

# Java面向对象

## 一，类与对象

今天学习一下关于Java的三大特性，封装，继承，多态。其实关于三大特性对于从事编程人员来说都是基本的了，毕竟只要接触Java这些都是先要认识的，接下来就系统学习一下。

### 1.1，面向对象简述

面向对象是一种现在最为流行的程序设计方法，几乎现在的所有应用都以面向对象为主了，最早的面向对象的概念实际上是由IBM提出的，在70年代的Smalltalk语言之中进行了应用，后来根据面向对象的设计思路，才形成C++，而由C++产生了Java这门面向对象的编程语言。

但是在面向对象设计之前，广泛采用的是面向过程，面向过程只是针对于自己来解决问题。面向过程的操作是以程序的基本功能实现为主，实现之后就完成了，也不考虑修改的可能性，面向对象，更多的是要进行子模块化的设计，每一个模块都需要单独存在，并且可以被重复利用，所以，面向对象的开发更像是一个具备标准的开发模式。

在面向对象定义之中，也规定了一些基本的特征：（1）封装：保护内部的操作不被破坏；（2）继承：在原本的基础之上继续进行扩充；（3）多态：在一个指定的范围之内进行概念的转换。

对于面向对象的开发来讲也分为三个过程：OOA（面向对象分析）、OOD（面向对象设计）、OOP（面向对象编程）。

### 1.2，类与对象的基本概念

类与对象是整个面向对象中最基础的组成单元。

**类**：是抽象的概念集合，表示的是一个共性的产物，类之中定义的是属性和行为（方法）；**对象**：对象是一种个性的表示，表示一个独立的个体，每个对象拥有自己独立的属性，依靠属性来区分不同对象。

可以一句话来总结出类和对象的区别：类是对象的抽象，对象是类的实例。类只有通过对象才可以使用，而在开发之中应该先产生类，之后再产生对象。类不能直接使用，对象是可以直接使用的。

### 1.3，类与对象的定义和使用

在Java中定义类，使用关键字class完成。语法如下：

```
class 类名称
{
    属性（变量）；
    行为（方法）；
    构造函数；
}
```

定义一个Person类

```

class Person
{
    // 类名称首字母大写
    String name ;
    int age ;
    public void tell()
    {
        // 没有static
        System.out.println("姓名：" + name + "，年龄：" + age) ;
    }
}

```

类定义完成之后，肯定无法直接使用。如果要使用，必须依靠对象，那么由于类属于引用数据类型，所以对象的产生格式（两种格式）如下：

（1）格式一：声明并实例化对象

```

类名称 对象名称 = new 类名称 () ;

```

（2）格式二：先声明对象，然后实例化对象：

```

类名称 对象名称 = null ;
对象名称 = new 类名称 () ;

```

引用数据类型与基本数据类型最大的不同在于：引用数据类型需要内存的分配和使用。所以，关键字new的主要功能就是分配内存空间，也就是说，只要使用引用数据类型，就要使用关键字new来分配内存空间。

当一个实例化对象产生之后，可以按照如下的方式进行类的操作：**对象.属性**：表示调用类之中的属性；**对象.方法()**：表示调用类之中的方法。

```

class Person
{
    String name ;
    int age ;
    public void get()
    {
        System.out.println("姓名：" + name + "，年龄：" + age);
    }
}

public class TestDemo
{
    public static void main(String args[])
    {
        Person per = new Person() ;// 声明并实例化对象
        per.name = "张三" ;//操作属性内容
        per.age = 30 ;//操作属性内容
        per.get() ;//调用类中的get()方法
    }
}

```

运行结果：

姓名：张三，年龄：30

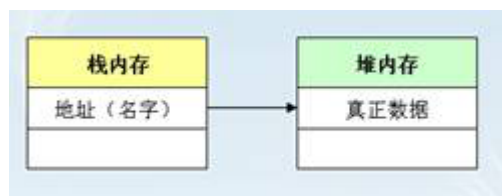
以上完成了一个类和对象的操作关系，下面换另外一个操作来观察一下：

```
class Person
{
    String name ;
    int age ;
    public void get()
    {
        System.out.println("姓名：" + name + "，年龄：" + age);
    }
}

public class TestDemo
{
    public static void main(String args[])
    {
        Person per = null;//声明对象
        per = new Person() ;//实例化对象
        per.name = "张三" ;//操作属性内容
        per.age = 30 ;//操作属性内容
        per.get() ;//调用类中的get()方法
    }
}
```

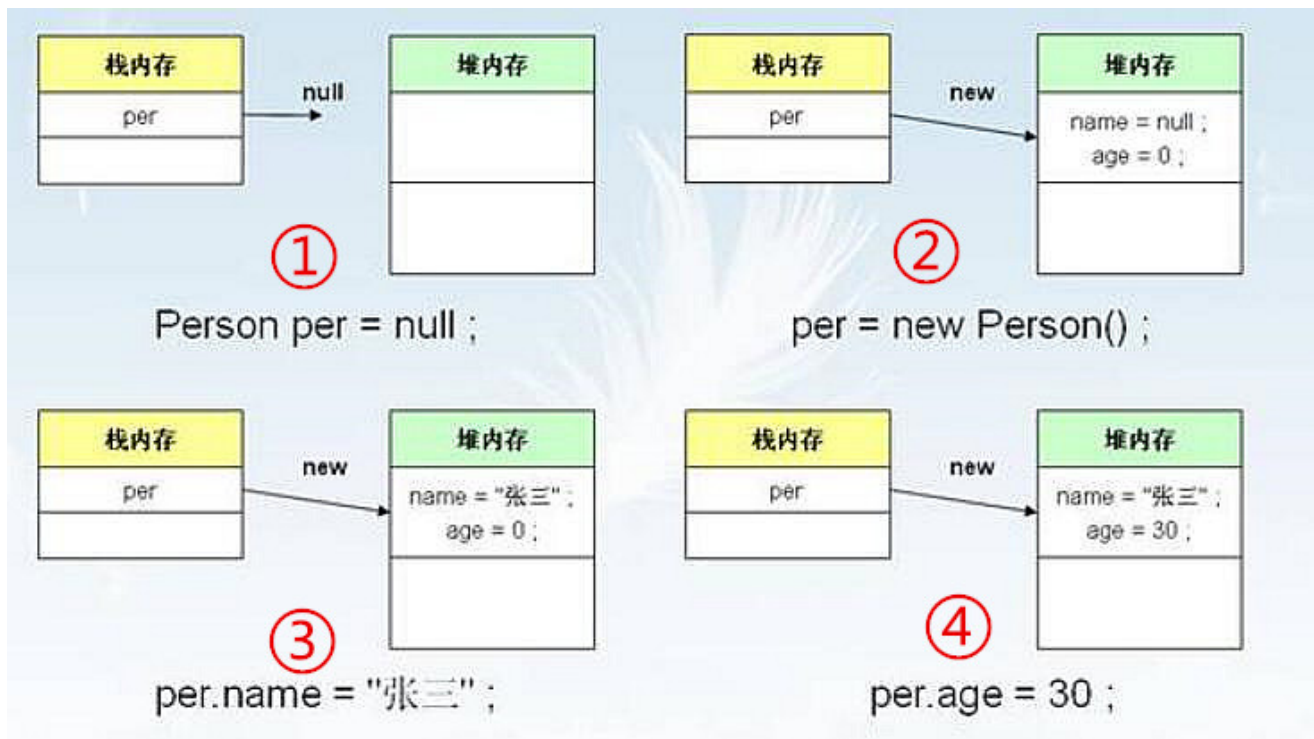
那么，问题来了，以上两种不同的实例化方式有什么区别呢？我们从内存的角度分析。首先，给出两种内存空间的概念：

- (1) 堆内存：保存对象的属性内容。堆内存需要用new关键字来分配空间；
- (2) 栈内存：保存的是堆内存的地址（在这里为了分析方便，可以简单理解为栈内存保存的是对象的名字）。



任何情况下，只要看见关键字new，都表示要分配新的堆内存空间，一旦堆内存空间分配了，里面就会有类中定义的属性，并且属性内容都是其对应数据类型的默认值。

于是，上面两种对象实例化对象方式内存表示如下：



两种方式的区别在于①②，第一种声明并实例化的方式实际就是①②组合在一起，而第二种先声明然后实例化是把①和②分步骤来。

另外，如果使用了没有实例化的对象，结果如何？如下：

```
class Person
{
    String name ;
    int age ;
    public void get()
    {
        System.out.println("姓名：" + name + "，年龄：" + age);
    }
}

public class TestDemo
{
    public static void main(String args[])
    {
        Person per = null;//声明对象
        //per = new Person() ;//实例化对象
        per.name = "张三" ;//操作属性内容
        per.age = 30 ;//操作属性内容
        per.get() ;//调用类中的get()方法
    }
}
```

运行结果：

```
Exception in thread "main" java.lang.NullPointerException
```

此时，程序只声明了Person对象，但并没有实例化Person对象（只有了栈内存，并没有对应的堆内存空间），则程序在编译的时候不会出现任何的错误，但是在执行的时候出现了上面的错误信息。这个错误信息表示的是“NullPointerException（空指向异常）”，这种异常只要是应用数据类型都有可能出现。

