Override  **public** **void** eat() {  System.***out***.println("用工具吃饭");  }    /\*\*  \* **@Title**: sleep  \* **@Description**: 实现父类的睡觉方法  \* **@see** com.igeek\_05.Animal#sleep()  \*/  @Override  **public** **void** sleep() {  System.***out***.println("在床上躺着睡觉");  }  //定义自身特有方法  /\*\*  \* **@Title**: study  \* **@Description**: 学习方法  \*/  **public** **void** study() {  System.***out***.println("学习书本知识与实践科学");  }    /\*\*  \* **@Title**: work  \* **@Description**: 工作方法  \*/  **public** **void** work() {  System.***out***.println("努力,用够时间");  }  } |

|  |
| --- |
| **package** com.igeek\_05;  /\*\*  \* **@ClassName**: Dog  \* **@Description**: 具体的子类Dog类  \* **@date** 2017年11月14日 下午5:22:41  \* Company www.igeekhome.com  \*  \*/  **public** **class** Dog **extends** Animal{  /\*\*  \* **@Title**: sleep  \* **@Description**: 实现父类的睡觉方法  \* **@see** com.igeek\_05.Animal#sleep()  \*/  @Override  **public** **void** sleep() {  System.***out***.println("在小狗窝里躺着睡觉");  }  } |

|  |
| --- |
| **package** com.igeek\_05;  /\*\*  \* **@ClassName**: SubClassTypeDemo  \* **@Description**: 子类多态的测试类  \* **@date** 2017年11月14日 下午3:26:43  \* Company www.igeekhome.com  \*  \* 多态后的类型转换问题:  \* 向上转型:多态本身是子类类型向父类类型向上转型  \* 向下转型(使用强制类型转换):  \* 父类类型 f = new 子类类型();  \* 子类类型 z = (子类类型)f;  \*  \*/  **public** **class** SubClassTypeDemo {  **public** **static** **void** main(String[] args) {    //使用多态,只表现出父类(等号左边的数据类型)的功能  Animal a = **new** Person();  a.eat();  a.sleep();  System.***out***.println("==============================");    //使用强制类型转换,将父类类型转为子类类型  Person p = (Person)a;  p.eat();  p.sleep();  p.study();  p.work();  System.***out***.println("==============================");    //在强制类型转换时,必须是什么对象,转为什么类型.  //如果使用错,使用了其他类型,则编译时不报错,运行时会报错,显示类型转换异常:java.lang.ClassCastException  Animal a2 = **new** Dog();  Person p2 = (Person)a2;  p2.eat();  p2.sleep();  p2.study();  p2.work();  System.***out***.println("==============================");  }  } |

## instanceof关键字

我们可以通过instanceof关键字来判断某个对象是否属于某种数据类型。如学生的对象属于学生类，学生的对象也属于人类。

使用格式：

boolean b = 对象 instanceof 数据类型;

如

Person p1 = new Student(); // 前提条件，学生类已经继承了人类

boolean flag = p1 instanceof Student; //flag结果为true

boolean flag2 = p2 instanceof Teacher; //flag结果为false

### 案例代码十

|  |
| --- |
| **package** com.igeek\_06;  /\*\*  \* **@ClassName**: Animal  \* **@Description**: 抽象父类Animal  \* **@date** 2017年11月14日 下午3:15:53  \* Company www.igeekhome.com  \*  \* 用于展示多态的抽象父类  \*  \* 吃饭,睡觉功能  \*/  **public** **abstract** **class** Animal {    /\*\*  \* **@Title**: eat  \* **@Description**: 普通的吃的方法  \*/  **public** **void** eat() {  System.***out***.println("动物吃了");  }    /\*\*  \* **@Title**: sleep  \* **@Description**: 抽象睡觉方法  \*/  **public** **abstract** **void** sleep();  } |

|  |
| --- |
| **package** com.igeek\_06;  /\*\*  \* **@ClassName**: Person  \* **@Description**: 具体子类Person  \* **@date** 2017年11月14日 下午3:23:23  \* Company www.igeekhome.com  \*  \* 用于展示多态的子类  \*  \* 在父类的共性功能之外,有自己的学习工作方法  \*/  **public** **class** Person **extends** Animal{  //重写父类方法  /\*\*  \* **@Title**: eat  \* **@Description**: 重写父类的吃方法  \* **@see** com.igeek\_06.Animal#eat()  \*/  @Override  **public** **void** eat() {  System.***out***.println("用工具吃饭");  }    /\*\*  \* **@Title**: sleep  \* **@Description**: 实现父类的睡觉方法  \* **@see** com.igeek\_06.Animal#sleep()  \*/  @Override  **public** **void** sleep() {  System.***out***.println("在床上躺着睡觉");  }  //定义自身特有方法  /\*\*  \* **@Title**: study  \* **@Description**: 学习方法  \*/  **public** **void** study() {  System.***out***.println("学习书本知识与实践科学");  }    /\*\*  \* **@Title**: work  \* **@Description**: 工作方法  \*/  **public** **void** work() {  System.***out***.println("努力,用够时间");  }  } |

