# 环境变量配置

变量名：JAVA\_HOME

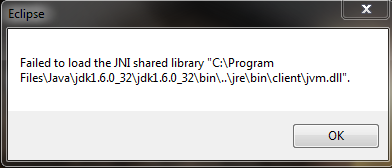
变量值：C:\Program Files\Java\jdk1.7.0\_51

变量名：Path

变量值：;%JAVA\_HOME%\bin;

说明：jdk可以安装在任何位置，不一定安装在C盘。修改了Java变量的值或者删除了Java变量，需要重启电脑才能生效。

# 启动eclipse遇到的问题



这个问题的解决方法：

打开eclipse时出现了Failed to load the JNI shared library，这个问题一般是eclipse和JDK位数不一样，那么如何判断eclipse和JDK位数不一样了，如何解决这个问题？首先查看两个软件的位数，然后再卸载一个，安装和另外一个位数一样的软件。

（1）首先，我们来查看JDK是多少位的，在搜索框中输入cmd，然后打开命令行窗口。

（2）在命令行中输入java -version ,查看JDK是多少位，一般显示64位的就是64为JDK，未显示的一般为32位的JDK。

（3）然后，我们来确定eclipse是多少位的。我们打开eclipse的安装目录，右击eclipse.ini文件以记事本的方式打开。

（4）查看--launcher.library下面的一行，如果显示的是win32.x86则表明eclipse是32位的，如果显示的win32.x86\_64，则是64位的。如果这个eclipse是32位的，JDK与eclipse不兼容，会出现Failed to load the JNI shared library这个问题。

（5）解决这种问题有两种，一是重新安装JDK，二是重新安装eclipse。因为安装JDK比较麻烦，所以这里我们选择第二种方法重新安装eclipse。

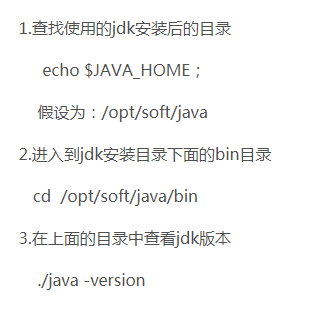
（6）在百度中搜索“eclipse”关键字，在搜索结果中，点击进入官网，进行下载，选择“Eclipse IDE for Java Developer”，然后选择相应的位数即可，我这里需要64位的eclipse，所以下载64位的。

（7）把下载下来的eclipse压缩包解压到你的软件盘下，进入eclipse根目录，点击“eclipse.exe”图标，等待一会儿就可以打开eclipse了。

# Java版本查看方法

命令：java –version

# linux系统查看java版本的方法



# Java命令

javac 类名.java //编译java文件

java 类名 //运行class文件

# jdk中的包

dt.jar是关于运行环境的类库,主要是swing的包

tools.jar是关于一些工具的类库

rt.jar包含了jdk的基础类库，也就是你在java doc里面看到的所有的类的class文件

# jar文件

1、JAR（Java Archive，Java 归档文件）是与平台无关的文件格式，它允许将许多文件组合成一个压缩文件。

2、JAR 文件格式以流行的 ZIP 文件格式为基础，与 ZIP 文件不同的是，JAR 文件不仅用于压缩和发布，而且还用于部署和封装库、组件和插件程序，并可被编译器和 JVM 这样的工具直接使用。在 JAR 中包含特殊的文件，如 manifests 和部署描述符，用来指示工具如何处理特定的 JAR文件。

