# 【OOP：面向对象编程\_第三天】

## 主要内容

1. 多态
2. Object类及常用方法
3. 组合关系

## 学习目标

|  |  |
| --- | --- |
| 知识点 | 要求 |
| 构造方法在继承中的调用过程 | 掌握 |
| 引入多态 | 熟悉 |
| 多态之向上转型 | 掌握 |
| 多态之向下转型 | 掌握 |
| 父类作为方法参数 | 掌握 |
| 父类作为方法返回值 | 掌握 |
| Object类的介绍 | 熟悉 |
| ==与equals方法的使用 | 掌握 |
| hashCode方法的使用 | 掌握 |
| toString方法的使用 | 掌握 |
| 组合关系的使用和练习 | 掌握 |

## 第一节 继承下的构造方法

### 1.1 继承情况下构造方法的调用过程

**继承条件下构造方法的执行顺序**

* 构造方法的第一条语句默认是super(),含义是调用父类无参数的构造方法
* 构造方法的第一条语句可以显式的指定为父类的有参数构造方法：super(.....)
* 构造方法的第一条语句可以显式的指定为当前类的构造方法：this(.....)

**注意事项**

* 每个类最好要提供无参数的构造方法
* 构造方法的第一条语句可以是通过super或者this调用构造方法，须是第一条语句
* 构造方法中不能同时使用super和this调用构造方法，并不是说不能同时出现this和super

#### 【示例1】继承情况下构造方法的调用过程

|  |
| --- |
| **package** com.bjsxt.constructorDemo;  **public class** Test1 {  **public static void** main(String[] args) {  S s =**new** S();  } } **class** F{  */\*public F(){  System.out.println("父类中无参的构造方法");  }\*/* **public** F(**int** i){  System.***out***.println(**"父类中有参数的构造方法"**);  }   **static** {  System.***out***.println(**"父类中的静态代码块"**);   }  {  System.***out***.println(**"父类中的普通代码块"**);  } }  **class** S **extends** F{  */\*  \* 构造方法不能被子类继承  \* 子类的构造方法一定会调用父类的构造方法  \* 在子类的构造方法中,使用super()的形式默认调用父类无参构造方法  \* 当父类中没有无参构造方法时,子类构造方法中必须显示书写super()并传入实参  \* super()必须是子类构造放的第一行  \*   \* 当父类中没有无参构造方法, 那么子类的构造方法中 要么就只能调用父类构造方法,要么就只能调用其他构造方法  \* \*/* **public** S (){  *// 调用父类无参数构造方法* **super**(10);   System.***out***.println(**"子类中无参构造方法"**);  }  **public** S(**int** i){  **this**();  }   **static** {  System.***out***.println(**"子类中的静态代码块"**);   }  {  System.***out***.println(**"子类中的普通代码块"**);  } } |

继承中的代码块执行顺序

|  |
| --- |
| **package** com.bjsxt.extendsDemo2;  **public class** Test1 {  **public static void** main(String[] args) {  **new** C();  **new** C();  } } **class** A{  {  System.***out***.println(**"A code block"**);  }  **static**{  System.***out***.println(**"A static code block"**);  }  **public** A(){  System.***out***.println(**"A constructor"**);  }  }  **class** B **extends** A {  {  System.***out***.println(**"B code block"**);  }  **static**{  System.***out***.println(**"B static code block"**);  }  **public** B(){  System.***out***.println(**"B constructor"**);  }  } **class** C **extends** B {  {  System.***out***.println(**"C code block"**);  }  **static**{  System.***out***.println(**"C static code block"**);  }  **public** C(){  System.***out***.println(**"C constructor"**);  }  } |

## 第二节 多态

多态（polymorphism）是面向对象三大特征之一。同一行为，通过不同的子类，可以体现出来的不同的形态。

### 2.1 引入和使用多态

**多态指的是同一个方法调用，由于对象不同可能会有不同的行为。**现实生活中，同一个方法，具体实现会完全不同。 比如：同样是调用人的“休息”方法，张三是睡觉，李四是旅游，王五是听音乐； 同样是调用人“吃饭”的方法，中国人用筷子吃饭，英国人用刀叉吃饭，印度人用手吃饭。

编译器类型指的是‘=’左边的类型，运行期类型指的是‘=’右边的类型。当有继承关系时，可能发生编译期类型和运行期类型不同的情况，即编译期类型是父类类型，运行期类型是子类类型。即：父类引用指向子类对象

