# 《疯狂Java讲义》源代码

<http://pan.baidu.com/s/1i5vhQlB>

# 1.this关键字

## 1.this关键字的重要特性以及使用

<http://paste.ubuntu.com/25211455/>

## 2.this关键字在什么时候不能省略

<http://paste.ubuntu.com/25211573/>

## 3.this关键字总结以及this关键字在无参构造函数中的作用

<http://paste.ubuntu.com/25212199/>

# java的继承

## 所有的class都继承自Object类

它里面包含了许多通用方法（也就是大部分类的对象都会有的行为）

这里介绍四种

1. equals(Object o)

判断两个个对象是否可以被认为是“相等”的

**public** **class** Test {

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

Dog a = **new** Dog();

Cat c = **new** Cat();

**if**(a.equals(c)) System.***out***.println("true");

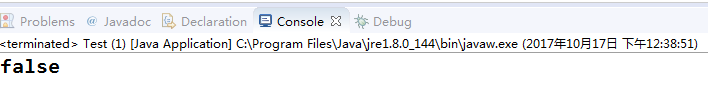
**else** System.***out***.println("false");

}

}

**class** Dog {}

**class** Cat {}



1. getClass()

让你知道对象是从哪里初始化的

**public** **class** Test {

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

Dog a = **new** Dog();

Cat c = **new** Cat();

System.***out***.println(a.getClass());

System.***out***.println(c.getClass());

}

}

**class** Dog {}

**class** Cat {}

3、hashCode()

给出对象的哈希码，可以认为是一个唯一的ID

**public** **class** Test {

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

Dog a = **new** Dog();

Cat c = **new** Cat();

System.***out***.println(a.hashCode());

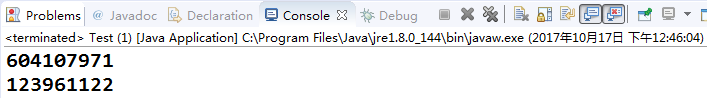
System.***out***.println(c.hashCode());

}

}

**class** Dog {}

**class** Cat {}



1. toString()

列出类的名称和一个我们不关心的数字，可能是地址什么的

**public** **class** Test {

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

Dog a = **new** Dog();

Cat c = **new** Cat();

System.***out***.println(a.toString());

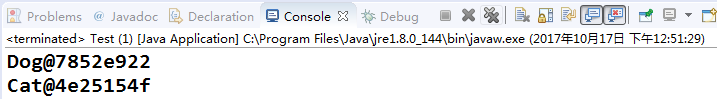
System.***out***.println(c.toString());

}

}

**class** Dog {}

**class** Cat {}



## 任何从ArrayList<Object>取出的东西都会被当作Object类型的引用而不管它原来是什么

使用Object类型的多态引用是需要付出一定的代价的

假如声明了一个存储Object型的myDogArrayList，用这个ArrayList来

存储Dog对象：

ArrayList<Object> myDogArrayList = new ArrayList<Object>();

然后向myDogArrayList中放入Dog对象：

Dog aDog = new Dog();

myDogArrayList.add(aDog);

进行完上面的操作之后，当从myDogArrayList中取出对象时

会出现错误：

Dog d = myDogArrayList.get(0)

无法通过编译，对ArrayList<Object>调用get()方法

会返回Object类型，编译器无法确认它是不是Dog





## 1、继承格式，使用extends：

<http://paste.ubuntu.com/25297296/>

## 2、java与C++在面向对象这方面，其中的一个不同之处就是java不支持多继承，而C++是支持的

## 3、构造方法是无法继承的（这一点C++和java一样，因为类名不一样，构造函数的名字也不一样）

## 4、super关键字的作用（访问父类各种数据）

<http://paste.ubuntu.com/25297522/>

## 5、子类中所有的构造方法默认都会先访问父类中空参的构造方法

## 因为子类会继承父类中的数据，可能还会使用父类的数据，所以在子类初始化前，一定要先完成父类数据的初始化

代码演示如下：（**super()**语句）

<http://paste.ubuntu.com/25297605/>

## 6、父类中没有无参构造方法时的解决方法（因为super()会自动访问父类无参构造函数）

<http://paste.ubuntu.com/25297848/>

# 3.static关键字和代码块（普通，构造，静态）

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**[普通代码块](http://paste.ubuntu.com/25302158/)**

**[构造代码块](http://paste.ubuntu.com/25302155/)**

**[静态代码块](http://paste.ubuntu.com/25302330/)**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

代码块一般分为以上三种

他们的执行顺序及特点如下：

普通代码块在main方法中定义，若在里面定义了变量，则代码块外部的语句无法访问代码块内部的变量

构造代码块一般写在类体中，每次类创建对象都会调用一次构造代码块，若有多个构造代码块，则按照先后顺序来执行

静态代码块在类加载时执行，也包括public类，也就是说静态代码块是随着类的加载执行的，因此它具有最高的优先级

且每一个static代码块只会执行一次，就是在类加载时执行

\*\*/

//代码演示：

class Code{

{

System.out.println("Code的构造块");

}

static{

System.out.println("Code的静态代码块");

}

public Code(){

System.out.println("Code的构造方法");

}

}

public class CodeBlock03{

{

System.out.println("CodeBlock03的构造块");

}

static{

System.out.println("CodeBlock03的静态代码块");

}

public CodeBlock03(){

System.out.println("CodeBlock03的构造方法");

}

public static void main(String[] args){

System.out.println("CodeBlock03的主方法");

new Code();

new Code();

new CodeBlock03();

new CodeBlock03();

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*

在这段代码中，会先执行public class CodeBlock03中的static代码块

~~输出"CodeBlock03的静态代码块"

因为肯定要先加载类，而加载类的同时，static语句就会被执行

然后进入main方法

~~输出"CodeBlock03的主方法"

接下来创建code对象，加载code类，执行code的static代码块

~~输出"Code的静态代码块"

然后再执行构造代码块

~~输出"Code的构造块"

接着Code构造方法执行

~~输出"Code的构造方法"

又一次创建Code对象，但这一次static代码块不会再执行，因为它只在类加载时执行，仅此一次

所以接下来会

~~输出"Code的构造块"

~~输出"Code的构造方法"

最后

~~输出"CodeBlock03的构造块"

~~输出"CodeBlock03的构造方法"

~~输出"CodeBlock03的构造块"

~~输出"CodeBlock03的构造方法"

创建了两次CodeBlock03对象，为什么static没有执行呢？

因为CodeBlock03在程序开始时已经执行过了，因为他是public类，是第一个加载的，static第一开始就执行过了

\*\*\*\*\*\*\*\*\*\*\*\*\*/

# 4.final关键字

## final：有点像C++中的const

## 修饰类,类不能被继承(final最后的  我是最后的了，没有类再能继承我)：

<http://paste.ubuntu.com/25323308/>

## 修饰变量，变量就成了常量，只能赋值一次：

<http://paste.ubuntu.com/25323314/>

## 修饰方法，方法就不能被重写，也就是说该方法的功能不能再被拓展：

<http://paste.ubuntu.com/25323288/>

# 多态及其应用实例

/\*

多态的概述：

事物存在的多种形态

多态的前提：

a、要有继承关系；

b、要有方法重写；

c、要有父类引用指向子类对象

\*/

多态的总结：

成员变量：

编译看左边（父类），运行看左边（父类）

成员方法：

编译看左边（父类），运行看右边（子类）----动态绑定

静态方法：

编译看左边（父类），运行看左边（父类）

既然是静态的，就不存在动态绑定这种东西了

只有非静态的成员方法才会存在编译看左边（父类），运行看右边（子类）的情况

**1、多态中成员变量的访问方式:**<http://paste.ubuntu.com/25331052/>

/\*

在多态中，成员变量与成员方法有着不同的特性

Animal A=new Cat();

A.eat();

在这段语句中，父类与子类都有eat方法，那么最终会调用子类方法

Animal A=new Cat();

System.out.println(A.num);

而在这段语句中，父类与子类中都有num这个变量；

但最终会调用父类的num

这就是多态中成员方法与成员函数的区别

\*/

**2、多态中成员方法的访问方式**:<http://paste.ubuntu.com/25331054/>

/\*

运行过程

多态中的方法，动态绑定，编译和运行时的情况是不一样的

比方说这条语句：

Animal A=new Cat();

A.eat();

在编译的时候，编译器会先去看看Animal类中是否有eat这个方法

然后在执行的时候，会去执行子类中的eat方法

如果将代码中父类的eat方法注释掉，则程序会报错，因为编译就没有通过

\*/

**3、多态中静态方法的访问方式**:<http://paste.ubuntu.com/25331086/>

### 多态的使用实例：

当父类与子类有共同的方法时，我们可以在主类中以一种更加方便的方式进行该共同方法的调用

我们可以在主类中建立一个method方法：

public static void method(父类 父类对象){

父类对象.共同的方法;

}

这样一来就能减少相当的代码量

详细代码：<http://paste.ubuntu.com/25332135/>

多态分析题

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public class A{

public static void main(String[] args){

Animal a=new Cat();

a.show();

}

}

class Animal{

public void show(){

show2();

}

public void show2(){

System.out.println("Cat show2");

}

}

class Cat extends Animal{

public void show2(){

System.out.println("Cat show2");

}

}

这段代码乍一看可能认为是错误的，

多态中，成员方法具有动态绑定的特性，即编译时看父类，执行时看子类

按照这样的逻辑，从代码上来看，子类中并没有show方法，因此会以为这段代码是无法正确执行的

但其实由于Cat继承了Animal，所以它是有show方法的，就是Animal中的方法，只不过隐含在其中，没有显示出来，

真正的Cat类代码应该是这样的：

class Cat extends Animal{

public void show(){

show2();

}

public void show2(){

System.out.println("Cat show2");

}

}

故而最后还是执行了show2方法

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# 6.方法的重写（子类重写父类方法）

方法的重写：

说白了就是子类中定义了与父类中一模一样的方法

子类想要对父类的功能进行拓展，这时候就要用super对父类的方法进行重写

说白了就是用 "super." 的方式调用父类中与子类同名的方法

如果不写 "super." 的话，父类中的同名方法将会被子类覆盖掉

/\*\*\*\*\*\*\*\*\*\*\*

public class c{

public static void main(String[] args){

zi s=new zi();

s.print();

s.method();

}

}

class fu{

public void print(){

System.out.println("fu print");

}

}

class zi extends fu{

public void method(){

System.out.println("zi print");

}

public void print(){

super.print();

System.out.println("zi print");

}

}

\*\*\*\*\*\*\*\*\*\*\*\*/

//注意事项：

## 1、子类不能重写父类中的private方法，因为子类根本就无法继承父类中的private方法

## 2、子类重写父类方法时，访问权限不能更低，否则编译会出错

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public class c{

public static void main(String[] args){

zi s=new zi();

s.print();

s.method();

}

}

class fu{

public void print(){

System.out.println("fu print");

}

}

class zi extends fu{

public void method(){

System.out.println("zi print");

}

//~~~~~~~~~~~~~

private void print(){

//~~~~~~~~~~~~~

super.print();

System.out.println("zi print");

}

}

// 错误提示：

// 错误: zi中的print()无法覆盖fu中的print()

        // private void print(){

                     // ^

// 正在尝试分配更低的访问权限; 以前为public

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

## 3、父类静态方法，子类也必须通过静态方法进行重写

## （其实静态的不算重写，静态只能覆盖）

使用super会出现这样的错误提示

错误: 无法从静态上下文中引用非静态 变量 super

                super.print();

所以无法重写，只能覆盖

/\*\*\*\*\*\*\*\*\*\*\*

public class c{

public static void main(String[] args){

zi s=new zi();

s.print();

s.method();

}

}

class fu{

public static void print(){

System.out.println("fu print");

}

}

class zi extends fu{

public void method(){

System.out.println("zi print");

}

public static void print(){

System.out.println("zi print");

}

}

\*\*\*\*\*\*\*\*\*\*\*\*/

# 7.抽象类与接口

**因为两者非常相似，就把它们写在了一起**

**抽象类**的概述及其特点：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

抽象类与抽象方法的定义需要使用关键字abstract

当我们无法具体描述一个类时

我们就可以把这样的类定义成抽象类

当你不知道一种方法具体是怎样实现的时候

我们就可以把这样的方法定义成抽象方法

抽象方法必存在于抽象类中

抽象类不一定有抽象方法

但有抽象方法的类一定是抽象类

抽象类不能被实例化：<http://paste.ubuntu.com/25336633/>

按照多态的方式，由具体的子类实例化

其实这也是多态的一种，抽象类多态

抽象类的子类：

要么是抽象类

要么重写父类中所有的的抽象方法

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~·~**

**接口**的概述及其特点：（跟抽象类很像）

## 接口的一个代码示例：

Outpu.java，第一段代码，定义一个接口Outpu

package heqile\_1**;**

public interface Output **{**

int MAX\_CACHE\_LINE **=** 50**;**

void out**();**

void getData**(**String mag**);**

**default** void print**(**String**...** msgs**)** **{**

**for(**String msg**:** msgs**)**

System**.**out**.**println**(**msg**);**

**}**

**default** void test**()** **{**

System**.**out**.**println**(**"默认的test()方法"**);**

**}**

static String staticTest**()** **{**

**return** "接口里的方法"**;**

**}**

**}**

Printer**.**java，第二段代码，定义实现类Printer

package heqile\_3**;**

**import** heqile**.**Output//com.heqile包中的Output接口

interface Product **{**

int getProduceTime**();**//定义接口Product，并在其中定义抽象方法getProduceTime()

**}**

public class Printer **implements** Output**,** Product **{**//定义类Printer来实现接口Output和Product

private String**[]** printData **=** **new** String**[**MAX\_CACHE\_LINE**];**//创建一个字符数组printData

private int dataNum **=** 0**;**//将成员变量dataNum初始化为0

public void out**()** **{**//定义out()方法,用来输出存进去的字符串

**while(**dataNum **>** 0**)** **{**

System**.**out**.**println**(**"打印机打印：" **+** printData**[**0**]);;**

System**.**arraycopy**(**printData**,** 1 **,** printData**,** 0**,** **--**dataNum**);**

//调用lang包中的arraycopy()方法

**}**

**}**

public void getData**(**String mString **)** **{**//下达任务，打印指定字符串

**if(**dataNum **>=** MAX\_CACHE\_LINE**)** **{**//如果超过规定负荷，则输出出错提示

System**.**out**.**println**(**"输出队列已满，添加失败"**);**

**}**

**else** **{**

printData**[**dataNum**++]** **=** mString**;**//如还可继续接收字符，就将其存进字符数组中，并将脚标向后移动

**}**

**}**

public int getProduceTime**()** **{**//实现类必须重写父接口方法

**return** 45**;**

**}**

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

Output o **=** **new** Printer**();**

o**.**getData**(**"轻量级 Java EE企业应用实战"**);**

o**.**getData**(**"疯狂Java讲义"**);**

o**.**out**();**

o**.**getData**(**"疯狂Android讲义"**);**

o**.**getData**(**"疯狂Ajax讲义"**);**

o**.**out**();**

o**.**print**(**"孙悟空"**,** "猪八戒"**,** "白骨精"**);**

o**.**test**();**

System**.**out**.**println**(**Output**.**staticTest**());**

Product p **=** **new** Printer**();**

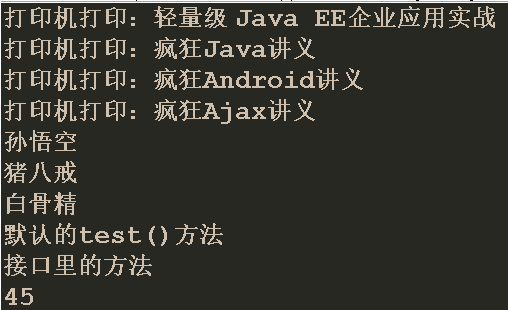
System**.**out**.**println**(**p**.**getProduceTime**());**

Object ojb **=** p**;**//所有接口类型的引用变量都可以直接赋给Object类型的变量

**}**

**}**

运行结果：



**利用多态的形式进行使用**

**Inter i=new Demo();**

**Demo是Inter的实现，class Demo implements Inter{}**

1、接口的定义用关键字:

interface 接口名 {}

接口中的方法都是抽象的

2、类实现接口用implements表示

class 类名 implements 接口名 {}

3、接口不能实例化(这一点与抽象类相似)

代码：<http://paste.ubuntu.com/25337256/>

接口的成员特点：

成员变量：

只能是常量，并且是静态公共的

默认修饰符：public static final(这三个关键字在顺序上不分先后)

构造方法：

接口没有构造方法

成员方法：

只能是抽象方法，不能定义非抽象方法，再有就是default修饰的默认方法和static修饰的类方法

默认修饰符：public abstract

**类与类，类与接口，接口与接口的关系：**

<http://paste.ubuntu.com/25337401/>

a、类与类：

继承关系，只能单继承，可以多层继承

b、类与接口：

实现关系，可以单实现，也可以多实现

并且还可以在继承一个类的同时实现多个接口

class Demo extends Object implements InterA,InterB{

public void printA(){

System.out.println("Inter");

}

}

c、接口与接口：

继承关系，可以单继承，也可以多继承

interface InterC extends InterA,InterB{

}

**抽象类与接口的设计理念区别：**

       抽象类：抽象类中定义的是该继承体系的共性功能

    接口：接口定义的是该继承体系的扩展功能

# 8.封装与隐藏

## 一、四种访问控制权限：

1、private：（当前类访问权限）

被修饰的类中的成员（变量和函数）只有在本类内部才能被访问

这个访问控制符最适合用来隐藏类的成员变量

2、default：（包访问权限）

这个是Java默认的一个修饰符，即在不写任何修饰符的情况下，自动添加的

所谓包访问权限，就是说一个类中的成员（变量和函数）如果没有使用任何修饰符,那么default访问控制的成员或外部类可被同一个包下的其他类访问

3、protected：（子类访问权限）

被protected修饰的成员既可以被同一个包里的其他类访问，

也可以被不同包里的类访问

通常情况下，如果一个方法被protected修饰，就是希望其子类来重写该方法

4、public：（公共访问权限）

这时四种访问控制权限中最宽松的访问控制级别

如果一个成员或者外部类使用public来进行修饰

那么这个成员或外部类就可以被所有类访问，不管访问类和被访问类是否在同一个包内，

或是否具有继承关系

访问控制级别表：

|private |default |protected |public

同一个类中 |可访问 |可访问 |可访问 |可访问

同一个包中 | |可访问 |可访问 |可访问

子类中 | | |可访问 |可访问

全局范围内 | | | |可访问

## 二、包的概念：

@@@@为了处理Java中的类的重名的问题，Oracle允许在类名前增加一个前缀来限定这个类

**Java引入了包（package）机制**，提供了类的多层命名空间，用于解决类名的冲突问题

Java允许将一组功能相关的类放在同一个package下，从而组成逻辑上的类库单元

将一个Java类放在指定package下的方式是，在java源文件的第一个非注释行上敲上如下代码:

package packageName;

eg:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package heqile;

public class A {

public static void main(String[] args){

System.out.println("heqile");

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

在命令行下进入这个Java文件路径，执行命令：

javac -d . A.java

即对A.java文件进行编译，并将生成的class文件保存在当前目录下

-d 选择保存路径

.就是保存在当前目录

但是在编译后，当前路径下并没有A.class文件，而是出现了名为heqile的文件夹

A.class文件就在这个目录下

在cmd中进入文件夹heqile的目录（注意：是文件夹heqile的目录，而不是A.class的目录）

执行命令java A

系统报错：找不到或无法加载主类 A.class

执行命令java heqile.A 运行正常

前面已经说过，Oracle公司在java中增添了package机制，就是给类的名字加前缀

而这个前缀就是包名（packageName）

所以在cmd中运行.class文件时也需要带上包名

### @@@@包和子包

上面已经有了A.class，此时我们再写一个Apple类

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package heqile.sub;//sub就是heqile的子包

public class Apple{

public static void main(String[] args){

}

public static void P(){

System.out.println("I have a pen");

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

此时要想在A中使用Apple类，就必须以这种形式：

heqile.sub.Apple.P();

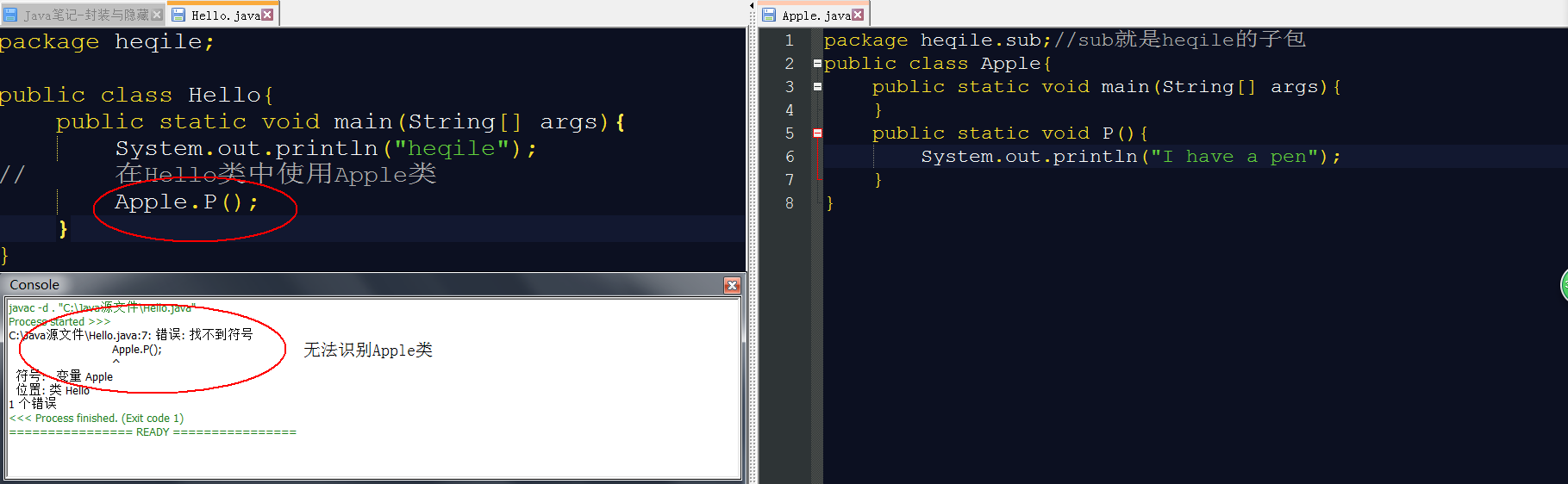
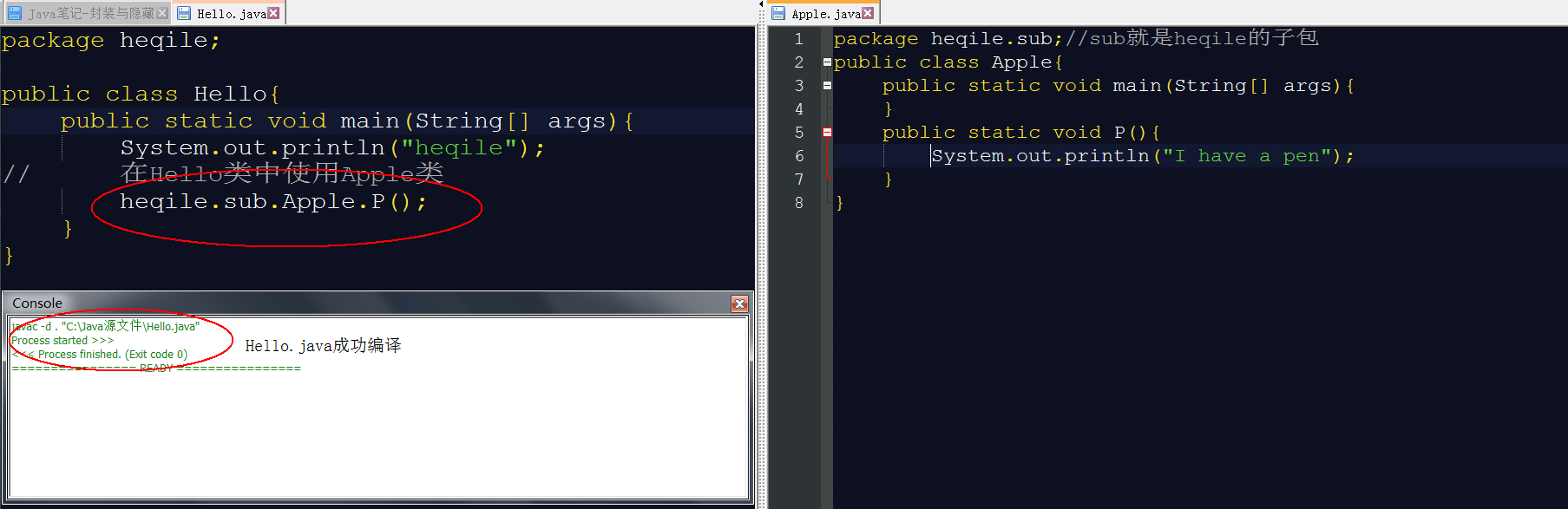
如果这样写

Apple.P();

编译报错，找不到Apple

因为此时Apple和A已经不在同一个包下，这时再使用Apple就需要用它的全称（包名+类名）

虽然A和Apple都在heqile包下，但仍然不能省略heqile



从这两张图中可以看到包名的效果

### @@@@import关键字

不同包中的类互相使用的时候，总是需要使用该类的全称，这样很麻烦

因此Java引入了关键字import用来导入指定的包

import导入单个类的方法：

import packageNmae.subpackage...ClassName;

例如上面的程序，要想导入Apple类，可这样操作：

import heqile.sub.Apple;

导入指定包中的全部类的方法：

import packageName.\*;

在上面的程序中，要想导入heqile.sub中全部的类，可这样操作：

import heqile.sub\*;

在一些极端情况下，import语句会失效，比如下面的：

import java.util.\*;

import java.sql.\*;

如果在接下来的程序中使用Date类，则会报错，因为java.util与java.sql中都有Date类

这时仍然需要在这样的类前面加上包名来区分

java.sql.Date d=new java.sql.Date();

JDK中还引入了静态导入的语法

可以用下面这种方式来导入指定包中的静态（变量/方法/整个类中的变量和方法）

import static package.subpackage...ClassName.fieldName;//导入静态变量

import static package.subpackage...ClassName.methodName;//导入静态方法

导入全部的静态变量和方法

import static package.subpackage...ClassName.\*;//导入类中所有的静态变量和方法

import总结起来就一句话：

减少代码量，导入包来省略包名的书写，

使用静态导入来省略的书写

eg:

import static java.lang.System.\*;

public class StaticImportTest{

public static void main(String[] args){

//此处直接省略了类名System

out,println("HelloWorld!!");

}

}

一段java程序的主体框架大致为：

package ……

import | import static ……

public classDefinition | interfaceDefinition | enumDefinition //0或1个public 类、接口或枚举定义

classDefinition | interfaceDefinition | enumDefinition //0或多个普通 类、接口或枚举定义

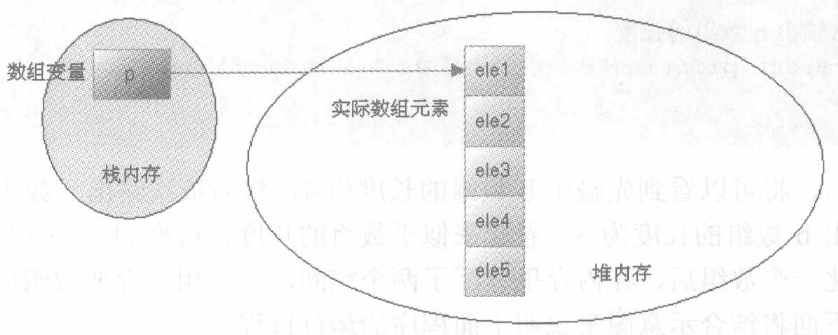
# 9.Java垃圾回收初步

## 数组：

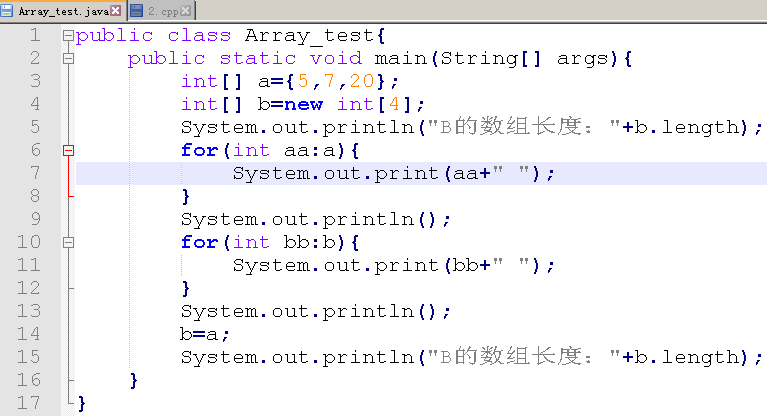
在Java中，数组引用变量只是一个引用，并非数组本身，这个引用变量可以指向任何有效的内存，只有当该引用指向有效内存之后，才可以通过该数组变量来访问数组元素。引用变量是访问真实数组的根本方式，即如果我们想访问一个数组，只能通过这个数组的引用变量来访问它。