|  |
| --- |
| **package** com.igeek\_06;  /\*\*  \* **@ClassName**: Dog  \* **@Description**: 具体的子类Dog类  \* **@date** 2017年11月14日 下午5:22:41  \* Company www.igeekhome.com  \*  \*/  **public** **class** Dog **extends** Animal{  /\*\*  \* **@Title**: sleep  \* **@Description**: 实现父类的睡觉方法  \* **@see** com.igeek\_06.Animal#sleep()  \*/  @Override  **public** **void** sleep() {  System.***out***.println("在小狗窝里躺着睡觉");  }  /\*\*  \* **@Title**: keepDoor  \* **@Description**: 狗的看门方法  \*/  **public** **void** keepDoor() {  System.***out***.println("在大门看门");  }  } |

|  |
| --- |
| **package** com.igeek\_06;  /\*\*  \* **@ClassName**: SubClassTypeDemo  \* **@Description**: 子类多态的测试类  \* **@date** 2017年11月14日 下午3:26:43  \* Company www.igeekhome.com  \*  \* 多态后的类型转换问题:  \* 向上转型:多态本身是子类类型向父类类型向上转型  \* 向下转型(使用强制类型转换):  \* 父类类型 f = new 子类类型();  \* 子类类型 z = (子类类型)f;  \*  \* instanceof关键字:用来判断对象是否属于某种数据类型  \*  \* 某个对象instanceof某个数据类型  \* (note:对象与类型必须拥有子父类关系才可以判断,无关系的两个类型不能判断!)  \*  \*/  **public** **class** SubClassTypeDemo {  **public** **static** **void** main(String[] args) {    //使用多态,只表现出父类(等号左边的数据类型)的功能  Animal a = **new** Person();  a.eat();  a.sleep();  System.***out***.println("==============================");    //使用强制类型转换,将父类类型转为子类类型  Person p = (Person)a;  p.eat();  p.sleep();  p.study();  p.work();  System.***out***.println("==============================");    //在强制类型转换时,必须是什么对象,转为什么类型.  //如果使用错,使用了其他类型,则编译时不报错,运行时会报错,显示类型转换异常:java.lang.ClassCastException  Animal a2 = **new** Dog();  //Person p2 = (Person)a2;  //p2.eat();  //p2.sleep();  //p2.study();  //p2.work();    *method*(p);  }    **public** **static** **void** method(Animal a) {    //先吃饭  a.eat();  //如果是狗,就看门,如果是人就学习工作  **if**(a **instanceof** Dog) {  //强转为狗  Dog d = (Dog)a;  //调用狗的方法  d.keepDoor();  }    //如果是,就看门,如果是人就学习工作  **if**(a **instanceof** Person) {  //强转为人  Person p = (Person)a;  //调用人的方法  p.study();  p.work();  }  }  } |

# static与final修饰符

## static简介

### 概述

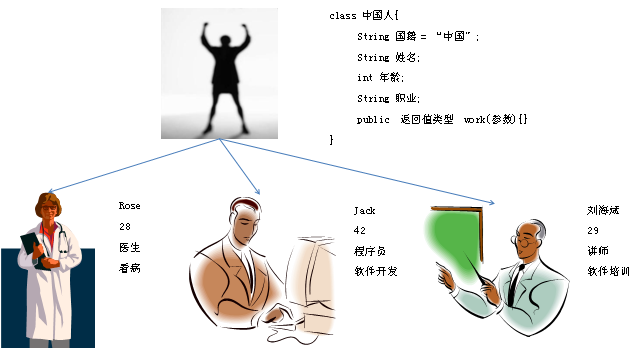
static是静态修饰符，一般修饰成员。被static修饰的成员属于类，不属于单个这个类的某个对象。