多态的要点：

1. 多态是方法的多态，不是属性的多态（多态与属性无关）。

2. 多态的存在要有3个必要条件：继承，方法重写，父类引用指向子类对象。

3. 父类引用指向子类对象后，用该父类引用调用子类重写的方法，此时多态就出现了。

使用父类做方法的形参和返回值，是多态使用最多的场合。即使增加了新的子类，方法也无需改变，符合开闭原则。

父类引用做方法的形参，实参可以是任意的子类对象，可以通过不同的子类对象实现不同的行为方式。另外即使增加了新的子类，方法也无需改变，提高了扩展性，符合开闭原则。

|  |
| --- |
| **扩展： 面向对象的设计原则之一：开闭原则OCP**   * 对扩展开放，对修改关闭 * 更通俗翻译：软件系统中的各种组件，如模块（Modules）、类（Classes）以及功能（Functions）等，应该在不修改现有代码的基础上，引入新功能 * 面向对象设计需要遵守的最高原则，也是最终目标 |

### 2.2 多态之向上转型

将子类对象赋给父类引用，称为向上转型（upcasting），自动进行类型转换。向上转型可以调用的子类继承的方法，但不能调用子类特有的方法。需要特别理解的是如果子类重写了父类的方法，向上转型后通过父类引用调用的却是真实子类重写的方法

#### 【示例2】 向上转型

|  |
| --- |
| **package** com.bjsxt.polymorphismDemo1;  **public class** Test1 {  **public static void** main(String[] args) {  */\* F f =new F();  f.eat();   S1 a =new S1();  a.eat();\*/   /\*  \* 子类可以声明成父类对象 应为子类完全拥有父类的属性和方法  \*  \* 1子类声明成父类对象 在调用方法时,执行的是子类重写父类的方法  \* 2子类声明成父类对象 只能调动父类中声明过的方法,不能调用子类中自定义的方法  \*  \* 子类声明成父类对象 编译时认为是父类对象 只能调用到父类中声明过的方法  \* 运行时认为是子类,运行子类中重写的父类的方法  \*  \* \*/    // F f=new S3();   // f.eat();  //  // f.play();   teatA*(**new** S1());  }   **public static void** teatA(F f){  f.eat();  }  }  **class** F{  **public void** eat(){  System.***out***.println(**"吃大米"**);  } }  **class** S1 **extends** F{  **public void** eat(){  System.***out***.println(**"吃面条"**);  }  **public void** play(){  System.***out***.println(**"打游戏"**);  } } **class** S2 **extends** F{  @Override  **public void** eat() {  System.***out***.println(**"吃饺子"**);  } } **class** S3 **extends** F{  @Override  **public void** eat() {  System.***out***.println(**"吃馒头"**);  } } **class** S4 **extends** F{  @Override  **public void** eat() {  System.***out***.println(**"吃烤肉"**);  } } |

### 2.3 多态之向下转型

将父类的引用变量转换为子类类型，称为向下转型（downcasting）。向下转型后就可以调用子类特有的方法了。

* 需要进行强制转换Chinese ch = (Chinese)pro;
* 强制转换不是做手术，必须转换成真实子类型，否则ClassCastException；
* 向下转型之前肯定发生了向上转型
* 为了避免ClassCastException,向下转型之前使用instanceof先判断一下

pro instanceof Italian  
对象 instanceof 类或者接口

使用instancof的前提：左边的对象和右边的类型在继承树上有上下级关系

#### 【示例3】向下转型

|  |
| --- |
| **package** com.bjsxt.polymorphismDemo2;  **public class** Test1 {  **public static void** main(String[] args) {  */\*  \* 父类对象一般不能声明成子类对象 父类不具备子类的功能  \* 可以通过强制转换将父类对象转为子类对象 通过编译  \* 如果父类对象原本就是由子类声明而成的,那就可以转换回子类  \* \*/* F f =**new** S1();  S1 s1 =(S1)f;   *methodX*(**new** S3());  }   **public static void** methodX(F f){  *// 如何判断传入的对象到底是哪个子类的对象呢?  // instanceof 判断一个对象是否属于某个类* System.***out***.println(f **instanceof** S1);  System.***out***.println(f **instanceof** S2);  System.***out***.println(f **instanceof** S3);  System.***out***.println(f **instanceof** S4);  } }  **class** F{  **public void** eat(){  System.***out***.println(**"吃大米"**);  } }  **class** S1 **extends** F{  **public void** eat(){  System.***out***.println(**"吃面条"**);  }  **public void** play(){  System.***out***.println(**"打游戏"**);  } } **class** S2 **extends** F{  @Override  **public void** eat() {  System.***out***.println(**"吃饺子"**);  } } **class** S3 **extends** F{  @Override  **public void** eat() {  System.***out***.println(**"吃馒头"**);  } } **class** S4 **extends** F{  @Override  **public void** eat() {  System.***out***.println(**"吃烤肉"**);  } } |