数组在内存中的存储：

实际的数组元素被存储在堆内存中（heap），数组引用变量被存储在栈内存（stack）中，如下图所示：



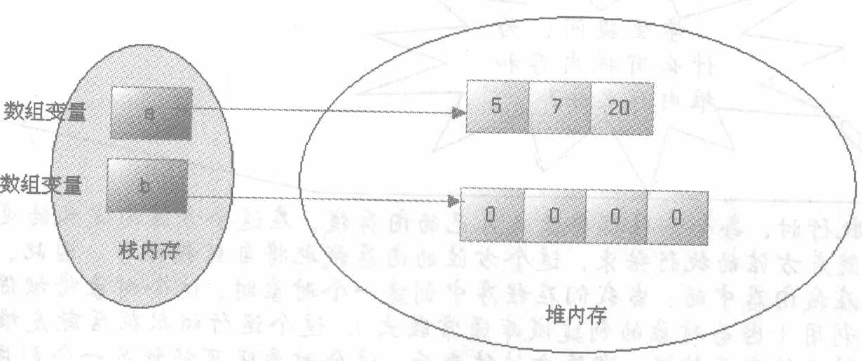
## 示例程序：



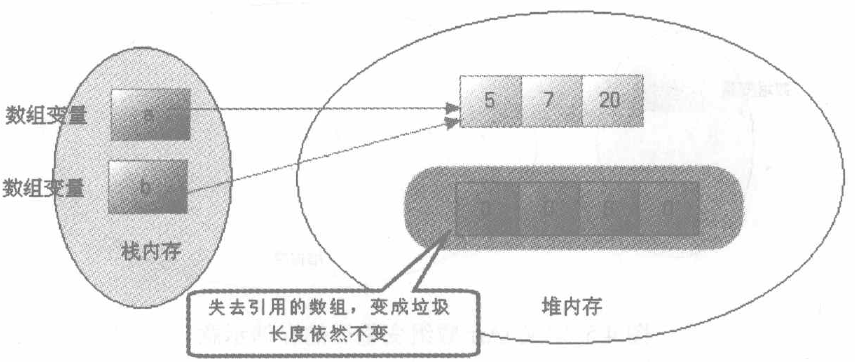
## 运行结果：b的长度变成了3，是因为引用变量b指向a所指向的数组所致



a和b只不过是数组的引用，它们与数组在内存中的存储状态是这样的：



**当程序执行到“a=b”时，b引用将指向a引用所指向的数组，此时原来b所指向的数组没有任何变量对其进行引用，它就成为了垃圾数组**



这时，Java垃圾回收机制会再合适的时候将其占用的内存收回，它就彻底消失了。

不光数组是这样，对象也是如此，如果没有任何引用指向该对象，那么这个对象将会被当作垃圾被Java回收。

# 10.类的组合

小贴士：

1.输出数组不一定非得用循环，还可以用下面这种方式进行遍历输出：

for(typName Parameter:arrayName){

System.out.println(typName);

}

示例：

int[] a={1,2,3};

for(int aa:a) {

System.out.println(aa);

}

2.instanceof 可用来判断一个对象是否可以进行强制类型转换；

要想进行强制类型转换，那么instanceof前的操作对象的类型要么与后面的类相同，

要么是后面的类的子类

eg:

Object hello="hello";

System.out.println(hello instanceof Object);

System.out.println(hello instanceof String);

这两个输出都是true，因为String是Object的子类

## @@@@利用类的组合实现类的复用

要想复用一个类，除了作为这个类的基类以外，我们还可以吧这个类嵌套在另一类当中

，由此就出现了类的组合。

我们把一个类的对象作为另一个类的组合成分，这样就可以通过这个对象来调用这个类

中的方法，下面给出继承和类的组合两种调用另一个类的方法的方式的代码：

类的继承：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public class One{

public static void main(String[] args){

Bird b=new Bird();

b.beat();

}

}

class Animal{

public void beat(){

System.out.println("心脏跳动");

}

}

class Bird extends Animal{

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

类的组合：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public class Test{

public static void main(String[] args){

Animal a=new Animal();

Bird b=new Bird(a);

b.beat();

}

}

class Animal{

public void beat(){

System.out.println("心脏跳动");

}

}

class Bird{

private Animal a;

public Bird(Animal a){

this.a=a;

}

public void beat(){

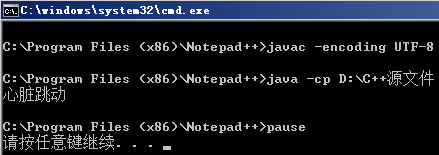
a.beat();

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

两者的输出：均为“心脏跳动”



关于继承与组合合：

继承是在已有类的基础上进行修改和完善，来使其适用于实际问题的解决

而组合则是为了表明两个类之间的包含与被包含关系

总结起来一句话就是，**继承强调的是（is是什么）关系**

而**组合强调的是（has有什么）关系**

# 11.包装类

虽然java是面向对象的语言，但是它仍然包含了8种基本数据类型，为了解决这8种基本数据类型的变量不能当成Object类型变量使用的问题，Java提供了**包装类**的概念

下表为基本数据类型与包装类的对应关系：

**基本数据类型** **包装类**

byte Byte

short Short

**int Integer**

long Long

**char Character**

float Float

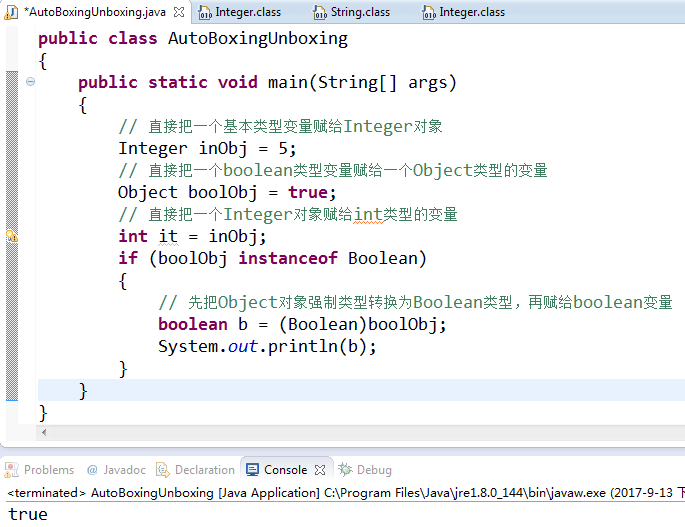
double Double

boolean Boolean

并且，为了减少工作量，java提供了自动装箱和拆箱功能

可以直接给Object对象赋值

示例：[代码](http://paste.ubuntu.com/25526962/)



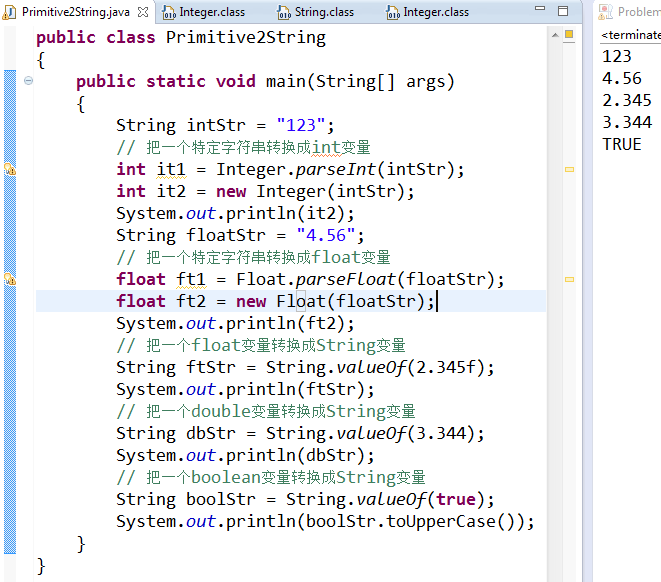
**除此之外，包装类还可以实现基本类型变量和字符串之间的转换，把字符串类型的值转换为基本类型的值有两种方式：**

1、利用包装类提供的**parseXxx(String s)静态方法**(除了Character之外的所有包装类都提供了该方法)

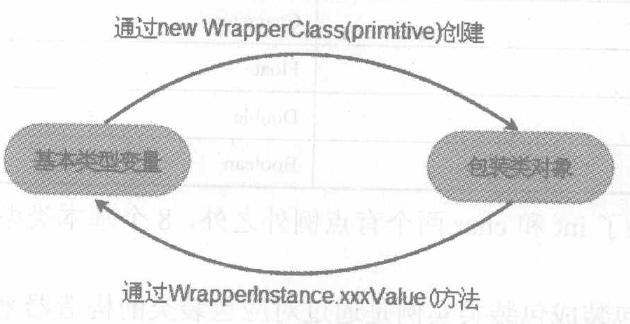
2、利用包装类提供的Xxx(String s)**构造器**

String类提供了多个重载**valueOf()方法**，用于将基本类型变量转换成字符串

示例：[代码](http://paste.ubuntu.com/25526970/)



基本类型变量和字符串之间的转换关系：



如果想要把基本数据类型转换为String型变量，有一种更加简单的方法，直接用”+”将基本类型变量和””连接在一起即可：  
 String intStr = 5 + "";

System.out.println(intStr);

**注意：**

包装类型的变量可以直接与数值类型的变量进行比较，这种比较是直接取出包装类实例所包装的值进行比较。。。。然而，两个包装类型的变量的比较就比较复杂了：

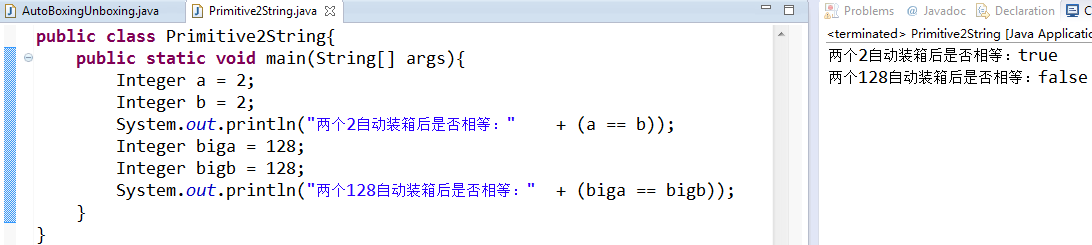
示例：[代码](http://paste.ubuntu.com/25527174/)



这个的输出结果是false，只有当两个包装类引用指向一个对象的时候才会返回true

自动装箱就是可以直接把一个基本类型值赋给一个包装实例，但这种情况下可能会出现一些特殊情形

示例：[代码](http://paste.ubuntu.com/25527323/)



当把128装箱之后再比较发现输出了false

这是为什么呢？

查看Integer的源代码就可知道，（其实我也不是非常明白）

他的工作原理是把一个-128~127之间的整数自动装箱成Integer实例，并放入一个名为cache的数组中缓存起来，

如果以后再有-128~127之间的整数自动装箱成一个Integer实例时，实际上就是直接指向对应的数组元素，因此-128~127之间的同一个整数自动装箱成Integer实例时，永远都是引用cache数组的同一个数组元素，所以他们全部相等。

但是当把一个不在-128~127范围内的整数自动装箱成Integer实例时，系统总是重新创建一个Integer实例，因而程序输出结果是false

## @@@@包装类的无符号算术运算功能：

示例：[代码](http://paste.ubuntu.com/25527781/)

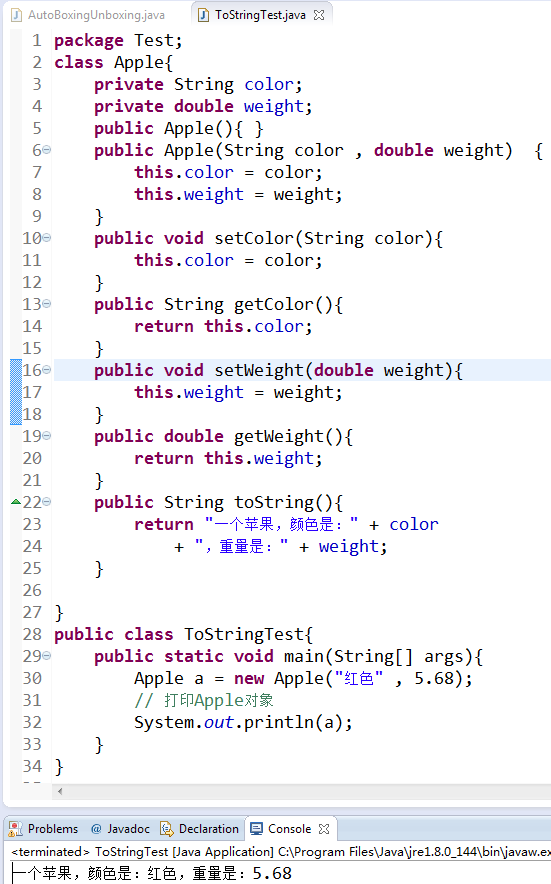
![L6X`H5PQOXDJ](HV)ZGAMFF](data:image/png;base64,)

# 12.处理对象

## @@@@打印对象和toString方法

toString方法是Object类的一个实例方法，又因为所有的类都是Object的子类，所以当执行语句System.out.println(p)时（p是一个对象），就相当于执行了System.out.println(p.toString())，toString()会返回一个字符串，用来显示主调对象的“自我描述”信息，toString()总是返回主钓对象实现类的“类名+@+hashCode”值，这个返回值并不能真正实现“自我描述”功能，因此需要重载该方法。

示例：[代码](http://paste.ubuntu.com/25552773/)



最后的输出语句，格式化的写法是这样的：

类名[field1=值1, field2=值2,.......]

Eg:

public String toString(){

return “Apple [color=” + color + “, weight = ” + weight + “]”;

}

通过重新定义toString()函数，实现了对象信息的完整输出。

## @@@@ == 和 equals方法

Java中测试两个变量是否相等的方式有两种，==和equals()方法。

利用==来判断两个变量是否相等时，如果两个变量是基本类型变量，且都是数值类型（数据类型不一定完全相同），只要两个变量的值相等，就返回true。

但是==在判断引用类型变量时具有一定的局限性，只有当两个引用类型变量指向同一个对象时，==才会返回true，以下程序演示了用==来比较两种类型变量是否相等的结果。

示例：[代码](http://paste.ubuntu.com/25552926/)

public class EqualTest {

public static void main(String[] args) {

int it = 65;

float fl = 65.0f;

System.*out*.println("65和65.0f是否相等？" + (it == fl));

char ch = 'A';

System.*out*.println("65和A是否相等？" + (it == ch));

String str1 = new String("hello");

String str2 = new String("hello");

System.*out*.println("str1和srt2是否相等？" + (str1 == str2));

System.*out*.println("str1是否equals srt2？" + (str1.equals(str2)));

}

}

简单地直接使用==来进行比较，输出了错误的结果，因此我们需要另一种判断方法来解决该问题。

首先来解释一下字符串直接量“hello”和new String(“hello“)的区别，要解释这个，还需要讲一下什么叫做常量池。（它俩的存储位置不同）

**常量池：**

常量池（constant pool）专门用来管理在编译时被确定并被保存在已编译的.class文件中的一些数据。它包括了关于类、方法、接口中的常量，还包括字符串常量。

JVM使用常量池管理字符串直接量的情形：

示例：[代码](http://paste.ubuntu.com/25554055/)



由该示例程序可以看出，能够放入常量池的都是显式表示的字符串，没有变量的参与，JVM常量池保证相同的字符串直接量只有一个，不会产生多个副本。

使用new String()创建的字符串对象是运行时创建出来的，它被保存在运行时内存（即堆内存）内，不会放入常量池中。

String类创建字符串对象的构造方法：

public String(char value[]) {

this.value = Arrays.copyOf(value, value.length);

}

其实就是传进去了一个字符数组。

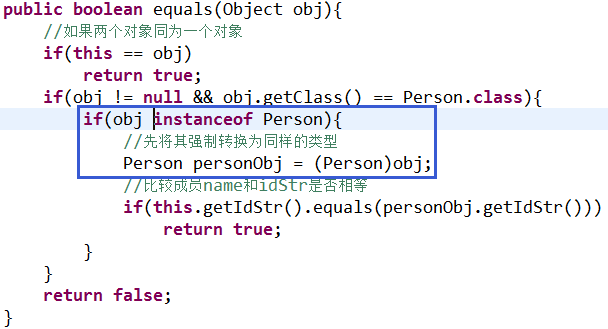
当Java直接使用形如“hello”这样的字符串直接量的时候，JVM会使用常量池来管理这些字符串；当使用new String(“hello”)时，JVM会先使用常量池来管理“hello”直接量，再调用String类的构造器来创建一个新的String对象，新创建的String对象被保存在堆内存中。换句话说，new String(“hello”)一共产生了两个字符串对象

要想真正使用equals()方法，需要像重载toString()方法一样来重载equals()方法。

**注意：**String类已经重写Object了的equals()方法，String的equals()方法判断两个字符串是否相等的标准是：只要这两个字符串所包含的字符序列相同，就返回true。

但许多我们自定义的类是需要我们自己去重写equals()方法，你标准由自己来制定。

示例：[代码](http://paste.ubuntu.com/25554331/)



Object默认提供的equals()方法只是比较对象的地址，即Object类提供的equals()方法和==运算符的比较结果完全相同。因此，在实际应用中常常需要重写equals()方法，相等的判断标准根据业务要求来决定。

# 13.类成员

所谓类成员，就是被static关键字所修饰的成员，static修饰的成员属于整个类，而不属于某一个实例。

**注意**：类成员不能访问实例成员，因为类成员是属于类的，类成员的作用域比实例成员的作用域更大，**完全可能出现类成员已经初始化完成，实例成员还未初始化的情况，如果允许类成员访问实例成员将会引起大量错误。**类成员是在类加载时刻初始化的。

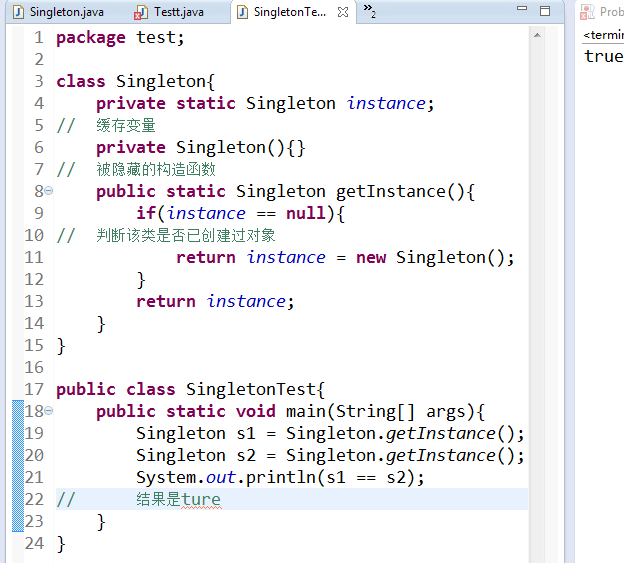
## @@@@单例(Singleton)类

在一些特殊的场景下，**不允许自由创建该类的对象**，而只允许为该类型创建一个对象，为了避免其他类创建该类对象，**需要把这个类的构造方法用private修饰**，从而达到隐藏该类构造器的目的。

一旦隐藏了一个类的构造器，就必须提供一个public方法作为该类的访问点，用于创建该类的对象，**注意方法必须用static修饰，由于隐藏了构造器，所以在调用这个方法之前是没有对象的，只能由类来调用。**

为了保证该类只有一个对象，需要在类中创建一个缓存变量来保存曾经创建过的对象，因为这个变量需要被上面的静态方法访问，所以也需要定义为static变量。

示例：[代码](http://paste.ubuntu.com/25576263/)



# 14.final修饰符详解

## @@@@宏替换（final变量），再谈常量池

一个变量被final修饰，

在定义该final变量时指定了初始值

该初始值可以在编译时就确定下来

满足以上三个条件的变量就不再是变量了，而是一个直接量

在如下的程序中，其实a根本就不存在，程序直接将所有的a变量替换成了5这个直接量。

public class FinalLocalTest{

public static void main(String[] args){

final int a = 5;

System.out.println(a);

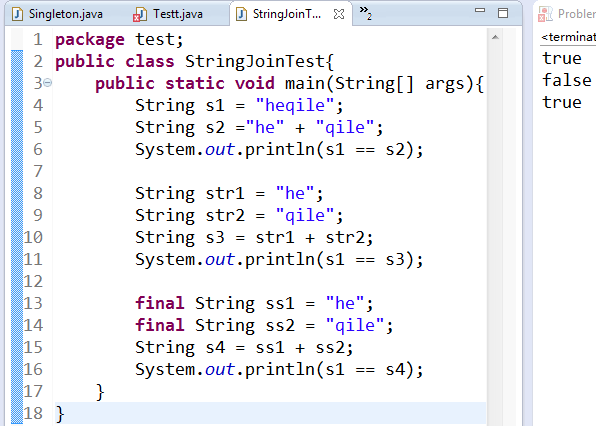
}

}

前面提到过常量池，下面再来明确一下它的定义，常量池里放的都是直接量，什么叫做直接量呢？就是没有变量参与，在编译时就已经确定下来的量（编译的那一刻）。

比如：每一个字符串都是一个String类型的对象，**”heqile”这就是一个直接量，可直接放入常量池中，它是不会变的，只是指向它的引用变量会改变而已**，但是如果是str = “heqile”，那么str就不是直接量，要想让str成为直接量，只需要在它的前面加上final修饰符即可。

示例：[代码](http://paste.ubuntu.com/25576443/)

 注意：final方法不可以被重写，但完全可以被重载。

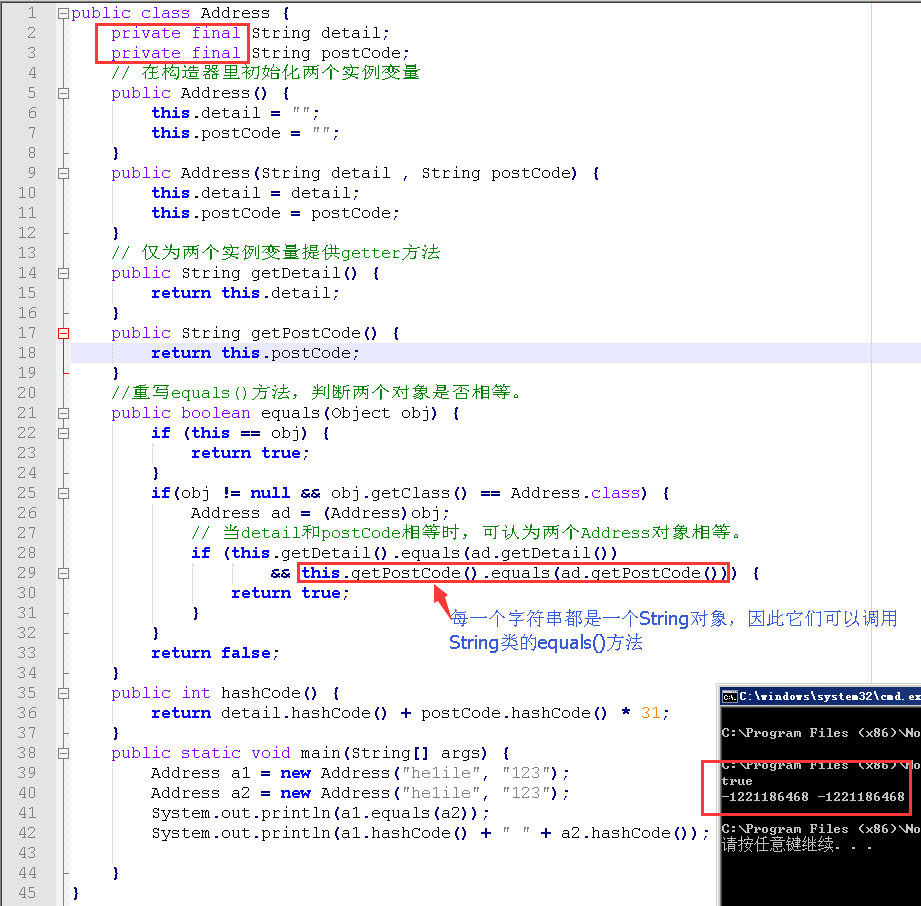
### @@@@不可变类

不可变类就是创建该类的实例后，该实例的实例变量是不可以被改变的，例如前面提到过的包装类和java.lang.String类都是不可变类，他们并没有提供修改实例变量的方法。

自定义不可变类需要遵守以下规则：

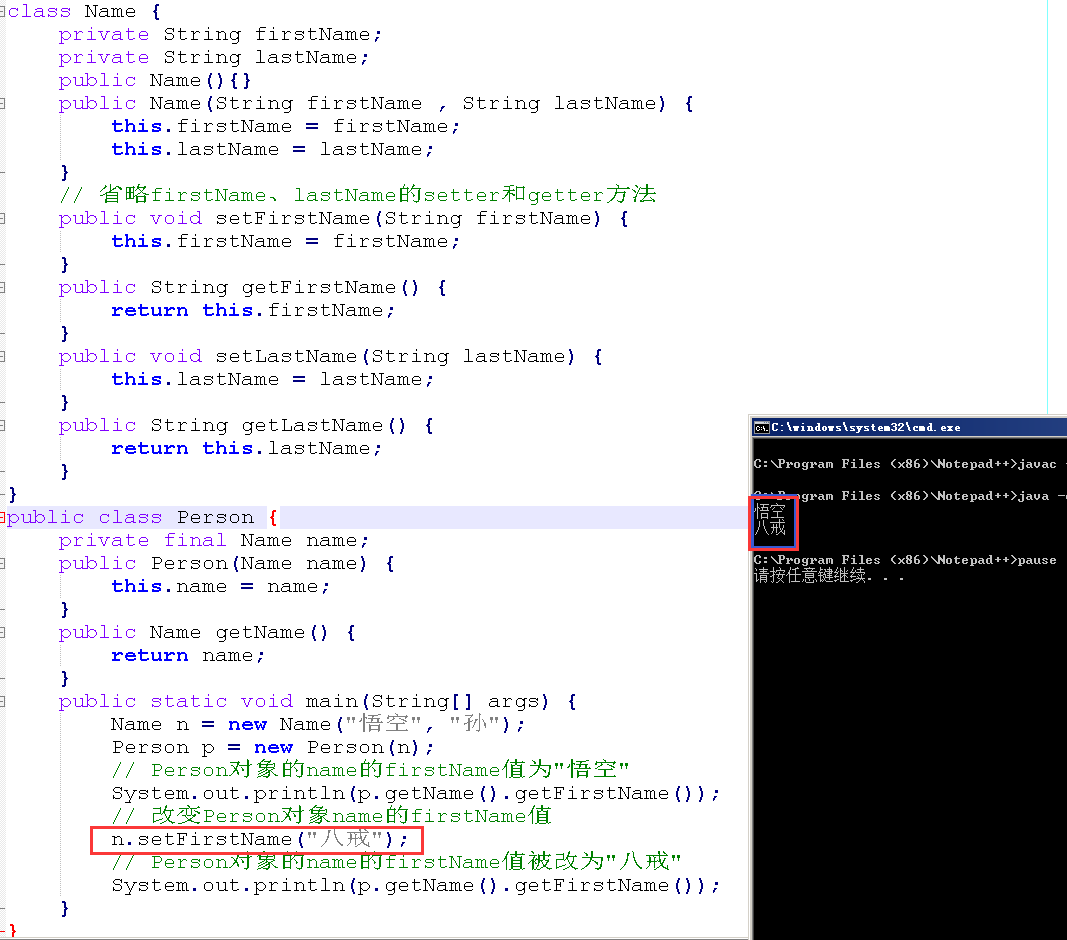
1. 使用private和final修饰符来修饰该类的成员变量
2. 提供带参数构造器
3. 仅提供getter()，不提供setter()
4. 如果有必要，需要重写Object类的hashCode()和equals()方法

示例：[代码](http://paste.ubuntu.com/25603933/)

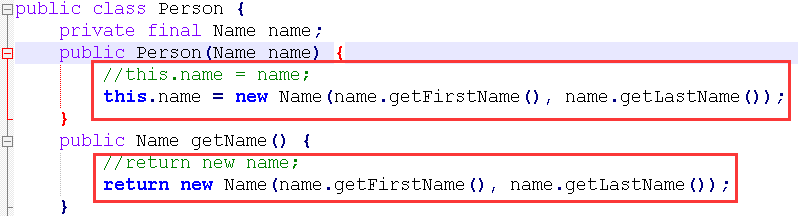


如果在设计不可变类的时候，在类中声明了一个类型可变的成员变量，那么这个不可变类就是失败的，因为它的成员变量的值是可以被修改的。

示例：[代码](http://paste.ubuntu.com/25603902/)



当一个不可变类中声明了类型可变的成员变量时，必须保护好该不可变类的引用类型的成员变量，在上面的程序中，要让程序无法访问到Person对象的name成员变量，将程序部分修改之后，即可解决该问题。  
示例：[代码](http://paste.ubuntu.com/25604070/)





修改之后，程序运行结果是这样的。

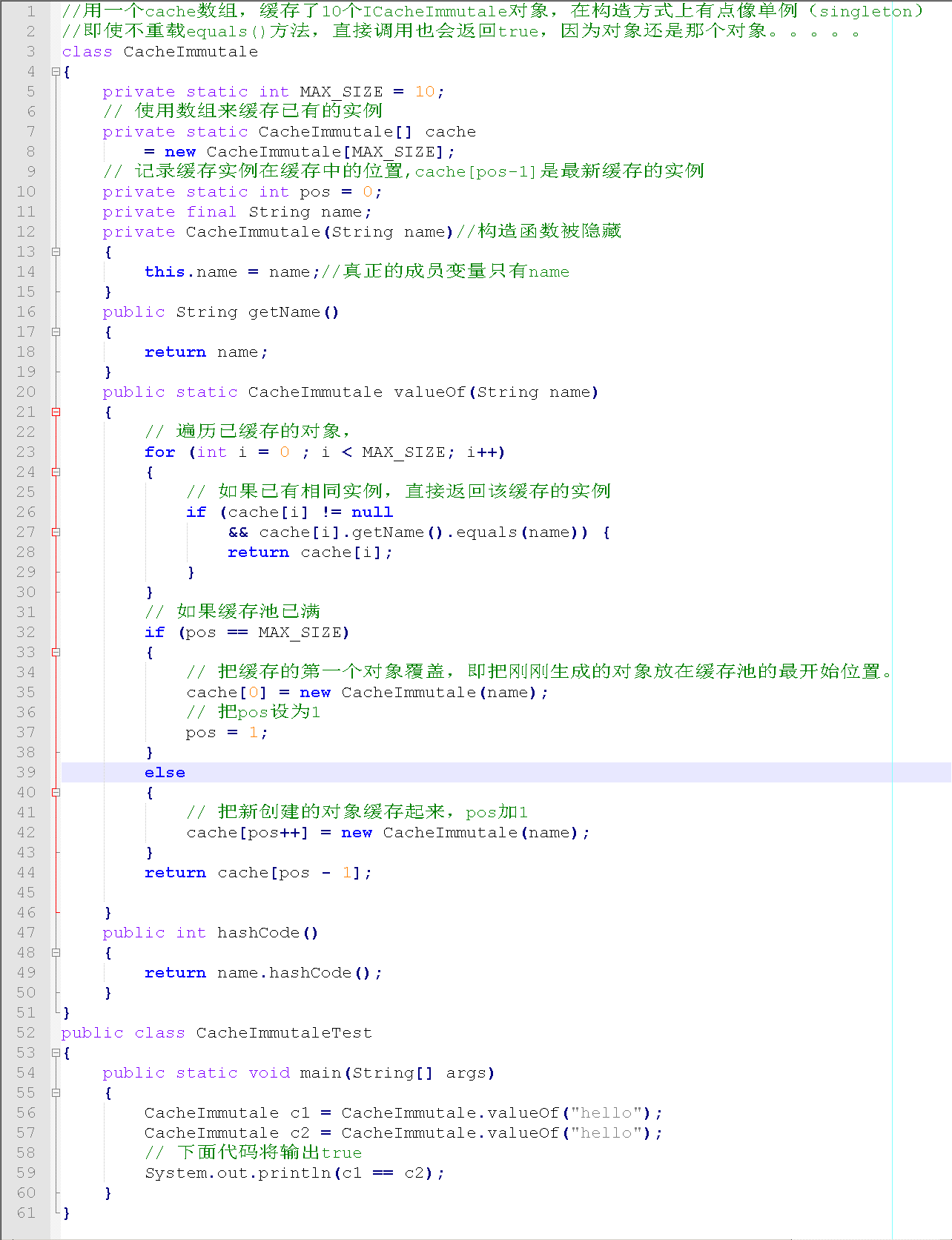
修改之后的代码，在Person类的构造函数并没有直接使用传进来的name对象，而是在构造函数中临时创建了一个Name对象，这样一来，更改在main函数中创建的Name对象并不会影响到Person对象的成员变量name，于是就达到了使外界无法访问不可变类成员的目的。