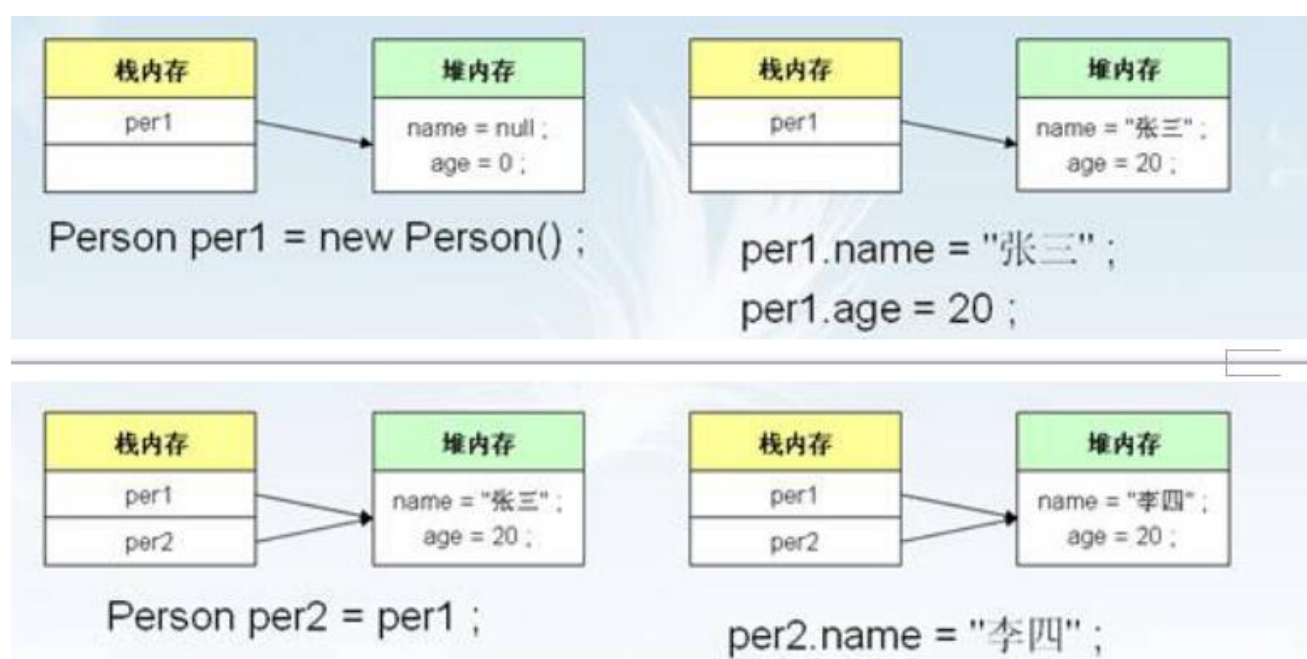
引用传递的精髓：同一块堆内存空间，可以同时被多个栈内存所指向，不同的栈可以修改同一块堆内存的内容。

下面通过若干个程序，以及程序的内存分配图，来进行代码的讲解。

我们来看一个范例：

```
class Person
{
    String name ;
    int age ;
    public void tell()
    {
        System.out.println("姓名：" + name + "，年龄：" + age) ;
    }
}
public class TestDemo
{
    public static void main(String args[])
    {
        Person per1 = new Person() ;           // 声明并实例化对象
        per1.name = "张三" ;
        per1.age = 20 ;
        Person per2 = per1 ; // 引用传递
        per2.name = "李四" ;
        per1.tell() ;
    }
}
```

对应的内存分配图如下：



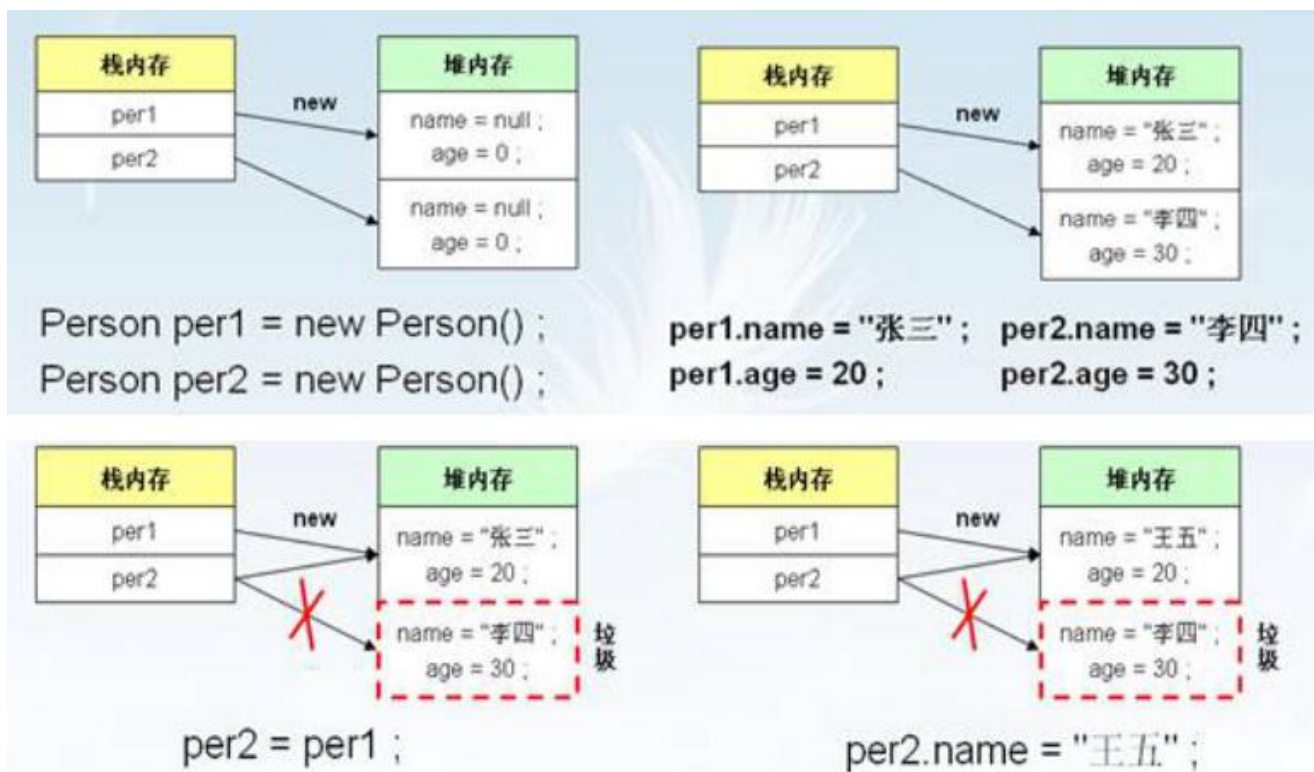
再来看另外一个：

```

class Person
{
    String name ;
    int age ;
    public void tell()
    {
        System.out.println("姓名：" + name + "，年龄：" + age) ;
    }
}
public class TestDemo
{
    public static void main(String args[])
    {
        Person per1 = new Person() ;           // 声明并实例化对象
        Person per2 = new Person() ;
        per1.name = "张三" ;
        per1.age = 20 ;
        per2.name = "李四" ;
        per2.age = 30 ;
        per2 = per1 ;// 引用传递
        per2.name = "王五" ;
        per1.tell() ;
    }
}

```

对应的内存分配图如下：



垃圾：指的是在程序开发之中没有任何对象所指向的一块堆内存空间，这块空间就成为垃圾，所有的垃圾将等待GC（垃圾收集器）不定期的进行回收与空间的释放。

## 1.4， 方法的定义和使用

方法的定义

```
[修饰符] 返回值类型 方法名 ( 参数类型 参数1 , 参数类型 参数2 ) [throws 异常]
{
    // 方法体 ;
    [return 返回值类型 ; ]
}
```

说明:

- 修饰符：访问控制符public|protected|private、静态修饰符static、抽象修饰符abstract、最终修饰符final等。
- 返回值类型: void(无返回值的方法) 或 数据类型 ( 有返回值的方法 )。
- 参数列表：多个参数时逗号分割也可以没有参数。

## 1.5, 构造函数

构造器 ( Constructor )：是一个用来创建对象的特殊方法，用来初始化对象的属性。

- 构造器的名字与类名相同构造器
- 没有返回值构
- 造器所包含的语句用来对所创建的对象进行初始化
- 没有参数的构造器称为“无参构造器”每个Java类都至少有一个构造器，如果该类没有显式地声明任何构造器，系统会默认地为该类提供一个不包含任何语句的无参构造器

```
public class Point
{
    public int x = 0;
    public int y = 0;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public Point()
    {
    }
}
```

## 二，封装

先来说说特性之一：封装

## 2.1，什么是封装

封装（Encapsulation）是面向对象方法的重要原则，就是把对象的属性和操作（或服务）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。