static修饰的成员被多个对象共享。

static修饰的成员属于类，但是会影响每一个对象。

被static修饰的成员又叫类成员，不叫对象的成员。

如下例中国籍变量，所有中国人国籍均应该为中国，不应各自定义各自的国籍，所以可以将国籍定义为static，属于类，被多个对象共享。



### 案例代码十一

**Chinese类**

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: Chinese  \* **@Description**: Chinese中国人类  \* **@date** 2017年11月15日 上午9:17:36  \* Company www.igeekhome.com  \*  \* 中国人类  \* 国籍,姓名,年龄,职业  \* 其中,国籍应该被共享  \* 姓名,年龄,职业各个对象有各个对象的值  \*  \*/  **public** **class** Chinese {  //静态成员,被多个对象共享  /\*\*  \* **@Fields** country : 国籍  \*/  **public** **static** String *country* = "中国";  //普通成员,每个对象的普通成员其内容不同  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;  /\*\*  \* **@Fields** age : 年龄  \*/  **private** **int** age;  /\*\*  \* **@Fields** work : 职业  \*/  **private** String work;  /\*\*  \* **@Title**: Chinese  \*/  **public** Chinese() {  **super**();  }  /\*\*  \* **@Title**: Chinese  \* **@param** name  \* **@param** age  \* **@param** work  \*/  **public** Chinese(String name, **int** age, String work) {  **super**();  **this**.name = name;  **this**.age = age;  **this**.work = work;  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  /\*\*  \* **@return** the age  \*/  **public** **int** getAge() {  **return** age;  }  /\*\*  \* **@param** age the age to set  \*/  **public** **void** setAge(**int** age) {  **this**.age = age;  }  /\*\*  \* **@return** the work  \*/  **public** String getWork() {  **return** work;  }  /\*\*  \* **@param** work the work to set  \*/  **public** **void** setWork(String work) {  **this**.work = work;  }  } |

**StaticDemo类**

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: StaticDemo  \* **@Description**: static,静态特征测试类  \* **@date** 2017年11月15日 上午9:22:05  \* Company www.igeekhome.com  \*  \* static是静态修饰符，一般修饰成员。  \* 被static修饰的成员属于类，不属于单个这个类的某个对象。  \*  \* static修饰的成员被多个对象共享。  \* static修饰的成员属于类，但是会影响每一个对象。  \* 被static修饰的成员又叫类成员，不叫对象的成员。  \*  \* 所有中国人国籍均应该为中国，不应各自定义各自的国籍，所以可以将国籍定义为static，属于类，被多个对象共享。  \* 国籍,姓名,年龄,职业  \* 其中,国籍应该被共享  \* 姓名,年龄,职业各个对象有各个对象的值  \*  \* 一般static修饰的成员,直接赋值  \*  \* 当多个对象共享使用同一个类中静态成员时,只要该值改变,就会影响所有的对象  \*  \*/  **public** **class** StaticDemo {  **public** **static** **void** main(String[] args) {  //创建人类对象,验证静态static修饰的成员被多个对象共享  Chinese c = **new** Chinese("王宝强", 34, "演员");  System.***out***.println(c.getAge()+"岁的"+c.getName()+"是"+c.*country*+"人");    Chinese c2 = **new** Chinese("周星驰", 48, "演员或导演或制片");  System.***out***.println(c2.getAge()+"岁的"+c2.getName()+"是"+c2.*country*+"人");    c.*country* = "中华人名共和国";  System.***out***.println("================================");    System.***out***.println(c.getAge()+"岁的"+c.getName()+"是"+c.*country*+"人");    System.***out***.println(c2.getAge()+"岁的"+c2.getName()+"是"+c2.*country*+"人");  }  } |

## 静态的使用方式及加载原理

### 原理概述

被static修饰的成员可以并且建议通过类名直接访问。也可以通过某个对象访到属于类的静态成员，原因即多个对象均属于一个类，共享使用同一个静态成员。

格式：

类名.静态成员变量名

类名.静态成员方法名(参数)

对象名.静态成员变量名 ------不建议，出现警告

对象名.静态成员方法名(参数) ------不建议，出现警告

### 案例代码十二

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: StaticDemo  \* **@Description**: static,静态特征测试类  \* **@date** 2017年11月15日 上午9:22:05  \* Company www.igeekhome.com  \*  \* 静态成员直接使用类名访问  \* 类名.静态成员变量名  \* 类名.静态成员方法名(参数)  \*  \*/  **public** **class** StaticDemo {  **public** **static** **void** main(String[] args) {  //创建人类对象,验证静态static修饰的成员被多个对象共享  Chinese c = **new** Chinese("王宝强", 34, "演员");  System.***out***.println(c.getAge()+"岁的"+c.getName()+"是"+Chinese.*country*+"人");    Chinese c2 = **new** Chinese("周星驰", 48, "演员或导演或制片");  System.***out***.println(c2.getAge()+"岁的"+c2.getName()+"是"+Chinese.*country*+"人");    Chinese.*country* = "中华人名共和国";  System.***out***.println("================================");    System.***out***.println(c.getAge()+"岁的"+c.getName()+"是"+Chinese.*country*+"人");    System.***out***.println(c2.getAge()+"岁的"+c2.getName()+"是"+Chinese.*country*+"人");    //类名.方法名()  Chinese.*method*();  }  } |

**StaticDemo2.java类**

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: StaticDemo2  \* **@Description**: 静态成员的访问  \* **@date** 2017年11月15日 上午9:32:02  \* Company www.igeekhome.com  \*  \* 静态成员只能直接访问静态成员  \*  \* 原因为:静态内容优先于对象存在  \*/  **public** **class** StaticDemo2 {    **public** **static** **void** main(String[] args) {  //由于静态内容随类的加载而加载,有类则有该方法  //但是此时是可以没有对象的  StaticDemo2.*method*();  }  /\*\*  \* **@Fields** name : 姓名的实例成员变量  \*/  **private** String name;    /\*\*  \* **@Title**: normalMethod  \* **@Description**: 普通的实例成员方法  \*/  **public** **void** normalMethod() {  System.***out***.println("普通方法");  }    /\*\*  \* **@Title**: method  \* **@Description**: 静态成员方法  \*/  **public** **static** **void** method() {  //静态方法不能直接访问非静态成员  //System.out.println(name);  //normalMethod();  }  } |