### @@@@缓存实例的不可变类

不可变类旳实例状态是不可改变的，可以很方便地被多个对象所共享，如果程序经常需要访问相同的不可变类实例，就应该考虑把该不可变实例缓存起来以减少系统开销，如果可能，应该将这个已经创建的不可变类的实例进行缓存。

缓存是软件设计中一个非常有用的模式，缓存的实现方式有很多种，不同的实现方式可能有较大的性能差别，（具体是什么差别，目前我也布吉岛）

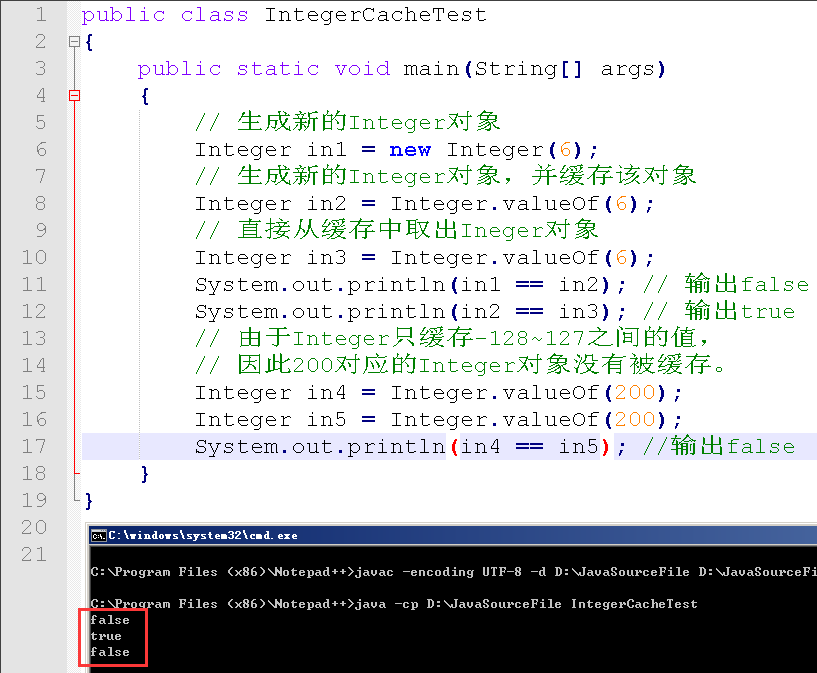
示例：[代码](http://paste.ubuntu.com/25604964/)



来看一下缓存在Java语言中的应用，拿包装类Integer来说吧。。在Integer包装类中，使用new直接创建对象，每次都会返回一个全新的Integer对象，但是如果调用valueOf()方法来创建对象，就会缓存该方法所创建的对象，下次需要相同的对象时，直接从缓存中将其取出来。

注意：只能缓存-128~127之间的值

示例：[代码](http://paste.ubuntu.com/25605028/)



# 内部类

**内部类的主要作用：**  
 1、提供更好的封装，可以把内部类隐藏在外部类（宿主类）之内，不允许同一个包内的其他类访问该类。就像是自己的成员一样，离开了外部类就没有任何意义。

1. 内部类成员可以直接访问外部类的私有数据，因为内部类被当作其外部类成员，同一个类的不同成员可以互相访问，**但是反过来却不行，**外部类无法访问内部类的实现细节，例如内部类的成员变量。
2. 匿名内部类适合用于创建那些仅需要使用一次的类，对于**接口中的[命令模式](#_命令模式)**，当需要传入一个Command对象时，，重新专门定义PrintCommand和AddCommand两个实现类可能没有太大的意义，因为这两个现实类可能只需要使用一次，在这种情况下，使用匿名内部类将更加方便。

**另：1、内部类比外部类可以多使用三个修饰符：private，static，protected**

1. **非静态内部类不能拥有静态成员**

## 非静态内部类

定义一个内部类是非常简单的，只要把一个类放在另外一个类中定义即可。此处的内部类**包括类中的任意位置**，甚至**在方法中也可以定义内部类**（方法里定义的内部类被称为**局部内部类**）。

内部类的**定义格式**如下：

**public** **class** OuterClass {

//定义内部类

}

**内部类比外部类多了两个访问控制符**是有原因的：

外部类的上一级程序单元是package，因此它的作用域只有两个，同一个包内（对应default//省略访问控制符）**||**任意位置（对应public//公开访问权限）；

而对于内部类而言，他的上一级程序单元是外部类，所以就拥有了4个作用域：同一个class（对应）**||**同一个package**||**父子类extends**||**任何位置，因此内部类可以使用四种访问控制权限。

**示例：**

**package** heqile;

**public** **class** Cow {

**private** **double** weight;

**public** Cow() {}

**public** Cow(**double** weight) {

**this**.weight = weight;

}

**private class CowLeg** {

**private** **double** length;

**private** String color;

**public** CowLeg() {}

**public** CowLeg(**double** length, String color) {

**this**.length = length;

**this**.color = color;

}

**public** **double** getLength() {

**return** length;

}

**public** **void** setLength(**double** length) {

**this**.length = length;

}

**public** String getColor() {

**return** color;

}

**public** **void** setColor(String color) {

**this**.color = color;

}

**public** **void** info() {

System.***out***.println("当前牛腿颜色是：" + color + "，高：" + length);

System.***out***.println("本牛腿所在奶牛重：" + weight);

//**内部类中的成员方法直接访问到了外部类的成员变量**

}

}

**public** **void** test() {

CowLeg c1 = **new** CowLeg(1.12, "黑白相间");

c1.info();

}

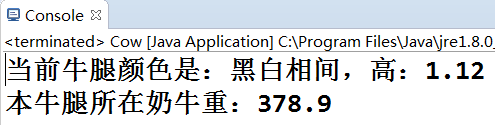
**public** **static** **void** main(String[] args) {

Cow cow = **new** Cow(378.9);

cow.test();

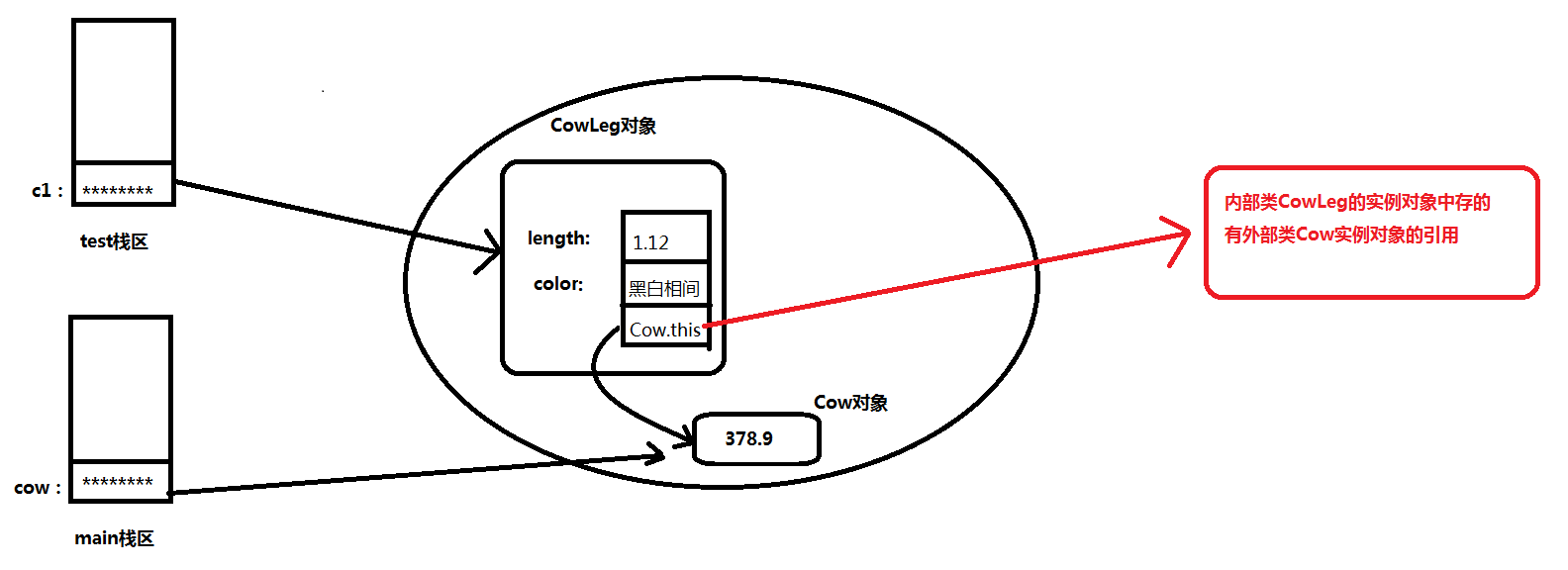
}

}



在packag的目录下会生成两个.class文件，一个是**OuterClass.class**，一个是**OuterClass$InnerClass.class**,前者是**外部类的文件形式**，后者是**内部类的文件形式。**

**上述程序运行时的内存示意图：**

CowLeg类的方法能够访问其外部类的private实例变量，是因为在非静态内部类对象里，保存了一个它所寄生的外部类对象的引用（**当调用非静态内部类实例方法时，必须有一个非静态内部类实例，非静态内部类实例必须寄生在外部类实例里**）

当在非静态内部类的方法访问某个变量时，系统优先在该**方法内**查找是否存在该名字的局部变量，如果存在就使用该变量；如果不存在，就到**该方法所在的内部类**中去找；如果不在，就到**该内部类的外部类**中去找，如果存在则使用，否则编译错误：找不到该变量。

**因此，如果*外部类成员变量、内部类成员变量与内部类方法里的局部变量*同名，则可*以通过this，外部类类名.this*作为限定来进行区分。**

**示例：**

**package** heqile;

**public** **class** DiscernVriable {

**private** String prop = "外部类的实例变量";

**private** **class** InClass {

**private** String prop = "内部类的实例变量";

**public** **void** info() {

String prop = "局部变量";

System.***out***.println("外部类的实例变量值：" + DiscernVriable.**this**.prop);

System.***out***.println("内部类的实例变量值：" + **this**.prop);

System.***out***.println("局部变量的值：" + prop);

}

}

**public** **void** test() {

InClass in = **new** InClass();

in.info();

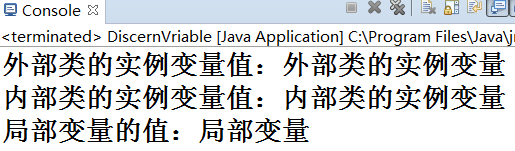
}

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

**new** DiscernVriable().test();

}

}



**----------------------------------------------我是一条严肃的分割线--------------------------------------------------**

**外部类不能直接访问内部类的private成员**，如果外部类需要访问非静态内部类的的成员，则必须显式创建非静态内部类对象来调用访问其实例成员。

**示例：**

**public** **class** Outer {

**private** **int** outProp = 9;

**class** Inner {

**private** **int** inProp = 5;

**public** **void** accessOuterProp() {

System.***out***.println("外部类的outProp值：" + outProp);

}

}

**public** **void** accessInnerProp() {

//System.out.println("内部类的inProp值：" + inProp);

//上面这条输出语句将会引起编译错误，外部类无法直接访问非静态内部类的实例变量

System.***out***.println("内部类的inProp值：" + **new** Inner().inProp);

//**创建了一个内部类对象，然后再使用该对象访问内部类的inProp变量**

}

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

Outer out = **new** Outer();

**//执行完这句代码之后，只是创建了外部类对象**

**//内部类对象还未被创建**

out.accessInnerProp();

}

}

**----------------------------------------------我是一条严肃的分割线--------------------------------------------------**

不允许外部类直接访问非静态内部类实例变量也是有原因的,因为非静态内部类对象必须是寄生在外部类对象里，所以**如果存在一个非静态内部类对象，那么就一定存在一个被它所寄生的外部类对象，故非静态内部类可以直接访问其外部类的成员变量。**

但是反过来却不一定，外部类对象里不一定寄生了一个非静态内部类对象，**因此外部类对象访问非静态内部类成员时，可能非静态普通内部类对象根本就不存在。**

**----------------------------------------------我是一条严肃的分割线--------------------------------------------------**

根据**静态成员不能访问非静态成员**的规则，外部类的静态方法、静态代码块不能访问非静态内部类，且不能使用非静态内部类定义变量、创建实例等。一句话就是：**不允许在外部类的静态成员中直接使用非静态内部类。**

**示例：**

public class StaticTest **{**

private class In **{}**

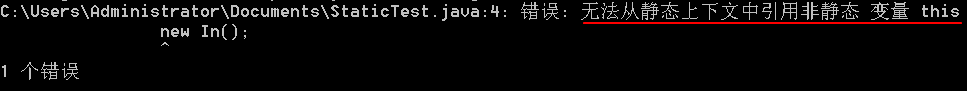
public static void main**(**String**[]** args**)** **{**

**new** In**();**

**}**

**}**

**上面这段代码会引发以下错误：**



**----------------------------------------------我是一条严肃的分割线--------------------------------------------------**

**Java不允许在非静态内部类中定义静态成员。**

**示例：**

**public** **class** InnerNoStatic {

**private** **class** InnerClass {

**static** {

System.***out***.println("Test Test");

}

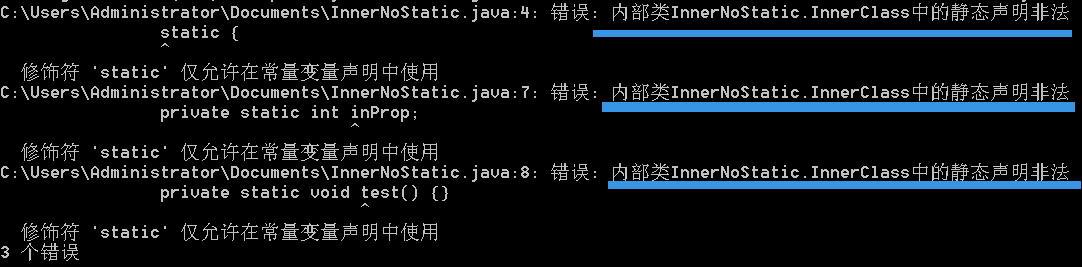
**private** **static** **int** *inProp*;

**private** **static** **void** test() {}

}

}

**上述代码会引发以下错误：**

-

## 静态内部类

如果使用static来修饰一个内部类，那么这个内部类就属于外部类本身，而不属于外部类的某个对象，被static修饰的内部类被称为**类内部类**或**静态内部类。**

静态内部类可以拥有静态成员和非静态成员，根据静态成员不能访问非静态成员的规则，**静态内部类**不能访问外部类的实例成员，**只能访问外部类的类成员**。即使是静态内部类的**实例方法也不能访问外部类的实例成员**，**只能访问外部类的静态成员**。

**示例：**

**public** **class** StaticInnerTest {

**private** **int** prop1 = 5;

**private** **static** **int** *prop2* =9;

**static** **class** StaticInnerClass {

**private** **static** **int** *age*;

**public** **void** accessOuterProp() {

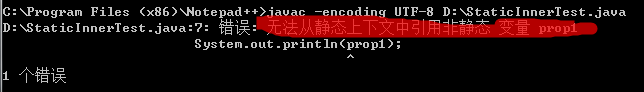
System.***out***.println(prop1);

**//这行代码将会引起错误，静态内部类无法访问外部类的实例变量**

System.***out***.println(*prop2*);

}

}

}

因为静态内部类是外部类的类相关的，而不是外部类的对象相关的。就是说，**静态内部类对象并不是寄生在外部类的实例中，而是寄生在外部类的类本身中**。**当静态内部类对象存在时，并不存在一个被它寄生的外部类对象，静态内部类对象只持有外部类对象的类引用，没有持有外部类对象的引用**。**如果允许静态内部类的实例方法访问外部类的实例成员，但找不到被寄生的外部类对象，将会引起编译错误**

**----------------------------------------------我是一条严肃的分割线--------------------------------------------------**

静态内部类是外部类的一个静态成员，因此外部类的所有方法、所有初始化块中可以使用静态内容来定义变量、创建对象等。

外部类依然不能直接访问静态内部类的成员，但可以使用静态内部类的类名作为调用者来访问静态内部类的**类成员**，也可以使用静态内部类对象作为调用者来访问静态内部类的**实例成员，**

**示例：**

**public** **class** AccessStaticInnerClass {

**static** **class** StaticInnerClass {

**private** **static** **int** *prop1* = 5;

**private** **int** prop2 = 9;

}

**public** **void** accessInnerProp() {

System.***out***.println(StaticInnerClass.*prop1*);

//直接用 "类名." 的方式调用静态内部类的类成员

//System.out.println(prop1);

//上面这行代码会引发错误，外部类不可直接访问内部类成员

System.***out***.println(**new** StaticInnerClass().prop2);

//通过静态内部类对象来访问静态内部类的实例成员

}

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

AccessStaticInnerClass asic = **new** AccessStaticInnerClass();

asic.accessInnerProp();

}

}



**----------------------------------------------我是一条严肃的分割线--------------------------------------------------**

除此之外，Java还允；许在接口中定义内部类，接口里定义的内部类默认使用public static修饰，就是说**接口内部类只能是静态内部类**。

如果为接口内部类指定访问控制符，则只能指定public访问控制符；如果定义接口内部类时省略访问控制符，则该内部类默认是public访问控制权限。

接口中可定义以内部接口,不过意义不大，因为**接口的作用是定义一个公共规范（暴露出来供大家使用）**，如果把这个接口定义成一个内部接口，就没有什么实际意义了。

**----------------------------------------------我是一条严肃的分割线--------------------------------------------------**

## 使用内部类

定义类的主要作用就是定义变量、创建实例和作为父类被继承。定义内部类的主要作用也是如此，但**使用内部类定义变量和创建实例则与外部类存在一些小小的差异**。下面分三种情况讨论内部类的用法。

### 在外部类内部使用内部类

在外部类中使用内部类时，与平常使用普通类没有太大区别，**唯一的区别就是：不能在外部类的静态成员中（包括静态方法和静态代码块）中使用非静态内部类（根据静态成员不能访问非静态成员的规则）**

### 在外部以外使用非静态内部类

如果想要在外部类以外的地方访问内部类（包括静态和非静态两种），则内部类不能

使用private访问控制权限，private修饰的内部类只能在外部类内部使用。对于使用其他的访问控制修饰的内部类，则能在访问控制符对应的访问权限内使用。

* 省略控制符的内部类，只能被与外部类处于**同一包中的其它类所访问**
* 使用protected修饰的内部类，可被与外部类处于**同一个包中的其它类和外部类的子类所访问**
* 使用public修饰的内部类，**可以在任何地方被访问**

**在外部类以外的地方定义内部类变量的语法格式如下：**

OutterClass.InnerClass varName

在外部类以外的地方使用内部类时，内部类完整的类名应该是OuterClass.InnerClas，如果是在外部类所在包的外头，还需要**缀上外部类的包名**。

在创建内部类对象之前，必须先创建外部类对象，因为内部类对象是寄生在外部类对象中的，在外部类以外的地方创建非静态内部类实例的语法格式如下：

OuterInstance.new InnerConstructor()

**示例：**外部类以外创建非静态内部类对象

**class** Out {

**class** In {

**public** In(String msg) {

System.***out***.println(msg);

}

}

}

**public** **class** CreatInnerInstance {

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

Out.In in = **new** Out().**new** In("测试信息");

/\*\*

\* 上面这行代码等同于以下的代码：

\*

\* 使用OuterClass.InnerClass的形式定义内部类变量

\* Out.In in;

\*

\* 创建外部类实例，非静态内部类实例将寄生在该示例中

\* Out out = new Out();

\*

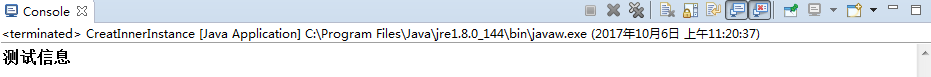
\* 通过外部类实例和new来调用内部类构造器创建非静态内部类实例

\* in = out.new In("测试信息");

\*/

}

}



从上面的代码中可以看出，**非静态内部类的构造器必须使用外部类对象来调用**

**----------------------------------------------我是一条严肃的分割线--------------------------------------------------**

下面示范如何让**子类继承父类中的内部类**：

**class** Out {

**class** In {

**public** In(String msg) {

System.***out***.println(msg);

}

}

}

**class** SubClass **extends** Out.In {

**public** SubClass(Out out) {

**out.super("hello");**

}

}

**public** **class** CreatInnerInstance {

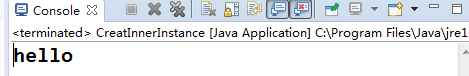
**public** **static** **void** main(String[] args) {

Out out = **new** Out();

SubClass sClass = **new** SubClass(out);

}

}



**重点解释一下语句out.super(“hello”)：**

非静态内部类In类的构造器必须使用外部类对象来调用，代码中super代表调用In类的构造器，out代表外部类对象。

从上面的代码可以看出，如果需要创建SubClass对象，就必须先创建一个Out对象。**因为SubClass是非静态内部类In类的子类，非静态内部类In对象里必须有一个对Out对象的引用，其子类SubClass对象里也应该持有对Out对象的引用**。当创建SubClass对象时传给该构造器的Out对象，就是SubClass对象里Out对象引用所指向的对象。

非静态内部类In对象和SubClass对象都必须持有指向Out对象的引用，但**两者还是有区别的**：

创建In类对象时，必须通过Out对象来调用关键字new，创建SubClass类的对象时，必须使用Out对象作为调用者来调用In类的构造器。

### 在外部类以外使用静态内部类

因为静态内部类是外部类**类相关**的，因此**创建静态内部类对象时无须创建外部类对象**。在外部类以外的地方创建静态内部类实例的语法如下：

New OuterClass.InnerConstructor()

**示例：**

**class** StaticOut {

**static** **class** StaticIn {

**public** StaticIn() {

System.***out***.println("静态内部类的构造器");

}

}

}

**public** **class** CreatStaticInnerInstance {

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

**StaticOut.StaticIn in = new StaticOut.StaticIn();**

/\*\*

\* 上面的代码可改为如下两行代码：

\*

\* 使用OuterClass.InnerClass的形式定义内部类变量

\* StaticOut.StaticIn in;

\*

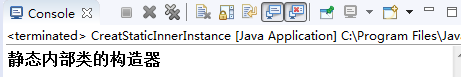
\* 通过new来调用内部类构造器创建静态内部类实例

\* in = new StaticOut.StaticIn();

\*/

}

}



从以上代码可以看出，不管是静态内部类还是非静态内部类，他们声明变量的语法完全一样。**区别只是**在创建内部类对象时，**静态内部类只需使用外部类即可调用构造器，而非静态内部类必须使用外部类对象来调用构造器**。

由于调用静态内部类的构造器时无须使用外部类对象，所以创建静态内部类的子类也比较简单，下面代码就为静态内部类StaticIn类定义了一个空的子类：

**class** StaticOut {

**static** **class** StaticIn {

**public** StaticIn() {

System.***out***.println("静态内部类的构造器");

}

}

}

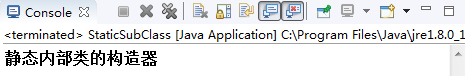
**public** **class** StaticSubClass **extends** StaticOut.StaticIn {

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

StaticSubClass ssc = **new** StaticSubClass();

}

}



从上面代码看以看出，当定义一个静态内部类时，其**外部类非常像一个包空间**。

相比之下，使用**静态内部类比使用非静态内部类要简单很多**，因此**当程序需要使用内部类时，应该优先考虑使用静态内部类**。

## 局部内部类

如果把一个**内部类放到方法里定义**，那么这个**类就被称为局部内部类，**局部内部类只在该方法中有效，因此**局部内部类不能使用访问控制符和static修饰。**

对于局部成员而言，不管是局部变量还是局部内部类，他们的**上一级程序单元都是方法而不是类**，**使用static修饰没有任何意义**；而且，由于**局部成员的作用域是所在方法**，**其他程序单元永远也不可能访问另一个方法中的局部成员**，所以所有的局部成员都**不能使用访问控制符修饰**。

**----------------------------------------------我是一条严肃的分割线--------------------------------------------------**

如果需要用局部内部类定义变量、创建实例或派生子类，那么只能在局部类所在的方法内进行。

实例：

**public** **class** LocalInnerClass {

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

**class** InnerBase {

**int** a;

}

**class** InnerSub **extends** InnerBase {

**int** b;

}

InnerSub is = **new** InnerSub();

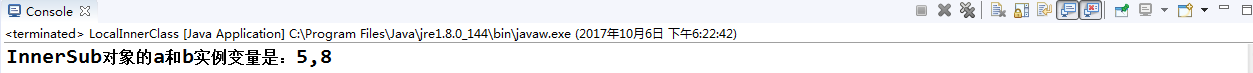
is.a = 5;

is.b = 8;

System.***out***.println("InnerSub对象的a和b实例变量是：" + is.a + "," + is.b);

}

}



编译上面的程序，会生成三个class文件：LocalInnerClass.class、LocalInnerClass$**1**InnerBase.class和LocalInnerClass$**1**InnerSub.class，注意到局部内部类的class文件的文件名比成员内部类的class文件的文件名多了一个数字，这是因为同一个类里不可能有两个同名的成员内部类，但是**同一个类里却可能存在两个及以上的处于不同方法中的同名局部内部类**，所以Java为局部内部类的class文件名中增加了一个数字，用于区分。

其实局部内部类十一个非常“**鸡肋**”的语法，在**实际开发中很少用到**，创建一个类，当然是希望多次复用这个类，但**局部内部类的作用于太小了**，离开了方法之后就失去作用了。

## Java8改进的匿名内部类

**匿名内部类适合创建那种只需要使用一次的类**，例如前面接口中讲到的命令模式中的**[Cmd](#_命令模式)**对象，**创建匿名内部类时会立即创建一个该类的实例，这个类定义立即消失**，因此匿名内部类是不能重复使用的。

定义匿名内部类的格式如下：

**new** 实现接口() | 父类构造器(实参列表) {

//匿名内部类的实体部分

}

匿名内部类必须继承一个父类，或者实现一个接口，但最多只能继承一个父类，实现一个接口 。

匿名内部类还要遵循以下两条规则：  
 1、**匿名内部类不能是抽象类**，因为抽象类是无法创建实例的

2、**匿名内部类不能定义构造器**，因为匿名内部类是没有类名的，因此也无法定义构造器，**但它可以定义初始化块**，通过实例初始化块来完成构造器需要完成的事情

最常用的创建匿名内部类的方式是需要创建某个接口类型的对象

**示例：**

**interface** Product {

**public** **double** getPrice();

**public** String getName();

}

**public** **class** AnonymousTest {

**public** **void** test(Product product ) {

System.***out***.println("购买了一个" + product.getName() + "，花掉了" + product.getPrice());

}

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

*AnonymousTest* ta = **new** AnonymousTest();

ta.test(**new** Product() **{**

**public double getPrice() {**

**return 567.8;**

**}**

**public String getName() {**

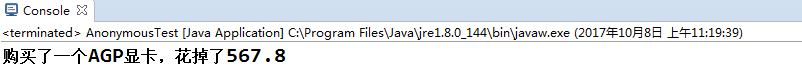
**return "AGP显卡";**

**}**

**}**);

}

}



上面代码的黄色部分，即为匿名内部类的类体部分。

AnonymousTest类中定义了一个test()方法，这个方法需要传递一个Product对象作为参数，但Product只是一个接口，无法直接创建对象，因此这里需要创建一个Product的实现类，如果该实现类需要重复使用，就把它单独定义成一个类，如果这个实现类只需要使用一次，那么就可以把它定义成匿名内部类，也就是上面的程序所采用的方式。

定义匿名内部类并不需要关键字class，而是在定义匿名内部类是直接生成该匿名内部类的对象。

由于匿名内部类不能是抽象类，所以**匿名内部类必须实现他的抽象父类或者接口里的所有抽象方法。**

**----------------------------------------------我是一条严肃的分割线--------------------------------------------------**

当通过**实现接口来创建匿名内部类**时，**匿名内部类不能显式创建构造器**，因此匿名内部类只有一个隐式的无参数构造器，故new接口名后的括号里不能传入参数值。

但如果**通过继承父类来创建匿名内部类**时，**匿名内部类将拥有和父类相似的构造器，此处的相似指的是拥有相同的形参列表**。

**示例：**

abstract class Device {

private String name;

public abstract double getPrice();

public Device(){}

public Device(String name) {

this.name = name;

}

// 此处省略了name的setter和getter方法

public void setName(String name) {

this.name = name;

}

public String getName() {

return this.name;

}

}

public class AnonymousInner {

public void test(Device d) {

System.out.println("购买了一个" + d.getName()

+ "，花掉了" + d.getPrice());

}

public static void main(String[] args) {

AnonymousInner ai = new AnonymousInner();

