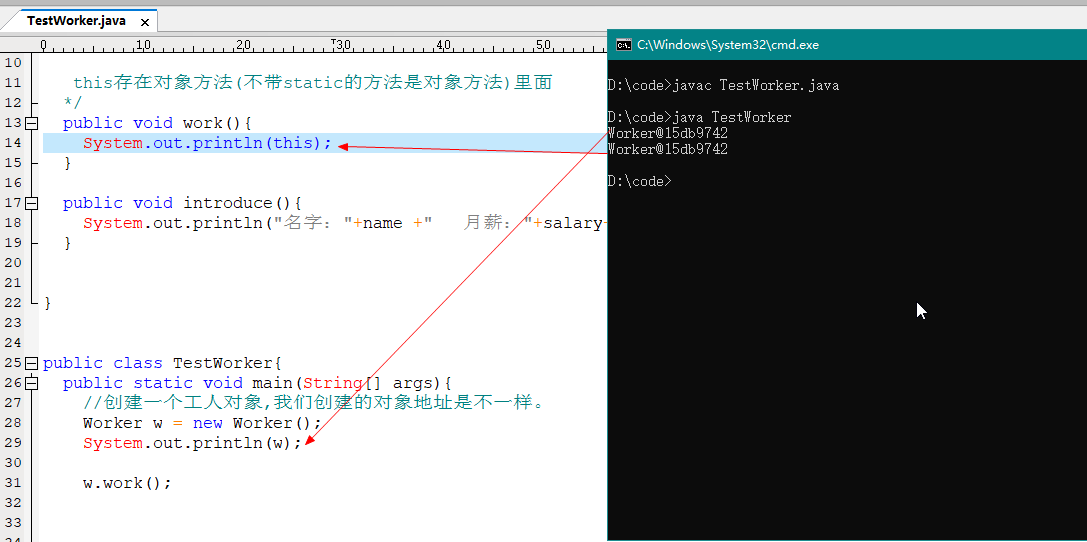
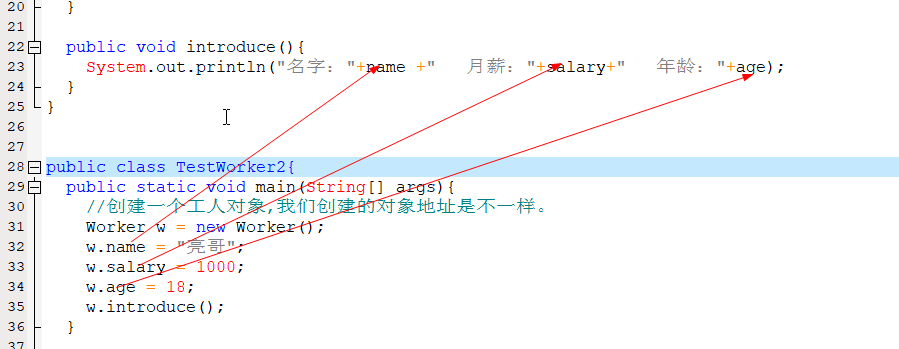
# This关键字

|  |
| --- |
| class Worker{    String name;    int salary;    int age;    /\*\*    this存在对象方法(不带static的方法是对象方法)里面  \*/  public void work(){  System.out.println(this);  }    public void introduce(){  System.out.println("名字："+name +" 月薪："+salary+" 年龄："+age);  }      }  public class TestWorker{  public static void main(String[] args){  //创建一个工人对象,我们创建的对象地址是不一样。  Worker w = new Worker();  System.out.println(w);  w.work();  }    } |



|  |
| --- |
| class Worker{    String name;    int salary;    int age;    /\*\*    this存在对象方法(不带static的方法是对象方法)里面    对象方法中隐藏了this关键字，this表示调用这个方法的那个对象    对象方法中this是可以省略的，效果是一样的  \*/  public void work(){  System.out.println(name);    }    public void introduce(){  System.out.println("名字："+name +" 月薪："+salary+" 年龄："+age);  }  }  public class TestWorker2{  public static void main(String[] args){  //创建一个工人对象,我们创建的对象地址是不一样。  Worker w = new Worker();  w.name = "亮哥";  w.salary = 1000;  w.age = 18;  w.introduce();  }    } |



this关键字的内存结构图：对象方法为什么是共享的。

栈 堆

main

0x98 new Worker()