- 将类的某些信息隐藏在类的内部，不允许外部程序进行直接的访问调用。
- 通过该类提供的方法来实现对隐藏信息的操作和访问。
- 隐藏对象的信息。
- 留出访问的对外接口。

举个比较通俗的例子，比如我们的USB接口。如果我们需要外设且只需要将设备接入USB接口中，而内部是如何工作的，对于使用者来说并不重要。而USB接口就是对外提供的访问接口。

说了这么多，那为什么使用封装？

## 2.2，封装的特点

- 对成员变量实行更准确的控制。
- 封装可以隐藏内部程序实现的细节。
- 良好的封装能够减少代码之间的耦合度。
- 外部成员无法修改已封装好的程序代码。
- 方便数据检查，有利于保护对象信息的完整性，同时也提高程序的安全性。
- 便于修改，提高代码的可维护性。

## 2.3，封装的使用

- 使用private修饰符，表示最小的访问权限。
- 对成员变量的访问，统一提供setXXX，getXXX方法。

定义一个Student实体对象类，属性为 [long id],[String name],[Integer sex]，

为其提供 get set 方法：

```
public class Student
{

    private Long id;
    private String name;
    private Integer sex;

    public Long getId()
    {
        return id;
    }

    public void setId(Long id)
    {
        this.id = id;
    }

    public String getName()
    {
        return name;
    }
}
```



```
    }

    public void setName(String name)
    {
        this.name = name;
    }

    public Integer getSex()
    {
        return sex;
    }

    public void setSex(Integer sex)
    {
        this.sex = sex;
    }
}
```

分析：对于上面的一个实体对象，我想大家都已经很熟悉了。将对象中的成员变量进行私有化，外部程序是无法访问的。但是我们对外提供了访问的方式，就是set和get方法。

而对于这样一个实体对象，外部程序只有赋值和获取值的权限，是无法对内部进行修改，因此我们还可以在内部进行一些逻辑上的判断等，来完成我们业务上的需要。

到这里应该就明白封装对于我们的程序是多么重要。下面再来说说继承的那点事。

## 三，继承

---

### 3.1，什么是继承

继承就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为。当然，如果在父类中拥有私有属性(private修饰)，则子类是不能被继承的。

### 3.2，继承的特点

#### 1，关于继承的注意事项：

只支持单继承，即一个子类只允许有一个父类，但是可以实现多级继承，及子类拥有唯一的父类，而父类还可以再继承。

子类可以拥有父类的属性和方法。

子类可以拥有自己的属性和方法。

子类可以重写覆盖父类的方法。

#### 2，继承的特点：

提高代码复用性。

父类的属性方法可以用于子类。

可以轻松的定义子类。

使设计应用程序变得简单。

## 3.3，继承的使用

**1，在父子类关系继承中，如果成员变量重名，则创建子类对象时，访问有两种方式。**

a，直接通过子类对象访问成员变量

等号左边是谁，就优先使用谁，如果没有就向上找。

b，间接通过成员方法访问成员变量

该方法属于谁，谁就优先使用，如果没有就向上找。

```
public class FU
{
    int numFU = 10;
    int num = 100;
    public void method()
    {
        System.out.println("父类成员变量：" + numFU);
    }
    public void methodFU()
    {
        System.out.println("父类成员方法!");
    }
}
```

```
public class Zi extends FU
{
    int numZi = 20;
    int num = 200;
    public void method()
    {
        System.out.println("子类成员变量：" + numFU);
    }
    public void methodZi()
    {
        System.out.println("子类方法!");
    }
}
```

```
public class ExtendDemo
{
```

```

public static void main(String[] args)
{
    FU fu = new FU();
    // 父类的实体对象只能调用父类的成员变量
    System.out.println("父类：" + fu.numFU);    // 结果：10

    Zi zi = new Zi();
    System.out.println("调用父类：" + zi.numFU); // 结果：10
    System.out.println("子类：" + zi.numZi);    // 结果：20

    /** 输出结果为200，证明在重名情况下，如果子类中存在则优先使用，
     *  如果不存在则去父类查找，但如果父类也没有那么编译期就会报错。
     */
    System.out.println(zi.num); // 结果：200
    /**
     *  通过成员方法调用成员变量
     */
    zi.method();    // 结果：10
}
}

```

## 2，同理：

成员方法也是一样的，创建的对象是谁，就优先使用谁，如果没有则直接向上找。**注意事项：**

无论是成员变量还是成员方法，如果没有都是向上父类中查找，绝对不会向下查找子类的。

## 3，在继承关系中，关于成员变量的使用：

局部成员变量：直接使用 本类成员变量：this.成员变量 父类成员变量：super.父类成员变量

```

int numZi = 10;
public void method()
{
    int numMethod = 20;
    System.out.println(numMethod); // 访问局部变量
    System.out.println(this.numZi); // 访问本类成员变量
    System.out.println(super.numFu); // 访问本类成员变量
}

```

## 3.4，重写，重载

### 重写(override)

是子类对父类的允许访问的方法的实现过程进行重新编写，返回值和形参都不能改变。**即外壳不变，核心重写！**

```

//父类
class Animal
{
    public void move()
    {
        System.out.println("动物行走！");
    }
}

```

```

}
//子类 dog
class Dog extends Animal
{
    public void move()
    {
        System.out.println("狗可以跑和走");
    }
}
//main方法
public class TestDog
{
    public static void main(String args[])
    {
        Animal a = new Animal(); // Animal 对象
        Animal b = new Dog(); // Dog 对象
        a.move();// 执行 Animal 类的方法
        b.move();//执行 Dog 类的方法
    }
}

```

重写的规则：

- 1，参数列表必须与被重写方法相同。
- 2，访问权限不能比父类中被重写的方法的访问权限更低（public>protected>(default)>private）。
- 3，父类成员的方法只能被它的子类重写。
- 4，被final修饰的方法不能被重写。
- 5，构造方法不能

### 重载(overload)

是在一个类里面，方法名字相同，而参数不同。返回类型可以相同也可以不同。每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。

最常用的地方就是构造器的重载。

```

public class Overloading
{
    public int test()
    {
        System.out.println("test1");
        return 1;
    }
    public void test(int a)
    {
        System.out.println("test2");
    }
    //以下两个参数类型顺序不同
    public String test(int a,String s)
    {
        System.out.println("test3");
    }
}

```

```

        return "returntest3";
    }
    public String test(String s,int a)
    {
        System.out.println("test4");
        return "returntest4";
    }
    public static void main(String[] args)
    {
        Overloading o = new Overloading();
        System.out.println(o.test());
        o.test(1);
        System.out.println(o.test(1,"test3"));
        System.out.println(o.test("test4",1));
    }
}

```

重载规则：

- 1，被重载的方法必须改变参数列表（参数个数或者类型不一样）。
- 2，被重载的方法可以改变返回类型。
- 3，被重载的方法可以改变访问修饰符。

## 3.5，this，super关键字

**super()关键字的用法** 1，子类的成员方法中，访问父类的成员变量。

2，子类的成员方法中，访问父类的成员方法。

3，子类的构造方法中，访问父类的构造方法。

**this关键字用法：** 1，本类成员方法中，访问本类的成员变量。 2，本类成员方法中，访问本类的另一个成员方法。 3，本类的构造方法中，访问本类的另一个构造方法。 注意：

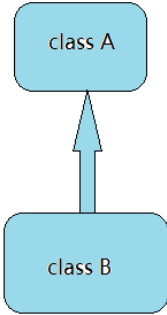
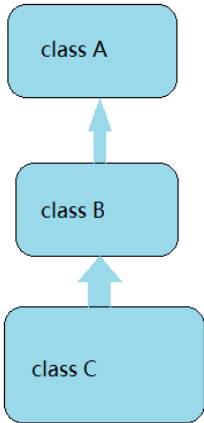
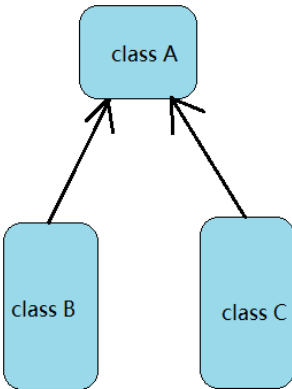
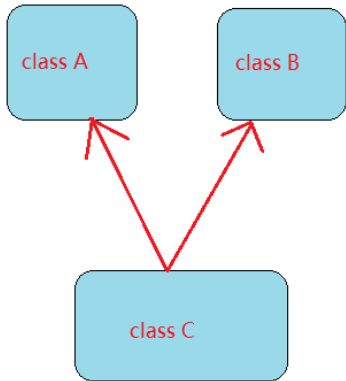
- this关键字同super一样，必须在构造方法的第一个语句，且是唯一的。
- this()与super()不能同时存在。

## 3.6，继承中的构造器

**继承关系中，父子类构造方法的访问特点：** 1，在子类构造方法中有一个默认隐含的super();调用，因此一定是先调用父类构造方法，再调用子类构造方法。 2，子类构造可以通过super();调用父类的重载构造。(重载) 3，super();的父类调用构造方法，必须在子类构造中的第一行，就是第一个;号结束的元素，并且只能调用一次。