### 2.4 父类作为方法参数

里氏代换原则;

白马,马也

乘白马,乘马也

黑马,马也

成黑马,乘马也

父类:马 子类:白马 黑马

父类能够出现的地方法,子类就能出现

娣,美人也

爱娣,非爱美人也

父类:美人 子类:娣

子类出现的地方,父类一般不能出现

通过对子类父类转型的学习,我们知道,子类可以正常转换为父类,那么在我们实际的编码中,该种转换的应用场景之一就是父类数据类型作为方法参数,其任意子类对象就可以作为方法的实参传入。

#### 【示例4】使用父类作为方法参数

张三去4S试驾车 奥迪 BMW BYD

|  |
| --- |
| **package** com.bjsxt.polymorphismDemo2;  **public class** Test2 {  **public static void** main(String[] args) {  Audi audi=**new** Audi();  BMW bmw=**new** BMW();  BYD byd=**new** BYD();   Person p=**new** Person();  p.dirve(audi);  p.dirve(bmw);  p.dirve(byd);  } } **class** Person{   **public void** dirve(Car car){  System.***out***.println(**"张三试驾"**);  car.run();  } }  **class** Car{  **public void** run(){  System.***out***.println(**"一辆汽车正在飞奔"**);  } } **class** Audi **extends** Car{  @Override  **public void** run() {  System.***out***.println(**"一辆奥迪正在飞奔"**);  } } **class** BMW **extends** Car{  @Override  **public void** run() {  System.***out***.println(**"一辆BMW正在飞奔"**);  } } **class** BYD **extends** Car{  @Override  **public void** run() {  System.***out***.println(**"一辆BYD正在飞奔"**);  } } |

### 2.5 父类作为方法返回值-简单工厂模式

不仅可以使用父类做方法的形参，还可以使用父类做方法的返回值类型，真实返回的对象可以是该类的任意一个子类对象。

案例：定义一个汽车工厂对象 ，这个工厂中可以统一返回某一类对象

#### 【示例5】使用父类作为返回值

|  |
| --- |
| **package** com.bjsxt.polymorphismDemo3;  **public class** Test1 {  **public static void** main(String[] args) {  Car car = CarFactory.*getCar*(**"BYD"**);  car.run();   *// 判断返回的对象到底是哪个子类声明而来的?* System.***out***.println(car **instanceof** Audi);  System.***out***.println(car **instanceof** BMW);  System.***out***.println(car **instanceof** BYD);  }  } *// 生产汽车的工厂* **class** CarFactory{  */\*  父类作为方法参数,其任何一个子类对象都可以是返回的结果  \*/* **public static** Car getCar(String brand){  **if**(brand.equals(**"奥迪"**)){  Audi audi = **new** Audi();  **return** audi;  }**else if**(**"BMW"**.equals(brand)){  **return new** BMW();  }**else**{  **return new** BYD();  }  } }  **class** Car{  **public void** run(){  System.***out***.println(**"一辆汽车正在飞奔"**);  } } **class** Audi **extends** Car{  @Override  **public void** run() {  System.***out***.println(**"一辆奥迪正在飞奔"**);  } } **class** BMW **extends** Car{  @Override  **public void** run() {  System.***out***.println(**"一辆BMW正在飞奔"**);  } } **class** BYD **extends** Car{  @Override  **public void** run() {  System.***out***.println(**"一辆BYD正在飞奔"**);  } } |