3、使用范围：一个 JAR 文件可以用于

（1） 用于发布和使用类库。

（2）作为应用程序和扩展的构建单元。

（3）作为组件、applet 或者插件程序的部署单位。

（4）用于打包与组件相关联的辅助资源。

# jdk1.6特性

1、Desktop类和SystemTray类

2、使用JAXB2(Java Architecture for XML Binding)来实现对象与XML之间的映射

3、理解STAX

4、使用Compiler API

5、轻量级 Http Server API

6、插入式注解处理 API

7、使用Console开发控制台程序

8、对脚本语言的支持

9、Common Annotations

10、Java GUI界面的显示

11、嵌入式数据库Derby

12、Web服务元数据

13、Jtable的排序和过滤

14、更简单更强大的JAX-WS（Java API for XML Web Services）

# jdk1.7特性

1、switch中可以使用字符串。

2、泛型实例化类型自动推断。

3、语法上支持集合，而不一定是数组。

4、新增一些获取环境信息的工具方法。

5、boolean类型反转，空指针安全，参与位运算。

6、两个char之间的equal。

7、安全的加减乘除。

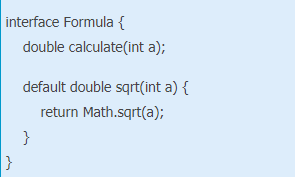
8、map集合支持并发请求。

# jdk1.8特性

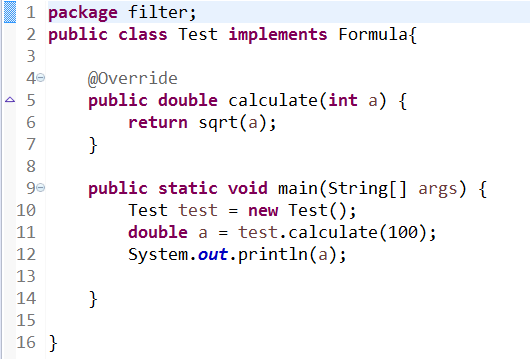
## 1、接口的默认方法

Java 8允许我们给接口添加一个非抽象的方法实现，只需要使用 default关键字即可，这个特征又叫做扩展方法。（在接口中可以添加扩展方法）

代码示例：



Formula接口在拥有calculate方法之外同时还定义了sqrt方法，实现了Formula接口的子类只需要实现一个calculate方法，默认方法sqrt将在子类上可以直接使用。



说明：在Java中只有单继承，如果要让一个类赋予新的特性，通常是使用接口来实现。

## 2、Lambda 表达式

Lambda 是一个匿名函数,我们可以把 Lambda 表达式理解为是一段可以传递的代码(将代码像数据一样进行传递)。可以写出更简洁、更灵活的代码。作为一种更紧凑的代码风格，使 Java的语言表达能力得到了提升。

Lambda表达式的写法：

Comparator<Integer> comparator = (x, y) -> Integer.compare(x, y);

Runnable runnable = () -> System.out.println("Lambda表达式");

我们会发现Lambda表达式的写法更加的简洁、灵活。它只关心参数和执行的功能（具体需要干什么，比如->后的Integer.compare(x, y)）。

Lambda表达式语法：

Lambda 表达式在Java 语言中引入了一个新的语法元 素和操作符。这个操作符为 “->” , 该操作符被称为 Lambda 操作符或剪头操作符。它将 Lambda 分为两个部分：

（1）左侧：指定了 Lambda 表达式需要的所有参数。

（2）右侧：指定了 Lambda 体,即 Lambda 表达式要执行的功能。

可以是如下的写法：

(params) -> expression

(params) -> statement

(params) -> { statements }

语法格式一：无参，无返回值，Lambda 体只需一条语句。

Runnable runnable = () -> System.out.println("lamda表达式");

语法格式二：Lambda 只需要一个参数

Consumer<String> consumer=(x)->System.out.println(x);

语法格式三：Lambda 只需要一个参数时,参数的小括号可以省略

Consumer<String> consumer=x->System.out.println(x);

语法格式四：Lambda 需要两个参数

Comparator<Integer> comparator = (x, y) -> Integer.compare(x, y);

语法格式五：当 Lambda 体只有一条语句时，return 与大括号可以省略

BinaryOperator<Integer> binaryOperator=(x,y)->(x+y);

语法格式六：数据类型可以省略,因为可由编译器推断得出，称为“类型推断"。

BinaryOperator<Integer> bo=(x,y)->{

System.out.println("Lambda");

return x+y;};