## final关键字

### final关键字的使用

final是最终修饰符，可以修饰类、成员方法、变量。

final修饰的类无法被继承。

final修饰的方法无法被重写。

final修饰的变量无法被再次赋值。

如：

final int a = 10; //则a无法被2次赋值

我们通常使用public static final来定义静态常量（如接口中的固定修饰符）

引用类型的变量用final修饰，是指其所引用的对象不能改变，即该变量引用的地址值不能改变。

### final修饰类无法被继承

#### 案例代码十三

FinalClass类

|  |
| --- |
| **package** com.igeek\_03\_02;  /\*\*  \* **@ClassName**: FinalClass  \* **@Description**: final类  \* **@date** 2017年11月15日 上午10:06:15  \* Company www.igeekhome.com  \*  \* 被final修饰的类，不能有子类，不能被继承  \*/  **public** **final** **class** FinalClass {  } |

FinalSubClass类

|  |
| --- |
| **package** com.igeek\_03\_02;  /\*\*  \* **@ClassName**: FinalSubClass  \* **@Description**: 子类  \* **@date** 2017年11月15日 上午10:07:15  \* Company www.igeekhome.com  \*  \* 被final修饰的类FinalClass不能被继承  \*/  //public class FinalSubClass extends FinalClass{  //  //} |

### final修饰的方法无法被重写

#### 案例代码十四

FinalClass2类

|  |
| --- |
| **package** com.igeek\_03\_03;  /\*\*  \* **@ClassName**: FinalClass2  \* **@Description**: 被final修饰方法  \* **@date** 2017年11月15日 上午10:10:33  \* Company www.igeekhome.com  \*  \* 被final修饰方法:无法被重写  \*  \*/  **public** **class** FinalClass2 {  /\*\*  \* **@Title**: method  \* **@Description**: final方法  \*/  **public** **final** **void** method() {  System.***out***.println("被final修饰的方法");  }  } |

FinalSubClass2类

|  |
| --- |
| **package** com.igeek\_03\_03;  /\*\*  \* **@ClassName**: FinalSubClass2  \* **@Description**: 子类  \* **@date** 2017年11月15日 上午10:13:16  \* Company www.igeekhome.com  \*  \* final修饰的方法无法被子类重写  \*/  **public** **class** FinalSubClass2 **extends** FinalClass2{  // final修饰的方法不能被重写  // public void method() {  // System.out.println("被final修饰的方法");  // }  } |

### final修饰变量无法被再次赋值

#### 案例代码十五

Person类

|  |
| --- |
| **package** com.igeek\_03\_04;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person类  \* **@date** 2017年11月15日 上午10:25:23  \* Company www.igeekhome.com  \*  \*/  **public** **class** Person {  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  } |

FinalDemo测试类

|  |
| --- |
| **package** com.igeek\_03\_04;  /\*\*  \* **@ClassName**: FinalDemo  \* **@Description**: final修饰变量  \* **@date** 2017年11月15日 上午10:18:42  \* Company www.igeekhome.com  \*  \* final是最终修饰符，可以修饰类、成员方法、变量  \*  \* final修饰的类无法被继承。  \* final修饰的方法无法被重写。  \* final修饰的变量无法被再次赋值,必须被赋值。(包括局部变量与成员变量)  \*  \* 现实开发过程中遇到的最常见成员常量的情况,连续使用public static final在一个类中定义一个常量,采取直接给常量赋值的方式.  \* 注意常量的命名规则:所有字母均大写,用\_连接  \*  \*/  **public** **class** FinalDemo {  // 成员位置定义静态常量  /\*\*  \* **@Fields** PI : PI常量  \*/  **public** **static** **final** **double** ***PI*** = 3.1415926;  **public** **static** **void** main(String[] args) {  // final修饰的变量无法被再次赋值  **int** a = 10;  a = 20;  **final** **int** b = 20;  // 被final修饰的变量不能二次赋值  // b = 30;  // 使用静态常量  **double** r = 2.5;  **double** area = r \* r \* FinalDemo.***PI***;  System.***out***.println(area);  // 为引用变量二次赋值,指p这个变量只能指向一个地址,不能指向第二个地址  **final** Person p = **new** Person();  p.setName("张三");  p.setName("李四");  System.***out***.println(p.getName());  // 地址不能改,为下边这句内容  // p = new Person();  }  } |