// 调用有参数的构造器创建Device匿名实现类的对象

ai.test(new Device("电子示波器") {

public double getPrice() {

return 67.8;

}

});

// 调用无参数的构造器创建Device匿名实现类的对象

Device d = new Device() {

// 初始化块

{

System.out.println("匿名内部类的初始化块...");

}

// 实现抽象方法

public double getPrice() {

return 56.2;

}

// 重写父类的实例方法

public String getName() {

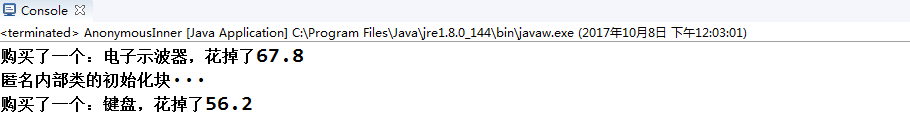
return "键盘";

}

};

ai.test(d);

}

}

上面程序创建了一个抽象父类Device类，这个抽象父类里包含两个构造器：一个无参数的，一个有参数的。

当创建匿名内部类时，必须实现接口或者抽象类中所有的抽象方法，**如果有需要，也可以重写父类中的普通方法，**上面的程序中的第二段黄色代码就重写了父类Device中的getName()方法，getName()是普通方法。

**----------------------------------------------我是一条严肃的分割线--------------------------------------------------**

在Java 8之前，Java要求局部内部类、匿名内部类访问的局部变量必须使用final修饰，从Java 8开始这个限制被取消了，Java 8更加智能：如果局部变量被匿名内部类访问，那么该局部变量相当于自动使用了final修饰。

**示例：**  
**interface** A {

**void** test();

}

**public** **class** ATest {

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

**int** age = 8;

A a = **new** A() {

**public** **void** test() {

**//在Java 8以前下面语句将提示错误：age必须使用final修饰**

**//从Java 8开始，匿名内部类、局部内部类允许访问非final局部变量**

System.***out***.println(age);

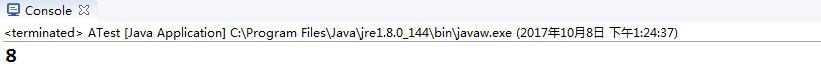
}

};

a.test();

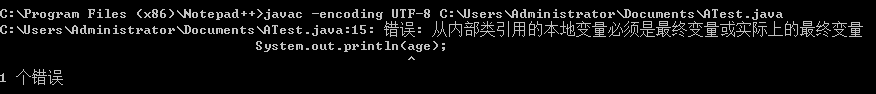
}

}



但是如果在上面的程序中int age = 8后面增加如下代码：  
 age = 2

将会引起编译错误，如下所示：



这是因为再次对age赋值，导致Java 8无法自动使用final修饰age局部变量。

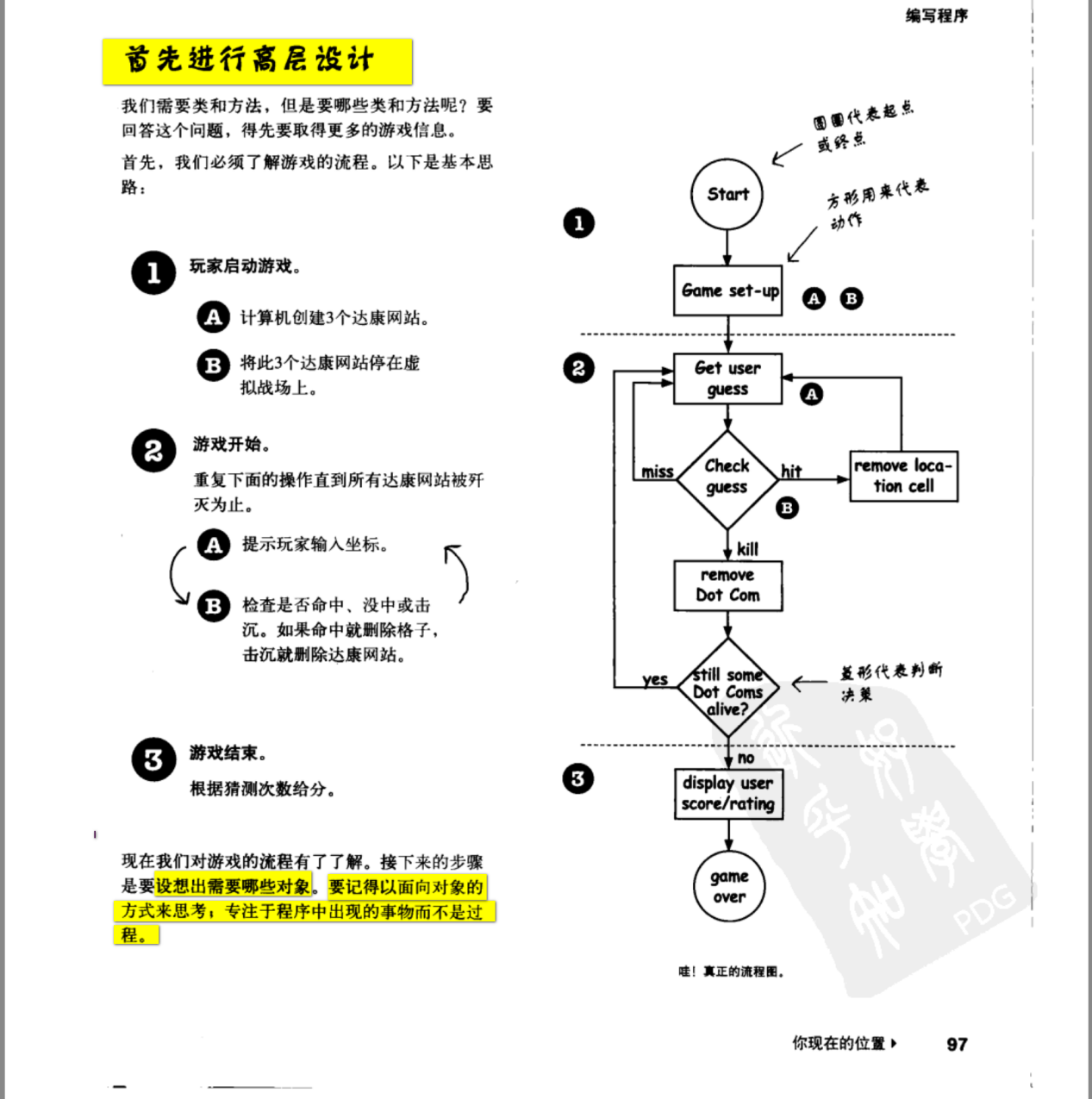
Java 8将这个功能称为“effectively final”，他的意思是对于**被匿名类访问的局部变量**，可以用final修饰，也可以不用final修饰，但**必须按照有final修饰的方法来使用，也就是一次赋值后不能再重新赋值。**

# 案例

## **“攻击网站”游戏设计案例（Simple Dot Com Game）**

在真正开始写游戏之前，需要先有一个宏观的架构以及流程图，然后再根据这些东西去设计相应的类和对象

先创建一个简单版的，之后会进行优化



**package** learn;

**import** java.util.\*;

**class** DotCom {

**private** **int**[] locationCells;

**private** **int** numOfHits;

**void** setLocationCells(**int**[] loc) {

locationCells = loc;

**int** bg = (**int**)(Math.*random*() \* 10) + 1;

**for**(**int** i = bg - 1; i < bg + 2; i++)

locationCells[i] = 1;

}

String checkYourself(**int** guess) {

**if**(numOfHits == 3)

**return** "kill";

**if**(locationCells[guess] == 1) {

locationCells[guess] = 0;

numOfHits ++;

**return** "hit";

}

**return** "miss";

}

}

**public** **class** SimpleDotComGame {

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

Scanner input = **new** Scanner(System.***in***);

**int**[] loc = **new** **int**[100];

DotCom dc = **new** DotCom();

dc.setLocationCells(loc);

**int** times = 0;

**while**(**true**) {

System.***out***.print("Enter a number ");

**int** guess = input.nextInt();

times++;

System.***out***.println("" + dc.checkYourself(guess));

**if**(dc.checkYourself(guess) == "kill") {

System.***out***.println("You took " + times + " guesses");

**break**;

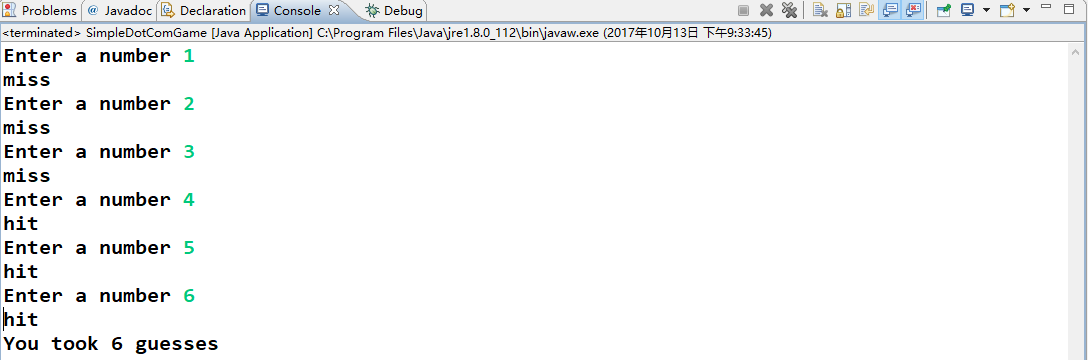
}

}

input.close();

}

}



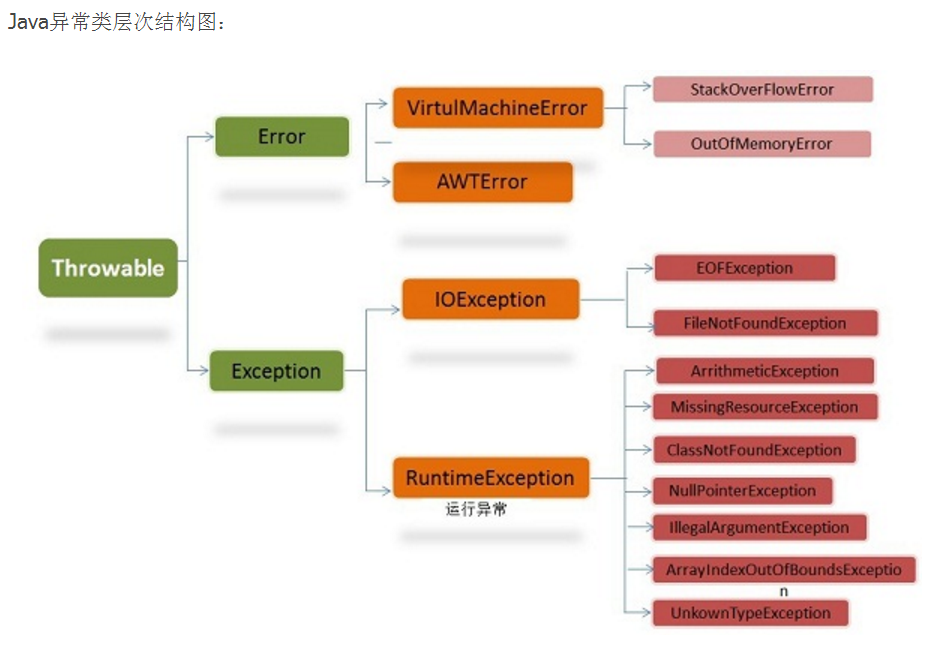
## 个人设计的游戏（Java小程序，无图形界面）



# 异常处理

## @@@@什么是异常

异常指不期而至的各种状况，如：文件找不到、网络连接失败、非法参数等。异常是一个事件，它发生在程序运行期间，干扰了正常的指令流程。Java通 过API中Throwable类的众多子类描述各种不同的异常。因而，Java异常都是对象，是Throwable子类的实例，描述了出现在一段编码中的错误条件。当条件生成时，错误将引发异常。



在Java中，所有的异常都有一个共同的祖先Throwable(可抛出)

#### ****1.Throwable：****

有两个重要的子类: Exception(异常)和Error(错误)，他俩都是Java异常处理的重要子类，各自都包含了大量的子类。

#### ****2.Error：****

程序无法处理的错误，这种类型的错误与程序的编写者所执行的操作无关，它所表示的是JVM出现的问题。

这些错误表示故障发生于虚拟机自身、或者发生在虚拟机试图执行应用时。

这些错误是不可查的，因为他们在应用程序的控制和处理能力之外，而且绝大多数是程序在运行是不允许出现的状况。

对于设计合理的程序来说，本质上也不应该试图去处理它所引起的异常状况。

在Java中，错误通过Error的子类描述。

#### ****3.Exception：****

这种异常是程序本身就可以处理的异常

Exception和Error的主要区别就是Exception是程序自身可以处理的异常，而Error是程序自身无法处理的异常

一般来说，Java的异常分为两种：**不可查的异常(checked Exceptions)-/-/-/可查的异常(unchecked Exceptions)**

不可查异常包括****运行时异常****（RunTimeException及其子类）和错误（Error）

****运行时异常：****都是RuntimeException类及其子类异常，如NullPointerException(空指针异常)、IndexOutOfBoundsException(下标越界异常)等，这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。

运行时异常的特点是Java编译器不会检查它，也就是说，当程序中可能出现这类异常，即使没有用try-catch语句捕获它，也没有用throws子句声明抛出它，也会编译通过。

****非运行时异常：****这种异常是一定要进行捕捉和处理的，否则编译是不会通过的

## @@@@异常处理的机制

#### ****1.抛出异常****

当一个方法引发异常时，方法会创建异常对象并交付运行时系统，异常对象中包含了异常类型和异常出现时的程序状态等异常信息，运行时系统负责寻找适合出了该异常的代码并执行。

#### ****2.捕获异常****

方法抛出异常之后，运行时系统就会开始寻找合适的****异常处理器(exception handler)****，潜在的异常处理器是异常发生时依次存留在调用栈中的方法的集合，(其实就是一系列的catch()方法)，当异常处理器**所能处理的异常类型与方法抛出的异常类型相符**时，那他就是合适的异常处理器。

运行时系统从发生异常的方法开始，依次回查调用栈中的方法，直至找到含有合适异常处理器的方法并执行。当运行时系统遍历调用栈而未找到合适 的异常处理器，则运行时系统终止。同时，意味着Java程序的终止。

  由于****运行时异常****的不可查性，为了更合理、更容易地实现应用程序，Java规定，运行时异常将由Java运行时系统自动抛出，****允许应用程序忽略运行时异常****。

       对于方法运行中可能出现的****Error****，当运行方法不欲捕捉时，Java****允许该方法不做任何抛出声明****。因为，大多数Error异常属于永远不能被允许发生的状况，也属于合理的应用程序不该捕捉的异常。

       对于所有的****可查异常****，Java规定：一个****方法必须捕捉，或者声明抛出方法之外****。也就是说，当一个方法选择不捕捉可查异常时，它必须声明将抛出异常。

从方法中抛出的任何异常都必须使用throws子句。

捕捉异常通过try-catch语句或者try-catch-finally语句实现。

     总体来说，Java规定：对于可查异常必须捕捉、或者声明抛出。允许忽略不可查的RuntimeException和Error。

## @@@@捕获异常

#### ****1. try-catch、finally语句:****

****Try-catch语句的一般语法形式：** try** {

// 可能会发生异常的程序代码

} **catch** (Type1 id1){

// 捕获并处置try抛出的异常类型Type1

}

**catch** (Type2 id2){

//捕获并处置try抛出的异常类型Type2

}

关键字try后面的大括号成为监控区，catch语句用来处理异常

**示例：**

**public** **class** TestException {

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

**int** a = 6;

**int** b = 0;

**try** { // try监控区域

**if** (b == 0) **throw** **new** ArithmeticException(); // 通过throw语句抛出异常

System.***out***.println("a/b的值是：" + a / b);

}

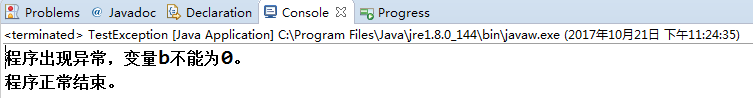
**catch** (ArithmeticException e) { // catch捕捉异常

System.***out***.println("程序出现异常，变量b不能为0。");

}

System.***out***.println("程序正常结束。");

}

}

事实上，“除数为0”等ArithmeticException，是RuntimException的子类。而运行时异常将由运行时系统自动抛出，不需要使用throw语句。

**public** **class** TestException {

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

**int** a = 6;

**int** b = 0;

**try** {

System.***out***.println("a/b的值是：" + a / b);

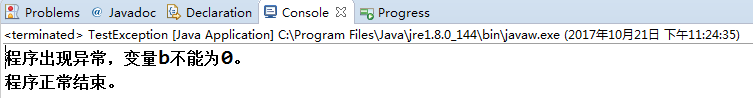
} **catch** (ArithmeticException e) {

System.***out***.println("程序出现异常，变量b不能为0。");

}

System.***out***.println("程序正常结束。");

}

}

这一个示例程序并没有使用throw语句来抛出异常，但还是正常捕获到了异常，因为RunTimeException的子类的实例（即异常）会由运行时系统会自动抛出。

上面说过，**运行时异常是允许被忽略的**，就算不声明抛出异常的行为，不进行捕获，最后也会**编译器自己也会报出异常。**

一旦某个catch捕获到匹配的异常类型，将进入异常处理代码。一经处理结束，就意味着整个try-catch语句结束。**其他的catch子句不再有匹配和捕获异常类型的机会。**

这句话的意识也就是说，不管写了多少个catch语句，只要有一个catch语句被匹配到并执行，那么其他的catch语句就再也不会被执行，因为整个try-catch语句块已经全部结束了。

#### ****2. try-catch-finally语句:****

这个语句块相比于上面的try-catch语句多了一个finally语句，finally语句中的内容总是会被执行，不管有没有发生异常。

**public** **class** TestException {

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

**int** i = 0;

String greetings[] = { " 何其乐最帅!", " 何其乐最帅!! ",

" 何其乐最帅!!!" };

**while** (i < 4) {

**try** {

// 特别注意循环控制变量i的设计，避免造成无限循环

System.***out***.println(greetings[i++]);

} **catch** (ArrayIndexOutOfBoundsException e) {

System.***out***.println("数组下标越界异常");

} **finally** {

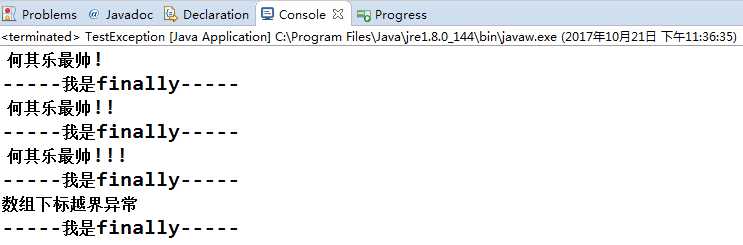
System.***out***.println("-----我是finally-----");

}

}

}

}



**如果将上面的try语句改成下面这个样子，将会造成死循环**（我都把eclipse整崩了）

**try** {

System.***out***.println(greetings[i]);

i++;

}

之所以会造成死循环，是因为，当i == 3时，就会出现数组下表越界的异常，从而造成i++语句无法被执行，直接进入下面的catch语句，结果等到try-catch-finally语句全部执行完毕之后，回到while判断循环是否能够继续时，发现i=3，仍然满足i < 4的条件，然后就造成了死循环。

****try 块：****用于捕获异常。其后可接零个或多个catch块，如果没有catch块，则必须跟一个finally块。

****catch 块：****用于处理try捕获到的异常。

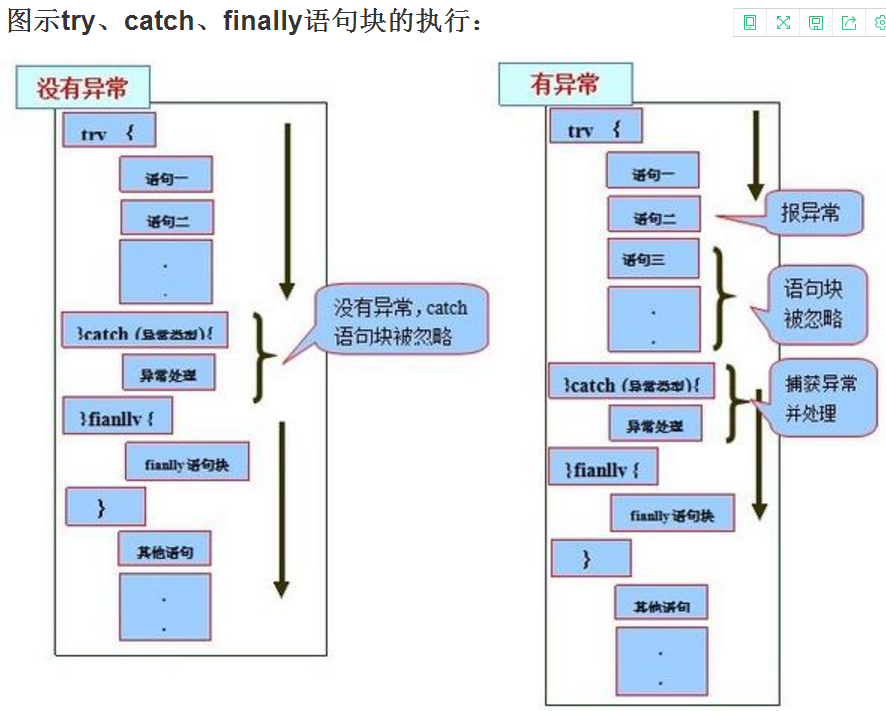
****finally 块：****无论是否捕获或处理异常，finally块里的语句都会被执行。

当在try块或catch块中遇到return语句时，finally语句块将在方法返回之前被执行。在以下4种特殊情况下，finally块不会被执行：  
 1）在finally语句块中发生了异常  
 2）在前面的代码中用了System.exit()退出程序  
 3）程序所在的线程死亡  
 4）关闭CPU。

#### ****3. try-catch-finally 规则(****[异常处理语句的语法规则](http://book.51cto.com/art/201009/227791.htm" \o "16.2.5 异常处理语句的语法规则" \t "http://blog.csdn.net/hguisu/article/details/6155636/_blank)****)：****

1)  必须在 try 之后添加 catch 或 finally 块。try 块后可同时接 catch 和 finally 块，但至少有一个块。  
 2) 必须遵循块顺序：若代码同时使用 catch 和 finally 块，则必须将 catch 块放在 try 块之后。  
 3) catch 块与相应的异常类的类型相关。  
 4) 一个 try 块可能有多个 catch 块。若如此，则执行第一个匹配块。即Java虚拟机会把实际抛出的异常对象依次和各个catch代码块声明的异常类型匹配，如果异常对象为某个异常类型或其子类的实例，就执行这个catch代码块，不会再执行其他的 catch代码块  
 5) 可嵌套 try-catch-finally 结构。  
 6) 在 try-catch-finally 结构中，可重新抛出异常。  
 7) 除了下列情况，总将执行 finally 做为结束：JVM 过早终止（调用 System.exit(int)）；在 finally 块中抛出一个未处理的异常；计算机断电、失火、或遭遇病毒攻击。

#### ****4. try、catch、finally语句块的执行顺序:****



## @@@@抛出异常

任何Java代码都是可以抛出异常的：

自己编写的代码、Java开发环境包中的代码、Java运行时系统

特别要注意throw和throws是不同的，不要混淆

**Throw**是**Java语句**用来抛出异常用的

**throws**是**Java方法**用来抛出异常用的

从方法中抛出的异常必须要用关键字throws

#### ****throws抛出异常****

将程序比作一辆汽车，当汽车出现了自己不能解决的额问题时，就需要开车的人来进行解决，这时候就相当于将问题**抛**了出来，Java程序就用throws来抛出自己在运行过程中的异常。

**Throws**是用来**声明**方法将要抛出的**异常类型**，如果抛出的异常类型是exception， 那么就是说该方法被声明为**抛出所有类型的异常**，因为，exception是所有异常的父类。

**另外，如果想要抛出多个异常，可以使用逗号对异常类型进行分割，如下所示：** methodname **throws** Exception1,Exception2,..,ExceptionN {

}

Exception1,Exception2,..,ExceptionN这些是声明要抛出的异常列表。

当方法抛出异常列表中的异常时，该方法并不会处理这个异常，而是抛给调用该方法的方法来进行catch处理：  
 **public** **class** TestException {

**static** **void** pop() **throws** NegativeArraySizeException {

// 定义方法并抛出NegativeArraySizeException异常

**int**[] arr = **new** **int**[-3]; // 创建数组

}

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

**try** { // try语句处理异常信息

*pop*(); // 调用pop()方法

} **catch** (NegativeArraySizeException e) {

System.***out***.println("pop()方法抛出的异常");// 输出异常信息

}

}

}

在上面这段程序中，由pop方法抛出异常，main方法捕获异常。

当时用throws将异常抛给调用者之后，如果调用者不想进行处理，可以继续向上抛出，但最终一定要有能处理该异常的调用者。

#### ****2.throws抛出异的规则****

  1) 如果是不可查异常（unchecked exception），即Error、RuntimeException 或它们的子类，那么可以不使用throws关键字来声明要抛出的异常，编译仍能顺利通 过，但在运行时会被系统抛出。

    2）必须声明方法可抛出的任何可查异常（checked exception）。即如果一个方法 可能出现受可查异常，要么用try-catch语句捕获，要么用throws子句声明将它抛出， 否则会导致编译错误

    3)仅当抛出了异常，该方法的调用者才必须处理或者重新抛出该异常。当方法的调 用者无力处理该异常的时候，应该继续抛出，而不是打肿脸充胖子。

4）调用方法必须遵循任何可查异常的处理和声明规则。若覆盖一个方法，则不能声明与覆盖方法不同的异常。声明的任何异常必须是被覆盖方法所声明异常的同类或父类。1

**void** method1() **throws** IOException{} //合法

//编译错误，必须捕获或声明抛出IOException

**void** method2(){

method1();

}

//合法，声明抛出IOException

**void** method3()**throws** IOException {

method1();

}

//合法，声明抛出Exception，IOException是Exception的子类

**void** method4()**throws** Exception {

method1();

}

//合法，捕获IOException

**void** method5(){

**try**{

method1();

} **catch**(IOException e) {…}

}

//编译错误，必须捕获或声明抛出Exception

**void** method6(){

**try**{

method1();

} **catch**(IOException e) {**throw** **new** Exception();}

}

//合法，声明抛出Exception

**void** method7()**throws** Exception{

**try**{

method1();

} **catch**(IOException e){**throw** **new** Exception();}

}

要么捕获被调用方法抛出的异常，要么再向上抛出被调用方法抛出异常的同类异常或者父类异常。

****判断一个方法可能会出现异常的依据如下：****  
      1）方法中有throw语句

    2）调用了其他方法，其他方法用throws子句声明抛出某种异常。

#### ****3.throw抛出异常（****是throw，没有s****）****

throw总是出现在函数体中，用来抛出一个Throwable类型的异常，throw语句后的代码终止，然后在包含它的所有的try块中寻找合适的catch，为什么说是所有的呢，因为这个try块可能存在于该函数中，也可能存在于该函数的调用者中，它是从里向外寻找合适的catch语句的。

如果在函数体中throw出了检查异常（[checked exceptions](#检查异常" \o "什么是checked exceptions)），那么在该函数首部也要用throws声明抛出异常的类型。

**import** java.lang.Exception;

**public** **class** TestException {

**static** **int** quotient(**int** x, **int** y) **throws** MyException { // 定义方法抛出异常

**if** (y < 0) { // 判断参数是否小于0

**throw** **new** MyException("除数不能是负数"); // 异常信息

}

**return** x/y; // 返回值

}

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

**int** a =3;

**int** b =-1;

**try** { // try语句包含可能发生异常的语句

**int** result = *quotient*(a, b); // 调用方法quotient()

} **catch** (MyException e) { // 处理自定义异常

System.***out***.println(e.getMessage()); // 输出异常信息

} **catch** (ArithmeticException e) { // 处理ArithmeticException异常，这个异常属于RunTimeException，运行时系统会自动抛出，因此上面的try监控区中并未对该异常做抛出处理

System.***out***.println("除数不能为0"); // 输出提示信息

} **catch** (Exception e) { // 处理其他异常

System.***out***.println("程序发生了其他的异常"); // 输出提示信息

}

}

}

**class** MyException **extends** Exception { // 创建自定义异常类

String message; // 定义String类型变量

**public** MyException(String ErrorMessagr) { // 父类方法

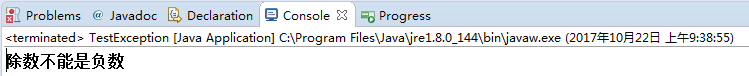
message = ErrorMessagr;

}

**public** String getMessage() { // 覆盖getMessage()方法

**return** message;

}

} 

#### ****4.自定义异常****

上面的代码中自定义了一个异常类，继承自Exception类。

使用Java内置的异常类可以描述在编程时出现的大部分异常情况。除此之外，用户还可以自定义异常。用户自定义异常类，只需继承Exception类即可。  
  在程序中使用自定义异常类，大体可分为以下几个步骤。  
 （1）创建自定义异常类。  
 （2）在方法中通过throw关键字抛出异常对象。  
 （3）如果在当前抛出异常的方法中处理异常，可以使用try-catch语句捕获并处理；否则在方法的声明处通过throws关键字指明要抛出给方法调用者的异常，继续进行下一步操作。  
 （4）在出现异常方法的调用者中捕获并处理异常。

# 枚举类型Enum

## 枚举类入门

Java5新增了enum这个关键字，他的地位和class、interface相同，用以定义枚举类。

**一个Java源文件中最多只能定义一个public访问权限的枚举类，且该Java源文件也必须和该枚举类的类名相同。**

但枚举类并不是普通类，它和普通类之间有以下几个差别：

* 枚举类可以实现一个或多个接口，使用enum定义的枚举类默认继承了java.lang.Enum类，而不是默认继承Object类，因此**枚举类不能显式继承其他父类**。（因为Java是单继承，已经默认继承了java.lang.Enum就不能再继承其他类了）