类型推断

上述 Lambda 表达式中的参数类型都是由编译器推断得出的。Lambda 表达式中无需指定类型，程序依然可以编译，这是因为 javac 根据程序的上下文，在后台 推断出了参数的类型。Lambda 表达式的类型依赖于上 下文环境，是由编译器推断出来的。这就是所谓的 “类型推断”。

### 示例

例1、用lambda表达式实现Runnable

new Thread( () -> System.out.println("In Java8, Lambda expression rocks !!") ).start();

例2：在[使用lambda](http://javarevisited.blogspot.sg/2012/03/how-to-loop-arraylist-in-java-code.html)或不使用lambda表达式的情况下迭代列表。你可以看到列表现在有了一个 forEach()  方法，它可以迭代所有对象，并将你的lambda代码应用在其中。

List features = Arrays.asList("Lambdas", "Default Method", "Stream API", "Date and Time API");

features.forEach(n -> System.out.println(n));

// 使用Java 8的方法引用更方便，方法引用由::双冒号操作符标示，

// 看起来像C++的作用域解析运算符

features.forEach(System.out::println);

说明：[列表循环](http://javarevisited.blogspot.sg/2012/10/jstl-foreach-tag-example-in-jsp-looping.html)的最后一个例子展示了如何在Java 8中使用方法引用（method reference）。你可以看到C++里面的双冒号、范围解析操作符现在在Java 8中用来表示方法引用。

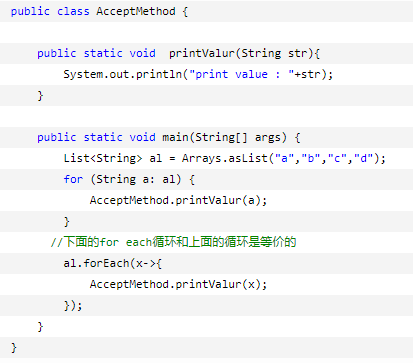
## 3、函数式接口

函数式接口，@FunctionalInterface，简称FI，简单的说，FI就是指仅含有一个抽象方法的接口，以@Functionalnterface标注，注意，这里的抽象方法指的是该接口自己特有的抽象方法，而不包含它从其上级继承过来的抽象方法。

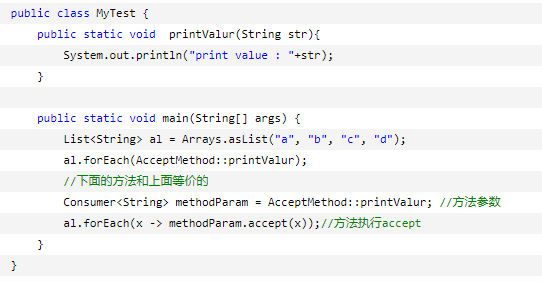
java8中双冒号的用法：把方法当做参数传到stream内部，使stream的每个元素都传入到该方法里面执行一下。

示例：

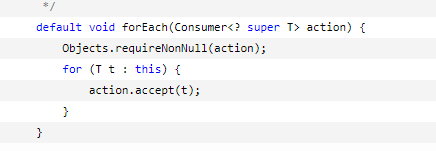
以前的代码一般是如此的：



现在JDK双冒号是：



在JDK8中，接口Iterable中默认实现了forEach方法，调用了 JDK8中增加的接口Consumer内的accept方法，执行传入的方法参数。JDK源码如下



## 4、方法与构造函数引用

## 5、Lambda 作用域

## 6、访问局部变量

## 7、访问对象字段与静态变量

## 8、访问接口的默认方法

## 9、Date API

Java 8 在包java.time下包含了一组全新的时间日期API。新的日期API和开源的Joda-Time库差不多，但又不完全一样。

### （1）Clock时钟

Clock类提供了访问当前日期和时间的方法，Clock是时区敏感的，可以用来取代 System.currentTimeMillis() 来获取当前的微秒数。某一个特定的时间点也可以使用Instant类来表示，Instant类也可以用来创建老的java.util.Date对象。