# 包与访问修饰符

## 包的概述及定义格式

### 包的概念

分包管理是组织软件项目结构的基本方式。我们将同类功能放到一个包中，方便管理。并且日常项目的分工也是以包作为边界。

包在文件系统中是以文件夹的形式存在的。类中定义的包必须与实际class文件所在的文件夹情况相统一，即定义包后，如果类在a包下，则生成的.class文件必须在a文件夹下，否则找不到类。

### 包的定义格式

使用公司域名反写，多级包全部小写，用”.”连接

如：极客营网址为igeekhome.com那么域名反写就为com.igeekhome

甲骨文公司oracle.com 那么域名反写就为 com.oracle

类中的定义格式： package 包名;

编译时：直接编译java文件即可

运行时：由于类的全名是包含包名的，所以必须运行加入了包名的全类名

### 案例代码十六

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: PackageDemo  \* **@Description**: 包的作用  \* **@date** 2018年1月19日 下午2:18:03  \* Company www.igeekhome.com  \*  \* 包：是java程序管理资源的方式  \*  \* 类中声明的包，必须与真正该.class文件所在的文件夹一致  \*  \* 声明的格式:  \* package 包(文件夹).包(文件夹).包(文件夹)...  \*  \*/  **public** **class** PackageDemo {  } |

## 类的全名访问

在访问类时，为了能够找到该类，使用类时，应该使用包含包名的类全名。

如：字符串String在使用时，可以按照如下方式访问：

java.lang.String s = “一个字符串”;

等价于 String s = “一个字符串”;

### 案例代码十七

Person类：

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person类  \* **@date** 2017年11月15日 上午10:54:11  \* Company www.igeekhome.com  \*  \*/  **public** **class** Person {  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;    /\*\*  \* **@Title**: Person  \* **@param** name  \*/  **public** Person(String name) {  **this**.name=name;  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  } |

其他包中的Person类：

|  |
| --- |
| **package** com.igeek\_02.other;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person类  \* **@date** 2017年11月15日 上午10:54:11  \* Company www.igeekhome.com  \*  \*/  **public** **class** Person {  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;    /\*\*  \* **@Title**: Person  \* **@param** name  \*/  **public** Person(String name) {  **this**.name=name;  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  } |

类全名创建对象测试类：

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: PackageDemo  \* **@Description**: package包的作用  \* **@date** 2017年11月15日 上午10:37:53  \* Company www.igeekhome.com  \*  \* 包:是java程序管理资源的方式  \*  \* 类中声明的包,必须与真正该.class文件所在的文件夹一致  \*  \* 声明格式:  \* package 文件夹(包).文件夹(包).文件夹(包)...  \*  \* 如何带包使用其他的类:应该加入全名,即包名+类名  \*/  **public** **class** PackageDemo {  **public** **static** **void** main(String[] args) {  //使用类全名访问JRE提供好的类  java.lang.String s = "abc";  System.***out***.println(s);    //使用类全名访问同包下的其他类  com.igeek\_02.Person person=**new** com.igeek\_02.Person("Jack");  person.setName("Rose");  System.***out***.println(person.getName());    //使用类全名访问不同包(跨包)下的其他类  com.igeek\_02.other.Person person2=**new** com.igeek\_02.other.Person("Rose");  System.***out***.println(person2.getName());  }  } |

## 带包的类访问

### 同包下省略包名

当被使用的类与使用的类在同一个文件夹下，或者被使用的类是java.lang包下时，我们通常可以省略掉类的包，直接使用类型，避免使用类全名。

### 跨包时导包访问

当被使用的类与使用的类不在同一个文件夹下时，我们可以通过导包的方式使用该类，避免使用类全名。

导包格式：

package后，class前

使用import 包名.包名.类名;

当多个文件夹下有相同的类名时，只能有一个导包使用，其他必须仍然书写全名。

当想一下导入包中多个类时，可以使用\*代表该包下的所有类。

### 案例代码十八

同包下的Person类：

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person类  \* **@date** 2017年11月15日 上午10:54:11  \* Company www.igeekhome.com  \*  \*/  **public** **class** Person {  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;    /\*\*  \* **@Title**: Person  \* **@param** name  \*/  **public** Person(String name) {  **this**.name=name;  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  } |

com.igeek\_03.other1包下的PersonOther类：

|  |
| --- |
| **package** com.igeek\_03.other1;  /\*\*  \* **@ClassName**: PersonOther  \* **@Description**: PersonOther类  \* **@date** 2017年11月15日 上午10:54:11  \* Company www.igeekhome.com  \*  \*/  **public** **class** PersonOther {  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;    /\*\*  \* **@Title**: Person  \* **@param** name  \*/  **public** PersonOther(String name) {  **this**.name=name;  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  } |

com.igeek\_03.other2包下的PersonOther类：

|  |
| --- |
| **package** com.igeek\_03.other2;  /\*\*  \* **@ClassName**: PersonOther  \* **@Description**: PersonOther类  \* **@date** 2017年11月15日 上午10:54:11  \* Company www.igeekhome.com  \*  \*/  **public** **class** PersonOther {  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;    /\*\*  \* **@Title**: Person  \* **@param** name  \*/  **public** PersonOther(String name) {  **this**.name=name;  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  } |