其中java.lang.Enum类实现了java.lang.Serializable和java.lang.Comparable两个接 口。

* 使用enum定义非抽象的枚举类默认会使用final修饰，因此枚举类不能派生子类。
* 枚举类的**构造器只能使用private访问控制符**，如果省略了构造器的访问控制符，则默认使用private修饰；如果强制指定访问控制符，则只能指定private修饰符。
* **枚举类的所有实例必须在枚举类的第一行显式列出**，否则这个枚举类永远都不能产生实例。列出这些实例时，系统会自动添加public static final修饰，无需程序员显式添加。

枚举类默认提供了一个**values()方法，该方法可以很方便地遍历所有的枚举值**。

Eg: Season枚举类

**public** **enum** SeasonEnum {

***SPRING***, ***SUMMER***, ***FALL***, ***WINTER***;

}

**定义枚举类时，需要显式列出所有的枚举值，所有的枚举值之间以”,”隔开，枚举列举结束后以”;”作为结束。这些枚举值代表了该枚举类所有可能的实例。**

如果需要使用该枚举类的某个实例，则可以使用**EnumClass.variable**的形式，eg: SeasonEnum.SPRING。

**package** justTest;

**public** **class** EnumTest {

**public** **void** judge(SeasonEnum s) {

//switch语句里的表达式可以是枚举值

**switch**(s) {

**case** ***SPRING***:

System.***out***.println("春暖花开，正好踏青");

**break**;

**case** ***SUMMER***:

System.***out***.println("夏日炎炎，适合游泳");

**break**;

**case** ***FALL***:

System.***out***.println("秋高气爽，进补及时");

**break**;

**case** ***WINTER***:

System.***out***.println("冬日雪飘，围炉赏雪");

**break**;

}

}

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

//枚举类的默认values()方法，可以返回该枚举类的所有实例

**for**(SeasonEnum s: SeasonEnum.*values*()) {

System.***out***.println(s);

}

//使用枚举实例时，可通过EnumClass.variable形式来访问

**new** EnumTest().judge(SeasonEnum.***SPRING***);

}

}

**enum** SeasonEnum {

***SPRING***, ***SUMMER***, ***FALL***, ***WINTER***;

}

JDK1.5 switch语句进行了扩展：switch的控制表达式可以是任何枚举类型。而且当switch控制表达式使用枚举类型时，后面case表达式中的值直接使用枚举值的名字，无需添加枚举类作为限定。

因为枚举类默认继承了java.lang.Enum类，因此所有的枚举类都可以使用java.lang.Enum类所包含的方法。以下为几个常用的方法：

* int CompareTo(E o):该方法用于**与指定枚举对象比较顺序**，只有相同类型的枚举实例才能使用这个方法。如果当前枚举对象在指定枚举对象**之后，返回正**整数；**之前则返回负**整数，**否则返回零。**
* String name():返回此枚举类实例的名称。
* int ordinal():返回枚举值在枚举类中的索引值（就是枚举值在枚举声明中的位置，**第一个枚举值的索引值为零**）
* String toString():返回枚举常量的名称，与name()方法相似，但toString()方法更常用。
* public static <T extends Enum<T>> T valueOf(Class<T>enum Type, String name):这是一个静态方法，用于返回**指定枚举类**中**指定名称**的枚举值。名称必须与在该枚举类声明枚举值时所用的标识符完全匹配，**不允许使用额外的空白字符。**

## 枚举类的成员变量、方法和构造器

枚举类也是一种类，只不过比较特殊，因此他一样可以定义成员变量、方法和构造器。

**package** justTest;

**public** **enum** Gender {

***MALE***, ***FEMALE***;

//定义一个public修饰的实例变量

**public** String name;

}

上面的Gender枚举定义了一个名为name的实例变量，并且将它定义成一个public访问权限的。

**package** justTest;

**public** **class** GenderTest {

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

//枚举类型通过valueof()方法来创建对象

Gender g = Enum.*valueOf*(Gender.**class**, "FEMALE");

//直接为枚举值的name实例变量赋值

g.name = "女";

//直接访问枚举值的name实例变量

System.***out***.println(g + "代表：" + g.name);

}

}

**enum** Gender {

***MALE***, ***FEMALE***;

//定义一个public修饰的实例变量

**public** String name;

}

从上面这个程序来看，在使用Gender枚举类时和使用一个普通类并没有多大的区别，差别只是Gender对象的产生方式不同，**枚举类的实例只能是枚举值，而不是随意地通过new来创建枚举对象。**

**Java应该把所有类设计成良好封装的类，**上面的Gender的name并不能直接设置成public访问权限，下面是改进后的枚举类Gender的代码：

**package** justTest;

**public** **enum** Gender {

***MALE***, ***FEMALE***;

//定义一个public修饰的实例变量

**private** String name;

**public** **void** setName(String name) {

**switch**(**this**) {

**case** ***MALE***:

**if**(name().equals("男")) **this**.name = name;

**else** {

System.***out***.println("参数错误");

**return**;

}

**break**;

**case** ***FEMALE***:

**if**(name.equals("女")) **this**.name = name;

**else** {

System.***out***.println("参数错误");

**return**;

}

**break**;

}

}

**public** String getName() {

**return** **this**.name;

}

}

上面switch语句有效地限制了setName()传入的参数不会被错误赋值，保证了FEMAEL枚举值的name变量只能是”女”，MALE枚举值的name变量只能是”男”。

但是这样写还不是最好，枚举类通常应该设计成不可变类，也就是说它的成员变量应该是不可变的，前面的修饰符应该是private final，既然值不可以改变，那就需要在构造器里给他们指定初始值了。

一旦为枚举值显式定义了带参数的构造器，列出枚举值时就必须对应地传入参数。

Eg：

**package** justTest;

**public** **enum** Gender {

***MALE*("男"), *FEMALE*("女");**

**private** **final** String name;

**private** Gender(String name) {

**this**.name = name;

}

**public** String getName() {

**return** **this**.name;

}

}

在枚举类中列出枚举值时，实际上就是调用构造器创建枚举类对象，只是这里无需使用new关键字，也无需显式调用构造器。

上面代码中的蓝色代码段实质上相当于：  
 public static final Gender MALE = new Gender(“男”);  
 public static final Gender FEMALE = new Gender(“女”);

## 实现接口的枚举类

枚举类也可以实现一个或者多个接口。与普通类实现一个或多个接口**完全**一样，枚举类实现一个或多个接口时，也需要实现该接口所包含的方法。

**public** **interface** GenderDesc {

**void** info();

}

在上面的GenderDesc上定义了一个info()方法，下面给出的Gender代码会实现这个接口并实现该接口中所包含的info()方法：  
 **public** **enum** Gender **implements** GenderDesc {

***MALE***,***FEMALE***;

**public** String name;

**public** **void** info() {

System.***out***.println("这是一个用于定义性别的枚举类");

}

}

如果需要每一个枚举值在调用info()方法时呈现出不同的行为方式，则可以**让每个枚举值分别来实现info()方法，每个枚举值提供不同的实现方式**。如下所示：

**package** justTest;

**public** **class** EnumTest {

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

Gender.***FEMALE***.info();

}

}

**enum** Gender **implements** GenderDesc {

***MALE*("男") {**

**public void info() {**

**System.*out*.println("这个枚举值代表男性");**

**}**

**},**

***FEMALE*("女") {**

**public void info() {**

**System.*out*.println("这个枚举值代表女性");**

**}**

**};**

**private** String name;

**private** Gender(String name) {

**this**.name = name;

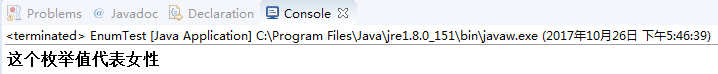
}

}

**interface** GenderDesc {

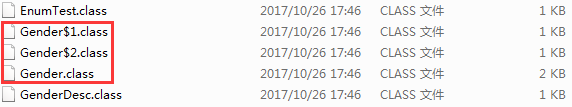
**void** info();

}



上面这段代码的**蓝色加粗花括号部分实际上就是匿名内部类**，在这种情况下，当创建MALE、FEMALE枚举值时，并不是直接创建Gender枚举类的实例，而是相当于创建Gender的匿名子类的实例。

编译上面的程序，将会**生成三个class文件：Gender.class、Gender$1.class、Gender$2.class**，这也正好证明了**MALE和FEMALE就是Gender匿名子类的实例**，而不是Gender类的实例。



前面说过，非抽象枚举类是不能派生子类的，默认被final修饰，但是上面的Gender是抽象类，**对于枚举类而言，只要包含抽象方法就是抽象类**，因为它实现了GenderDesc接口，所以它包含了接口中的抽象方法。

## 包含抽象方法的枚举类

假设有一个Operation枚举类，它的4个枚举值PLUS,MINUS,TIMES,DIVIDE分别代表加、减、乘、除4种运算，该枚举类需要定义一个eval()方法来完成计算。

这就应用到了上面的匿名内部类来实现抽象方法的方式，需要给每一个枚举值赋予不同的实现：

**package** justTest;

**public** **enum** Operation {

***PLUS*** {

**public** **double** eval(**double** x , **double** y) {

**return** x + y;

}

},

***MINUS*** {

**public** **double** eval(**double** x , **double** y) {

**return** x - y;

}

},

***TIMES*** {

**public** **double** eval(**double** x , **double** y) {

**return** x \* y;

}

},

***DIVIDE*** {

**public** **double** eval(**double** x , **double** y) {

**return** x / y;

}

};

// 为枚举类定义一个抽象方法

// 这个抽象方法由不同的枚举值提供不同的实现

**public** **abstract** **double** eval(**double** x, **double** y);

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

System.***out***.println(Operation.***PLUS***.eval(3, 4));

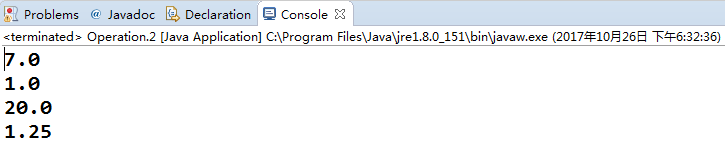
System.***out***.println(Operation.***MINUS***.eval(5, 4));

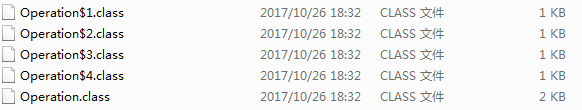
System.***out***.println(Operation.***TIMES***.eval(5, 4));

System.***out***.println(Operation.***DIVIDE***.eval(5, 4));

}

}





枚举类里定义抽象方法时不能使用abstract关键字将枚举类定义成抽象类（系统会自动为它添加abstract关键字），但因为枚举类需要**显式创建枚举值**，而不是作为父类，所以定义每个枚举值时必须为抽象方法提供实现，否则将出现编译错误。

枚举（enum），是指一个经过排序的、被打包成一个单一实体的项列表。一个枚举的实例可以使用枚举项列表中任意单一项的值。枚举在各个语言当中都有着广泛的应用，**通常用来表示诸如颜色、方式、类别、状态等等数目有限、形式离散、表达又极为明确的量**。Java从JDK5开始，引入了对枚举的支持。

## 不使用枚举类型带来的问题

在枚举类型出现之前，如果想要表示一组特定的**离散值，**使用下面这种方式**：  
public** **class** Entity {

**public** **static** **final** **int** ***VIDEO*** = 1;//视频

**public** **static** **final** **int** ***AUDIO*** = 2;//音频

**public** **static** **final** **int** ***TEXT*** = 3;//文字

**public** **static** **final** **int** ***IMAGE*** = 4;//图片

**private** **int** id;

**private** **int** type;

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** **int** getType() {

**return** type;

}

**public** **void** setType(**int** type) {

**this**.type = type;

}

}

这样做的缺点有：

（1）代码可读性差、易用性低。由于setType()方法的参数是int型的，在阅读代码的时候往往会让读者感到一头雾水，根本不明白这个2到底是什么意思，代表的是什么类型。  
而这样的话，问题又来了。这样做，客户端必须对这些常量去建立理解，才能了解如何去使用这个东西。说白了，在调用的时候，如果用户不到Entity类中去看看，还真不知道这个参数应该怎么传、怎么调。像是setType(2)这种用法也是在所难免，因为它完全合法，不是每个人都能够建立起用常量名代替数值，从而增加程序可读性、降低耦合性的意识。

（2）类型不安全。在用户去调用的时候，必须保证类型完全一致，同时取值范围也要正确。像是setType(-1)这样的调用是合法的，但它并不合理，今后会为程序带来种种问题。也许你会说，加一个有效性验证嘛，但是，这样做的话，又会引出下面的第(3)个问题。

（3）耦合性高，扩展性差。假如，因为某些原因，需要修改Entity类中常量的值，那么，所有用到这些常量的代码也就都需要修改——当然，要仔细地修改，万一漏了一个，那可不是开玩笑的。同时，这样做也不利于扩展。例如，假如针对类别做了一个有效性验证，如果类别增加了或者有所变动，则有效性验证也需要做对应的修改，不利于后期维护。

## 枚举的用法

因为在Java中一个枚举就是一个类，它也可以有属性和方法，并且实现接口。只是所有的枚举都继承自**java.lang.Enum**类，因此**enum不可以再继承其他的类**。

在下面的这段程序中，每一个枚举值都对应一个int型数值，而且不同的枚举值也会有不同的int数值，它和普通类别一样，也可以声明构造器和一些自己的各种各样的方法：

**public** **enum** TypeEnum {

***VIDEO***(1), ***AUDIO***(2), ***TEXT***(3), ***IMAGE***(4);

**int** value;

TypeEnum(**int** value) {

**this**.value = value;

}

**public** **int** getValue() {

**return** value;

}

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

TypeEnum type = TypeEnum.***TEXT***;//type的value属性值为3。

System.***out***.println(type.getValue());//屏幕输出3。

}

}

这个枚举类为字符串和int数值建立起了一个映射关系

如果要为每个枚举值指定属性，则**在枚举中必须声明一个参数为属性（属性指的是枚举序列中括号里对应字段的类型）对应类型的构造方法（不能是public）**。**否则**编译器将给出**The constructor TypeEnum(int, String) is undefined的错误**。在此例中，属性为int型，因此构造方法应当为int型。除此之外，**还可以为枚举指定多个属性**，如：

**public** **enum** TypeEnum {

***VIDEO***(1, "视频"), ***AUDIO***(2, "音频"), ***TEXT***(3, "文本"), ***IMAGE***(4, "图像");

**int** value;

String name;

TypeEnum(**int** value, String name) {

**this**.value = value;

**this**.name = name;

}

**public** **int** getValue() {

**return** value;

}

**public** String getName() {

**return** name;

}

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

TypeEnum type = TypeEnum.***VIDEO***;

System.***out***.println(type.toString());

}

}

**下面的代码可以实现根据一个int型值生成枚举值的方法**

**public** **enum** TypeEnum {

***VIDEO***(1, "视频"), ***AUDIO***(2, "音频"), ***TEXT***(3, "文本"), ***IMAGE***(4, "图像");

**int** value;

String name;

TypeEnum(**int** value, String name) {

**this**.value = value;

**this**.name = name;

}

**public** **int** getValue() {

**return** value;

}

**public** String getName() {

**return** name;

}

**public** TypeEnum getByValue(**int** value) {

**for**(TypeEnum typeEnum : TypeEnum.*values*()) {//values()方法会返回该枚举的所有值

**if**(typeEnum.value == value) {

**return** typeEnum;

}

}

//遍历所有的枚举值，找到属性值与value相等的并返回该枚举值

**throw** **new** IllegalArgumentException("No element matches " + value);

}

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

TypeEnum type = TypeEnum.***VIDEO***;

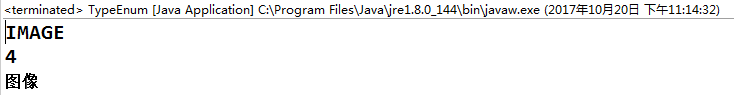
System.***out***.println(type.getByValue(4));

System.***out***.println(type.getByValue(4).getValue());

System.***out***.println(type.getByValue(4).getName());

}

}



# 图形界面

## Java图形界面设计：容器（JFrame）

### Java基本类（JFC）JavaFoundationClasses

它是由一些软件包组成，这些软件包主要包含下面这些应用程序接口（API）：

* 抽象窗口工具集（AWT）AbstractWindowsToolkit
* Swing构件
* Java2D应用程序接口（2D API）
* 兼容程序接口

**AWT是JFC的核心**

AWT为JFC的构成提供了以下的基本结构：

·代理事件模型

·轻量构件

·剪贴板和数据传输

·打印和无鼠标操作

#### AWT抽象窗口工具包

AWT的Widget依赖于作业平台的自身原有的Widget功能，但是不同的平台所提供的Widget不尽相同，如此一来，Java就会打破它自己一直以来宣传的宗旨“一次编译，到处运行”，AWT提供的都是重量级的组件，是由本地方法来实现其功能的。

AWT的特点：

·AWT中的图形函数与操作系统所提供的图形函数之间有着一一对应的关系

·AWT通过牺牲功能来实现其平台无关性

·AWT不能满足图形化用户界面发展的需要

#### Swing包

为了解决AWT组件的缺陷，尤其是跨平台的问题，Java推出了Swing包，Swing是在AWT的基础上构建的一套新的图形界面组件，是由100%的Java写成的。

Swing并不是使用本地方法来实现图形功能的，因此Swing组件被称为轻量级组件。

Swing是在AWT的基础上构建出来的，并不是取代AWT，AWT是根基，Swing是发展。

## AWT和Swing的区别

AWT提供了一套与本地图形界面进行交互的接口，AWT中的图形函数和操作系统所提供的图形函数之间有着一一对应的关系，我们把它称之为peers。**这也就是说，当我们利用AWT来构建图形用户界面的时候，实际上是在利用操作系统所提供的图形库**。这时候问题就来了，不同的操作系统所提供的功能是不一样的，为了实现Java所宣称的“一次编译到处运行”，AWT只能牺牲其一部分功能来实现平台无关性。这也就意味着，AWT所提供的图形功能是各种通用性操作系统所提供的图形功能的交集。由于AWT是依靠本地方法实现

其功能的，我们通常把AWT控件称为重量级控件。

Swing是在AWT的基础上构建的一套新的图形界面系统，它提供了AWT能够提供的所有功能，并且用纯粹的Java代码对AWT的功能进行了大幅度的扩充。由于Swing并没有使用本地方法来实现图形功能，我们通常把Swing控件称为轻量级控件。

AWT和Swing之间的基本区别：AWT 是基于本地方法的C/C++程序，其运行速度比较快；Swing是基于AWT 的Java程序，其运行速度比较慢。对于一个嵌入式应用来说，目标平台的硬件资源往往非常有限，而应用程序的运行速度又是项目中至关重要的因素。在这种矛盾的情况下，简单而高效的AWT 当然成了嵌入式Java的第一选择。而在普通的基于PC或者是工作站的标准Java应用中，硬件资源对应用程序所造成的限制往往不是项目中的关键因素，所以在标准版的Java中则提倡使用Swing，也就是通过**牺牲速度**来实现应用程序的功能。

AWT 是抽象窗口组件工具包，是 java 最早的用于编写图形节目应用程序的开发包。

Swing 是为了解决 AWT 存在的问题而新开发的包，它以 AWT 为基础的

## Swing基本框架

由上图也可以看出，Swing是继承自AWT的。

//设置窗体可见，没有该语句，窗体将不可见，此语句必须有，否则没有界面就没有如何意义了

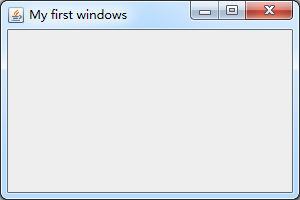
f.setVisible( **true**);

//用户单击窗口的关闭按钮时程序执行的操作

f.setDefaultCloseOperation(3);

}

}



重点说明一下，setDefaultCloseOperation这个方法的参数：

它的参数是固定的，只能是0,1,2,3。

分别对应着:

1. -> DO\_NOTHING\_ON\_CLOSE

啥也不干

1 -> HIDE\_ON\_CLOSE

隐藏窗口

1. -> DISPOSE\_ON\_CLOSE

隐藏并释放该窗体，但继续运行应用程序，释放了窗体中占用的资源

1. -> EXIT\_ON\_CLOSE

使用System exit退出应用程序

默认情况下，它的参数总是被设定为1，也就是隐藏窗口

## 布局管理器

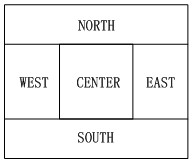
布局管理器一共有7种：

FlowLayout、BorderLayout、GridLayout、GridBagLayout、CardLayout、绝对定位、BoxLayout

### FlowLayout

就是流式布局，组件是一行一行的排列，每满一行就另起一行。

### BorderLayout

边界布局，就是把容器分为North、South、Center、West、East五部分。 

### GridLayout

Grid意为格子，顾名思义就是把容器分割成纵横线分割的网格，每个网格所占的区域大小相同。

向GridLayout布局管理器的容器添加组件时，默认是从左往右从上到下的顺序进行添加。

GridLayout共有两个构造器：  
 GridLayout(int rows, int cols)：采用指定的行数、列数，以及默认的横向间距和纵向间距将容器分割成多个网格。

GridLayout(int rows, int cols, int hgap, int vgap)：采用指定的行数、列数进行网格分割，间距也是指定的，参数是hgap和vgap（horizontal，vertical）。

示例代码：

**import** java.awt.\*;

**import** **static** java.awt.BorderLayout.\*;

**public** **class** GridLayoutTest

{

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

{

Frame f = **new** Frame("计算器");

Panel p1 = **new** Panel();

p1.add(**new** TextField(30));

f.add(p1 , ***NORTH***);

Panel p2 = **new** Panel();

// 设置Panel使用GridLayout布局管理器

p2.setLayout(**new** GridLayout(3, 5 , 4, 4));

String[] name = {"0" , "1" , "2" , "3"

, "4" , "5" , "6" , "7" , "8" , "9"

, "+" , "-" , "\*" , "/" , "."};

// 向Panel中依次添加15个按钮

**for** (**int** i = 0 ; i < name.length; i++ )

{

p2.add(**new** Button(name[i]));

}

// 默认将Panel对象添加Frame窗口的中间

f.add(p2);

// 设置窗口为最佳大小

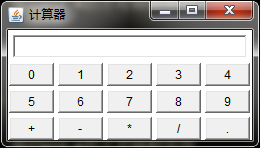
f.pack();

// 将窗口显示出来（Frame对象默认处于隐藏状态）

f.setVisible(**true**);

}

}



### GridBagLayout

GridBagLayout布局管理的功能是最强大的，但也是最复杂的，它与GridLayout的不同点在于：GridBagLayout中的一个组件可以跨越一个或多个网格，当窗口大小发生变化时，GridBagLayout布局管理器可以准确地控制窗口各部分的拉伸。

**使用GridBagLayout的步骤如下**：

1. 创建GridBagLayout布局管理器，并指定GUI容器使用该布局管理器。

GridBagLayout gb = new GridBagLayout();

continer.setLayout(gb);

1. 创建GridBadConstraints对象，并设置该对象的相关属性（用于设置受该对象控制的GUI组件的大小、跨越性等）

GridBagConstraints gbc = new GridBagConstraints();

gbc.gridx = 2; //设置受该对象控制的GUI组件位于网格的横向索引

gbc.gridy = 1; //设置受该对象控制的GUI组件位于网格的纵向索引

gbc.gridwidth = 2; //设置受该对象控制的GUI组件横向跨越多少网格 gbc.gridheight = 1; //设置受该对象控制的GUI组件纵向跨越多少网格

1. 调用GridBagLayout对象的方法来建立GridBagConstraints对象和受控制组件之间的关联。

gb.setConstraints(c, gbc); //设置c组件受gbc对象控制

1. 添加组件与采用普通布局管理器添加组件的方法完全一样

container.add(c);

重复2~4步骤即可向容器中添加多个GUI组件，gbc对象可重复使用，只要改变其属性值即可，不需要多次创建GridBagConstraints对象。

从上面可以看出，**使用GridBagLayout的关键在于GridBagConstraints，它才是精确控制每一个GUI组件的核心类**，该类具有如下几个属性：

* gridx、gridy：控制纵横索引（左上角网格的索引是0、0）
  + 这两个值还可以是GridBagConstraints.RELATIVE（默认值），它表明当前组件紧跟在上一个组件之后。
* gridwidth、gridheight：设置受该对象控制的GUI组件横向纵向跨越多少个网格，**默认值均为1，**如果这两个属性值为GridBagConstraints.REMAINDER，则表明受该对象控制的GUI组件是横向、纵向最后一个组件；如果这两个属性值为GridBagConstraints.RELATIVE，则表明受该对象控制的GUI组件是横向、纵向倒数第二个组件。
* fill：设置受该对象控制的GUI组件如何占据空白区域，该属性的取值如下：
  + GridBagConstraints.NONE：GUI组件不扩大
  + GridBagConstraints.HORIZONTAL：GUI组件水平扩大以占据空白区域
  + GridBagConstraints.VERTICAL：GUI组件垂直扩大以占据空白区域
  + GridBagConstraints.BOTH：GUI组件水平、垂直同时扩大以占据空白区域
* ipadx、ipady：设置GUI组件内部纵、横方向填充大小，即在该组件的最小尺寸的基础上还需增大多少。如果设置了这两个属性，则组件横向大小为最小宽度再加上ipadx\*2像素，纵向大小为最小高度再加上ipady\*2像素。
* insets：设置GUI组件的外部填充大小，即该组件边界和显示区域边界之间的距离。
* anchor：设置GUI组件在其显示区域中的定位方式吗，如下所示：
  + GridBagConstraints.

CENTER（中间）

NORTH（上中）

NORTHWEST（左上角）

NORTHEAST（右上角）

SOUTH（下中）

SOUTHEAST（右下角）

SOUTHWEST（左下角）

EAST（右中）

WEST（左中）

* Weightx、weighty：设置GUI组件占据多余空间的水平、垂直增加比例（也叫权重，weight的直译），这两个属性的默认值是0，即该组件不占据多余空间。假设某个容器的水平线上包括三个GUI组件，它们的水平增加比例分别是1、2、3，当容器宽度增加60像素时，则三个组建的像素增加数值分别为10、20、30。如果其增加比例为0，则表示不会增加。

注意：如果想要让某个GUI组件的大小随着容器的增大而增大，则必须同时设置控制该组件的GridBagConstraints对象的fill属性和weightx、weighty属性。

示例代码：

**package** justTest;

**import** java.awt.\*;

**public** **class** GridBagTest

{

**private** Frame f = **new** Frame("测试窗口");

**private** GridBagLayout gb = **new** GridBagLayout();

**private** GridBagConstraints gbc = **new** GridBagConstraints();

**private** Button[] bs = **new** Button[10];

**public** **void** init()

{

f.setLayout(gb);

**for** (**int** i = 0; i < bs.length ; i++ )

{

bs[i] = **new** Button("按钮" + i);

}

// 所有组件都可以横向、纵向上扩大

gbc.fill = GridBagConstraints.BOTH;

gbc.weightx = 1;

addButton(bs[0]);

addButton(bs[1]);

addButton(bs[2]);

// 该GridBagConstraints控制的GUI组件将会成为横向最后一个元素

gbc.gridwidth = GridBagConstraints.REMAINDER;

addButton(bs[3]);

// 该GridBagConstraints控制的GUI组件将横向上不会扩大

gbc.weightx = 0;

addButton(bs[4]);

// 该GridBagConstraints控制的GUI组件将横跨两个网格

gbc.gridwidth = 2;

addButton(bs[5]);

// 该GridBagConstraints控制的GUI组件将横跨一个网格

gbc.gridwidth = 1;

// 该GridBagConstraints控制的GUI组件将纵向跨两个网格

gbc.gridheight = 2;

// 该GridBagConstraints控制的GUI组件将会成为横向最后一个元素

gbc.gridwidth = GridBagConstraints.REMAINDER;

addButton(bs[6]);

// 该GridBagConstraints控制的GUI组件将横向跨越一个网格，纵向跨越2个网格。

gbc.gridwidth = 1;

gbc.gridheight = 2;

// 该GridBagConstraints控制的GUI组件纵向扩大的权重是1

gbc.weighty = 1;

addButton(bs[7]);

// 设置下面的按钮在纵向上不会扩大

gbc.weighty = 0;

// 该GridBagConstraints控制的GUI组件将会成为横向最后一个元素

gbc.gridwidth = GridBagConstraints.REMAINDER;

// 该GridBagConstraints控制的GUI组件将纵向上横跨一个网格

gbc.gridheight = 1;

addButton(bs[8]);

addButton(bs[9]);

f.pack();

f.setVisible(**true**);

}

**private** **void** addButton(Button button)

{

gb.setConstraints(button, gbc);

f.add(button);

}

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

{

**new** GridBagTest().init();

}

}

上面的代码所创建出的窗口中，虽然设置了4、5按钮在横向上是不会扩大的，但是由于受到最顶层4个窗口的影响，仍然会在横向伸长；同理，虽然设置了8、9按钮纵向上不会扩大，但因为受到按钮7的影响，所以按钮9依然会变高（按钮8不变）。

### CardLayout

CardLayout最显著的特点就是它不以空间来管理布局，而是以时间来管理它的组件，它将加入的所有组件看成一叠卡片，每次只有最上面的那个Component才可见。

CardLayout提供了两个构造器：

* CardLayout()：创建默认的CardLayout布局管理器
* CardLayout(int hgap, int vgap)：通过指定卡片与容器左右边界的间距（hgap）、上下边界（vgap）的间距来创建CardLayout布局管理器

CardLayout用于控制组件可见的5个常用方法如下：

* first(Container target)：显示target容器的第一张卡片
* last(Container target)：显示target容器中的最后一张卡片
* previous(Container target)：显示target容器中的前一张卡片
* next(Container target)：显示target容器的后一张卡片
* show(Container target, String name)：显示target容器中指定名字的卡片

示例代码：

**package** justTest;

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** CardLayoutTest

{

Frame f = **new** Frame("测试窗口");

String[] names = {"第一张" , "第二张" , "第三张"

, "第四张" , "第五张"};

Panel pl = **new** Panel();

**public** **void** init()

{

**final** CardLayout c = **new** CardLayout();

pl.setLayout(c);

**for** (**int** i = 0 ; i < names.length ; i++)

{

pl.add(names[i] , **new** Button(names[i]));

}

Panel p = **new** Panel();

ActionListener listener = e ->

{

**switch**(e.getActionCommand())

{

**case** "上一张":

c.previous(pl);

**break**;

**case** "下一张":

c.next(pl);