## 3.7，关于继承的注意事项

1，Java语言是单继承的，一个子类只能有唯一——一个父类 2，Java语言可以是多级继承，一个子类有一个父类，一个父类还可以有一个父类。 3，一个子类只有一个父类，但是一个父类可以有多个子类。

单继承 <pre>public class A{} public class B extends B{}</pre>	多级继承 <pre>public class A{} public class B extends A{} public class C extends B{}</pre>	同一个父类多个子类 <pre>public class A{} public class B extends A{} public class C extends A{}</pre>	<del>多继承  <pre>public class A{} public class B{} public class C extends A,B{}</pre> </del>
 <pre>graph BT     B[class B] --&gt; A[class A]</pre>	 <pre>graph BT     C[class C] --&gt; B[class B]     B --&gt; A[class A]</pre>	 <pre>graph BT     B[class B] --&gt; A[class A]     C[class C] --&gt; A</pre>	<p>错误, Java不支持多继承</p>  <pre>graph BT     C[class C] --&gt; A[class A]     C --&gt; B[class B]</pre>

## 3.8 , Object类

Java Object 类是所有类的父类，也就是说 Java 的所有类都继承了 Object，**子类可以使用 Object 的所有方法。**

Object 类位于 java.lang 包中，编译时会自动导入，我们创建一个类时，如果没有明确继承一个父类，那么它就会自动继承 Object，成为 Object 的子类。

Object 类可以显示继承，也可以隐式继承，以下两种方式时一样的：

显示继承:

```
public class Student extends Object
{
}
```

隐式继承:

```
public class Student
{
}
```

类的方法

序号	方法 & 描述
1	<a href="#">protected Object clone()</a> 创建并返回一个对象的拷贝
2	<a href="#">boolean equals(Object obj)</a> 比较两个对象是否相等
3	<a href="#">protected void finalize()</a> 当 GC (垃圾回收器)确定不存在对该对象的有更多引用时，由对象的垃圾回收器调用此方法。
4	<a href="#">Class getClass()</a> 获取对象的运行时对象的类
5	<a href="#">int hashCode()</a> 获取对象的 hash 值
6	<a href="#">void notify()</a> 唤醒在该对象上等待的某个线程
7	<a href="#">void notifyAll()</a> 唤醒在该对象上等待的所有线程
8	<a href="#">String toString()</a> 返回对象的字符串表示形式
9	<a href="#">void wait()</a> 让当前线程进入等待状态。直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法。

目前我们只用知道，所有的类都是object类的子类，所以也都具备这些方法就可以了。

## 四，多态

### 4.1，什么是多态

多态是同一个行为具有多个不同表现形式或形态的能力。

### 4.2，多态的特点

- 1，消除类型之间的耦合关系，实现低耦合。
- 2，灵活性。
- 3，可扩充性。
- 4，可替换性。

### 4.3，多态的体现形式

- 继承
- 父类引用指向子类的对象
- 重写

**注意：在多态中，编译看左边，运行看右边**

```
public class MultiDemo
{
    public static void main(String[] args)
```

```

    {
        // 多态的引用，就是向上转型
        Animals dog = new Dog();
        dog.eat();

        Animals cat = new Cat();
        cat.eat();

        // 如果要调用父类中没有的方法，则要向下转型
        Dog dogDown = (Dog)dog;
        dogDown.watchDoor();

    }
}
class Animals
{
    public void eat()
    {
        System.out.println("动物吃饭！");
    }
}
class Dog extends Animals
{
    public void eat()
    {
        System.out.println("狗在吃骨头！");
    }
    public void watchDoor()
    {
        System.out.println("狗看门！");
    }
}
class Cat extends Animals
{
    public void eat()
    {
        System.out.println("猫在吃鱼！");
    }
}
}

```

## 4.4，向上转型

1，格式：父类名称 对象名 = new 子类名称();

含义：右侧创建一个子类对象，把它当作父类来使用。

注意：向上转型一定是安全的。

缺点：一旦向上转型，子类中原本特有的方法就不能再被调用了。



动物

eat(); // 抽象

向上转型

本来是猫  
还原为猫

本来是猫  
当做狗  
错误!

猫

eat() {鱼}

狗

eat() {SHIT}

1. 对象的向上转型，其实就是多态写法：  
格式：父类名称 对象名 = new 子类名称();  
含义：右侧创建一个子类对象，把它当做父类来看待使用。  
注意事项：向上转型一定是安全的。从小范围转向了大范围，从小范围的猫，向上转换为更大范围的动物。

类似于：  
double num = 100; // 正确，int --> double，自动类型转换。

Animal animal = new Cat();  
创建了一只猫，当做动物看待，没问题。

2. 对象的向下转型，其实是一个【还原】的动作。  
格式：子类名称 对象名 = (子类名称) 父类对象;  
含义：将父类对象，【还原】成为本来的子类对象。

Animal animal = new Cat(); // 本来是猫，向上转型成为动物  
Cat cat = (Cat) animal; // 本来是猫，已经被当做动物了，还原回来成为本来的猫

注意事项：  
a. 必须保证对象本来创建的时候，就是猫，才能向下转型成为猫。  
b. 如果对象创建的时候本来不是猫，现在非要向下转型成为猫，就会报错。ClassCastException

类似于：int num = (int) 10.0; // 可以      int num = (int) 10.5; // 不可以，精度损失

4.5 ,包

包是一种松散的类的集合,通常把需要在一起工作的类(互相访问)放入一个包。在Java语言程序设计中，通常需要定义许多类；就像利用“文件夹”把许多文件组织在一起，使硬盘管理的文件更清晰、更有条理一样；Java利用“包”把一些需要在一起操作的类组织在一起，以便程序员更好地管理操作这些类。引入的原因：容易找到和使用类避免名称冲突 控制访问定义：包是一个相关的类和接口的集合,它可以提供访问保护和名称空间管理例如：基本的类在 java.lang中；用于输入和输出的类在java.io中。

声明包在源文件的开始加上：

```
package 包名(要求全部小写)(反域名制) package cn.com.sun ;
```

使用包中的类如使用不同包的类，必须加入 import 关键字

4.6 ,访问权限修饰符

作用域	类	同一-package	子孙	其他package
public	√	√	√	√
protected	√	√	√	×
friendly	√	√	×	×
private	√	×	×	×

不写时默认为friendly

五，Static,Final关键字

在java的关键字中，static和final是两个我们必须掌握的关键字。不同于其他关键字，他们都有多种用法，而且在一定环境下使用，可以提高程序的运行性能，优化程序的结构。下面我们先来了解一下static关键字及其用法。

5.1，Static关键字

## 1.修饰成员变量

在我们平时的使用当中，static最常用的功能就是修饰类的属性和方法，让他们成为类的成员属性和方法，我们通常将用static修饰的成员称为类成员或者静态成员，这句话挺起来都点奇怪，其实这是相对于对象的属性和方法来说的。请看下面的例子：

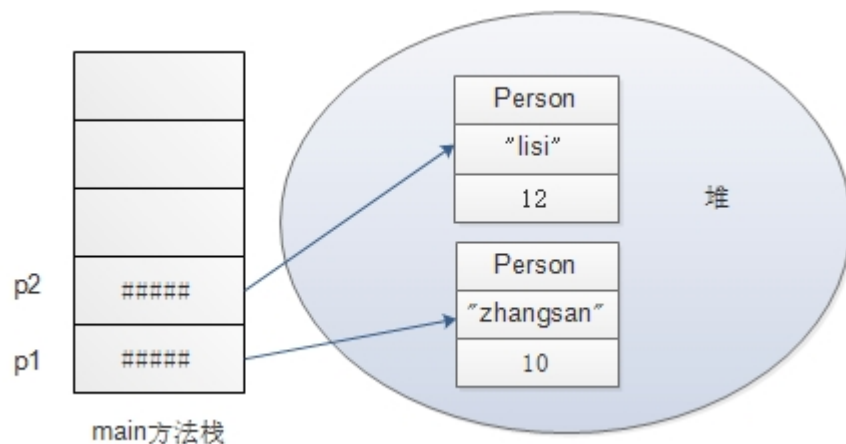
```
public class Person
{
    String name;
    int age;

    public String toString()
    {
        return "Name:" + name + ", Age:" + age;
    }

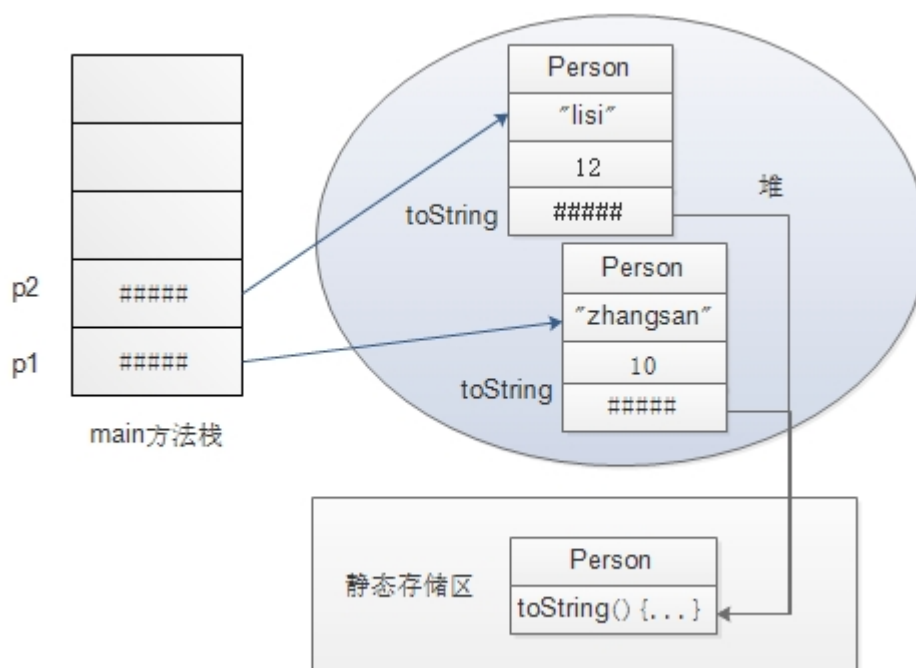
    public static void main(String[] args)
    {
        Person p1 = new Person();
        p1.name = "zhangsan";
        p1.age = 10;
        Person p2 = new Person();
        p2.name = "lisi";
        p2.age = 12;
        System.out.println(p1);
        System.out.println(p2);
    }
}

/**Output
 * Name:zhangsan, Age:10
 * Name:lisi, Age:12
 *///~
}
```