0x99 new Worker ()

name = "亮哥";

salary = 1000;

age = 18;

introduce ()

Worker w

Worker w1

0x98

0x99

name = "明哥";

salary = 2000;

age = 16;

数据共享区

Worker.class

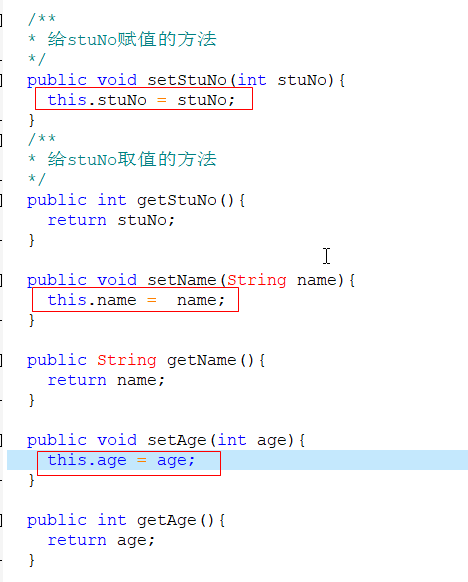
work()

introduce (){

System.out.println("名字："+this.name +" 月薪："+this.salary+" 年龄："+this.age);

}

This在封装中的应用：



# 构造器

用来创建对象的方法。

作用：创建对象，给对象赋初始值；

构造器本身是一个方法：

没有返回值，也没有void。

构造器的方法名必须和类名一致

在方法中定义要初始化的参数列表

## 默认构造器

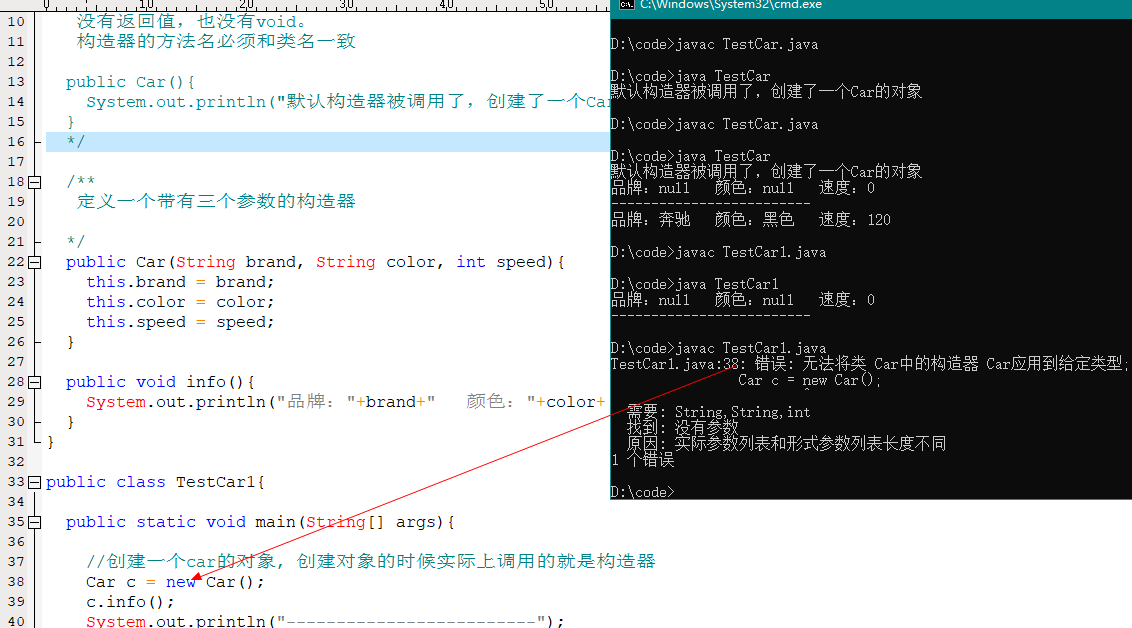
默认构造器是隐藏的

|  |
| --- |
| class Car{    String brand;    String color;    int speed;    /\*\*  没有返回值，也没有void。  构造器的方法名必须和类名一致  \*/  public Car(){  System.out.println("默认构造器被调用了，创建了一个Car的对象");  }    public void info(){  System.out.println("品牌："+brand+" 颜色："+color+ " 速度："+speed);  }  }  public class TestCar{    public static void main(String[] args){    //创建一个car的对象, 创建对象的时候实际上调用的就是构造器  Car c = new Car();    }  } |