**break**;

**case** "第一张":

c.first(pl);

**break**;

**case** "最后一张":

c.last(pl);

**break**;

**case** "第三张":

c.show(pl , "第三张");

**break**;

}

};

// 控制显示上一张的按钮

Button previous = **new** Button("上一张");

previous.addActionListener(listener);

// 控制显示下一张的按钮

Button next = **new** Button("下一张");

next.addActionListener(listener);

// 控制显示第一张的按钮

Button first = **new** Button("第一张");

first.addActionListener(listener);

// 控制显示最后一张的按钮

Button last = **new** Button("最后一张");

last.addActionListener(listener);

// 控制根据Card名显示的按钮

Button third = **new** Button("第三张");

third.addActionListener(listener);

p.add(previous);

p.add(next);

p.add(first);

p.add(last);

p.add(third);

f.add(pl);

f.add(p , BorderLayout.***SOUTH***);

f.pack();

f.setVisible(**true**);

}

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

{

**new** CardLayoutTest().init();

}

}

上面的代码使用了事件编程，下面的章节会介绍到。

### 绝对定位

Java可以对GUI组件进行绝对定位，绝对定位就是设置组件的位置（x，y）宽（width）和高（height），这四个参数即可绝对定位一个组件。

在Java容器中采用绝对定位的步骤如下：

* 将Container的布局管理器设成null：setLayout(null)
* 向容器中添加组件时，先调用setBounds()或setSize()方法来设置组件的大小、位置，或者直接创建GUI组件时通过构造参数指定该组件的大小、位置，然后将该组件添加到容器中。

示例程序：

**package** justTest;

**import** java.awt.\*;

**public** **class** NullLayoutTest

{

Frame f = **new** Frame("测试窗口");

Button b1 = **new** Button("第一个按钮");

Button b2 = **new** Button("第二个按钮");

**public** **void** init()

{

// 设置使用null布局管理器

f.setLayout(**null**);

// 下面强制设置每个按钮的大小、位置

b1.setBounds(20, 30, 90, 28);

f.add(b1);

b2.setBounds(120, 30, 120, 35);

f.add(b2);

f.setBounds(50, 50, 200, 100);

f.setVisible(**true**);

}

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

{

**new** NullLayoutTest().init();

}

}

不过使用绝对定位也是有代价的，它丧失了跨平台特性。

### BoxLayout

虽然GridBagLayout功能强大，但它实在是太复杂了，所以Swing引入了一个新的布局管理器：BoxLayout，它保留了GridBagLayout的很多优点，但是却没有那么复杂。BoxLayout可以在垂直和水平两个方向上添加GUI组件，BoxLayout提供了如下一个简单的构造器：

BoxLayout(Container target, int axis)：指定创建基于target容器的BoxLayout布局管理器，其中的组件按照axis的顺序进行排列。其中axis有BoxLayout.X\_AXIS（横向）和BoxLayout.Y\_AXIS（纵向）两个方向。

示例代码：

**package** justTest;

**import** java.awt.\*;

**import** javax.swing.\*;

**public** **class** BoxLayoutTest

{

**private** Frame f = **new** Frame("测试");

**public** **void** init()

{

f.setLayout(**new** BoxLayout(f , BoxLayout.***Y\_AXIS***));

// 下面按钮将会垂直排列

f.add(**new** Button("第一个按钮"));

f.add(**new** Button("按钮二"));

f.pack();

f.setVisible(**true**);

}

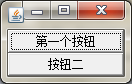
**public** **static** **void** main(String[] args)

{

**new** BoxLayoutTest().init();

}

}



BoxLayout常和Box容器结合使用，Box是一个特殊的容器，它有点向Panel容器，但该容器默认使用BoxLayout布局管理器，Box提供了如下两个静态方法来创建Box对象：

* createHorizontalBox()：创建一个水平排列组件的Box容器
* createVerticalBox()：创建一个垂直排列组件的Box容器

创建完Box容器之后，就可以使用Box来盛装普通的GUI组件，然后将这些Box组件添加到其他容器中，从而形成整体的窗口布局。

示例代码：

**package** justTest;

**import** java.awt.\*;

**import** javax.swing.\*;

**public** **class** BoxTest

{

**private** Frame f = **new** Frame("测试");

// 定义水平摆放组件的Box对象

**private** Box horizontal = Box.*createHorizontalBox*();

// 定义垂直摆放组件的Box对象

**private** Box vertical = Box.*createVerticalBox*();

**public** **void** init()

{

horizontal.add(**new** Button("水平按钮一"));

horizontal.add(**new** Button("水平按钮二"));

vertical.add(**new** Button("垂直按钮一"));

vertical.add(**new** Button("垂直按钮二"));

f.add(horizontal , BorderLayout.***NORTH***);

f.add(vertical);

f.pack();

f.setVisible(**true**);

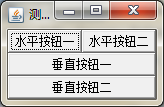
}

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

{

**new** BoxTest().init();

}

}   
 

BoxLayout没有提供设置间距的构造器和方法，但是它采用了另外一种方式来控制组件的间距——BoxLayout使用Glue（橡胶）、Strut（支架）和RigidArea（刚性区域）的组件来控制组件间的距离：

* Glue代表可以在纵向横向两个方向上同时拉伸的空白组件（间距）
* Strut代表可以在横向纵向任意一个方向上拉伸的空白组件（间距）
* RigidArea代表不可拉伸的空白组件（间距）

Box提供了如下5个静态方法来创建Glue、Strut和RigidArea

* createHorizontalGlue()：创建一条水平Glue（可在两个方向上同时拉伸的间距）
* createVerticalGlue()：创建一条垂直Glue（可在两个方向上同时拉伸的间距）
* createHorizontalStrut(int width)：创建一条指定宽度的水平Strut（可在垂直方向上拉伸的间距）
* createVerticalStrut(int height)：创建一条指定高度的垂直Strut（可在水平方向上拉伸的间距）
* createRigidArea(Dimension d)：创建指定宽度、高度的RigidArea（不可拉伸的间距）

**上面的Dimension是一个类，表示尺寸，可以通过它的构造函数Dimension(int width, int height)来创建一个指定了宽和高的一个区域。**

上面的5个静态方法的返回值均为Component对象（代表间距），程序会将这些分割Component添加到两个普通的GUI组件之间，用以控制组件之间的间距。

示例代码：

**package** justTest;

**import** java.awt.\*;

**import** javax.swing.\*;

**public** **class** BoxSpaceTest

{

**private** Frame f = **new** Frame("测试");

// 定义水平摆放组件的Box对象

**private** Box horizontal = Box.*createHorizontalBox*();

// 定义垂直摆放组件的Box对象

**private** Box vertical = Box.*createVerticalBox*();

**public** **void** init()

{

horizontal.add(**new** Button("水平按钮一"));

horizontal.add(Box.*createHorizontalGlue*());

horizontal.add(**new** Button("水平按钮二"));

// 水平方向不可拉伸的间距，其宽度为10px

horizontal.add(Box.*createHorizontalStrut*(10));

horizontal.add(**new** Button("水平按钮三"));

vertical.add(**new** Button("垂直按钮一"));

vertical.add(Box.*createVerticalGlue*());

vertical.add(**new** Button("垂直按钮二"));

// 垂直方向不可拉伸的间距，其高度为10px

vertical.add(Box.*createVerticalStrut*(10));

vertical.add(**new** Button("垂直按钮三"));

f.add(horizontal , BorderLayout.***NORTH***);

f.add(vertical);

f.pack();

f.setVisible(**true**);

}

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

{

**new** BoxSpaceTest().init();

}

}



## AWT常用组件

AWT需要调用运行平台的图形界面来创建和平台一致的对等体，因此AWT只能使用所有平台都支持的公共组件，所以AWT只提供了一些常用的GUI组件。

### 1、基本组件

AWT提供了如下基本组件：

* Button
* Canvas（用于绘图的画布）
* Checkbox（复选框组件，也可以变成单选框组件）
* CheckboxGroup（用于将多个Checkbox组件组合成一组，一组Checkbox组件将只有一个可以被选中，即全部变成单选框组件）
* Choice（下拉式选择框组件）
* Frame（窗口，在GUI程序里通过该类创建窗口）
* Lable（标签类，用于放置提示性文本）
* List（列表框组件，可以添加多条项目）
* Panel（不能单独存在的基本容器类，必须放到其他容器中）
* Scrollbar（滑动条组件）
* ScrollPane（带水平及垂直滚动条的容器组件）
* TextArea（多行文本域）
* TextField（单行文本域）

查阅API文档获取相应的构造器。

示例程序：

**package** justTest;

**import** java.awt.\*;

**import** javax.swing.\*;

**public** **class** CommonComponent

{

Frame f = **new** Frame("测试");

// 定义一个按钮

Button ok = **new** Button("确认");

CheckboxGroup cbg = **new** CheckboxGroup();

// 定义一个单选框（处于cbg组），初始处于被选中状态

Checkbox male = **new** Checkbox("男" , cbg , **true**);

// 定义一个单选框（处于cbg组），初始处于没有选中状态

Checkbox female = **new** Checkbox("女" , cbg , **false**);

// 定义一个复选框，初始处于没有选中状态

Checkbox married = **new** Checkbox("是否已婚？" , **false**);

// 定义一个下拉选择框

Choice colorChooser = **new** Choice();

// 定义一个列表选择框

List colorList = **new** List(6, **true**);

// 定义一个5行、20列的多行文本域

TextArea ta = **new** TextArea(5, 20);

// 定义一个50列的单行文本域

TextField name = **new** TextField(50);

**public** **void** init()

{

colorChooser.add("红色");

colorChooser.add("绿色");

colorChooser.add("蓝色");

colorList.add("红色");

colorList.add("绿色");

colorList.add("蓝色");

// 创建一个装载了文本框、按钮的Panel

Panel bottom = **new** Panel();

bottom.add(name);

bottom.add(ok);

f.add(bottom , BorderLayout.***SOUTH***);

// 创建一个装载了下拉选择框、三个Checkbox的Panel

Panel checkPanel = **new** Panel();

checkPanel.add(colorChooser);

checkPanel.add(male);

checkPanel.add(female);

checkPanel.add(married);

// 创建一个垂直排列组件的Box，盛装多行文本域、Panel

Box topLeft = Box.*createVerticalBox*();

topLeft.add(ta);

topLeft.add(checkPanel);

// 创建一个水平排列组件的Box，盛装topLeft、colorList

Box top = Box.*createHorizontalBox*();

top.add(topLeft);

top.add(colorList);

// 将top Box容器添加到窗口的中间

f.add(top);

f.pack();

f.setVisible(**true**);

}

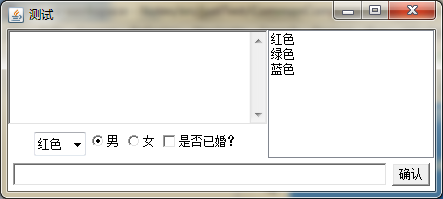
**public** **static** **void** main(String[] args)

{

**new** CommonComponent().init();

}

}



### 对话框（Dialog）

Dialog是Window类的子类，是一个容器类，属于特殊组件。对话框是可以独立存在的顶级窗口，因此用法与普通窗口的用法几乎完全一样，关于对话框，有以下两点需要注意：

* 对话框通常依赖于其他窗口，就是通常有一个parent窗口
* 对话框有非模式（non-modal）和模式（modal）两种，当某个模式对话框被打开之后，该模式对话框总是位于它依赖的窗口上；在模式对话框被关闭之前，它依赖的窗口无法获得**焦点**

对话框有多个重载的构造器，它的构造器可能有如下三个参数：

* owner：指定该对话框所依赖的窗口，既可以是窗口也可以是对话框
* title：指定该对话框的窗口标题
* modal：指定该对话框是否是模式的，可以是true或false

示例代码：  
 **package** justTest;

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** DialogTest

{

Frame f = **new** Frame("测试");

Dialog d1 = **new** Dialog(f, "模式对话框" , **true**);

Dialog d2 = **new** Dialog(f, "非模式对话框" , **false**);

Button b1 = **new** Button("打开模式对话框");

Button b2 = **new** Button("打开非模式对话框");

**public** **void** init()

{

d1.setBounds(0 , 0 , 300, 400);

d2.setBounds(0 , 0 , 300, 400);

b1.addActionListener(e -> d1.setVisible(**true**));

b2.addActionListener(e -> d2.setVisible(**true**));

f.add(b1 , BorderLayout.***NORTH***);

f.add(b2);

f.pack();

f.setVisible(**true**);

}

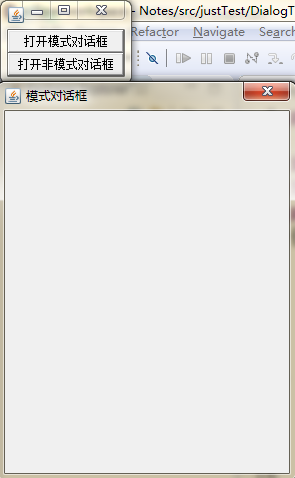
**public** **static** **void** main(String[] args)

{

**new** DialogTest().init();

}

}



Dialog还有一个子类：FileDialog（文件对话框，用于打开或保存文件）。

FileDialog也提供了几个构造器，可分别支持parent、title和mode三个参数，parent和title指定文件对话框所属的父窗口和标题；而**mode用于指定该文件对话框进行什么操作（打开文件/保存文件），该参数一共有两种：FileDialog.LOAD、FileDialog.SAVE。**

FIleDialog并不能指定自己是模式的还是非模式的，它依赖于**运行平台**的实现，如果运行平台的文件对话框是模式的，那么FileDialog就是模式的，否则FileDialog就是非模式的。

FIleDialog提供了下面两个方法来获取被打开/保存文件的路径

* **getDirectory()：**获取FileDialog被打开/保存文件的绝对路径
* **getFile()：**获取FileDialog被打开/保存文件的文件名

实例代码：

**package** justTest;

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** FileDialogTest

{

Frame f = **new** Frame("测试");

// 创建两个文件对话框

FileDialog d1 = **new** FileDialog(f

, "选择需要打开文件" , FileDialog.***LOAD***);

FileDialog d2 = **new** FileDialog(f

, "选择保存文件的路径" , FileDialog.***SAVE***);

Button b1 = **new** Button("打开文件");

Button b2 = **new** Button("保存文件");

**public** **void** init()

{

b1.addActionListener(e ->

{

d1.setVisible(**true**);

// 打印出用户选择的文件路径和文件名

System.***out***.println(d1.getDirectory()

+ d1.getFile());

});

b2.addActionListener(e ->

{

d2.setVisible(**true**);

// 打印出用户选择的文件路径和文件名

System.***out***.println(d2.getDirectory()

+ d2.getFile());

});

f.add(b1);

f.add(b2 , BorderLayout.***SOUTH***);

f.pack();

f.setVisible(**true**);

}

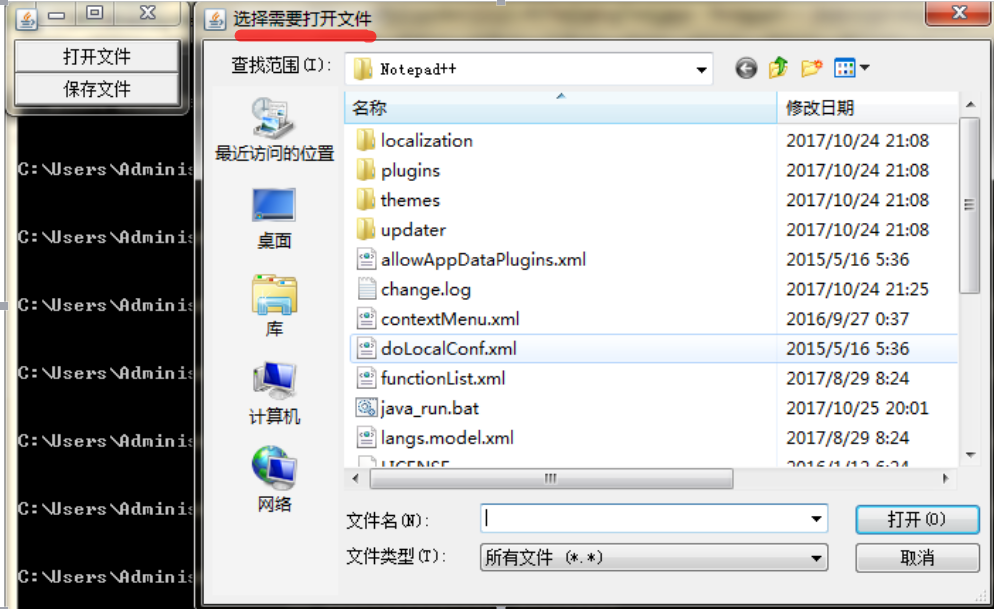
**public** **static** **void** main(String[] args)

{

**new** FileDialogTest().init();

}

}

 从运行结果可以看出，点击button之后弹出的文件对话框就是Windows（即Java运行平台）提供的文件对话框。

## 事件处理

前面已经介绍了各种各样的组件，但是他们**并不具备响应事件的能力**，想要对事件进行处理，需要对事件进行监听，而Frame和组件本身是没有这样的功能能的。

### Java事件模型的流程

在事件处理的过程中，主要涉及到三类对象：

* Event Source（事件源）
  + 通常指各个组件
* Event（事件）
  + 事件**封装了GUI组件上发生的特定事情**，程序通过Event对象来获得GUI组件上所发生事件的详细信息
* Event Listener（事件监听器）
  + 监听事件源发生的事件，并对其作出处理

AWT把用户在GUI组件上的操作都封装成了对应的EVent对象，事件会触发**事件监听器**，事件监听器会调用对应的事件处理器（即事件监听器的实例方法）做出响应

AWT的事件处理机制是一种委派式事件处理方式——普通组件（事件源）将事件的处理工作委托给特定的对象（事件监听器）；当该源事件发生指定的事件时，就通知所委托的事件监听器，由事件监听器来处理这个事件

示例代码：

**package** justTest;

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** EventQs

{

**private** Frame f = **new** Frame("测试事件");

**private** Button ok = **new** Button("确定");

**private** TextField tf = **new** TextField(30);

**public** **void** init()

{

// 使用addActionListener向组件注册事件监听器

ok.addActionListener(**new** OkListener()); //②

f.add(tf);

f.add(ok , BorderLayout.***SOUTH***);

f.pack();

f.setVisible(**true**);

}

// 定义事件监听器类

**class** OkListener **implements** ActionListener //①

{

// 下面定义的方法就是事件处理器，用于响应特定的事件

**public** **void** actionPerformed(ActionEvent e) //③

{

System.***out***.println("用户单击了ok按钮");

tf.setText("Hello World");

}

}

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

{

**new** EventQs().init();

}

}

1. 实现事件监听类，该监听类是一个特殊的Java类，必须实现一个XxxListener接口
2. 创建普通组件（事件源），创建事件监听器对象
3. 调用addXxxListener()方法将事件监听器对象注册给普通组件（事件源）。

当事件源上发生指定的事件时，AWT会触发事件监听器，由事件监听器调用相应的方法（事件处理器）来处理事件，事件源上所发生的事件会作为参数传入事件处理器

### 事件和事件监听器

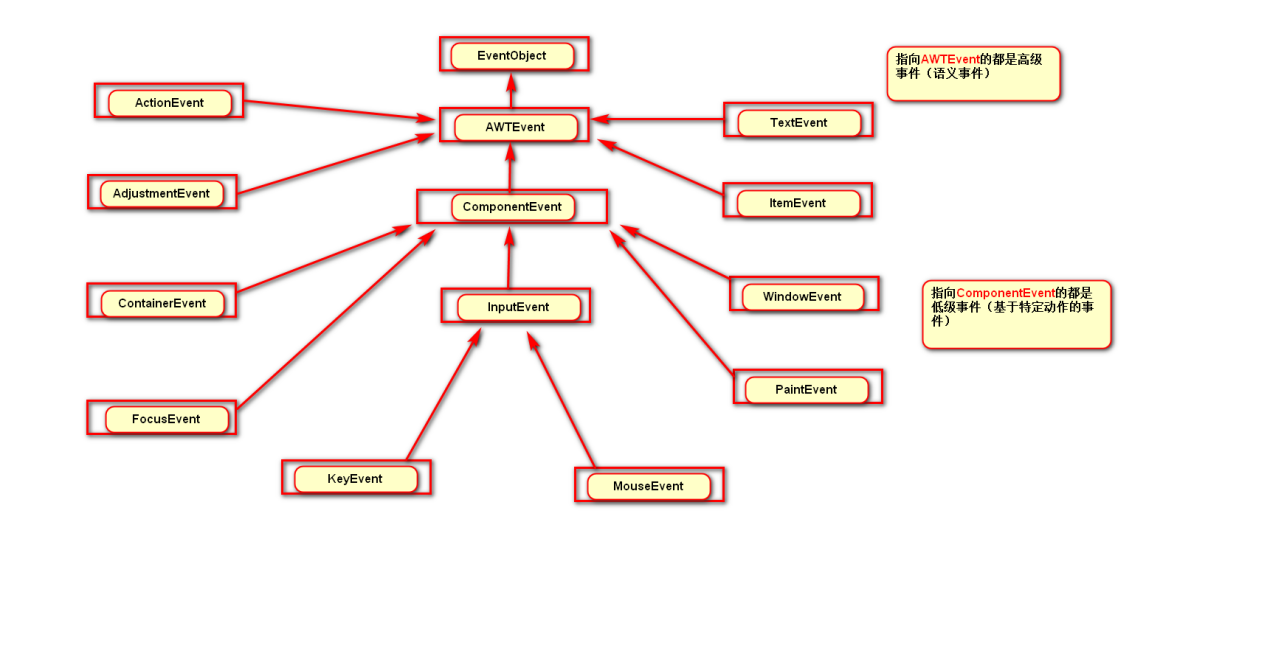
事件监听器必须实现事件监听器接口，AWT提供了大量的事件监听器接口用于实现不同类型的事件处理器，来监听不同的事件。

AWT中还提供了丰富的事件类，用于封装不同组件上所发生的特定操作——AWT的事件类都是AWTEvent类的子类，AWTEvent是EventObject（EventObject类代表更广义的事件对象，包括Swing组件上所触发的事件、数据库连接所触发的事件等）的子类

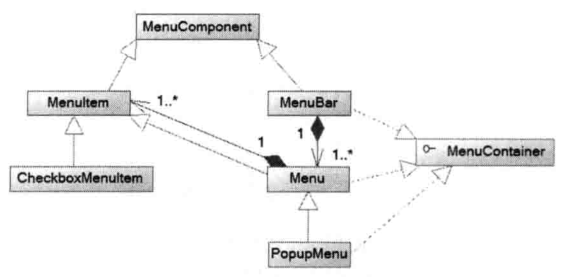
AWT事件分为两大类：低级事件、高级事件

* 低级事件
  + 低级事件是指基于特定动作的事件，当组件得到焦点、失去焦点时触发**焦点事件**
    - ComponentEvent：组件事件，触发该事件的是：
      * 组件的尺寸变化、位置的移动、显示和隐藏状态
    - ContainerEvent：容器事件
      * 容器中添加、删除组件会触发该事件
    - FocusEvent：**焦点**事件
      * 组件得到焦点或失去焦点
    - KeyEvent：键盘事件
      * 按键被按下、松开、单击时触发该事件
    - MouseEvent：鼠标事件
      * 进行单击、按下、松开、移动鼠标等动作时触发该事件
    - PaintEvent：组件绘制事件
      * 该事件是一个特殊的事件类型，当GUI组件调用update/paint方法来呈现自身时触发该事件，该事件并非专用于事件处理模型
* 高级事件（语义事件）
  + 高级事件是基于语义的事件，它可以不和特定的事件相关联，而依赖于触发该事件的类。比如：在TextField中按Enter键会触发ActionEvent事件，在滑动条上移动滑块会触发AdjustmentEvent事件，选中项目列表中的某一项就会触发ItemEvent事件
    - ActionEvent：动作事件，当按钮、菜单项被单击，在TextField中按Entet键时触发该事件
    - AdjustmentEvent：调节事件，在滑动条上移动滑块以调节数值时触发该事件
    - ItemEvent：选项事件，当用户选中某项，或取消选中某项时触发该事件
    - TextEvent：文本事件，当文本框、文本域里的文本发生改变时触发该事件

AWT事件继承层次图：



## AWT菜单



AWT菜单组件的关系，黑色菱形代表组合关系，虚线空白三角代表实现，实线空白三角代表泛化

从上图中可以看出：

MenuBar组合了Menu，而Menu又组合了MenuItem

也就是说，Menu由一个一个的MenuItem组成，而MenuBar又由一个一个的Mean组成

Menu的子类PopupMenu可以用来创建右键菜单

# Lambda表达式

Lambda表达式是Java8新增的一个特性，他相当于**给方法传递了一个代码块作为参数**

**Lambda表达式和匿名内部类还是有着千丝万缕的关系的**

**看代码：**

**interface Command {**

**void process(int[] target);**

**}**

**class ProcessArray {**

**public void process(int[] target, Command cmd) {**

**cmd.process(target);**

**}**

**}**

**class AddCommand implements Command {**

**public void process(int[] target) {**

**int sum = 0;**

**for(int tmp: target)**

**sum += tmp;**

**System.out.println("数组元素的总和是:" + sum);**

**}**

**}**

**public class CommandTest {**

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

**ProcessArray pa = new ProcessArray();**

**int[] target = {3, -4, 6, 4};**

**//注意，这里的参数列表和上面的ProcessArray类中的成员函数process()方法中的参数列表是一致的，第一个是数组，第二个是cmd对象**

**pa.process(target, (int[] array)->{**

**int sum = 0;**

**for (int tmp : array )**

**{**

**sum += tmp;**

**}**

**System.out.println("数组元素的总和是:" + sum);**

**});**

**pa.process(target, new AddCommand());**

**}**

**}**

**---------------------------------------------------------------------------------------------------------------------------------**

**Lambda的语法格式如下：**

* **形参**
* **箭头(->)**
* **代码块，使用花括号用来表示结束，如果只有一条语句，可以省略花括号**
  + **其实Lambda是有返回值的，不过它的return可以被省略，Lambda会自动返回这条语句的值**

**先看一段代码：**

**i****interface Eatable {**

**void taste();**

**}**

**interface Flyable {**

**void fly(String weather);**

**}**

**interface Addable {**

**int add(int a , int b);**

**}**

**public class LambdaQs {**

**// 调用该方法需要Eatable对象**

**public void eat(Eatable e) {**

**System.out.println(e);**

**e.taste();**

**}**

**// 调用该方法需要Flyable对象**

**public void drive(Flyable f) {**

**System.out.println("我正在驾驶：" + f);**

**f.fly("【碧空如洗的晴日】");**

**}**

**// 调用该方法需要Addable对象**

**public void test(Addable add) {**

**System.out.println("5与3的和为：" + add.add(5, 3));**

**}**

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

**LambdaQs lq = new LambdaQs();**

**// Lambda表达式的代码块只有一条语句，可以省略花括号。**

**lq.eat(()-> System.out.println("苹果的味道不错！"));**

**// Lambda表达式的形参列表只有一个形参，省略圆括号**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**lq.drive(weather -> {**

**System.out.println("今天天气是：" + weather);**

**System.out.println("直升机飞行平稳");**

**});**

**//如果不使用Lambda表达式来实现的话，可以用下面蓝色字体的匿名内部类的形式来实现**

**//总之，只要实现Flyable接口中的flyable()方法即可**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**lq.drive(new Flyable() {**

**public void fly(String weather) {**

**System.out.println("现在的天气状况是：" + weather);**

**}**

**});**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**// Lambda表达式的代码块只有一条语句，省略花括号**

**// 代码块中只有一条语句，即使该表达式需要返回值，也可以省略return关键字。**

**lq.test((a , b)->a + b);**

**}**

**}**

从上面这段代码中，我们可以看出其实Lambda表达式被当成了一个**任意类型**的对象

**它的存在主要就是为了以一种非常简洁的语法实现函数式接口的抽象方法，**

**所谓函数式接口：**

**就是只包含一个抽象方法的接口**

**看代码：**

/\*\*

\* java8专门门为函数式接口提供了@FunctionalInterface注解，

\* 该注解通常放在接口定义前面，该注解对程序功能没有任何作用，

\* 它只是用来告诉编译器执行更严格检查————检查该接口必须是函数式接口，

\* 否则编译就会报错

\*/ **@FunctionalInterface**

**interface FkTest {**

**void run();**

**}**

**public class LambdaTest {**

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

**// Runnable接口中只包含一个无参数的方法**

**// Lambda表达式代表的匿名方法实现了Runnable接口中唯一的、无参数的方法**

**// 因此下面的Lambda表达式创建了一个Runnable对象**

**Runnable r = () -> {**

**for(int i = 0 ; i < 100 ; i ++)**

**System.out.println();**

**};**

**// // 下面代码报错: 不兼容的类型: Object不是函数接口**

**// Object obj = () -> {**

**// for(int i = 0 ; i < 100 ; i ++)**

**// {**

**// System.out.println();**

**// }**

**// };**

**//使用类型强制转换**

**Object obj1 = (Runnable)() -> {**

**for(int i = 0 ; i < 100 ; i ++)**

**System.out.println();**

**};**

**// 同样的Lambda表达式可以被当成不同的目标类型，唯一的要求是：**

**// Lambda表达式的形参列表与函数式接口中唯一的抽象方法的形参列表相同**

**Object obj2 = (FkTest)() -> {**

**for(int i = 0 ; i < 100 ; i ++)**

**System.out.println();**

**};**

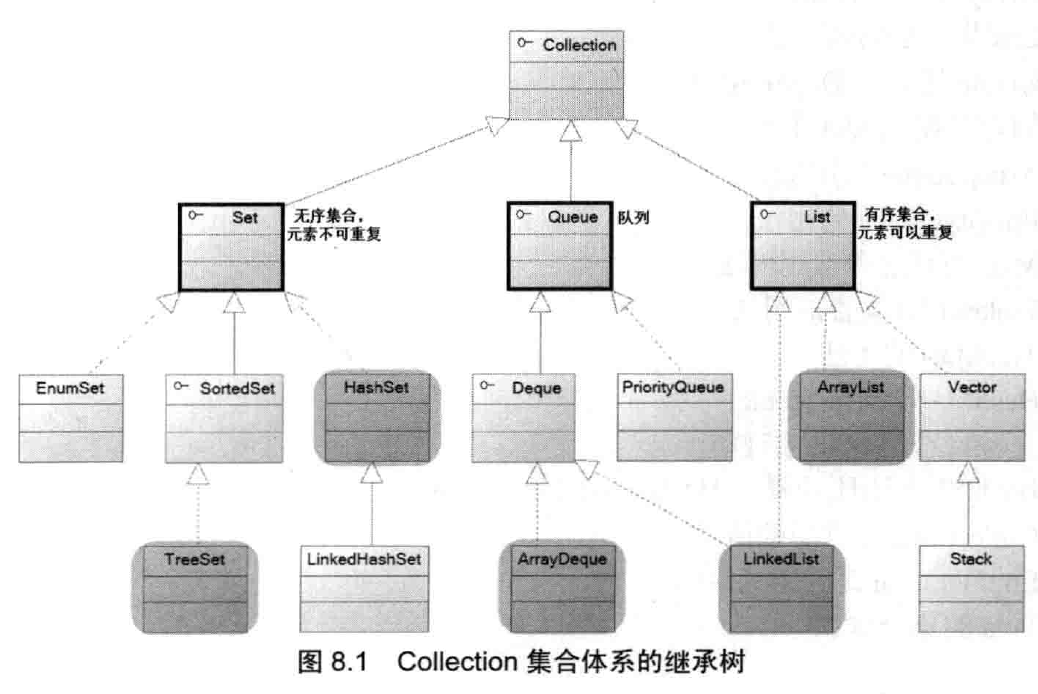
**}**

**}**

**---------------------------------------------------------------------------------------------------------------------------------**