上面的代码我们很熟悉，根据Person构造出的每一个对象都是独立存在的，保存有自己独立的成员变量，相互不会影响，他们在内存中的示意如下：



从上图中可以看出，p1和p2两个变量引用的对象分别存储在内存中堆区域的不同地址中，所以他们之间相互不会干扰。但其实，在这当中，我们省略了一些重要信息，相信大家也都会想到，对象的成员属性都在这了，由每个对象自己保存，那么他们的方法呢？实际上，不论一个类创建了几个对象，他们的方法都是一样的：



从上面的图中我们可以看到，两个Person对象的方法实际上只是指向了同一个方法定义。这个方法定义是位于内存中的一块不变区域（由jvm划分），我们称它为方法区（静态存储区）。这一块存储区不仅存放了方法的定义，实际上从更大的角度而言，它存放的是各种类的定义，当我们通过new来生成对象时，会根据这里定义的类的定义去创建对象。多个对象仅会对应同一个方法，这里有一个让我们充分信服的理由，那就是不管多少的对象，他们的方法总是相同的，尽管最后的输出会有所不同，但是方法总是会按照我们预想的结果去操作，即不同的对象去调用同一个方法，结果会不尽相同。

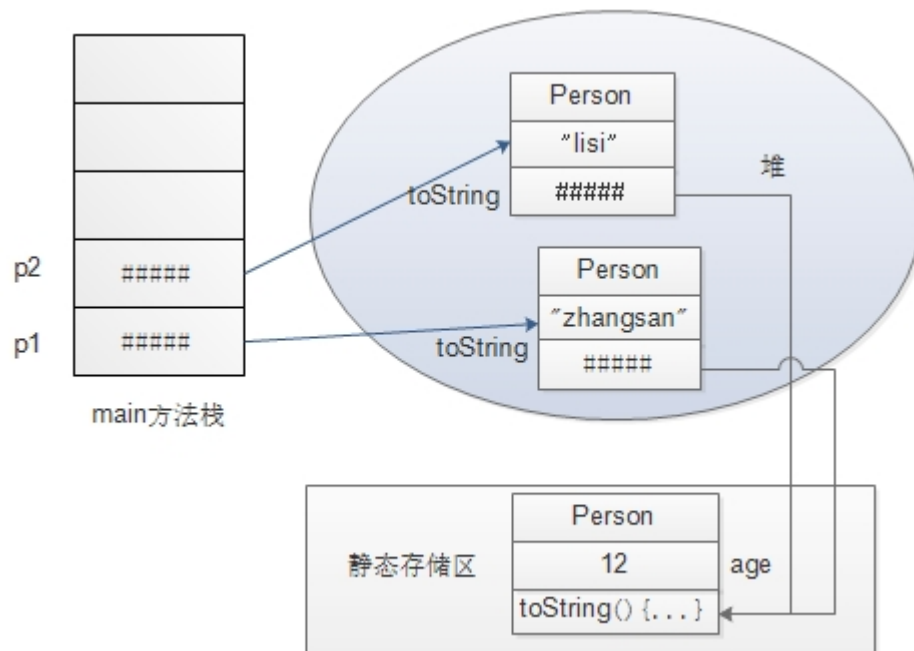
我们知道，static关键字可以修饰成员变量和方法，来让它们变成类的所属，而不是对象的所属，比如我们将Person的age属性用static进行修饰，结果会是什么样呢？请看下面的例子：

```
public class Person
{
    String name;
    static int age;

    /* 其余代码不变... */

    /**Output
     * Name:zhangsan, Age:12
     * Name:lisi, Age:12
     *///~
    }
```

我们发现，结果发生了一点变化，在给p2的age属性赋值时，干扰了p1的age属性，这是为什么呢？我们还是来看他们在内存中的示意：



我们发现，给age属性加了static关键字之后，Person对象就不再拥有age属性了，age属性会统一交给Person类去管理，即多个Person对象只会对应一个age属性，一个对象如果对age属性做了改变，其他的对象都会受到影响。我们看到此时的age和toString()方法一样，都是交由类去管理。

虽然我们看到static可以让对象共享属性，但是实际中我们很少这么用，也不推荐这么使用。因为这样会让该属性变得难以控制，因为它在任何地方都有可能被改变。如果我们想共享属性，一般我们会采用其他的办法：

```
public class Person
{
    private static int count = 0;
    int id;
    String name;
    int age;

    public Person()
    {
        id = ++count;
    }

    public String toString()
    {
        return "Id:" + id + ", Name:" + name + ", Age:" + age;
    }

    public static void main(String[] args)
    {
        Person p1 = new Person();
        p1.name = "zhangsan";
        p1.age = 10;
        Person p2 = new Person();
        p2.name = "lisi";
        p2.age = 12;
        System.out.println(p1);

        System.out.println(p2);
    }
}
```

```

    }
    /**Output
     * Id:1, Name:zhangsan, Age:10
     * Id:2, Name:lisi, Age:12
     *///~
    }

```

上面的代码起到了给Person的对象创建一个唯一id以及记录总数的作用，其中count由static修饰，是Person类的成员属性，每次创建一个Person对象，就会使该属性自加1然后赋给对象的id属性，这样，count属性记录了创建Person对象的总数，由于count使用了private修饰，所以从类外面无法随意改变。

## 2.修饰成员方法

static的另一个作用，就是修饰成员方法。相比于修饰成员属性，修饰成员方法对于数据的存储上面并没有多大的变化，因为我们从上面可以看出，方法本来就是存放在类的定义当中的。static修饰成员方法最大的作用，就是可以使用"类名.方法名"的方式操作方法，避免了先要new出对象的繁琐和资源消耗，我们可能会经常在帮助类中看到它的使用：

```

public class PrintHelper
{
    public static void print(Object o)
    {
        System.out.println(o);
    }

    public static void main(String[] args)
    {
        PrintHelper.print("Hello world");
    }
}

```

上面便是一个例子（现在还不太实用），但是我们可以看到它的作用，使得static修饰的方法成为类的方法，使用时通过“类名.方法名”的方式就可以方便的使用了，相当于定义了一个全局的函数（只要导入该类所在的包即可）。不过它也有使用的局限，一个static修饰的类中，不能使用非static修饰的成员变量和方法，这很好理解，因为static修饰的方法是属于类的，如果去直接使用对象的成员变量，它会不知所措（不知该使用哪一个对象的属性）。

## 3.静态块

在说明static关键字的第三个用法时，我们有必要重新梳理一下一个对象的初始化过程。以下面的代码为例：

```

class Book
{
    //构造函数
    public Book(String msg)
    {
        System.out.println(msg);
    }
}

```

```

}

public class Person
{
    //普通属性
    Book book1 = new Book("book1成员变量初始化");
    //静态属性
    static Book book2 = new Book("static成员book2成员变量初始化");
    //构造函数
    public Person(String msg)
    {
        System.out.println(msg);
    }
    //普通属性
    Book book3 = new Book("book3成员变量初始化");
    //静态属性
    static Book book4 = new Book("static成员book4成员变量初始化");
    //main函数
    public static void main(String[] args)
    {
        Person p1 = new Person("p1初始化");
    }
    /**Output
     * static成员book2成员变量初始化
     * static成员book4成员变量初始化
     * book1成员变量初始化
     * book3成员变量初始化
     * p1初始化
     * ///~
     */
}