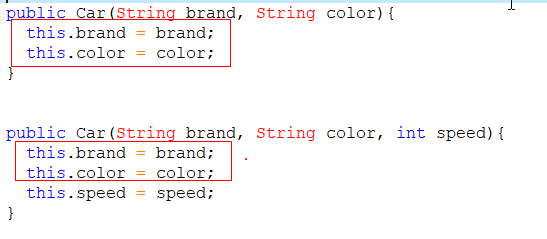
## 有参数的构造器

|  |
| --- |
| class Car{    String brand;    String color;    int speed;    /\*\*  没有返回值，也没有void。  构造器的方法名必须和类名一致  \*/  public Car(){  System.out.println("默认构造器被调用了，创建了一个Car的对象");  }      /\*\*  定义一个带有三个参数的构造器    \*/  public Car(String brand, String color, int speed){  this.brand = brand;  this.color = color;  this.speed = speed;  }    public void info(){  System.out.println("品牌："+brand+" 颜色："+color+ " 速度："+speed);  }  }  public class TestCar{    public static void main(String[] args){    //创建一个car的对象, 创建对象的时候实际上调用的就是构造器  Car c = new Car();  c.info();  System.out.println("-------------------------");    //使用有参数的构造器创建对象  Car c1 = new Car("奔驰","黑色",120);  c1.info();      }  } |

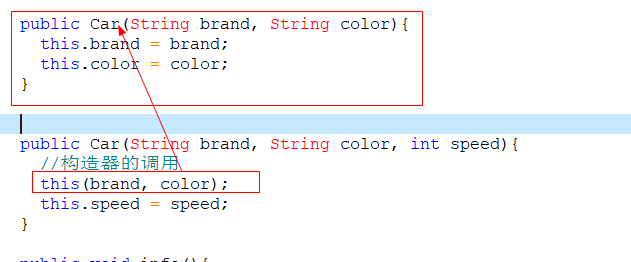
如果类存在有参的构造器的话，默认的构造器会被覆盖。如果还想要使用这个默认的构造器，就必须要显式的定义出来。



## 构造器之间的相互调用

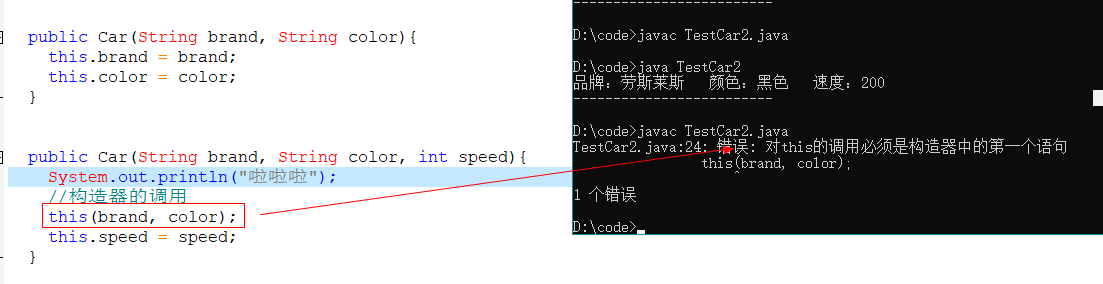


调用2个参数的构造器



|  |
| --- |
| class Car{    String brand;    String color;    int speed;      public Car(){    }      public Car(String brand, String color){  this.brand = brand;  this.color = color;  }    public Car(String brand, String color, int speed){  //构造器的调用  this(brand, color);  this.speed = speed;  }    public void info(){  System.out.println("品牌："+brand+" 颜色："+color+ " 速度："+speed);  }  }  public class TestCar2{    public static void main(String[] args){      Car c = new Car("劳斯莱斯", "黑色", 200);  c.info();  System.out.println("-------------------------");      }  } |