以上代码其实是简单工厂模式的实现，它是解决大量对象创建问题的一个解决方案。将创建和使用分开，工厂负责创建，使用者直接调用即可。简单工厂模式的基本要求是

* 定义一个static方法，通过类名直接调用
* 返回值类型是父类类型，返回的可以是其任意子类类型
* 传入一个字符串类型的参数，工厂根据参数创建对应的子类产品

|  |
| --- |
| **扩展： 设计模式**   * 有23中经典的设计模式，是一套被多数人知晓、经过分类编目的、反复使用的优秀代码设计经验的总结。每个设计模式均是特定环境下特定问题的处理方法。 * 面向对象设计原则是面向对象设计的基石，面向对象设计质量的依据和保障，设计模式是面向对象设计原则的经典应用。 * 简单工厂模式并不是23中经典模式的一种，是其中工厂方法模式的简化版 |

### 2.6抽象方法和抽象类

**·抽象方法**

使用abstract修饰的方法，没有方法体，只有声明。定义的是一种“规范”，就是告诉子类必须要给抽象方法提供具体的实现。

**·抽象类**

使用abstract修饰的类。通过abstract方法定义规范，然后要求子类必须定义具体实现。通过抽象类，我们就可以做到严格限制子类的设计，使子类之间更加通用。

问题1：Animal an = new Animal();没有一种动物，名称是Animal，所以Animal不能被实例化

解决：抽象类  
问题2：子类必须重写父类的某个方法，否则出现编译错误  
解决：抽象方法

|  |
| --- |
| **package** com.bjsxt.abstractDemo;  **public class** Test {  **public static void** main(String[] args) {  Erge erge=**new** XiaodiB();  erge.doSomethingBad();  erge.toujimogou();  } }**abstract class** Ladeng{  **public abstract void** doSomethingBad();  **public void** eat(){   }  **public** Ladeng(){   }  }  **abstract class** Erge **extends** Ladeng {  **public abstract void** toujimogou();  **public** Erge(){  **super**();  } }  **class** XiaodiB **extends** Erge {  **public void** doSomethingBad(){  System.***out***.println(**"炸地铁"**);  }   @Override  **public void** toujimogou() {  System.***out***.println(**"坑蒙拐骗偷"**);  }   **public** XiaodiB(){  **super**();  } } |

**抽象类的使用要点:**

* 有抽象方法的类只能定义成抽象类
* 抽象类不能实例化，即不能用new来实例化抽象类。
* 抽象类必须有构造方法，创建子类对象的时候使用
* 一个抽象类至少0个抽象方法，至多（所有的方法都是抽象方法）个抽象方法
* 子类必须重写父类的抽象方法，不重写就提示编译错误；或者子类也定义为抽象类
* override 重写 implements 实现  
   父类的方法是抽象的，需要子类实现；父类的方法不是抽象的，子类可以重写

### 本节作业

1. 向下转型的相关注意事项
2. 面向对象设计原则有哪些
3. 23种设计模式分为哪3大类，分别是哪些

## 第三节 Object类和组合关系

### 3.1 Object类的介绍

Object类是所有Java类的根基类，也就意味着所有的Java对象都拥有Object类的属性和方法。如果在类的声明中未使用extends关键字指明其父类，则默认继承Object类。Object类中定义了一些JAVA所有的类都必须具有的一些方法

**面试题目：请你写出Object类的6个方法**

|  |  |
| --- | --- |
| **方法摘要** | |
| boolean | **[equals](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \l "equals(java.lang.Object))**([Object](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \o "java.lang 中的类) obj)  指示其他某个对象是否与此对象“相等”。 |
| [Class](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Class.html" \o "java.lang 中的类)<?> | **[getClass](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \l "getClass())**() 返回此 Object 的运行时类。 |
| int | **[hashCode](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \l "hashCode())**() 返回该对象的哈希码值。 |
| void | **[notify](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \l "notify())**()  唤醒在此对象监视器上等待的单个线程。 |
| void | **[notifyAll](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \l "notifyAll())**()    唤醒在此对象监视器上等待的所有线程。 |
| [String](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[toString](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \l "toString())**() 返回该对象的字符串表示。 |
| void | **[wait](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \l "wait())**()  在其他线程调用此对象的 [notify()](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \l "notify()) 方法或 [notifyAll()](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \l "notifyAll()) 方法前，导致当前线程等待。 |
| void | **[wait](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \l "wait(long))**(long timeout)    在其他线程调用此对象的 [notify()](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \l "notify()) 方法或 [notifyAll()](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \l "notifyAll()) 方法，或者超过指定的时间量前，导致当前线程等待。 |
| void | **[wait](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \l "wait(long, int))**(long timeout, int nanos)     在其他线程调用此对象的 [notify()](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \l "notify()) 方法或 [notifyAll()](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \l "notifyAll()) 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量前，导致当前线程等待。 |