代码示例：

Clock clock = Clock.systemDefaultZone();

long mill = clock.millis();

System.out.println(mill);

Instant instant = clock.instant();

Date date = Date.from(instant);

System.out.println(date);

### （2）Timezones 时区

在新API中时区使用ZoneId来表示。时区可以很方便的使用静态方法of来获取到。时区定义了到UTS时间的时间差，在Instant时间点对象到本地日期对象之间转换的时候是极其重要的。

代码示例：

System.out.println(ZoneId.getAvailableZoneIds());  
// prints all available timezone ids

ZoneId zone1 = ZoneId.of("Europe/Berlin");  
ZoneId zone2 = ZoneId.of("Brazil/East");  
System.out.println(zone1.getRules());  
System.out.println(zone2.getRules());

// ZoneRules[currentStandardOffset=+01:00]  
// ZoneRules[currentStandardOffset=-03:00]

### （3）LocalTime本地时间

LocalTime 定义了一个没有时区信息的时间，例如 晚上10点，或者 17:30:15。下面的例子使用前面代码创建的时区创建了两个本地时间。

代码示例：

LocalTime now1 = LocalTime.now(zone1);  
LocalTime now2 = LocalTime.now(zone2);

System.out.println(now1.isBefore(now2));  // false

long hoursBetween = ChronoUnit.HOURS.between(now1, now2);  
long minutesBetween = ChronoUnit.MINUTES.between(now1, now2);

System.out.println(hoursBetween);       // -3  
System.out.println(minutesBetween);     // -239

北京时区：Etc/GMT+8

### （4）LocalDate 本地日期

## 10、Annotation 注解

# jdk1.9新特性

1、Jigsaw模块化项目

2、简化进程API

3、轻量级JSON API

4、钱和货币的API

5、改善锁的竞争机制

6、代码分段缓存

7、只能java编译

8、http2.0客户端

9、kulla计划

# 理解面向对象

1、Java语言是纯碎的面向对象的程序设计语言，这主要表现为Java完全支持面向对象的三种基本特征：封装、继承、多态。Java语言完全以对象为中心，Java语言的最小程序单位是类，整个Java程序由一个一个的类组成。

2、Java完全支持使用对象、类、继承、封装、消息等基本概念来进行程序设计，允许从现实世界中客观存在的事物（即对象）出发来构造。软件系统，在系统构造中尽可能运用人类的自然思维方式，实际上这些优势是所有面向对象编程语言的共同特征，面向对象的方式实际上由OOA(面向对象分析)、OOD（面向对象设计）、OOP（面向对象编程）三个部分有机组成，其中OOA、OOD的结构需要使用一种方式来描述并记录，目前统一采用UML（统一建模语言）来描述并记录OOA和OOD的结果。

3、采用面向对象方式开发的软件系统，其最小的程序单元是类，这些类可以生成系统中的多个对象，而这些对象则直接映像成客观世界的各种事物。

4、面向对象的软件系统由多个类组成，类代表了客观世界中具有某种特征的一类事物，这类事物往往有一些内部的状态数据，当然程序没有必要记录该事物所有的状态数据，程序只要记录业务关心的状态数据即可。

5、面向对象的语言不仅使用类来封装一类事物的内部状态数据，而且类会提供操作这些状态数据的方法，还会为这类事物的行为特征提供相应的实现，这种实现也是方法，因此可以得到如下基本公式：

状态数据+方法=类定义，从这个等式来看，面向对象比面向过程的编程粒度要大，面向对象的程序单位是类，面向过程的程序单位是函数（相当于方法），因此面向对象比面向过程更简单、易用。

6、抽象也是面向对象的重要部分，抽象就是忽略一个主题中与当前目标无关的那些方面，以便更充分的注意与当前目标有关的方便，抽象并不打算了解全部问题，而只是考虑部分问题，例如，需要考察Person对象时，我们不可能在程序中把Person的所有细节都定义出来，通常只能定义Person的部分数据、部分行为特征，而这些数据、行为特征是软件系统所关心的部分。