构造器调用构造器，被调用的构造器一定要放在第一行。



# Static关键字

## Static修饰在属性上

语法：static数据类型 变量名

使用static修饰的属性我们认为是类的属性，不带static修饰的属性我们认为是对象的属性。

栈 堆

name = “亮哥”

Age = 18

Addr = “火星”

0x98 New Person()

0x99 New Person()

Person p

Person p1

0x98

name = “发哥”

Age = 68

Addr = “地球”

0x99

数据共享区

Person.class

static int pcount =0;

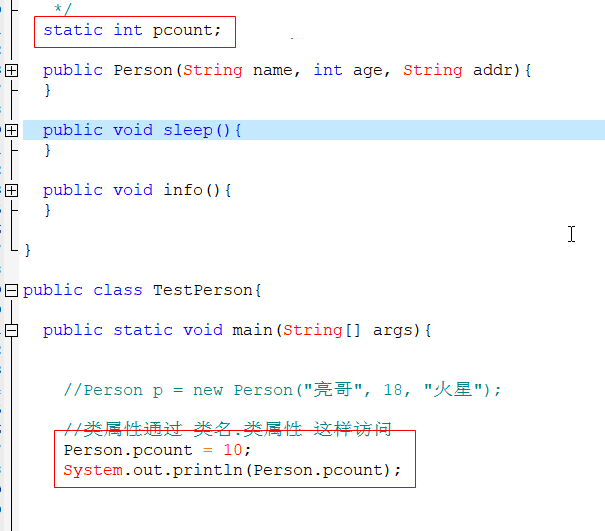
对象属性的生命周期

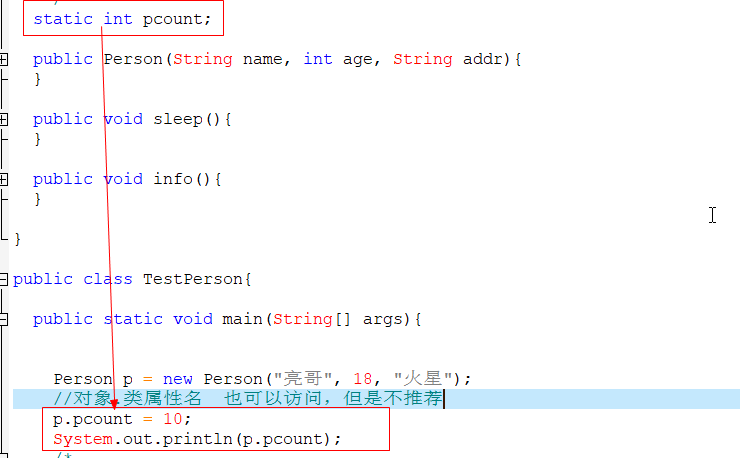
随着对象的创建二产生，对象如果被gc回收，属性自然也就不存在了

Static类属性生命周期

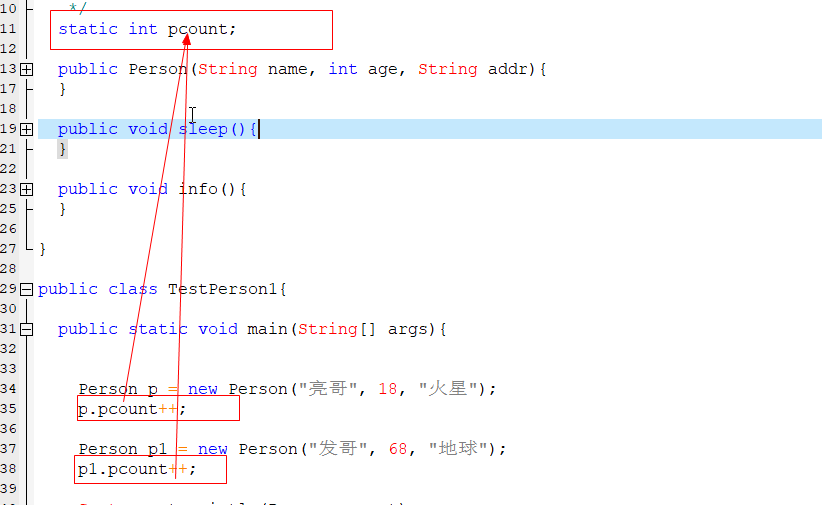
类文件被载入jvm的时候就会自动的初始化static的属性，jvm销毁后就会销毁类文件，static的属性也就被销毁了。