以上方法是Object类的所有方法吗？不是 是Object类所有的public方法。 除此之外可能还有private、protected、默认的方法

|  |  |
| --- | --- |
| **方法摘要** | |
| protected  [Object](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \o "java.lang 中的类) | **[clone](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \l "clone())**()      创建并返回此对象的一个副本。 |
| protected  void | **[finalize](mk:@MSITStore:C:\\京南2019\\2.尚学堂京南校区资料库\\02.开发文档列表\\01.JDK%20API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \l "finalize())**()  当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。 |

|  |
| --- |
| **扩展：native关键字**   * 一个native方法就是一个Java调用非Java代码的接口。一个native方法是指该方法的实现由非Java语言实现，比如用C或C++实现。 * 在定义一个native方法时，并不提供实现体，因为其实现体是由非Java语言在外面实现的。Java语言本身不能对操作系统底层进行访问和操作，但是可以通过JNI接口调用其他语言来实现对底层的访问。 * JNI是Java本机接口（Java Native Interface），是一个本机编程接口，它是Java软件开发工具箱（Java Software Development Kit，SDK）的一部分。JNI允许Java代码使用以其他语言编写的代码和代码库。Invocation API（JNI的一部分）可以用来将Java虚拟机（JVM）嵌入到本机应用程序中，从而允许程序员从本机代码内部调用Java代码。 |

### 3.2 ==和equals方法

“==”代表比较双方是否相同。如果是基本类型则表示值相等，如果是引用类型则表示地址相等即是同一个对象。

Object类中定义有：public boolean equals(Object obj)方法，提供定义“对象内容相等”的逻辑。比如，判断两个Dog是否是一个Dog，要求color、age、nickName等所有属性都相同。

Object 的 equals 方法默认就是比较两个对象的hashcode，是同一个对象的引用时返回 true 否则返回 false。显然，这无法满足子类的要求，可根据要求重写equals方法。

测试==

|  |
| --- |
| **public class** Test1 {  **public static void** main(String[] args) {  */\*  \* ==  \* 两端如果是基本数据类型,那么就是单纯判断值是否相同  \* 两端是引用类型,那么判断的引用的地址是否相同,判断是否指向内存上的同一个数据  \* \*/* **int** a=10;  **int** b=10;  System.***out***.println(a==b);    String s =**new** String(**"asdf"**);  String s2=**new** String(**"asdf"**);  System.***out***.println(s);  System.***out***.println(s2);  System.***out***.println(s==s2);   String s3=**"abc"**;  String s4=**"abc"**;  System.***out***.println(s3==s4);   System.***out***.println(s.equals(s2));  System.***out***.println(s3.equals(s4));   } } |

#### 【示例7】重写equals方法

|  |
| --- |
| **package** com.bjsxt.objectDemo1;  **import** com.bjsxt.homework.Person;  **import** java.util.Objects;  **public class** Test2 {  **public static void** main(String[] args) {  Dog d1=**new** Dog(**"金毛"**,**"金黄"**,10);  Dog d2=**new** Dog(**"金毛"**,**"金黄"**,10);   System.***out***.println(d1==d2);  */\*\*  \* 判断自定义类的两个对象属性值是否全部相同  \* 重写equals方法即可 alt+insert  \* 使用equals方法去判断  \*   \*/* System.***out***.println(d1.equals(d2));   String s1 =**new** String(**"asdf"**);  String s2 =**new** String(**"asdf"**);  System.***out***.println(s1==s2);  System.***out***.println(s1.equals(s2));  } }  **class** Dog{  **private** String **type**;  **private** String **color**;  **private int month**;   **public void** showInfo(){  System.***out***.println(**color**+**"颜色的"**+**month**+**"个月的"**+**type**);  }  *// alt +insert* @Override  **public boolean** equals(Object o) {  **if** (**this** == o) {  **return true**;  }  **if** (o == **null** || getClass() != o.getClass()){  **return false**;  }  Dog dog = (Dog) o;  **return month** == dog.**month** &&  Objects.*equals*(**type**, dog.**type**) &&  Objects.*equals*(**color**, dog.**color**);  }    **public** Dog() {  }   **public** Dog(String type, String color, **int** month) {  **this**.**type** = type;  **this**.**color** = color;  **this**.**month** = month;  }   **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** getMonth() {  **return month**;  }   **public void** setMonth(**int** month) {  **this**.**month** = month;  } } |