7、面向对象支持如下几个功能：

（1）对象是面向对象方法中最基本的概念，它的基本特点有：标识唯一性、分类性、多态性、封装性、模块独立性。

（2）类是具有共同属性、共同方法的一类事物，类是对象的抽象，对象是类的实例，而类是整个软件系统最小的程序单元，类的封装性将各种信息细节隐藏起来，并通过公用方法来暴露该类对外提供的功能，从而提高了类的内聚性，降低了对象之间的耦合性。

（3）对象间的这种相互合作需要一个机制协助进行，这样的机制称为“消息”，消息是一个实例与另一个实例之间相互通信的机制。

（4）在面向对象的方法中，类之间共享属性和操作的机制称为继承，继承具有传递性。

8、通常而言，“基于对象”也使用了对象，但是无法利用现有的对象模板产生新的对象类型，继而产生新的对象，也就是说，“基于对象”没有继承的特点，而“多态”则需要继承，没有了继承的概念也就无从谈论“多态”，面向对象方法的三大基本特征（封装、继承、多态）缺一不可，例如：js语言就是基于对象的，它使用了一些封装好的对象，调用对象的方法，设置对象的属性，但是它们无法让开发者派生新的类，开发者只能使用现有对象的方法和属性。

9、判断一门语言是否是面向对象的，通常可以使用继承和多态加以判断，“面向对象”和“基于对象”都实现了“封装”的概念，但是面向对象实现了“继承和多态”，而“基于对象”没有实现这些。

10、对象就是面向对象程序设计的中心，对象是人们要进行研究的任何事物，从最简单的整数到复杂的飞机等均可看作对象，它不仅能表示具体的事物，还能表示抽象的规则、计划或事件。

11、对象具有状态，一个对象用数据值来描述它的状态，Java通过为对象定义属性来描述对象的状态，对象还有操作，这些操作可以改变对象的状态，对象的操作也被称为对象的行为，Java通过为对象定义方法来描述对象的行为。

12、对象实现了数据和操作的结合，对象把数据和对数据的操作封装成一个有机的整体，因此面向对象提供了更大的编程粒度，因此对程序员来说，更易于掌握和使用。

13、对象是Java程序的核心，所以Java里的对象具有唯一性，每个对象都有一个标识来引用它，如果某个对象失去了标识，这个对象将变成垃圾，只能等着系统垃圾回收机制来回收它。Java语言不允许直接访问对象，而是通过对对象的引用来操作对象。

14、开发者定义了Java类之后，就可以使用new关键字来创建指定类的对象了，每个类可以创建任意多个对象，多个对象的属性值可以不同--这表现为不同对象的数据存在差异。

# 面向接口编程

## 概念

面向接口编程很重要的一点就是接口回调，用接口声明的变量称作接口变量，属于引用型变量，可以存放实现该接口的类的实例的引用，也就是对象引用。

接口回调：**可以把实现某一接口的类创建的对象的引用赋值给该接口声明的接口变量，那么该接口变量就可以调用被类实现的接口方法**。比如：

Com com;      //接口

ImplCom object;    //实现接口的对象

com = object;        //接口回调，com会根据对象的不同而实现不同的接口方法，回调类重写接口方法.