```

上面的例子中，Person类中组合了四个Book成员变量，两个是普通成员，两个是static修饰的类成员。我们可以看到，当我们new一个Person对象时，static修饰的成员变量首先被初始化，随后是普通成员，最后调用Person类的构造方法完成初始化。也就是说，在创建对象时，static修饰的成员会首先被初始化，而且我们还可以看到，如果有多个static修饰的成员，那么会按照他们的先后位置进行初始化。

实际上，static修饰的成员的初始化可以更早的进行，请看下面的例子：

```

class Book
{
    //构造函数
    public Book(String msg)
    {
        System.out.println(msg);
    }
}

public class Person
{
    //普通属性
    Book book1 = new Book("book1成员变量初始化");

    //静态属性

```

```

static Book book2 = new Book("static成员book2成员变量初始化");
//构造函数
public Person(String msg)
{
    System.out.println(msg);
}
//普通属性
Book book3 = new Book("book3成员变量初始化");
//静态属性
static Book book4 = new Book("static成员book4成员变量初始化");
//静态方法
public static void funStatic()
{
    System.out.println("static修饰的funStatic方法");
}

//main函数
public static void main(String[] args)
{
    Person.funStatic();
    System.out.println("*****");
    Person p1 = new Person("p1初始化");
}
/**Output
 * static成员book2成员变量初始化
 * static成员book4成员变量初始化
 * static修饰的funStatic方法
 * *****
 * book1成员变量初始化
 * book3成员变量初始化
 * p1初始化
 * ///~
 */
}

```

在上面的例子中我们可以发现两个有意思的地方，第一个是当我们没有创建对象，而是通过类去调用类方法时，尽管该方法没有使用到任何的类成员，类成员还是在方法调用之前就初始化了，这说明，当我们第一次去使用一个类时，就会触发该类的成员初始化。第二个是当我们使用了类方法，完成类的成员的初始化后，再new该类的对象时，static修饰的类成员没有再次初始化，这说明，static修饰的类成员，在程序运行过程中，只需要初始化一次即可，不会进行多次的初始化。

回顾了对对象的初始化以后，我们再来看static的第三个作用就非常简单了，那就是当我们初始化static修饰的成员时，可以将他们统一放在一个以static开始，用花括号包裹起来的块状语句中：

```

class Book
{
    public Book(String msg)
    {
        System.out.println(msg);
    }
}

public class Person

```

```

{
    //普通属性
    Book book1 = new Book("book1成员变量初始化");
    //声明静态属性，但未赋值
    static Book book2;
    //静态代码块
    static
    {
        book2 = new Book("static成员book2成员变量初始化");
        book4 = new Book("static成员book4成员变量初始化");
    }
    //构造方法
    public Person(String msg)
    {
        System.out.println(msg);
    }
    //普通属性
    Book book3 = new Book("book3成员变量初始化");
    //声明静态属性，但未赋值，虽然book4变量在static代码块的下面，但是不影响
    static Book book4;

    public static void funStatic()
    {
        System.out.println("static修饰的funStatic方法");
    }

    public static void main(String[] args)
    {
        Person.funStatic();
        System.out.println("*****");
        Person p1 = new Person("p1初始化");
    }
    /**Output
     * static成员book2成员变量初始化
     * static成员book4成员变量初始化
     * static修饰的funStatic方法
     * *****
     * book1成员变量初始化
     * book3成员变量初始化
     * p1初始化
     * ///~
    */
}

```

我们将上一个例子稍微做了一下修改，可以看到，结果没有二致。

## 4.静态导包

相比于上面的三种用途，第四种用途可能了解的人就比较少了，但是实际上它很简单，而且在调用类方法时会更方便。以上面的“PrintHelper”的例子为例，做一下稍微的变化，即可使用静态导包带给我们的方便：



```

/* PrintHelper.java文件 */
package com.soft863.test;

public class PrintHelper
{

    public static void print(Object o)
    {
        System.out.println(o);
    }
}

```

```

/* Main.java文件 */

import static com.soft863.test.PrintHelper.*;

public class Main
{
    public static void main( String[] args )
    {
        print("Hello World!");
    }
    /**Output
     * Hello World!
     *///~
}

```

上面的代码来自于两个java文件，其中的PrintHelper很简单，包含了一个用于打印的static方法。而在App.java文件中，我们首先将PrintHelper类导入，这里在导入时，我们使用了static关键字，而且在引入类的最后还加上了“.\*”，它的作用就是将PrintHelper类中的所有类方法直接导入。不同于非static导入，采用static导入包后，在不与当前类的方法名冲突的情况下，无需使用“类名.方法名”的方法去调用类方法了，直接可以采用“方法名”去调用类方法，就好像是该类自己的方法一样使用即可。

## 总结

static是java中非常重要的一个关键字，而且它的用法也很丰富，主要有四种用法：

1. 用来修饰成员变量，将其变为类的成员，从而实现所有对象对于该成员的共享；
2. 用来修饰成员方法，将其变为类方法，可以直接使用“类名.方法名”的方式调用，常用于工具类；
3. 静态块用法，将多个类成员放在一起初始化，使得程序更加规整，其中理解对象的初始化过程非常关键；
4. 静态导包用法，将类的方法直接导入到当前类中，从而直接使用“方法名”即可调用类方法，更加方便。

## 5.2, Final关键字

### 1.修饰数据

在编写程序时，我们经常需要说明一个数据是不可变的，我们成为常量。在java中，用final关键字修饰的变量，只能进行一次赋值操作，并且在生存期内不可以改变它的值。更重要的是，final会告诉编译器，这个数据是不会修改的，那么编译器就可能会在编译时期就对该数据进行替换甚至执行计算，这样可以对我们的程序起到一点优化。不过在针对基本类型和引用类型时，final关键字的效果存在细微差别。我们来看下面的例子：

```

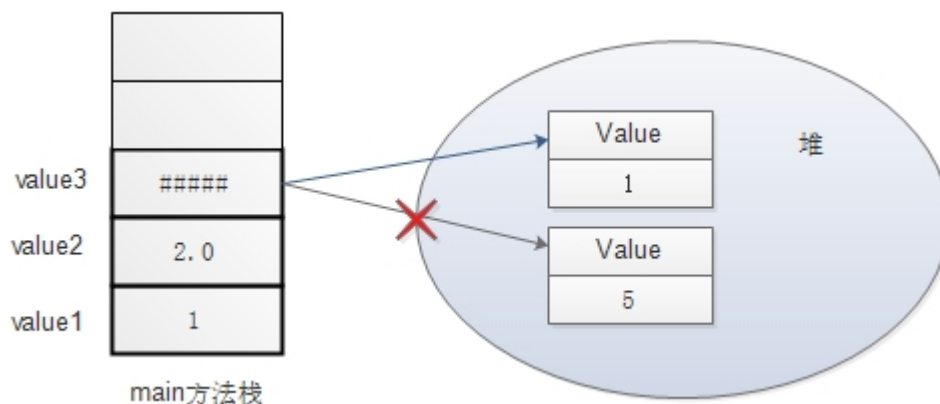
class Value
{
    int v;
    public Value(int v)
    {
        this.v = v;
    }
}

public class FinalTest
{
    final int f1 = 1;
    final int f2;
    public FinalTest()
    {
        f2 = 2;
    }

    public static void main(String[] args)
    {
        final int value1 = 1;
        //报错
        // value1 = 4;
        final double value2;
        value2 = 2.0;
        final Value value3 = new Value(1);
        value3.v = 4;
    }
}

```

上面的例子中，我们先来看一下main方法中的几个final修饰的数据，在给value1赋初始值之后，我们无法再对value1的值进行修改，final关键字起到了常量的作用。从value2我们可以看到，final修饰的变量可以不在声明时赋值，即可以先声明，后赋值。value3是一个引用变量，这里我们可以看到final修饰引用变量时，只是限定了引用变量的引用不可改变，即不能将value3再次引用另一个Value对象，但是引用的对象的值是可以改变的，从内存模型中我们看的更加清晰：



上图中，final修饰的值用粗线条的边框表示它的值是不可改变的，我们知道引用变量的值实际上是它所引用的对象的地址，也就是说该地址的值是不可改变的，从而说明了为什么引用变量不可以改变引用对象。而实际引用的对象实际上是不受final关键字的影响的，所以它的值是可以改变的。

另一方面，我们看到了用final修饰成员变量时的细微差别，因为final修饰的数据的值是不可改变的，所以我们必须确保在使用前就已经对成员变量赋值了。因此对于final修饰的成员变量，我们有且只有两个地方可以给它赋值，一个是声明该成员时赋值，另一个是在构造方法中赋值，在这两个地方我们必须给它们赋初始值。

最后我们需要注意的一点是，同时使用static和final修饰的成员在内存中只占据一段不能改变的存储空间。

## 2.修饰方法参数

前面我们可以看到，如果变量是我们自己创建的，那么使用final修饰表示我们只会给它赋值一次且不会改变变量的值。那么如果变量是作为参数传入的，我们怎么保证它的值不会改变呢？这就用到了final的第二种用法，即在我们编写方法时，可以在参数前面添加final关键字，它表示在整个方法中，我们不会（实际上是不能）改变参数的值：

```
public class FinalTest
{
    /* ... */

    public void finalFunc(final int i, final Value value)
    {
        // i = 5; 不能改变i的值
        // v = new Value(); 不能改变v的值
        value.v = 5; // 可以改变引用对象的值
    }
}
```