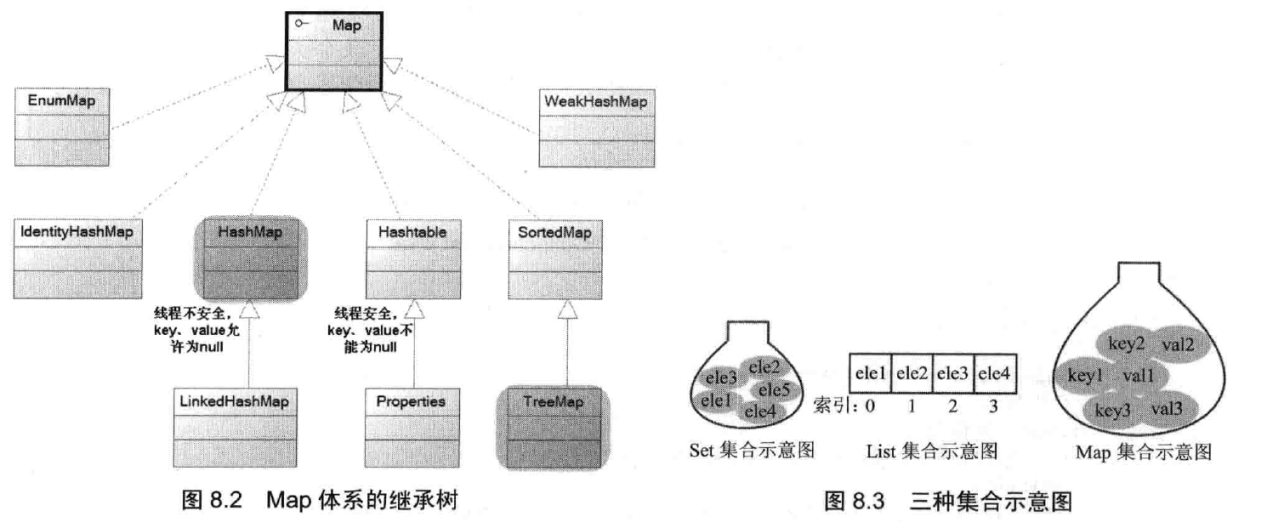
# Java集合

## Java集合概述



**Collection集合体系继承树，图中只是列出了比较常用的，还有一些子接口/实现类并没有显示出来。还有一个队列Queue，在后面介绍，阴影部分为常用实现类/集合**

**从图中可以看出Java的集合类主要由两个接口派生：Collection和Map，他俩是Java集合框架的根接口。**



**Map的继承树：**

**常用的三种容器是：Map、Set、List，他们都是接口**

**他们的访问方式也是不同的：**

**Map通过key值来进行元素的访问**

**Set直接通过元素本身进行访问，因为Set容器中是不允许存在相同的元素的，且其中的元素使无序放置的**

**List通过元素对应的索引进行访问**

## Collection和Iterator接口

下面的代码示范了Collection集合中的基本操作：

**---------------------------------------------------------------------------------------------------------------------------------**

package justTest;

import java.util.\*;

public class CollectionTest

{

public static void main(String[] args)

{

Collection c = new ArrayList();

// 添加元素

c.add("孙悟空");

// 虽然集合里不能放基本类型的值，但Java支持自动装箱

c.add(6);

System.out.println("c集合的元素个数为:" + c.size()); // 输出2

// 删除指定元素

c.remove(6);

System.out.println("c集合的元素个数为:" + c.size()); // 输出1

// 判断是否包含指定字符串

System.out.println("c集合是否包含\"孙悟空\"字符串:"

+ c.contains("孙悟空")); // 输出true

c.add("轻量级Java EE企业应用实战");

System.out.println("c集合的元素：" + c);

Collection books = new HashSet();

books.add("轻量级Java EE企业应用实战");

books.add("疯狂Java讲义");

System.out.println("c集合是否完全包含books集合？"

+ c.containsAll(books)); // 输出false

// 用c集合减去books集合里的元素

c.removeAll(books);

System.out.println("c集合的元素：" + c);

// 删除c集合里所有元素

c.clear();

System.out.println("c集合的元素：" + c);

// 控制books集合里只剩下c集合里也包含的元素

books.retainAll(c);//相当于 books - c

System.out.println("books集合的元素:" + books);

}

}

**---------------------------------------------------------------------------------------------------------------------------------**

上面代码中的

System.out.println("c集合的元素：" + c);

之所以能这样写，是因为所有的Collection实现类都重写了toString()方法

### 1、使用Lambda表达式遍历集合

**Java8位Iterator接口新增了一个****ForEach(Consumer action)默认方法，该方法所需参数的类型是一个函数式接口，而Iterator接口是Collection的父接口，因此所有的Collection集合都能使用forEach(Consumer action)方法来遍历元素。**

**在程序使用****forEach(Consumer action)方法遍历集合元素时，程序会依次将集合中的元素传给****Consumer的accept(T t)方法//accdept()方法中的T意思是Templet（模板），可以传入任意类型的参数//（它是Consumer函数式接口中唯一的抽象方法）。正因为Consumer是函数式接口，我们才可以使用Lambda表达式来对集合元素进行遍历。**

**示例代码：**

**---------------------------------------------------------------------------------------------------------------------------------**

**import java.util.\*;**

**public class CollectionEach {**

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

**//创建一个HashSet集合**

**Collection books = new HashSet();**

**books.add("轻量级Java EE企业级应用实战");**

**books.add("疯狂Java讲义");**

**books.add("疯狂Android讲义");**

**//调用forEach()方法遍历集合**

**//forEach()方法的参数列表是一个Consumer对象，而Consumer中的抽象方法accept需要一个对象作为参数，因此就需要为其提供一个参数obj，因此只需要给出一个Lambda表达式实现该函数式接口中的抽象方法accept(T t)即可**

**books.forEach(obj -> System.out.println("迭代集合元素" + obj));**

**}**

**}**

**---------------------------------------------------------------------------------------------------------------------------------**

### 使用Java8增强的Iterator遍历集合元素

Iterator是一个接口，它是Java集合框架的成员，与Collection和Map不同的是：  
 Collection和Map都是用来盛装其他对象的容器

Iterator则是用来遍历集合元素的（迭代访问），其对象被称作迭代器

Iterator接口中一共定义了如下4个方法：

* + - boolean hasNext()：:如果被迭代的集合元素还没有被遍历完，返回true
    - Object next()：返回集合里的下一个元素
    - void remove()：删除集合里上一次next方法返回的元素
    - void forEachRemaining(Consumer action)，这是Java 8为Iterator新增的默认方法，该方法可以使用Lambda表达式来遍历集合元素

注意：  
 remove方法只能在next方法被调用之后才能使用，且只能用一次

示例代码：

**---------------------------------------------------------------------------------------------------------------------------------**

import java.util.\*;

public class IteratorTest {

public static void main(String[] args) {

//创建一个HashSet集合

Collection books = new HashSet();

books.add("轻量级Java EE企业级应用实战");

books.add("疯狂Java讲义");

books.add("疯狂Android讲义");

Iterator it = books.iterator();

int count = 0;

while(it.hasNext()) {

count++;

String book = (String)it.next();

System.out.println(book);

//根据这个计数器，可以看出来，remove()方法删除的就是集合中上次next()方法返回 //的元素

if(count == 1)

it.remove();

System.out.println(books);

book = "测试字符串";

}

System.out.println(books);

}

}

**---------------------------------------------------------------------------------------------------------------------------------**

Iterator是依附于Collection对象的，如果仅有iterator对象而没有集合，那么Iterator的存在没有任何意义，它本身就是用来遍历集合的，如果连集合都没有，它也就失去了存在的意义

在Iterator遍历集合的时候，不能改变集合中的元素

### 使用Lambda表达式遍历Iterator

当然就是使用Iterator中所提供的forEachRemaining(Consumer action)方法，和上面的forEach(Consumer action)差不多

代码示例：  
**---------------------------------------------------------------------------------------------------------------------------------**

import java.util.\*;

public class IteratorEach {

public static void main(String[] args) {

Collection books = new HashSet();

books.add("轻量级Java EE企业级应用实战");

books.add("疯狂Java讲义");

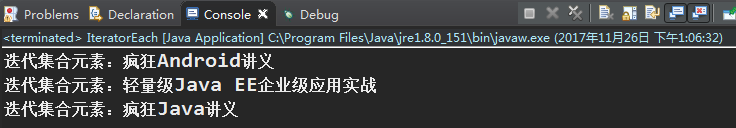
books.add("疯狂Android讲义");

Iterator it = books.iterator();

it.forEachRemaining(obj -> System.out.println("迭代集合元素：" + obj));

}

}



**---------------------------------------------------------------------------------------------------------------------------------**

### 使用foreach循环遍历集合元素

使用foreach循环进行遍历，不需要获得集合的迭代器

代码示例：  
**---------------------------------------------------------------------------------------------------------------------------------**

import java.util.\*;

public class ForeachTest

{

public static void main(String[] args)

{

Collection books = new HashSet();

books.add(new String("轻量级Java EE企业应用实战"));

books.add(new String("疯狂Java讲义"));

books.add(new String("疯狂Android讲义"));

for (Object obj : books)

{

String book = (String)obj;

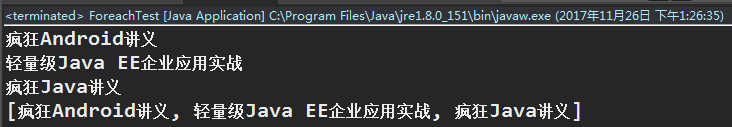
System.out.println(book);

}

System.out.println(books);

}

}



**---------------------------------------------------------------------------------------------------------------------------------**

### 使用Java 8新增的Predicate操作集合

Predicate意为“谓词”，作修饰之用，其实Predicate是一个函数式接口，它是Collection集合类在Java 8中新增的方法removeIf(Predicate filter)的参数类型，Filter意为过滤，也就是筛选条件，因此可以直接使用Lambda表达式来实现

Predicate中的抽象方法是boolean Test(T t)，T意为模板，因此传入的参数是待筛选的对象

示例代码：

**---------------------------------------------------------------------------------------------------------------------------------**

import java.util.\*;

public class PredicateTest {

public static void main(String[] args) {

Collection books = new HashSet();

books.add(new String("轻量级Java EE企业应用实战"));

books.add(new String("疯狂Java讲义"));

books.add(new String("疯狂iOS讲义"));

books.add(new String("疯狂Ajax讲义"));

books.add(new String("疯狂Android讲义"));

// 使用Lambda表达式（目标类型是Predicate）过滤集合

books.removeIf(ele -> ((String)ele).length() < 10);

//他只会输出长度大于10的元素

System.out.println(books);

}

}

**---------------------------------------------------------------------------------------------------------------------------------**

dafsgPredicate可以充分简化集合的运算，以下是三个需求：  
 计算集合中含有”疯狂”的元素的个数

计算集合中含有”Java”的元素的个数

计算集合中串长度大于10的元素个数

代码示例：

**---------------------------------------------------------------------------------------------------------------------------------**

import java.util.\*;

public class PredicateTest {

public static void mian(String[] args) {

Collection books = new HashSet();

books.add("疯狂Java讲义");

books.add("疯狂Android讲义");

books.add("轻量级java EE企业级应用开发");

//contains()方法是String类的一个成员方法

System.out.println(callAll(books, ele -> ((String)ele).contains("疯狂")));

System.out.println(callAll(books, ele -> ((String)ele).contains("Java")));

System.out.println(callAll(books, ele -> ((String)ele).length() > 10));

}

public static int callAll(Collection books, Predicate p) {

int total = 0;

for(Object obj: books) {

if(p.test(obj)) {

total++;

}

}

return total;

}

}

**---------------------------------------------------------------------------------------------------------------------------------**

### 使用Java 8新增的Stream操作集合

Java 8中新增了几个流式API：Stream， IntStream，LongStream，DoubleStream

Stream是通用的流接口，其余三个分别代表Int，Long，Double的流接口

每一个Stream接口都有一个内部接口Builder，这个Builder接口了一个对应的函数式接口Consumer

eg:  
 IntStream接口中的Builder继承的就是IntConsumer函数式接口，这个函数式接口中的抽象类是accept(int value)

Stream就是流，可以把数据流理解成流水线，在流水线上，我们可以对流进行任意的操作，但是当**流水线走到尽头之后，就无法再进行操作了**，只能从数据员再次获得一个流进行操作

代码示例：

**---------------------------------------------------------------------------------------------------------------------------------**

import java.util.stream.\*;

public class IntStreamTest {

public static void main(String[] args) {

/\*\*

\* 重点看下面这段蓝色代码:

\* IntStream中有一个内部接口Builder

\* public static Builder builder() {

\* return new Streams.IntStreamBuilderImpl();

\* }

\*

\* static final class IntStreamBuilderImpl

\* extends AbstractStreamBuilderImpl<Integer, Spliterator.OfInt>

\* implements IntStream.Builder, Spliterator.OfInt {

\* }

\* IntStreamBuilderImpl是Streams类中的一个静态内部类

\* builder方法会返回一个IntStreamBuilderImpl类型的对象，因为IntStreamBuilderImpl

\* 实现了IntStream.Builder，因此返回类型值可以写成Builder

\* 由此，通过Intstream的静态方法builder得到了一个IntStreamBuilderImpl型对象

\* IntStreamBuilderImpl实现了Builder中的抽象方法accept()

\* 继承了Builder中的默认方法

\* default Builder add(int t) {

\* accept(t);

\* return this;

\* }

\* 执行完add之后会返回当前对象，可继续执行add，在结尾的时候

\* 通过Build中被重写的方法build将BUilder还原成IntStream

\*/

IntStream is = IntStream.builder()

.add(20)

.add(13)

.add(-2)

.add(18)

.build();

// 下面调用聚集方法的代码每次只能执行一个

System.out.println("is所有元素的最大值：" + is.max().getAsInt());

System.out.println("is所有元素的最小值：" + is.min().getAsInt());

System.out.println("is所有元素的总和：" + is.sum());

System.out.println("is所有元素的总数：" + is.count());

System.out.println("is所有元素的平均值：" + is.average());

System.out.println("is所有元素的平方是否都大于20:"

+ is.allMatch(ele -> ele \* ele > 20));

System.out.println("is是否包含任何元素的平方大于20:"

+ is.anyMatch(ele -> ele \* ele > 20));

// 将is映射成一个新Stream，新Stream的每个元素是原Stream元素的2倍+1

IntStream newIs = is.map(ele -> ele \* 2 + 1);

// 使用方法引用的方式来遍历集合元素

newIs.forEach(System.out::println); // 输出41 27 -3 37

}

}

**---------------------------------------------------------------------------------------------------------------------------------**

# Java UML

## UML中的关系

UML中的关系主要有依赖、关联（聚合，组合）、泛化和实现等，这些关系在UML图中均由箭头和线条来进行表示

实现方式区别：

1. 依赖关系：关系对象出现在局部变量或者方法的参数里，或者关系类的静态方法被调用
2. 关联关系：关系对象出现在实例变量中
   * 1. 聚合关系:关系对象出现在实例变量中
     2. 合成关系：关系对象出现在实例变量中
3. 泛化: extends
4. 实现： implements

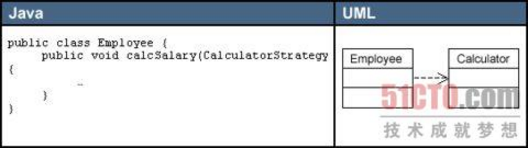
### 依赖（Dependency）

依赖关系描述了一种，“类与类之间” 一种 “使用与被使用” 的关系

依赖关系在Java中的具体代码表现形式为B为A的构造器或方法中的局部变量、方法或构造器的参数、方法的返回值，或者A调用B的静态方法

简单来说，就是 A 类中, 出现了 B 类，可以简单理解为use-a

一般用一条指向被依赖事物的虚线表示



### 关联（Association）

通常情况下，关联关系是通过类的成员变量来实现的。

代码示例：

**---------------------------------------------------------------------------------------------------------------------------------**

//公司类

public class Company {

//一个公司雇员，公司与雇员之间就是一种关联关系。

private Employee emp ;

}

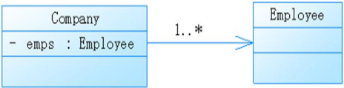
public class Employee {

}

**---------------------------------------------------------------------------------------------------------------------------------**

雇员作为公司的属性，不同于上面的依赖。依赖的话，雇员的改变会影响公司，显然不是。在这里雇员仅仅作为公司的一个属性而存在。因此Employee与Company之间是关联关系。关联关系还可以细分为聚合和组合两种。

一般用实线连接有关联的不同的两个类



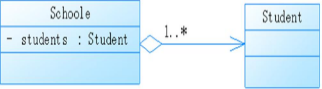
每一个Company可以有很多个Employee，但一个Employee只单独属于一个Company

#### 聚合（Aggregation）

聚合是一种特殊的关联。它描述了“has a”关系，表示整体对象拥有部分对象。

聚合是较强的关联关系,强调的是整体与部分的关系，聚合关系作为关联关系的一个分支，同样也是用成员变量的方式实现的

一般用带一个空心菱形（整体的一端-学校）的实线表示

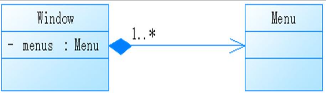


#### 组合（Composition）

组合是聚合的一种形式，他具有更强的拥有关系，强调整体与部分的生命周期是一致的。

例如Windows窗口和窗口上的菜单就是组合关系，如果窗口被销毁，那么窗口上的菜单也就不复存在了。

一般用带实心菱形(整体的一端)的实线来表示



#### 聚合和组合的区别

两者的区别在于，聚合是has-a关系，而组合是contain-a关系

在整体与部分关系方面，聚合要比组合弱一些

### 泛化（Generalization）

泛化就是父类和子类之间的关系，子类继承父类的所有结构和行为，在子类中可以增添新的结构和行为，也可以覆盖父类的行为

一般用一个带空心箭头的实线表示泛化关系



意为Tiger类继承自Animal类

### 实现（Realization）

实现关系指定两个实体之间的一个合约。换言之，一个实体定义一个合约，而另一个实体保证履行该合约。对类来说，就是一个类实现了一个接口。

一般用一条指向接口的虚线表示



# Java设计模式

## [耦合](https://www.baidu.com/s?wd=%E8%80%A6%E5%90%88&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9mvfYnvmsPAczuj6srjmL0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErHczPW6snjbsnHTznWfkPWb4" \t "https://zhidao.baidu.com/question/_blank)性：

也称块间联系。指软件系统结构中各模块间相互联系紧密程度的一种度量。 模块之间联系越紧密，其[耦合](https://www.baidu.com/s?wd=%E8%80%A6%E5%90%88&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9mvfYnvmsPAczuj6srjmL0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErHczPW6snjbsnHTznWfkPWb4" \t "https://zhidao.baidu.com/question/_blank)性就越强，模块的独立性则越差。模块间耦合 高低取决于模块间接口的复杂性、调用的方式及传递的信息。

## 面向接口编程

### 标识接口

在有些Java接口中并**没有声明任何方法**，这样的接口就叫做**标识接口**：

**interface** example {}

很明显，标识接口**对它的实现类没有任何语义上的要求**，它仅仅用来**充当一个标识的作用**，用来表明**实现它的类属于一个特定的类型**。这个标签类似于汽车的标志图标，每当人们看到一个汽车的标志图标时，就能知道这款汽车的品牌。在使用的时候会经常用**instanceof**来判断实例对象的类型**是否实现了一个给定的标识接口**。

**下面通过一个例子来详细说明标识接口的作用。例如要开发一款游戏，游戏里面有一个人物角色专门负责出去寻找有用的材料，假设这个人物只收集矿石和武器，而不会收集垃圾，下例通过标识接口来实现这个功能。**

**import** java.util.ArrayList;

**interface** Stuff{}

//矿石

**interface** Ore **extends** Stuff{}

//武器

**interface** Weapon **extends** Stuff{}

//垃圾

**interface** Rubbish **extends** Stuff{}

//金矿

**class** Gold **implements** Ore{

**public** String toString(){

**return** "Gold";

}

}

//铜矿

**class** Copper **implements** Ore{

**public** String toString(){

**return** "Copper";

}

}

//枪

**class** Gun **implements** Weapon{

**public** String toString(){

**return** "Gun";

}

}

//榴弹

**class** Grenade **implements** Weapon{

**public** String toString(){

**return** "Grenade";

}

}

**class** Stone **implements** Rubbish{

**public** String toString(){

**return** "Stone";

}

}

**public** **class** Test{

**public** **static** ArrayList<Stuff> collectStuff(Stuff[] s){

ArrayList<Stuff> al=**new** ArrayList<Stuff>();

**for**(**int** i=0;i<s.length;i++){

**if**(!(s[i] **instanceof** Rubbish))

al.add(s[i]);

}

**return** al;

}

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

Stuff[] s={**new** Gold(),**new** Copper(),**new** Gun(),**new** Grenade(),**new** Stone()};

ArrayList<Stuff> al=*collectStuff*(s);

System.***out***.println("The usefull Stuff collected is:");

**for**(**int** i=0;i<al.size();i++)

System.***out***.println(al.get(i));

}

}

 在上例中，设计了三个接口：Ore、Weapon和Rubbish分别代表矿石、武器和垃圾，只要是实现Ore或Weapon的类，游戏中的角色都会认为这是有用的材料，例如：Gold、Copper、Gun、Grenade，因此会收集，只要是实现Rubbish的类，都会被认为是无用的东西，例如Stone，因此不会被收集。

如此一来，这个角色就不会收集Rubbish及其子类的对象了。

### 工厂模式

上面的程序示例中定义了一个Printer类，继承了Output接口，那么下面问题来了：

现在有这样一个场景，假设程序中有一个Conputer类需要组合一个输出设备，那么有两种方式，一种是让Computer组合一个Printer类，一种是组合一个Output类，如果说组合了Printer类，那么将来如果想要对Printer进行优化，就需要更改Computer的源代码，如果只有一个Computer类组合了Printer还好，但要是Printer被大量的类组合，优化Printer将是一个巨大的工程，因此组合Output接口是一个更好的选择，这样就可以将Computer类与Printer类完全隔离，

Computer类的定义如下：

public class Computer **{**

private Output out**;**

public Computer**(**Output out**)** **{**

//out是指向实现类对象的一个引用

**this.**out **=** out**;**

**}**

public void keyIn**(**String msg**)** **{**

out**.**getData**(**msg**);**

**}**

public void print**()** **{**

out**.**out**();**

**}**

**}**

上面的Computer类已经完全与Printer类分离，只是与接口耦合，并且Computer不再负责创建Output对象，可以再提供一个Output工厂来负责生产Output对象，

OuputFactory类代码实现如下：

public class OutputFactory **{**

public Output getOutput**()** **{**

**return** **new** Printer**();**

//定义getOutput()方法，直接返回一个新的Printer对象

**}**

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

OutputFactory of **=** **new** OutputFactory**();**

Computer c **=** **new** Computer**(**of**.**getOutput**());**

c**.**keyIn**(**"疯狂Java讲义"**);**

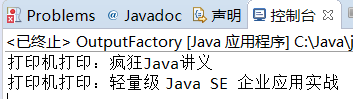
c**.**keyIn**(**"轻量级 Java SE 企业应用实战"**);**

c**.**print**();**

**}**

**}**

运行结果：



这个实例很好的利用了接口的特点，如果以后需要优化Printer()类，只需让优化类实现Output接口，并把OutputFactory类中的getOutput()方法稍作更改即可。

### 命令模式

再考虑另外一种场景，某个方法需要完成一种特定的行为，但这个行为的具体实现无法确定，必须等到执行该方法时才可以确定。再具体一点就是：假设某个方法需要遍历某个数组的数组元素，但无法确定在遍历数组时如何处理这些元素，需要在调用该方法时指定具体的处理行为。

说得直白一点，就有点类似于我们需要把“处理行为”作为参数传递给方法。

**注：**在Ruby这种语言中，确实有这种操作，允许一个代码块作为参数；现在Java 8中已经增加了Lambda表达式，通过该表达式可以传入代码块作为参数。

示例：

package hql**;**

interface Command **{**

void process**(**int**[]** target**);**

//该方法用来封装”处理行为“

**}**

//通过Command接口，来实现ProcessArray类和具体”处理行为“的分离

//使用Command接口代表了对数组的处理行为，但Command也没有提供真正的处理，

//只有等到需要调用ProcessArray对象的process()方法时，才真正传入

//一个Command对象，才确定数组的处理行为

class ProcessArray **{**

public void process**(**int**[]** target**,** Command cmd**)** **{**

cmd**.**process**(**target**);**

**}**

**}**

class PrintCommand **implements** Command **{**

public void process**(**int**[]** target**)** **{**

**for(**int tmp**:** target**)**

System**.**out**.**println**(**"迭代输出目标数组的元素：" **+** tmp**);**

**}**

**}**

class AddCommand **implements** Command **{**

public void process**(**int**[]** target**)** **{**

int sum **=** 0**;**

**for(**int tmp**:** target**)**

sum **+=** tmp**;**

System**.**out**.**println**(**"数组元素的总和是：" **+** sum**);**

**}**

**}**

public class CommandTest **{**

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

ProcessArray pa **=** **new** ProcessArray**();**

int**[]** target **=** **{**3**,** **-**4**,** 6**,** 4**};**

pa**.**process**(**target**,** **new** PrintCommand**());**

System**.**out**.**println**(**"---------我是一条正经的分割线---------"**);**

pa**.**process**(**target**,** **new** AddCommand**());**

**}**

**}**

我只想说，接口是真的强大。。。

### Adapter模式

**Adapter设计模式，即适配器设计模式，相当于接口中的转接头**

示例：

**interface** MobileStorage {

**void** read();

**void** write();

}

**class** FlashDisk **implements** MobileStorage {

**public** **void** read() {

System.out.println("I can read");

}

**public** **void** write() {

System.out.println("I can write");

}

}

**class** MP3 **implements** MobileStorage {

**public** **void** read() {

System.out.println("I can read");

}

**public** **void** write() {

System.out.println("I can write");

}

}

**class** MobileDisk **implements** MobileStorage {

**public** **void** read() {

System.out.println("I can read");

}

**public** **void** write() {

System.out.println("I can write");

}

}

**class** SuperStorage {

**public** **void** rd() {

System.out.println("reading");

}

**public** **void** wt() {

System.out.println("writing");

}

}

**class** SuperStorageAdapter **implements** MobileStorage {

**private** SuperStorage superStorage;

**public** **void** setSuperStorage(SuperStorage superStorage) {

**this**.superStorage = superStorage;

}

**public** **void** read() {

superStorage.rd();

}

**public** **void** write() {

superStorage.wt();

}

}

**class** Computer {

**private** MobileStorage usbDriver;

**public** **void** setUsbdriver(MobileStorage usbDriver) {

**this**.usbDriver = usbDriver;

}

**public** **void** readData() {

usbDriver.read();

}

**public** **void** writeData() {

usbDriver.write();

}

}

**public** **class** Test {

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

Computer pComputer = **new** Computer();

SuperStorage storage = **new** SuperStorage();

SuperStorageAdapter superStorageAdapter = **new** SuperStorageAdapter();

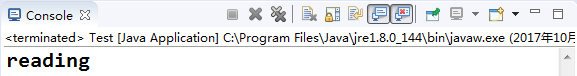
superStorageAdapter.setSuperStorage(storage);

pComputer.setUsbdriver(superStorageAdapter);

pComputer.readData();

}

}



**黄色部分为关键代码，为后来新增加的设备SuperStorage设计了一个转接头adapter，从而可以连接到Computer上**

### 策略模式

其实就是使用抽象类或者借口定义一个抽象方法，然后写出多个实现类来实现该方法

最后将最高层的抽象类或者接口类型的引用作为一个类的成员变量，根据该类对象的构造函数中传入的参数，来决定具体使用哪一个实现类的方法。

看代码吧：

abstract class Strategy {

public abstract void method();

}

class StrategyImplA extends Strategy {

public void method() {

System.*out*.println("这是第一个实现");

}

}

class StrategyImplB extends Strategy {

public void method() {

System.*out*.println("这是第二个实现");

}

}

class StrategyImplC extends Strategy {

public void method() {

System.*out*.println("这是第三个实现");

}

}

class Context {

Strategy stra;

public Context(Strategy stra) {

this.stra = stra;

}

public void doMethod() {

stra.method();

}

}

public class Test {

public static void main(String[] args) {

Context ctx = new Context(new StrategyImplA());

ctx.doMethod();

ctx = new Context(new StrategyImplB());

ctx.doMethod();

ctx = new Context(new StrategyImplC());

ctx.doMethod();