使用接口编程的最重要的核心思想就是使用接口回调及接口变量存放实现该接口类的对象的引用，从而接口变量就可以回调类实现的接口方法。

## 示例

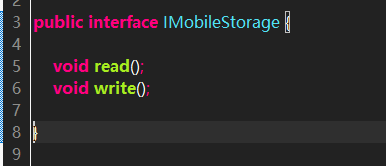
我们举一个例子去说明：

**定义**：现在我们要开发一个应用，模拟移动存储设备的读写，即计算机与U盘、MP3、移动硬盘等设备进行数据交换。

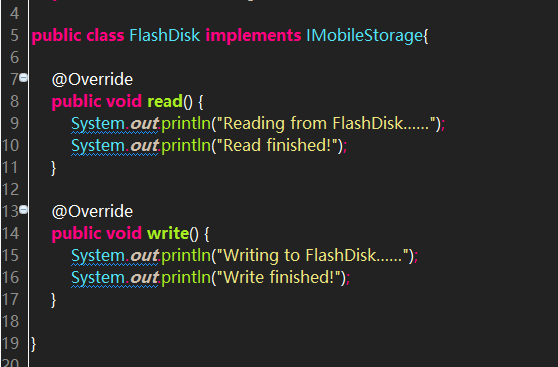
**上下文环境**：已知要实现U盘、MP3播放器、移动硬盘三种移动存储设备，要求计算机能同这三种设备进行数据交换，并且以后可能会有新的第三方的移动存储设备，所以计算机必须有扩展性，能与目前未知而以后可能会出现的存储设备进行数据交换。各个存储设备间读、写的实现方法不同，U盘和移动硬盘只有这两个方法，MP3Player还有一个PlayMusic方法。