## 3.修饰方法

第三种方式，即用final关键字修饰方法，它表示该方法不能被覆盖。这种使用方式主要是从设计的角度考虑，即明确告诉其他可能会继承该类的程序员，不希望他们去覆盖这个方法。这种方式我们很容易理解，然而，关于private和final关键字还有一点联系，这就是类中所有的private方法都隐式地指定为是final的，由于无法在类外使用private方法，所以也就无法覆盖它。

## 4.修饰类

了解了final关键字的其他用法，我们很容易可以想到使用final关键字修饰类的作用，那就是用final修饰的类是无法被继承的。

上面我们讲解了final的四种用法，然而，对于第三种和第四种用法，我们却甚少使用。这不是没有道理的，从final的设计来讲，这两种用法甚至可以说是鸡肋，因为对于开发人员来讲，如果我们写的类被继承的越多，就说明我们写的类越有价值，越成功。即使是从设计的角度来讲，也没有必要将一个类设计为不可继承的。Java标准库就是一个很好的反例，特别是Java 1.0/1.1中Vector类被如此广泛的运用，如果所有的方法均未被指定为final的话，它可能会更加有用。如此有用的类，我们很容易想到去继承和重写他们，然而，由于final的作用，导致我们对Vector类的扩展受到了一些阻碍，导致了Vector并没有完全发挥它应有的全部价值。

## 总结

final关键字是我们经常使用的关键字之一，它的用法有很多，但是并不是每一种用法都值得我们去广泛使用。它的主要用法有以下四种：

1. 用来修饰数据，包括成员变量和局部变量，该变量只能被赋值一次且它的值无法被改变。对于成员变量来讲，我们必须在声明时或者构造方法中对它赋值；

2. 用来修饰方法参数，表示在变量的生存期中它的值不能被改变；
3. 修饰方法，表示该方法无法被重写；
4. 修饰类，表示该类无法被继承。

上面的四种方法中，第三种和第四种方法需要谨慎使用，因为在大多数情况下，如果是仅仅为了一点设计上的考虑，我们并不需要使用final来修饰方法和类。

## 六，抽象类与接口

### 6.1， 抽象类与抽象方法

在了解抽象类之前，先来了解一下抽象方法。抽象方法是一种特殊的方法：它只有声明，而没有具体的实现。抽象方法的声明格式为：

```
public abstract void open();
```

抽象方法必须使用abstract关键字进行修饰。如果一个类含有抽象方法，则称这个类为抽象类，抽象类必须在类前用abstract关键字修饰。因为抽象类中无具体实现的方法，所以不能用抽象类创建对象。

```
public abstract class Door
{
    public abstract void open();

    public abstract void close();
}
```

从这里可以看出，抽象类就是为了继承而存在的，如果你定义了一个抽象类，却不去继承它，那么等于白白创建了这个抽象类，因为你不能用它来做任何事情。对于一个父类，如果它的某个方法在父类中没有任何意义，必须根据子类的实际需求来进行不同的实现，那么就可以将这个方法的声明为abstract方法，此时这个类就不是abstract类了。

包含抽象方法的类称为抽象类，但并不意味着抽象类中只能有抽象方法，它和普通类一样，同样可以拥有成员变量和普通的成员方法。注意，抽象类和普通类的主要有三点区别：

- 1) 抽象方法必须为public或者protected（因为如果为private，则不能被子类继承，子类便无法实现该方法），缺省情况下默认为public。
- 2) 抽象类不能用来创建对象；
- 3) 如果一个类继承于一个抽象类，则子类必须实现父类的抽象方法。如果子类没有实现父类的抽象方法，则必须将子类也定义为abstract类。

其他方面，抽象类普通的类并没有区别。

### 6.2， 接口

接口，英文称作interface，在软件工程中，接口泛指供别人调用的方法或者函数。从这里，我们可以体会到Java语言设计者的初衷，它是对行为的抽象。在Java中，定一个接口的形式如下：

```
public interface TestInterface
{

    public void open();

    public void close();

}
```

接口中可以含有变量和方法。但是要注意，接口中的变量会被隐式地指定为public static final变量（并且只能是public static final变量，用private修饰会报编译错误），而方法会被隐式地指定为public abstract方法且只能是public abstract方法（用其他关键字，比如private、protected、static、final等修饰会报编译错误），并且接口中所有的方法不能有具体的实现，也就是说，接口中的方法必须都是抽象方法。

从这里可以隐约看出接口和抽象类的区别，接口是一种极度抽象的类型，它比抽象类更加“抽象”，并且一般情况下不在接口中定义变量。

要让一个类遵循某组特地的接口需要使用implements关键字，具体格式如下：

```
public class TestInterfaceImpl implements TestInterface
{

    @Override
    public void open()
    {
        System.out.println("开门...");
    }

    @Override
    public void close()
    {
        System.out.println("关门...");
    }

}
```

可以看出，允许一个类实现多个特定的接口。如果一个非抽象类实现了某个接口，就必须实现该接口中的所有方法。对于实现某个接口的抽象类，可以不实现该接口中的抽象方法。

**在JDK8之后：**

1，default修饰，接口里允许定义默认的方法，但默认方法也可以覆盖重写。 2，接口里允许定义静态方法。

**接口的注意事项：** 1，不能通过接口的实现类对象去调用接口中的静态方法。 正确语法：接口名称调用静态方法。

### 接口当中的常量的使用：

1，接口当中定义的常量：可以省略public static final。 2，接口当中定义的常量：必须进行赋值。 3，接口当中定义的常量：常量的名称要全部大写，多个名称之间使用下划线进行分割。

### 使用接口的注意事项：

1，接口是没有静态代码块或者构造方法 2，一个类的直接父类是唯一的，但是一个类可以同时实现多个接口。 3，如果实现类没有覆盖重写接口中所有的抽象方法，那么实现类就必须是一个抽象类 4，如果实现类中实现多个接口，存在重复的抽象方法，那么只需要覆盖重写一次即可。 5，在Java中，如果实现类的直接继承父类与实现接口发生冲突时，父类优先级高于接口。

### 接口之间的关系：

1，多个接口之间是继承关系。 2，多个父接口当中默认方法如果重复，那么子接口必须进行默认方法的覆盖重写。

## 6.3接口与抽象类的区别

### 语法层面上的区别

- 1) 抽象类可以提供成员方法的实现细节，而接口中只能存在public abstract 方法；
- 2) 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是public static final类型的；
- 3) 接口中不能含有静态代码块以及静态方法，而抽象类可以有静态代码块和静态方法；
- 4) 一个类只能继承一个抽象类，而一个类却可以实现多个接口。

### 设计层面上的区别

1) 抽象类是对一种事物的抽象，即对类抽象，而接口是对行为的抽象。抽象类是对整个类整体进行抽象，包括属性、行为，但是接口却是对类局部（行为）进行抽象。

举个简单的例子，飞机和鸟是不同类的事物，但是它们都有一个共性，就是都会飞。那么在设计的时候，可以将飞机设计为一个类Airplane，将鸟设计为一个类Bird，但是不能将飞行这个特性也设计为类，因此它只是一个行为特性，并不是对一类事物的抽象描述。此时可以将飞行设计为一个接口Fly，包含方法fly()，然后Airplane和Bird分别根据自己的需要实现Fly这个接口。然后至于有不同种类的飞机，比如战斗机、民用飞机等直接继承Airplane即可，对于鸟也是类似的，不同种类的鸟直接继承Bird类即可。

从这里可以看出，继承是一个“是不是”的关系，而接口实现则是“有没有”的关系。如果一个类继承了某个抽象类，则子类必定是抽象类的种类，而接口实现则是有没有、具备不具备的关系，比如鸟是否能飞（或者是否具备飞行这个特点），能飞行则可以实现这个接口，不能飞行就不实现这个接口。

2) 设计层面不同，抽象类作为很多子类的父类，它是一种模板式设计。而接口是一种行为规范，它是一种辐射式设计。

什么是模板式设计？最简单例子，大家都用过ppt里面的模板，如果用模板A设计了ppt B和ppt C，ppt B和ppt C公共的部分就是模板A了，如果它们的公共部分需要改动，则只需要改动模板A就可以了，不需要重新对ppt B和ppt C进行改动。

而辐射式设计，比如某个电梯都装了某种报警器，一旦要更新报警器，就必须全部更新。也就是说对于抽象类，如果需要添加新的方法，可以直接在抽象类中添加具体的实现，子类可以不进行变更；而对于接口则不行，如果接口进行了变更，则所有实现这个接口的类都必须进行相应的改动。

## 案例：门和警报的例子

```
public abstract class TestAbstractDoor
{
    public abstract void open();

    public abstract void close();
}
```