Static属性的使用。





类属性对每一个这个类所产生的对象都是共享的



## static关键字修饰在方法上

栈 堆

name = “亮哥”

Age = 18

Addr = “火星”

0x98 New Person()

0x99 New Person()

Person p

Person p1

0x98

name = “发哥”

Age = 68

Addr = “地球”

0x99

数据共享区

Person.class

Static int pcount;

Sleep(){

System.out.println(this.name+"在睡觉");

}

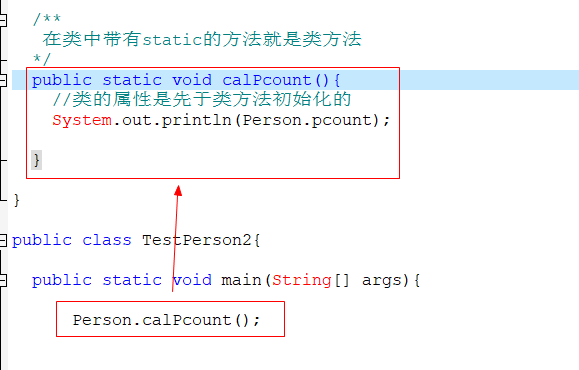
Static calPcount(){

System.out.println(Person.pcount);

}

对象方法是共享的，但是这个对象方法一定是创建了对象之后才能调用。对象方法的生命周期是在对象创建之后。

类方法是在static的属性初始化完毕初始化。类方法的调用也是通过类名.方法名



Static类方法只能访问static类属性。



对象方法可以访问static的类属性和类方法。

|  |
| --- |
| class Person{    String name;    int age;    String addr;  /\*\*  类属性  \*/  static int pcount;    public Person(String name, int age, String addr){  this.name = name;  this.age = age;  this.addr = addr;  }      /\*\*    在类中不带有static的方法是对象方法    \*/  public void sleep(){  System.out.println(name+"在睡觉");  System.out.println(pcount);  //调用对象方法  info();  //对象方法能够调用类方法  calPcount();  }    public void info(){  System.out.println("姓名："+name+ " 年龄："+age+ " 地址："+addr);  }    /\*\*  在类中带有static的方法就是类方法  \*/  public static void calPcount(){  //如果在同一个类中static的方法访问static的属性可以省略类名字。  System.out.println(pcount);    }    }  public class TestPerson4{    public static void main(String[] args){    Person p = new Person("aa", 18, "ggg");  p.sleep();          }  } |

# 单例模式

懒汉

|  |
| --- |
| class Singleton{    //定义一个当前类的属性  static Singleton s;      /\*\*  构造器的初始化  \*/  private Singleton(){    }    /\*\*  创建一个公有的方法返回一个Singleton这个类的对象的实例  \*/  public static Singleton getInstance(){  if(s == null){  s = new Singleton();  }  return s;  }      }  /\*\*  单例模式：一个类只能创建一个对象, 懒汉  \*/  public class TestSingleton{    public static void main(String[] args){  Singleton s = Singleton.getInstance();  System.out.println(s);    Singleton s1 = Singleton.getInstance();  System.out.println(s1);    Singleton s2 = Singleton.getInstance();  System.out.println(s2);    }    } |

饿汉

|  |
| --- |
| class Singleton{    //定义一个当前类的属性  static Singleton s = new Singleton();  /\*\*  构造器的初始化  \*/  private Singleton(){    }    /\*\*  创建一个公有的方法返回一个Singleton这个类的对象的实例  \*/  public static Singleton getInstance(){    return s;  }      }  /\*\*  单例模式：一个类只能创建一个对象, 饿汉  \*/  public class TestSingleton1{    public static void main(String[] args){  Singleton s = Singleton.getInstance();  System.out.println(s);    Singleton s1 = Singleton.getInstance();  System.out.println(s1);    Singleton s2 = Singleton.getInstance();  System.out.println(s2);    }    } |