}

}

我自己的代码：

# Java的输入与输出

我要在这里说一下制表符，一个中文字符所占的位置相当于两个英文字符

## File类

File类是专门用来操作文件和目录的，它并不能访问文件本身的内容，输入/输出流可以做到

讲英文输入改为全角输入，可以解决此问题

### 1、访问文件和目录

直接看例子吧，都是一些方法，熟练掌握即可，就像记忆Linux里面那些命令一样，他们之间有着相通之处

代码：

**---------------------------------------------------------------------------------------------------------------------------------**

import java.io.\*;

public class FileTest {

public static void main(String[] args) throws IOException {

// 以当前路径来创建一个File对象

File file = new File("C:\\Users\\Administrator\\Documents\\测试文档\\文件操作测试目录");

// 直接获取文件名，输出一点

//这返回的哪里是文件名啊，分明就是文件所在路径的最后一个目录的名字

System.*out*.println(file.getName());

// 获取相对路径的父路径可能出错，下面代码输出null

System.*out*.println(file.getParent());

// 获取绝对路径

System.*out*.println(file.getAbsoluteFile());

// 获取上一级路径

System.*out*.println(file.getAbsoluteFile().getParent());

// 在当前路径下创建一个临时文件

File tmpFile = File.*createTempFile*("aaa", ".heqile", file);

// 指定当JVM退出时删除该文件

//tmpFile.deleteOnExit();

// 以系统当前时间作为新文件名来创建新文件

File newFile = new File(System.*currentTimeMillis*() + "");

System.*out*.println("newFile对象是否存在：" + newFile.exists());

// 以指定newFile对象来创建一个文件

// System.out.println(newFile.createNewFile());

// 以newFile对象来创建一个目录，因为newFile已经存在，

// 所以下面方法返回false，即无法创建该目录

System.*out*.println(newFile.mkdir());

// 使用list()方法来列出当前路径下的所有文件和路径

String[] fileList = file.list();

System.*out*.println("====当前路径下所有文件和路径如下====");

for (String fileName : fileList)

System.*out*.println(fileName);

// listRoots()静态方法列出所有的磁盘根路径。

File[] roots = File.*listRoots*();

System.*out*.println("====系统所有根路径如下====");

for (File root : roots)

System.*out*.println(root);

}

}

**---------------------------------------------------------------------------------------------------------------------------------**

上面的代码到了很多常用的文件操作方法，有很多地方我也没有看懂，

但是有一个方法我想说些一下，就是mkdir()方法和mkdirs()方法

他们两个虽然只有一个s的差别，但作用还是很不同的:  
 如下代码所示：  
  import java.io.\*;

public class Test {

public static void main(String[] args) {

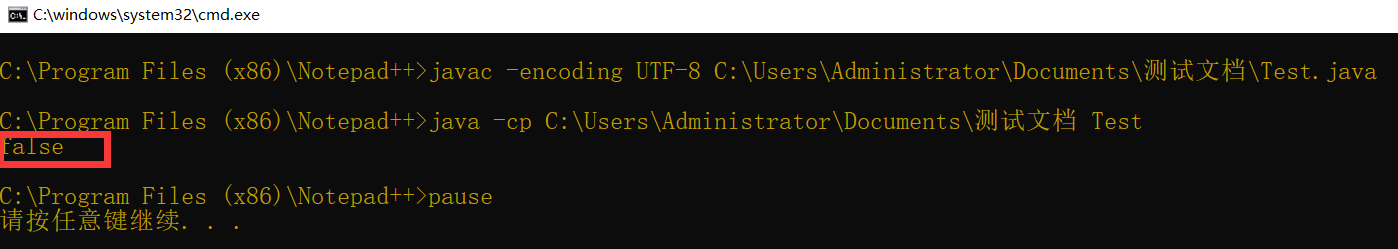
File file = new File("D:\\test\\test2");

System.out.println(file.mkdir());

}

}

使用mkdir()方法时，必须要保证当前对象的**最后一级目录之前的父目录都是存在的，否则是创建不出来test2目录的**



输出false，创建失败，因为D盘下并不存在test这个目录

**而mkdirs()相比之下啊就要比mkdir()方法更强大一些，它可以按照当前对象的抽象路径完全建立目录，即使父目录是不存在的，它也能创建出来**

**这是使用mkdirs()方法建立出来的：**

**import java.io.\*;**

**public class Test {**

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

**File file = new File("D:\\test0\\test1\\test2\\test3");**

**System.out.println(file.mkdirs());**

**}**

**}**

这个方法我也得说一下File newFile = new File(System.currentTimeMillis() + "");

尽管我现在也不是很清楚他的具体作用，但是我还是要说。。。。。

import java.io.\*;

public class Test {

public static void main(String[] args) {

File newFile = new File("C:\\Users\\Administrator\\Documents\\测试文档\\文件操作测试目录"

+ System.currentTimeMillis() + "");

}

}

看结果：  
 

我每执行一次这个代码，他就会在我所给的路径的最后一个目录的父目录中，士大夫创建一个前缀后最后一个目录一样后面跟一大堆数字的一个目录，后面跟的那些数字，是用当前时间减去1970-01-01 00:00:00的到的毫秒数，其实就跟时间戳一样

### 2、文件过滤器

先看代码：

**---------------------------------------------------------------------------------------------------------------------------------**

import java.io.\*;

public class FilenameFilterTest {

public static void main(String[] args) {

File file = new File("C:\\Users\\Administrator\\Documents\\测试文档");

// 使用Lambda表达式（目标类型为FilenameFilter）实现文件过滤器。

// 如果文件名以.java结尾，或者文件对应一个路径，返回true

String[] nameList = file.list((dir, name) -> name.endsWith(".java")

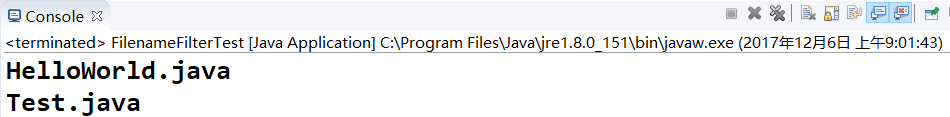
|| new File(name).isDirectory());

for(String name : nameList)

System.out.println(name);

}

}



**---------------------------------------------------------------------------------------------------------------------------------**

**List()方法可以接收一个****FilenameFilter参数，FilenameFilter是一个函数式接口，它里面有一个accept()抽象方法，该方法将依此对指定File的所有文件和子目录进行迭代**

**因此，使用该方法，就要实现accept(File dir, String name)抽象方法，上面使用的正是Lambda表达式，我们只需要给出规则就行，告诉accept什么样的文件应该被输出**

## 2、字节流和字符流

### InputStream和Reader

不管啦，直接示范一下程序读取自身的效果：  
**---------------------------------------------------------------------------------------------------------------------------------**

import java.io.\*;

public class FileInputStreamTest {

public static void main(String[] args) throws IOException {

// 创建字节输入流

//要给出路径，不然会报错，说他找不到指定文件

FileInputStream fis = new FileInputStream(

"D:\\eclipse\_workspace\\Test\\src\\FileInputStreamTest.java");

// 创建一个长度为1024的“竹筒”

byte[] bbuf = new byte[1024];

// 用于保存实际读取的字节数

int hasRead = 0;

// 使用循环来重复“取水”过程

while ((hasRead = fis.read(bbuf)) > 0 )

// 取出“竹筒”中水滴（字节），将字节数组转换成字符串输入！

System.*out*.print(new String(bbuf , 0 , hasRead ));

// 关闭文件输入流，放在finally块里更安全

fis.close();

}

}

**---------------------------------------------------------------------------------------------------------------------------------**



对上面的程序进行分析：

首先，要创建一个FileInputStream 对象，它的含参构造器中需要一个参数，这个参数就是数据源的绝对路径（要包含数据源即文件名）

然后，需要声明一个byte（字节）数组，以及一个int型变量，用来保存实际读取到的字符数量，最后在输出部分使用String的含参构造器，将其组装成字符进行输出

### OutputStream和Writer

说完了读取文件，现在来说输出到文件

先上代码：

**---------------------------------------------------------------------------------------------------------------------------------**  
import java.io.\*;

public class FileWriterTest

{

public static void main(String[] args)

{

try(

FileWriter fw = new FileWriter("C:\\Users\\Administrator\\Desktop\\poem.txt"))

{

String teString = "何其乐";

fw.write(teString + "锦瑟 - 李商隐\r\n" + "锦瑟无端五十弦，一弦一柱思华年。\r\n");

fw.write("庄生晓梦迷蝴蝶，望帝春心托杜鹃。\r\n");

fw.write("沧海月明珠有泪，蓝田日暖玉生烟。\r\n");

fw.write("此情可待成追忆，只是当时已惘然。\r\n");

}

catch (IOException ioe)

{

ioe.printStackTrace();

}

}

}

**---------------------------------------------------------------------------------------------------------------------------------**

**分析代码：**

**首先，需要给一个**FileWriter 对象，他的含参构造器也需要一个参数，就是输出文件的具体路径，（也可以不给路径，默认是本Java文件所在目录的父目录）这个路径必须得是存在的，除了文件名,意思就是输出文件可以是不存在的，但是他所在的路径必须是已经存在的，不然是无法运行的

然后调用上面创建出来的FileWriter对象的write()方法即可，wirte的参数就是一堆字符串，可以是字符串变量，也可以直接就是现成的字符

我突然想起了[常量池](#常量池)

还有就是，上面为什么使用try语句呢？因为这些流在使用完毕之后，都是要close掉的，try语句又正好有自动关闭资源的功能，try语句执行完之后，流也就被自动关闭了

## 输入输出流体系

### 处理流的用法

判段处理流的方法也是非常简单的，只要看它构造器的参数是什么就可以了，只要不是一个物理节点（也就是一个绝对路径，文件名什么的），那么他就是一个处理流，处理流的构造器参数一般是一个已经存在的流，就像下面的代码，是一个FileOutputStream对象

看代码啦： PrintStream处理流

**---------------------------------------------------------------------------------------------------------------------------------**

import java.io.\*;

public class PrintStreamTest

{

public static void main(String[] args)

{

try(

FileOutputStream fos = new FileOutputStream("test.txt");

PrintStream ps = new PrintStream(fos))

{

// 使用PrintStream执行输出

ps.println("普通字符串");

// 直接使用PrintStream输出对象

ps.println(new PrintStreamTest());

}

catch (IOException ioe)

{

ioe.printStackTrace();

}

}

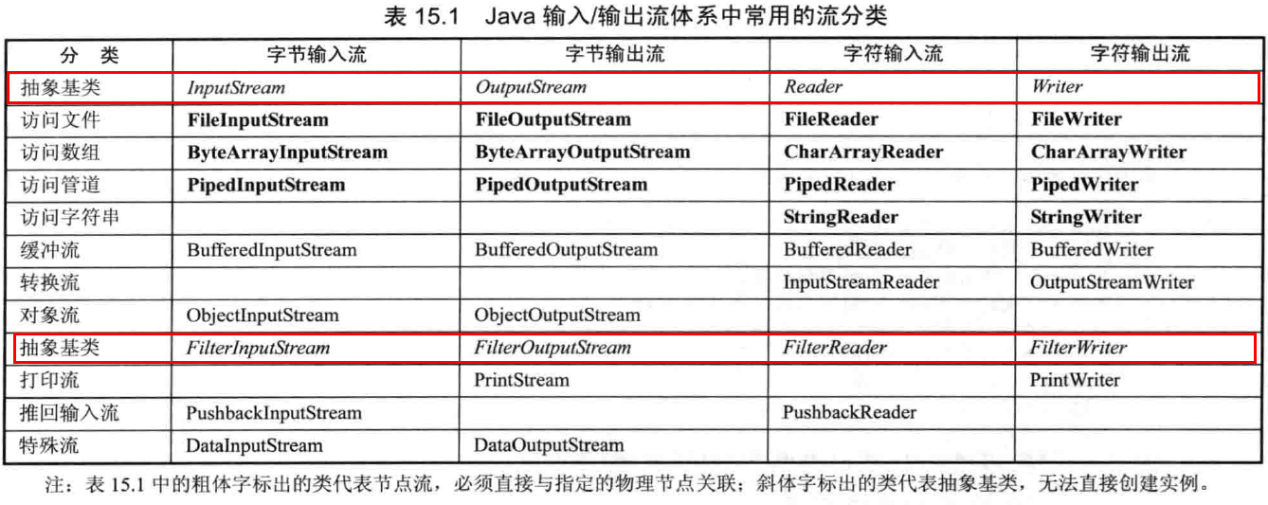
}

**---------------------------------------------------------------------------------------------------------------------------------**

的确是比上面的字节流字符流的输入输出简单了很多啊，先创建一个FileOutputStream 对象，然后在使用PrintStream对其进行包装即可，就可以直接使用println()方法来输出到文件了

而且我们只要把处理流关掉就行了，当处理流被关掉之后，他包装的节点流或者字符流也会自动关闭

### 2、输入/输出流体系



书上说了，哈哈哈（妈的，突然想笑），书上说字节流的功能要比字符流强大，因为计算机中的所有东西都用二进制存储的，就是一个一个的bit，构成了bit流，所以说他是根，但是操作起来比较复杂，还需要使用特定的方式来将这些字节流转换成字符，这肯定会使编程变得复杂，所以有一个不成文的规定，如果输入输出的是文本内容，那我们就用字符流，如果是二进制内容，那我们就用字节流来进行处理

下面，我们来示范一下，把字符串作为物理节点的字符输入/输出流的方法：  
**---------------------------------------------------------------------------------------------------------------------------------**

import java.io.\*;

public class StringNodeTest

{

public static void main(String[] args)

{

String src = "从明天起，做一个幸福的人\n"

+ "喂马，劈柴，周游世界\n"

+ "从明天起，关心粮食和蔬菜\n"

+ "我有一所房子，面朝大海，春暖花开\n"

+ "从明天起，和每一个亲人通信\n"

+ "告诉他们我的幸福\n";

char[] buffer = new char[32];

int hasRead = 0;

try(

StringReader sr = new StringReader(src))

{

// 采用循环读取的访问读取字符串

while((hasRead = sr.read(buffer)) > 0)

{

System.*out*.print(new String(buffer ,0 , hasRead));

}

}

catch (IOException ioe)

{

ioe.printStackTrace();

}

try(

// 创建StringWriter时，实际上以一个StringBuffer作为输出节点

// 下面指定的20就是StringBuffer的初始长度

StringWriter sw = new StringWriter())

{

// 调用StringWriter的方法执行输出

sw.write("有一个美丽的新世界，\n");

sw.write("她在远方等我,\n");

sw.write("哪里有天真的孩子，\n");

sw.write("还有姑娘的酒窝\n");

System.*out*.println("----下面是sw的字符串节点里的内容----");

// 使用toString()方法返回StringWriter的字符串节点的内容

System.*out*.println(sw.toString());

}

catch (IOException ex)

{

ex.printStackTrace();

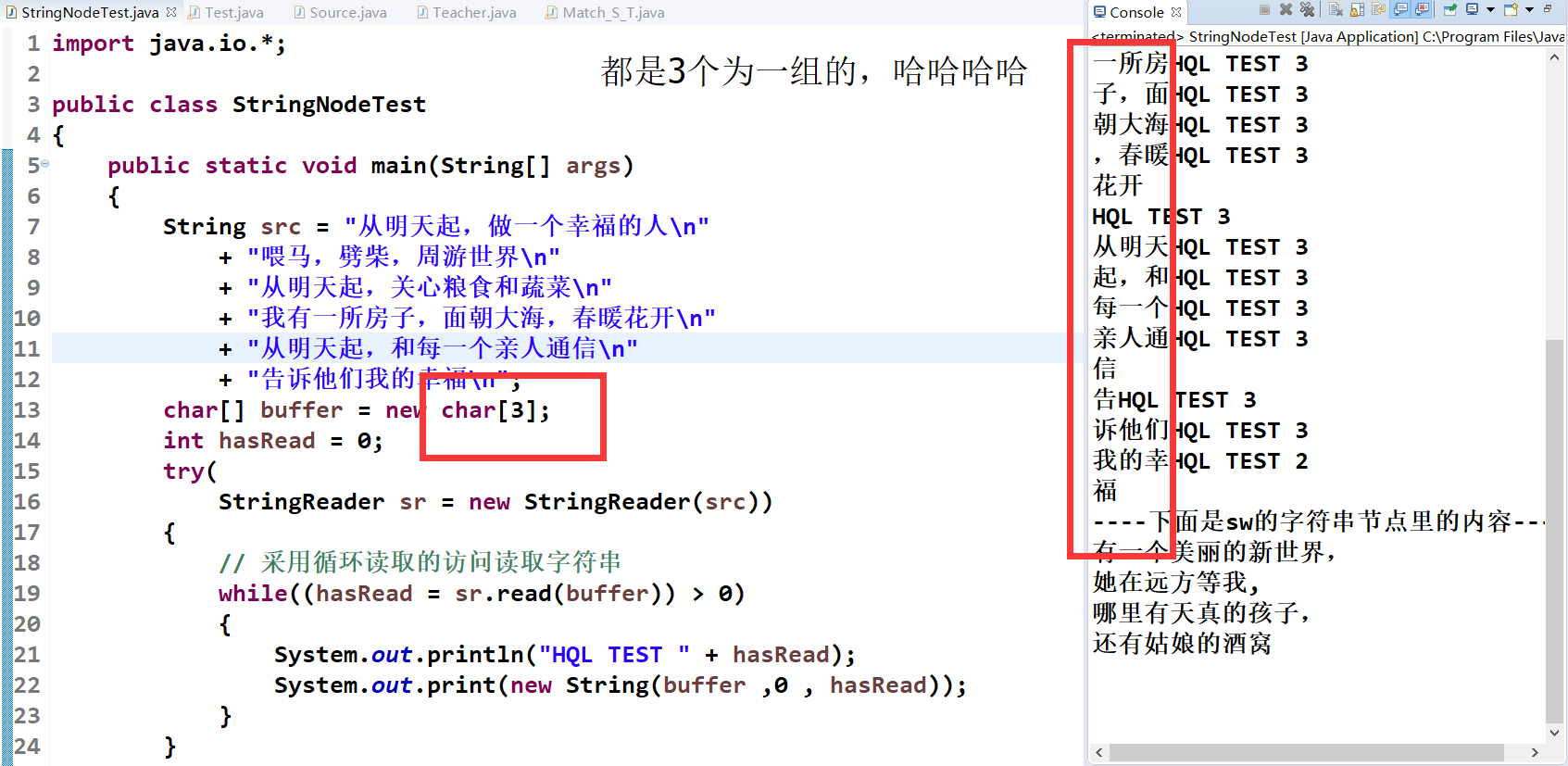
}

}

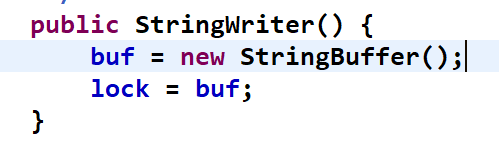
}

**---------------------------------------------------------------------------------------------------------------------------------**

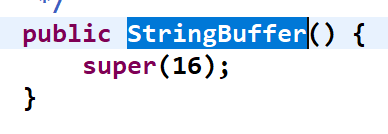
**仔细一看，其实和字节/字符流是很相似的，只不过物理节点由文件换成了数组，**

**而且，我们还可以通过更改buff数组的长度，来控制每次输出多长的字符串，如下图所示：**

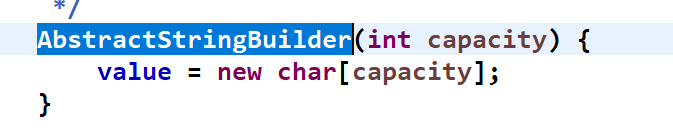
**创建StringWriter时，实际上以一个StringBuffer作为输出节点，因为String是不可变的对象，所以它新创建了一个StringBuffer对象这是StringWriter的无参构造器，返回一个StringBuffer对象**



**我今天才知道原来StringBuffer是一个类，它的无参构造器是这个样子的**

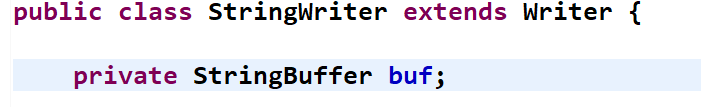


它使用了父类AbstractStringBuilder**的含参构造器**



看吧，最后给他的就是一个字符数组，capacity是容量/体积的意思，就是这个字符数组的长度

现在我们知道了，StringWriter对象默认的StringBuffer对象是一个长度为16的字符数组，就是说，初始的StringBuffer容量是16，不过他是可以追加的，可以变得很大很大，哈哈哈哈哈



我也不知道，为什么StringWriter要使用StringBuffer来存放字符串，而不是String

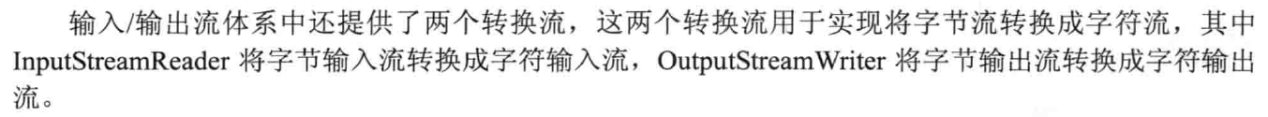
在万能的互联网上稍微搜索了一下，我找到了正确答案，为什么要用StringBuffer呢？？

哈哈哈，因为，再追加字符串这方面，StringBuffer要比String强啊，

String对象追加字符串，会另外开辟一块内存空间，而StringBuffer对象在追加字符串的时候，是在原来的基础上新增空间来实现的，因为StringWriter就是用来往字符串中追加字符的，所以采用StringBuffer是理所当然的啊

### 3、转换流

不想写了，偷个懒，直接截一张图



没什么好说的,直接看代码吧>>>>>>>>

**---------------------------------------------------------------------------------------------------------------------------------**

import java.io.\*;

public class KeyinTest

{

public static void main(String[] args)

{

try(

// 将Sytem.in对象转换成Reader对象

//System类里面有一个静态常量in，类型是InputStream

//InputStreamReader这个构造器以System.in作为参数

//直接返回InputStreamReader对象

InputStreamReader reader = new InputStreamReader(System.*in*);

//然后又将Reader对象包装成了BufferedReader

//为什么要这样呢？目的是为了使用BufferedReader的readLine()方法

BufferedReader br = new BufferedReader(reader))

{

String line = null;

// 采用循环方式来一行一行的读取

while ((line = br.readLine()) != null)

{

// 如果读取的字符串为"exit"，程序退出

if (line.equals("exit"))

{

System.*exit*(1);

}

// 打印读取的内容

System.*out*.println("输入内容为:\n" + line);

}

}

catch (IOException ioe)

{

ioe.printStackTrace();

}

}

}

**---------------------------------------------------------------------------------------------------------------------------------**

其实代码里的注释我已经补充的很详细了，之所以又用BufferedReader对InputStreamReader对象进行了包装，是为了一次读取一行，话说。。。。。。BufferedReader的对象如何读取输入我也不太清楚呢。。

看一下，看一下BufferedReader对象的方法，它里面有一个Reader()方法，

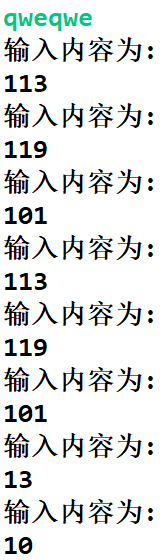
public int read() throws IOException {

return sd.read();

}

就是这样的，sd是，InputStreamReader的一个常量，是StreamDecoder的对象

private final StreamDecoder sd;

StreamSDecoder就是字节流向字符流转换的关键类，但是不知道为什么，当我想去看一下StreamDecoder的源代码的时候，却找不到，关联的源码压缩包中找不到这个类，嗨呀，好气啊。。。我又从网上下载了一个Java\_1.8 doc压缩包,也不知道好不好使，不管怎样，先看一下他输出的内容吧：  
 

输出了一大堆编码，也不知道是按照什么编码的，

虽然我没有StreamDecoder的源代码，但根据推测

sd.read()的返回值应该是int型的

算了，找不到源码，先这样吧

### 4、推回输入流

我要去做智慧树的期末考试题了，待会儿再回来。。。。。

做完了，找不到答案，全是蒙的，不过总分还行，80。。哈哈哈哈

看代码，重点看注释：

**---------------------------------------------------------------------------------------------------------------------------------**

import java.io.\*;

public class PushbackTest

{

public static void main(String[] args)

{

try(

// 创建一个PushbackReader对象，指定推回缓冲区的长度为64

PushbackReader pr = new PushbackReader(new FileReader(

"D:\\eclipse\_workspace\\Test\\src\\PushbackTest.java") , 64))

{

char[] buf = new char[32];

// 用以保存上次读取的字符串内容

String lastContent = "";

int hasRead = 0;

// 循环读取文件内容

//PushBackReader对象所调用的read方法，其实是实现的

//Reader的抽象方法

//read方法读取到的字符数量

//如果数据源中还有足够的字符，那么他每一次读取到的字符数量

//就是read方法中参数（一个字符数组）的容量

//在本例中，hasRead的值在一段时间内一直都是32

while ((hasRead = pr.read(buf)) > 0)

{

System.*out*.println("heqile" + hasRead);

// 将读取的内容转换成字符串

String content = new String(buf , 0 , hasRead);

int targetIndex = 0;

// 将上次读取的字符串和本次读取的字符串拼起来，

// 查看是否包含目标字符串, 如果包含目标字符串

//indexOf方法

//如果源字符串中包含目标字符串，那么

//就会返回目标字符串的第一个字符在源字符串

//中的相应位置

//如果目标字符串不存在，返回值是-1

if ((targetIndex = (lastContent + content)

.indexOf("new PushbackReader")) >= 0)

{

// 将本次内容和上次内容一起推回缓冲区

//使用toCharArrray方法，将字符串转换成了字符数组

pr.unread((lastContent + content).toCharArray());

// 重新定义一个长度为targetIndex的char数组

if(targetIndex > 32)

//targetIndex > 32，那么原来的buf数组就不够用了，

//因此需要再开辟一个符合要求的容量的数组

{

buf = new char[targetIndex];

}

//我突然明白了这个代码到底是怎么回事了

//请看图片[图片](D:\\include_heqile的云文档\\文档图片\\IMG_20171208_100108.jpg)

// 再次读取指定长度的内容（就是目标字符串之前的内容）

pr.read(buf , 0 , targetIndex);

// 打印读取的内容

System.*out*.print(new String(buf , 0 ,targetIndex));

System.*exit*(0);

}

else

{

// 打印上次读取的内容

System.*out*.print(lastContent);

// 将本次内容设为上次读取的内容

lastContent = content;

}

}

}

catch (IOException ioe)

{

ioe.printStackTrace();

}

}

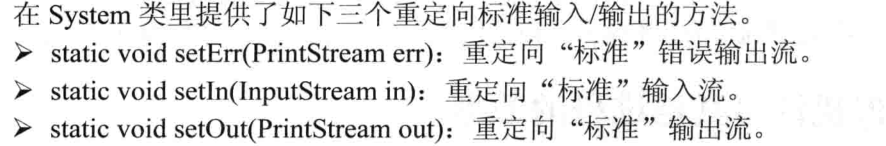
}

**---------------------------------------------------------------------------------------------------------------------------------**

这个代码，我之所以看了这么长时间，一方面是手机坏了，有点蓝瘦，另一方面是自己的思考方式有问题，就是那个推回操作，我搞了半天也搞不明白，最后终于弄懂了，之所以哈皮了这么长时间，仅仅是因为我脑海中的字符流的方向的问题，我上面的图片上画出来的字符流的流向是从右往左的，而Java讲义上的字符流都是从左往右流的，我就是卡在这个点上了，我开始是参考的后者的流向，然后在分析代码的时候，就把字符的流向整反了。。。。哎呀，我也不知道自己在说什么，反正就是跟空间，，方向什么的有关。。。。。

### 5、重定向标准输入/输出

这都是我的第三次写重定向了，总是没保存上。。。。



重定向说的就是System这个类里面的三个方法，正常情况下执行System.out.println()方法，会在屏幕上输出信息，而经过重定向之后System的标准输入输出就不是键盘和显示器了，而是自定义的Stream流水

比如下面这个代码：

将输出重定向到了一个名为out.txt的文本文件中

**---------------------------------------------------------------------------------------------------------------------------------**

import java.io.\*;

public class RedirectOut

{

public static void main(String[] args)

{

try(

// 一次性创建PrintStream输出流

PrintStream ps = new PrintStream(new FileOutputStream("out.txt")))

{

// 将标准输出重定向到ps输出流

System.*setOut*(ps);

// 向标准输出输出一个字符串

System.*out*.println("普通字符串");

// 向标准输出输出一个对象

System.*out*.println(new RedirectOut());

}

catch (IOException ex)

{

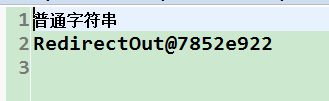
ex.printStackTrace();

}

}

}

out.txt文件的内容



**---------------------------------------------------------------------------------------------------------------------------------**

下面我们再来演示一下，如何重定向到标准输入

看代码：  
**---------------------------------------------------------------------------------------------------------------------------------**

import java.util.\*;

import java.io.\*;

public class RedirectIn

{

public static void main(String[] args)

{

try(

FileInputStream fis = new FileInputStream("C:\\Users\\Administrator\\Desktop\\input.txt"))

{

// 将标准输入重定向到fis输入流

System.*setIn*(fis);

// 使用System.in创建Scanner对象，用于获取标准输入

Scanner sc = new Scanner(System.*in*);

// 增加下面一行将只把回车作为分隔符

//useDelimiter这个方法可以让我们自定义输入的分隔符，当前

//分隔符是空格，也可以传一个参数 "\n"，用回车来作为分隔符

sc.useDelimiter(" ");

// 判断是否还有下一个输入项

while(sc.hasNext())

{

// 输出输入项

//注意：在读取输入时，只能接收String，如果需要int型的数据

//需使用Interger的静态方法parseInt()来将字符转换成对应的数字

System.*out*.println("键盘输入的内容是：" + Integer.*parseInt*(sc.next()));

}

}

catch (IOException ex)

{

ex.printStackTrace();

}

}

}

**---------------------------------------------------------------------------------------------------------------------------------**

**到目前为止，有了重定向输入和输出，基本上就可以实现数据的文件读入和输出了，哈哈哈哈哈**