或者使用接口

```
public interface TestInterfaceDoor
{
    public void open();

    public void close();
}
```

但是现在如果我们需要门具有报警alarm()的功能，那么该如何实现？下面提供两种思路：

- 1) 将这三个功能都放在抽象类里面，但是这样一来所有继承于这个抽象类的子类都具备了报警功能，但是有的门并不一定具备报警功能；
- 2) 将这三个功能都放在接口里面，需要用到报警功能的类就需要实现这个接口中的open()和close()，也许这个类根本就不具备open()和close()这两个功能，比如火灾报警器。

从这里可以看出，Door的open()、close()和alarm()根本就属于两个不同范畴内的行为，open()和close()属于门本身固有的行为特性，而alarm()属于延伸的附加行为。因此最好的解决办法是单独将报警设计为一个接口，包含alarm()行为,Door设计为单独的一个抽象类，包含open和close两种行为。再设计一个报警门继承Door类和实现Alarm接口。

再去定义一个警报接口

```
public interface TestInterfaceAlarm
{
    public void alarm();
}
```

具体实现类

```
//会报警的门
public class TestAlarmDoor extends TestAbstractDoor implements TestInterfaceAlarm
{
    @Override
    public void alarm()
    {

```

```

        // TODO Auto-generated method stub

    }

    @Override
    public void open()
    {
        // TODO Auto-generated method stub

    }

    @Override
    public void close()
    {
        // TODO Auto-generated method stub

    }

}

```

```

//不会报警的门，就不用实现TestInterfaceAlarm接口，所以也就没有报警的方法
public class TestDoor extends TestAbstractDoor
{

    @Override
    public void open()
    {
        // TODO Auto-generated method stub

    }

    @Override
    public void close()
    {
        // TODO Auto-generated method stub

    }

}

```

## 七，枚举

### 7.1，理解枚举类型

枚举类型是Java 5中新增特性的一部分，它是一种特殊的数据类型，之所以特殊是因为它既是一种



类(class)类型却又比类类型多了些特殊的约束，但是这些约束的存在也造就了枚举类型的简洁性、安全性以及便捷性。下面先来看看什么是枚举？如何定义枚举？

```
public class DayDemo
{
    public static final int MONDAY =1;

    public static final int TUESDAY=2;

    public static final int WEDNESDAY=3;

    public static final int THURSDAY=4;

    public static final int FRIDAY=5;

    public static final int SATURDAY=6;

    public static final int SUNDAY=7;

}
```

上述的常量定义常量的方式称为 `int` 枚举模式，这样的定义方式并没有什么错，但它存在许多不足，如在类型安全和使用方便性上并没有多少好处，如果存在定义 `int` 值相同的变量，混淆的几率还是很大的，编译器也不会提出任何警告，因此这种方式在枚举出现后并不提倡，现在我们利用枚举类型来重新定义上述的常量，同时也感受一把枚举定义的方式，如下定义周一到周日的常量。

```
//枚举类型，使用关键字enum
enum Day
{
    MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

相当简洁，在定义枚举类型时我们使用的关键字是 `enum`，与 `class` 关键字类似，只不过前者是定义枚举类型，后者是定义类类型。枚举类型Day中分别定义了从周一到周日的值，这里要注意，值一般是大写的字母，多个值之间以逗号分隔。同时我们应该知道的是枚举类型可以像类(class)类型一样，定义为一个单独的文件，当然也可以定义在其他类内部，更重要的是枚举常量在类型安全性和便捷性都很有保证，如果出现类型问题编译器也会提示我们改进，但务必记住枚举表示的类型其取值是必须有限的，也就是说每个值都是可以枚举出来的，比如上述描述的一周共有七天。那么该如何使用呢？如下

```
public class EnumDemo
{
    public static void main(String[] args)
```

```

    {
        //直接引用
        Day day =Day.MONDAY;
    }

}
//定义枚举类型
enum Day
{
    MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

```

就像上述代码那样，直接引用枚举的值即可，这便是枚举类型的最简单模型。

## 7.2，枚举实现的原理

我们大概了解了枚举类型的定义与简单使用后，现在有必要来了解一下枚举类型的基本实现原理。

实际上在使用关键字 `enum` 创建枚举类型并编译后，编译器会为我们生成一个相关的类，这个类继承了

Java API 中的 `java.lang.Enum` 类，也就是说通过关键字 `enum` 创建枚举类型在编译后事实上也是一个类类型而且该类继承自 `java.lang.Enum` 类。

## 7.3，编译器生成的 `values` 方法与 `valueOf` 方法

`values()` 方法和 `valueOf(String name)` 方法是编译器生成的 `static` 方法，因此从前面的分析

中，在 `Enum` 类中并没出现 `values()` 方法，但 `valueOf()` 方法还是有出现的，只不过编译器生成的

`valueOf()` 方法需传递一个 `name` 参数，而 `Enum` 自带的静态方法 `valueOf()` 则需要传递两个方法，从前面反编译后的代码可以看出，编译器生成的 `valueOf` 方法最终还是调用了 `Enum` 类的 `valueOf` 方法，下面通过代码来演示这两个方法的作用：

```

Day[] days2 = Day.values();
System.out.println("day2:"+Arrays.toString(days2));
Day day = Day.valueOf("MONDAY");
System.out.println("day:"+day);

/**
 输出结果：
  day2:[MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY]
  day:MONDAY
 */

```

从结果可知道，`values()` 方法的作用就是获取枚举类中的所有变量，并作为数组返回，而 `valueOf(String name)` 方法与 `Enum` 类中的 `valueOf` 方法的作用类似根据名称获取枚举变量，只不过编译器生成的 `valueOf` 方法更简洁些只需传递一个参数。这里我们还必须注意到，由于 `values()` 方法是由编译器插入到枚举类中的 `static` 方法，所以如果我们将枚举实例向上转型为 `Enum`，那么 `values()` 方法将无法被调用，因为 `Enum` 类中并没有 `values()` 方法，`valueOf()` 方法也是同样的道理，注意是一个参数的。

```
//正常使用
Day[] ds=Day.values();
//向上转型Enum
Enum e = Day.MONDAY;
//无法调用,没有此方法
//e.values();
```

## 7.4 , 向enum类添加方法与自定义构造函数

重新定义一个日期枚举类, 带有 `desc` 成员变量描述该日期的对于中文描述, 同时定义一个 `getDesc` 方法, 返回中文描述内容, 自定义私有构造函数, 在声明枚举实例时传入对应的中文描述, 代码如下:

```
public enum Day2
{
    MONDAY("星期一"),
    TUESDAY("星期二"),
    WEDNESDAY("星期三"),
    THURSDAY("星期四"),
    FRIDAY("星期五"),
    SATURDAY("星期六"),
    SUNDAY("星期日");//记住要用分号结束

    private String desc;//中文描述

    /**
     * 私有构造,防止被外部调用
     * @param desc
     */
    private Day2(String desc)
    {
        this.desc=desc;
    }

    /**
     * 定义方法,返回描述,跟常规类的定义没区别
     * @return
     */
    public String getDesc()
    {
        return desc;
    }

    /**
     * 覆盖
     * @return
     */
    @Override
    public String toString()
    {
        return desc;
    }
}
```

```

public static void main(String[] args)
{
    for (Day2 day:Day2.values())
    {
        System.out.println("name:"+day.name()+
            ",desc:"+day.getDesc());
    }
}

/**
输出结果:
name:MONDAY,desc:星期一
name:TUESDAY,desc:星期二
name:WEDNESDAY,desc:星期三
name:THURSDAY,desc:星期四
name:FRIDAY,desc:星期五
name:SATURDAY,desc:星期六
name:SUNDAY,desc:星期日
*/
}

```

从上述代码可知，在 `enum` 类中确实可以像定义常规类一样声明变量或者成员方法。但是我们必须注意到，如果打算在 `enum` 类中定义方法，务必在声明完枚举实例后使用分号分开，倘若在枚举实例前定义任何方法，编译器都会报错，无法编译通过，同时即使自定义了构造函数且 `enum` 的定义结束，我们也永远无法手动调用构造函数创建枚举实例，毕竟这事只能由编译器执行。

## 7.5，枚举与switch

关于枚举与switch是个比较简单的话题，使用switch进行条件判断时，条件参数一般只能是整型，字符型。而枚举型确实也被switch所支持，在java 1.7后switch也对字符串进行了支持。这里我们简单看一下switch与枚举类型的使用：

```

enum Color
{
    GREEN,RED,BLUE
}

public class EnumDemo4
{
    public static void printName(Color color)
    {
        switch (color){
            case BLUE: //无需使用Color进行引用
                System.out.println("蓝色");
                break;
            case RED:
                System.out.println("红色");
                break;
            case GREEN:
                System.out.println("绿色");
        }
    }
}

```

```
        break;
    }
}

public static void main(String[] args)
{
    //调用
    printName(Color.BLUE);
    printName(Color.RED);
    printName(Color.GREEN);
    /*结果*/
    //蓝色
    //红色
    //绿色
}
}
```