### 3.3 hashCode方法

哈希码:hash码,散列码,是一种无序不重复的一串十六进制数据,Object类中定义的方法作用就是为每一个对象生成一个独立的哈希码,用以区分每个对象,我们可以通过重写的方式自定义哈希码的声明规则,让其和属性值相关联.一般在重写equals方法时,都会重写hashcode方法

#### 【示例8】重写hashCode方法

|  |
| --- |
| **package** com.bjsxt.objectDemo2;  **import** java.util.Objects;  **public class** Test1 {  **public static void** main(String[] args) {  Dog d1=**new** Dog(**"金毛"**,**"金黄"**,10);  Dog d2=**new** Dog (**"金毛"**,**"金黄"**,10);   System.***out***.println(d1==d2);  System.***out***.println(d1.equals(d2));   System.***out***.println(d1.hashCode());  System.***out***.println(d2.hashCode());    } }  **class** Dog{  **private** String **type**;  **private** String **color**;  **private int month**;   **public void** showInfo(){  System.***out***.println(**color**+**"颜色的"**+**month**+**"个月的"**+**type**);  }   @Override  **public boolean** equals(Object o) {  **if** (**this** == o) **return true**;  **if** (o == **null** || getClass() != o.getClass()) **return false**;  Dog dog = (Dog) o;  **return month** == dog.**month** &&  Objects.*equals*(**type**, dog.**type**) &&  Objects.*equals*(**color**, dog.**color**);  }   *// 只要对象的属性值相同,返回的哈希码就是相同的* @Override  **public int** hashCode() {  **return** Objects.*hash*(**type**, **color**, **month**);  }   **public** Dog() {  }   **public** Dog(String type, String color, **int** month) {  **this**.**type** = type;  **this**.**color** = color;  **this**.**month** = month;  }   **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** getMonth() {  **return month**;  }   **public void** setMonth(**int** month) {  **this**.**month** = month;  } } |

### 3.4 toString方法

Object 类中定义类一个方便我们快捷查看对象属性信息的一个方法,toString的目的是返回一个对象的字符串表达形式,Object类中定义的方法规则为类的全路径名+哈希码,我们可以通过重写的方式重新定义该方法

#### 【示例9】重写toString方法

|  |
| --- |
| **package** com.bjsxt.objectDemo2;  **import** java.util.Objects;  **public class** Test1 {  **public static void** main(String[] args) {  Dog d1=**new** Dog(**"金毛"**,**"金黄"**,10);  Dog d2=**new** Dog (**"金毛"**,**"金黄"**,10);   System.***out***.println(d1==d2);  System.***out***.println(d1.equals(d2));   System.***out***.println(d1.hashCode());  System.***out***.println(d2.hashCode());   String d1String = d1.toString();  System.***out***.println(d1String);   } }  **class** Dog{  **private** String **type**;  **private** String **color**;  **private int month**;     @Override  **public boolean** equals(Object o) {  **if** (**this** == o) **return true**;  **if** (o == **null** || getClass() != o.getClass()) **return false**;  Dog dog = (Dog) o;  **return month** == dog.**month** &&  Objects.*equals*(**type**, dog.**type**) &&  Objects.*equals*(**color**, dog.**color**);  }   *// 只要对象的属性值相同,返回的哈希码就是相同的* @Override  **public int** hashCode() {  **return** Objects.*hash*(**type**, **color**, **month**);  }   @Override  **public** String toString() {  **return "Dog{"** +  **"type='"** + **type** + **'\''** +  **", color='"** + **color** + **'\''** +  **", month="** + **month** +  **'}'**;  }   **public** Dog() {  }   **public** Dog(String type, String color, **int** month) {  **this**.**type** = type;  **this**.**color** = color;  **this**.**month** = month;  }   **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** getMonth() {  **return month**;  }   **public void** setMonth(**int** month) {  **this**.**month** = month;  } } |