不同包中各个类的导入测试类：

|  |
| --- |
| **package** com.igeek\_03;  //导包声明  **import** com.igeek\_03.other1.PersonOther;  //不能导入相同名称的类,所以,当多个文件夹下有相同的类名时，只能有一个导包使用，其他必须仍然书写全名  //import com.igeek\_03.other2.PersonOther;  /\*\*  \* **@ClassName**: PackageAccessDemo  \* **@Description**: 不同包下面各个类的访问方式  \* **@date** 2017年11月15日 上午11:17:49  \* Company www.igeekhome.com  \*  \* JRE提供的java.lang包下的类,是可以直接使用类名,无需加包名的,比如String  \*  \* 同包访问时,可以省略全名,使用类名  \*  \* 跨包访问:  \* 可以不使用全名,使用导包的方式声明出要使用的类时哪个  \*  \* 导包格式：  \* package后，class前  \* 使用import 包名.包名.类名;  \*/  **public** **class** PackageAccessDemo {  **public** **static** **void** main(String[] args) {    //JRE提供的lang包下的类,是可以直接使用类名,无需加包名的,比如String  String s = "hit the road";    //同包访问时,可以省略全名,使用类名  Person person = **new** Person("Jack");  System.***out***.println(person.getName());    //跨包访问,需要先导包,再使用  PersonOther Person2 = **new** PersonOther("Rose");  System.***out***.println(Person2.getName());    //不能导入相同名称的类,所以,当多个文件夹下有相同的类名时，只能有一个导包使用，其他必须仍然书写全名  com.igeek\_03.other2.PersonOther Person3 = **new** com.igeek\_03.other2.PersonOther("Rose");  System.***out***.println(Person3.getName());    }  } |

## 访问权限

### 四种权限访问修饰符

在Java中提供了四种访问权限，使用不同的访问权限时，被修饰的内容会有不同的访问权限，以下表来说明不同权限的访问能力：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | public | protected | 空的(default) | private |
| 同一类中 | √ | √ | √ | √ |
| 同一包中(子类与无关类) | √ | √ | √ |  |
| 不同包的子类 | √ | √ |  |  |
| 不同包中的无关类 | √ |  |  |  |

归纳一下：在日常开发过程中，编写的类、方法、成员变量的访问

* 要想仅能在本类中访问使用private修饰；
* 要想本包中的类都可以访问不加修饰符即可；
* 要想本类的子类可以访问使用protected修饰
* 要想任意包中的任意类都可以访问使用public修饰。
* 注意：如果类用public修饰，则类名必须与文件名相同。一个文件中只能有一个public修饰的类。

### 同包中权限访问测试

#### 案例代码十九

**AccessClass类**

|  |
| --- |
| **package** com.igeek\_04\_02;  /\*\*  \* **@ClassName**: AccessClass  \* **@Description**: 测试访问权限的类  \* **@date** 2017年11月15日 下午2:24:35  \* Company www.igeekhome.com  \*  \*/  **public** **class** AccessClass {    /\*\*  \* **@Title**: method  \* **@Description**: 私有化的方法  \*/  **private** **void** method() {  System.***out***.println("私有化的方法");  }    /\*\*  \* **@Title**: method2  \* **@Description**: 默认权限的方法  \*/  **void** method2() {  System.***out***.println("默认权限的方法");  }    /\*\*  \* **@Title**: method3  \* **@Description**: 受保护的方法  \*/  **protected** **void** method3() {  System.***out***.println("受保护的方法");  }    /\*\*  \* **@Title**: method4  \* **@Description**: 公共的方法  \*/  **public** **void** method4() {  System.***out***.println("公共的方法");  }    /\*\*  \* **@Title**: testAccess  \* **@Description**: 本类当中的权限测试方法  \*/  **public** **void** testAccess() {  method();  method2();  method3();  method4();  }  **public** **static** **void** main(String[] args) {    AccessClass ac = **new** AccessClass();    //本类当中,可以访问所有的方法  ac.method();  ac.method2();  ac.method3();  ac.method4();  }  } |

**AccessDemo类**

|  |
| --- |
| **package** com.igeek\_04\_02;  /\*\*  \* **@ClassName**: AccessDemo  \* **@Description**: 访问权限的测试类  \* **@date** 2017年11月15日 下午2:26:38  \* Company www.igeekhome.com  \*  \*/  **public** **class** AccessDemo {  **public** **static** **void** main(String[] args) {    AccessClass ac = **new** AccessClass();    //同包下,可以访问其他类的非私有方法  //ac.method();  ac.method2();  ac.method3();  ac.method4();  }  } |