## 代码实现

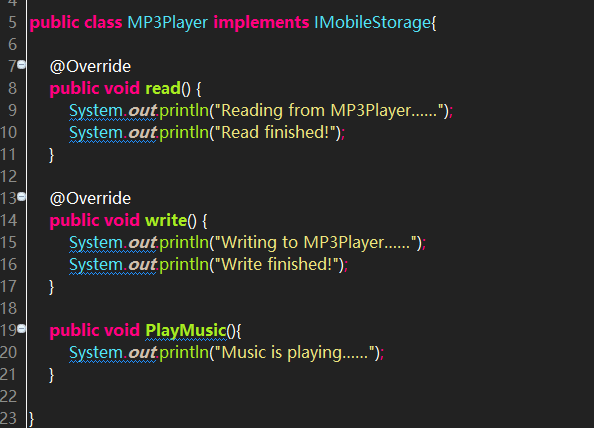
首先编写IMobileStorage接口：



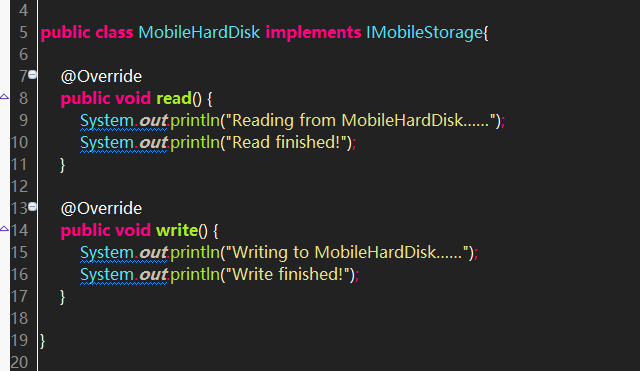
FlashDisk类



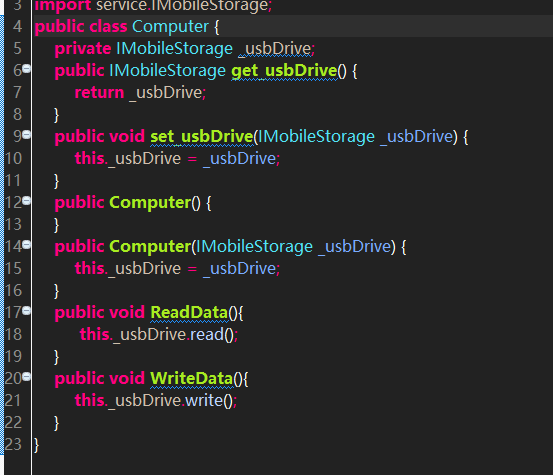
MP3Player类



MobileHardDisk类

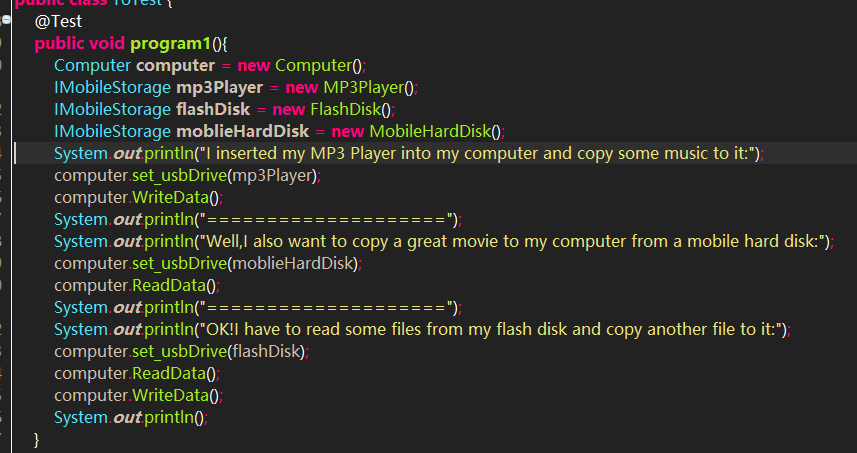


可以看到，它们都实现了IMobileStorage接口，并重写了各自不同的Read和Write方法。下面，我们来写Computer类：

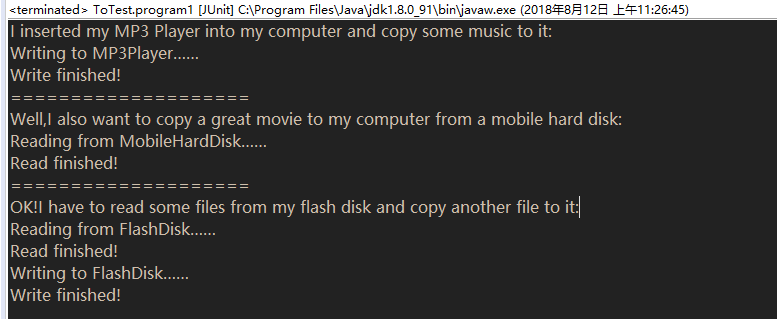


其中的UsbDrive就是可替换的移动存储设备，之所以用这个名字，是为了让大家觉得直观，就像我们平常使用电脑上的USB插口插拔设备一样

下面我们写测试代码：

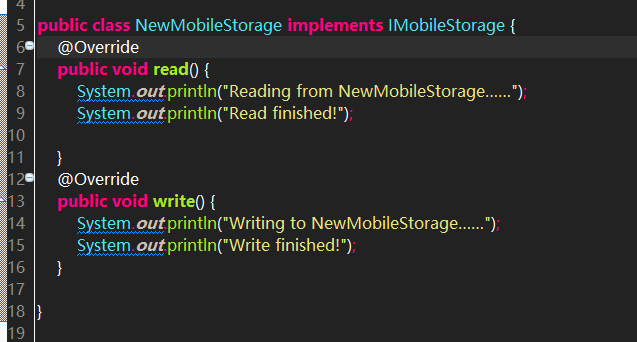


运行结果：

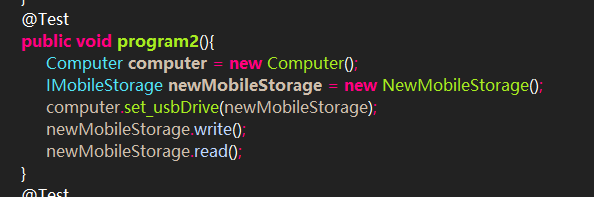


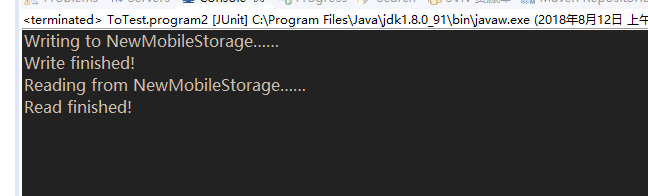
刚过了一个星期，就有人送来了新的移动存储设备NewMobileStorage，让我测试能不能用，我微微一笑，心想这不是小菜一碟，让我们看看面向接口编程的威力吧！将测试程序修改成如下