原型模式:根据一个原型克隆出多个属性相同但是在内存上是独立的对象

创建一个Dog类 实例化Dog 对象,思考如何通过一个Dog对象复刻出多个属性值相同的Dog对象 ,通过clone方法(浅克隆)完成

|  |
| --- |
| **package** com.bjsxt.objectDemo;  **import** java.util.Objects;  **public class** Dog **implements** Cloneable{  **private** String **type**;  **private** String **color**;  **private int month**;   @Override  **public** Object clone() **throws** CloneNotSupportedException {  **return super**.clone();  }   @Override  **public** String toString() {  **return "Dog{"** +  **"type='"** + **type** + **'\''** +  **", color='"** + **color** + **'\''** +  **", month="** + **month** +  **'}'**;  }   **public** Dog(String type, String color, **int** month) {  **this**.**type** = type;  **this**.**color** = color;  **this**.**month** = month;  }   **public** Dog() {  }   **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** getMonth() {  **return month**;  }   **public void** setMonth(**int** month) {  **this**.**month** = month;  } }  测试代码  **package** com.bjsxt.objectDemo;  **public class** Test2 {  **public static void** main(String[] args) **throws** CloneNotSupportedException {   */\*  \* clone  \* 1直接重写即可  \* 2要克隆的对象的所属类必须实现 Cloneable接口  \* 3clone方法重写后,推荐将访问修饰符修改为public  \*  \* \*/* Dog d1=**new** Dog(**"金毛"**,**"金色"**,10);  System.***out***.println(d1);  Dog d2=(Dog)d1.clone();  System.***out***.println(d2);  System.***out***.println(d1==d2);   d2.setType(**"泰迪"**);  System.***out***.println(d1);  System.***out***.println(d2);    } } |

### 本节作业

1. 描述组合关系的概念
2. 通过组合关系体现
3. Object类的常用方法及其作用
4. 说明继承条件下构造方法的执行过程

动物园丢失一只大熊猫,该熊猫color黑白色,month 30个月,name 猫淘淘 ,公安部门在园区外面发现很多野生大熊猫,判断哪一只是丢失的大熊猫;

|  |
| --- |
| **package** com.bjsxt.objectDemo3;  **import** java.util.Objects;  **public class** Test1 {  **public static void** main(String[] args) {  *// 丢失的那只大熊猫* Panda panda=**new** Panda(**"黑白"**,**"猫淘淘"**,30);   *// 发现几只大熊猫* Panda panda1=**new** Panda(**"黑白"**,**"猫淘淘"**,30);  Panda panda2=**new** Panda(**"黄白"**,**"熊大"**,30);  Panda panda3=**new** Panda(**"棕白"**,**"熊二"**,30);   *findPanda*(panda,panda3,panda2,panda1);  }  **public static void** findPanda(Panda target,Panda ... pandas){  System.***out***.println(**"丢失的大熊猫信息为:"**+ target.toString());  **for**(Panda panda:pandas){  System.***out***.println(**"当前这只熊猫信息为:"**+panda.toString());  **if**(target.equals(panda)){  System.***out***.println(**"这只熊猫就是丢失那只"**);  **break**;  }   }   } } **class** Panda{  **private** String **color**;  **private** String **name**;  **private int month**;  */\*  该类两个对象的属性值相同,equals方法即返回true 否则 返回false  \*/* @Override  **public boolean** equals(Object o) {  **if** (**this** == o) **return true**;  **if** (o == **null** || getClass() != o.getClass()) **return false**;  Panda panda = (Panda) o;  **return month** == panda.**month** &&  Objects.*equals*(**color**, panda.**color**) &&  Objects.*equals*(**name**, panda.**name**);  }  */\*  该类两个对象的属性值相同,hashCode 返回的哈希码即相同  \*/* @Override  **public int** hashCode() {  **return** Objects.*hash*(**color**, **name**, **month**);  }   @Override  **public** String toString() {  **return "Panda{"** +  **"color='"** + **color** + **'\''** +  **", name='"** + **name** + **'\''** +  **", month="** + **month** +  **'}'**;  }   **public** Panda() {  }   **public** Panda(String color, String name, **int** month) {  **this**.**color** = color;  **this**.**name** = name;  **this**.**month** = month;  }   **public** String getColor() {  **return color**;  }   **public void** setColor(String color) {  **this**.**color** = color;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public int** getMonth() {  **return month**;  }   **public void** setMonth(**int** month) {  **this**.**month** = month;  }  } |