### 跨包访问

#### 案例代码二十

**AccessDemo类**

|  |
| --- |
| **package** com.igeek\_04\_03;  **import** com.igeek\_04\_02.AccessClass;  /\*\*  \* **@ClassName**: AccessDemo  \* **@Description**: 访问权限的测试类  \* **@date** 2017年11月15日 下午2:26:38  \* Company www.igeekhome.com  \*  \*/  **public** **class** AccessDemo {  **public** **static** **void** main(String[] args) {    AccessClass ac = **new** AccessClass();    //测试跨包访问,只能访问公共方法  //ac.method();  //ac.method2();  //ac.method3();  ac.method4();  }  } |

### 不同包下子父类之间访问

#### 案例代码二十一

**AccessDemo类**

|  |
| --- |
| **package** com.igeek\_04\_04;  **import** com.igeek\_04\_02.AccessClass;  /\*\*  \* **@ClassName**: AccessDemo  \* **@Description**: 访问权限的测试类  \* **@date** 2017年11月15日 下午2:26:38  \* Company www.igeekhome.com  \*  \* AccessDemo类继承父类AccessClass  \*/  **public** **class** AccessDemo **extends** AccessClass{    **public** **static** **void** main(String[] args) {    AccessClass ac = **new** AccessClass();    //测试跨包访问,只能访问公共方法  //ac.method();  //ac.method2();  //ac.method3();  ac.method4();    AccessDemo ad=**new** AccessDemo();  ad.method0();  }    /\*\*  \* **@Title**: method0  \* **@Description**: 子类可以访问父类的公共与受保护方法,无论是否同包  \*/  **public** **void** method0() {  //super.method();  //super.method2();  **super**.method3();  **super**.method4();  }  } |

# 内部类

## 内部类基本概念

内部类即在类中定义另外一个类，是一个相对概念。

内部类分为成员内部类与局部内部类。

定义时是一个正常定义类的过程，同样包含各种修饰符、继承与实现关系等。

内部类可以直接访问外部类的所有成员。

外部类编译后会出现两个class文件。

(在日常的企业级开发中，我们很少会使用到内部类来实现业务逻辑)

## 成员内部类

定义在成员位置的内部类

定义格式

public class Outer{

class Inner{

//其他代码

}

}

访问格式：

Outer.Inner x = new Outer().new Inner();

### 代码案例二十二

**NestedClass成员内部类**

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: NestedClass  \* **@Description**: 成员内部类  \* **@date** 2017年11月15日 下午4:02:57  \* Company www.igeekhome.com  \*  \* 外部类  \*/  **public** **class** NestedClass {  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name = "Jack";    /\*\*  \* **@ClassName**: InnerClass  \* **@Description**: 成员内部类  \* **@date** 2017年11月15日 下午4:03:49  \* Company www.igeekhome.com  \*  \*/  **public** **class** InnerClass {    **public** **void** innerMethod() {  System.***out***.println("内部类的方法被调用了");  System.***out***.println("外部类的成员变量name的值为:"+name);  }  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  } |

**NestedClassDemo成员内部类测试类**

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: NestedClassDemo  \* **@Description**: 成员内部类测试类  \* **@date** 2017年11月15日 下午4:05:03  \* Company www.igeekhome.com  \*  \* 内部类是一个相对概念，即类中定义类  \* 内部类可以直接访问外部类的所有成员  \*  \* 内部类：  \* 成员内部类：定义在成员位置  \* 局部内部类：定义在局部位置，即一个方法内  \*  \* 成员内部类创建对象，必须有外部类对象，才能有内部类对象  \* Outer.Inner x = new Outer().new Inner();  \*/  **public** **class** NestedClassDemo {  **public** **static** **void** main(String[] args) {  //创建成员内部类对象  NestedClass.InnerClass inner = **new** NestedClass().**new** InnerClass();  inner.innerMethod();  }  } |

## 局部内部类

定义在局部位置的内部类

定义格式

public class Outer{

public void method{

class Inner{

//其他代码

}

}

}

访问格式：

只能在外部类的方法中创建对象并访问。

### 案例代码二十三

**NestedClass局部内部类**

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: NestedClass  \* **@Description**: 局部内部类  \* **@date** 2017年11月15日 下午4:02:57  \* Company www.igeekhome.com  \*  \* 外部类  \*/  **public** **class** NestedClass {  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name = "Jack";    /\*\*  \* **@Title**: outerMethod  \* **@Description**: 外部类的方法  \*/  **public** **void** outerMethod() {  /\*\*  \* **@ClassName**: InnerClass  \* **@Description**: 局部内部类  \* **@date** 2017年11月15日 下午4:10:17  \* Company www.igeekhome.com  \*/  **class** InnerClass {    **public** **void** innerMethod() {  System.***out***.println("内部类的方法被调用了");  System.***out***.println("外部类的成员变量name的值为:"+name);  }  }    //定义好类之后可以使用类  InnerClass inner = **new** InnerClass();  inner.innerMethod();  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  } |

**NestedClassDemo局部内部类测试类**

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: NestedClassDemo  \* **@Description**: 局部内部类测试类  \* **@date** 2017年11月15日 下午4:05:03  \* Company www.igeekhome.com  \*  \* 内部类是一个相对概念，即类中定义类  \* 内部类可以直接访问外部类的所有成员  \*  \* 内部类：  \* 成员内部类：定义在成员位置  \* 局部内部类：定义在局部位置，即一个方法内，符合先定义再使用！  \*  \* 成员内部类创建对象，必须有外部类对象，才能有内部类对象  \* Outer.Inner x = new Outer().new Inner();  \*/  **public** **class** NestedClassDemo {  **public** **static** **void** main(String[] args) {  //创建外部类对象  NestedClass nc = **new** NestedClass();  //通过外部类对象调用方法访问局部内部类对象的操作  nc.outerMethod();  }  } |

## 匿名内部类

### 概述

匿名内部类是局部内部类的一种。

过程：

临时定义一个类型的子类

定义后即刻创建刚刚定义的这个类的对象

目的：

匿名内部类是创建某个类型子类对象的快捷方式。

我们为了临时定义一个类的子类，并创建这个子类的对象而使用匿名内部类。

### 匿名内部类格式

将定义子类与创建子类对象两个步骤由一个格式一次完成。

虽然是两个步骤，但是两个步骤是连在一起的、即时的。

匿名内部类如果不定义变量引用，则也是匿名对象。格式如下：

new 父类(){

//重写需要重写的方法

};

例如：

已经存在的父类：

public abstract class Person{

public abstract void eat();

}

定义并创建该父类的子类对象，并用多态的方式赋值给父类引用变量

Person p = new Person(){

public void eat() {

System.out.println(“我吃了”);

}

};

### 案例代码二十四

**Animal抽象父类**

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: Animal  \* **@Description**: 抽象父类Animal  \* **@date** 2017年11月15日 下午4:18:17  \* Company www.igeekhome.com  \*  \* 吃饭,睡觉的抽象方法  \*/  **public** **abstract** **class** Animal {  //抽象方法  /\*\*  \* **@Title**: eat  \* **@Description**: 吃饭的方法  \*/  **public** **abstract** **void** eat();  /\*\*  \* **@Title**: sleep  \* **@Description**: 睡觉的方法  \*/  **public** **abstract** **void** sleep();  } |

**Dog具体子类**

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: Dog  \* **@Description**: 子类Dog类  \* **@date** 2017年11月15日 下午4:19:33  \* Company www.igeekhome.com  \*  \* 定义子类,重写父类方法  \*/  **public** **class** Dog **extends** Animal {  //重写方法  /\*\*  \* **@Title**: eat  \* **@Description**: 狗吃饭的方法  \* **@see** com.igeek\_04.Animal#eat()  \*/  @Override  **public** **void** eat() {  System.***out***.println("吃骨头");  }  /\*\*  \* **@Title**: sleep  \* **@Description**: 狗睡觉的方法  \* **@see** com.igeek\_04.Animal#sleep()  \*/  @Override  **public** **void** sleep() {  System.***out***.println("趴着睡");  }  } |

**AnonymousInnerClass匿名内部类的测试类**

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: AnonymousInnerClass  \* **@Description**: 匿名内部类的测试类  \* **@date** 2017年11月15日 下午4:20:54  \* Company www.igeekhome.com  \*  \* 匿名内部类  \*  \* 目的:创建某个类(接口)的子类对象  \* 步骤1:定义带名字的子类  \* 步骤2:创建对象  \*  \* 匿名内部类是创建某个类型子类对象的快捷方式  \* 我们为了临时定义一个类的子类，并创建这个子类的对象而使用匿名内部类。  \*  \* 将定义子类与创建子类对象两个步骤由一个格式一次完成  \*  \* 匿名内部类格式:  \* new 父类(){  \* //重写需要重写的方法  \* };  \*/  **public** **class** AnonymousInnerClass {    **public** **static** **void** main(String[] args) {    //使用有名字的子类(多态使用)  Animal ad = **new** Dog();  ad.eat();  ad.sleep();    //使用匿名内部类  Animal a = **new** Animal(){  /\*\*  \* **@Title**: eat  \* **@Description**: 吃草的方法  \* **@see** com.igeek\_04.Animal#eat()  \*/  @Override  **public** **void** eat() {  System.***out***.println("吃草");  }  /\*\*  \* **@Title**: sleep  \* **@Description**: 睡觉的方法  \* **@see** com.igeek\_04.Animal#sleep()  \*/  @Override  **public** **void** sleep() {  System.***out***.println("仰着睡");  }  };    a.eat();  a.sleep();    }  } |

重点和总结

1、抽象类和抽象方法

2、接口的基本操作，及和抽象类的区别

4、实现接口、或继承父类，带来的多态特性

5、static与final修饰符

6、package打包，import导入包，及四种访问修饰符

7、内部类的各种形式：成员内部类、局部内部类、匿名内部类