编写NewMobileStorage类



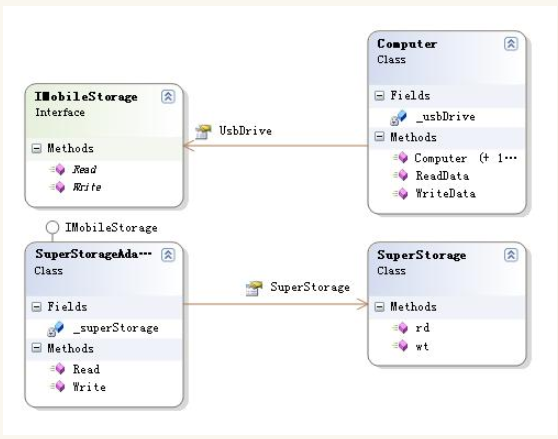
编写测试代码：

.运行结果：

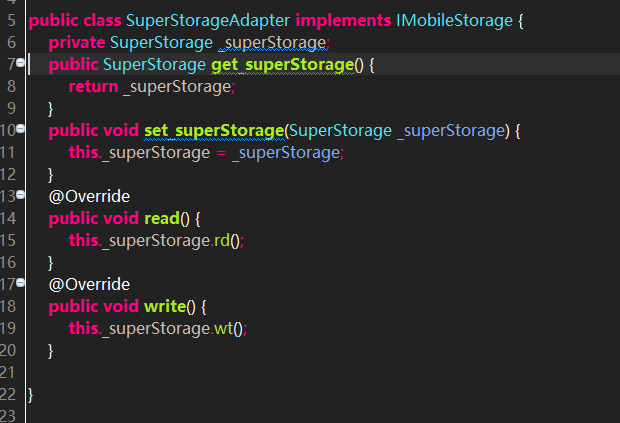


又过了几天，有人通知我说又有一个叫SuperStorage的移动设备要接到我们的Computer上，我心想来吧，管你是“超级存储”还是“特级存储”，我的“面向接口编程大法”把你们统统搞定。

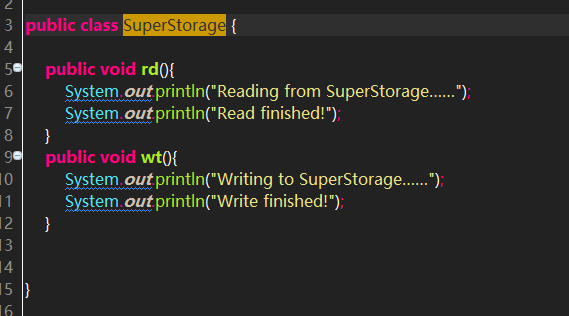
　　但是，当设备真的送来，我傻眼了，开发这个新设备的团队没有拿到我们的IMobileStorage接口，自然也没有遵照这个约定。这个设备的读、写方法不叫Read和Write，而是叫rd和wt，这下完了……不符合接口啊，插不上。但是，不要着急，我们回到现实来找找解决的办法。我们一起想想：如果你的Computer上只有USB接口，而有人拿来一个PS/2的鼠标要插上用，你该怎么办？想起来了吧，是不是有一种叫“PS/2-USB”转换器的东西？也叫适配器，可以进行不同接口的转换。对了！程序中也有转换器。

　　这里，我要引入一个设计模式，叫“Adapter”。它的作用就如现实中的适配器一样，把接口不一致的两个插件接合起来。由于本篇不是讲设计模式的，而且Adapter设计模式很好理解，所以我就不细讲了，先来看我设计的类图吧：  
  
　　如图所示，虽然SuperStorage没有实现IMobileStorage，但我们定义了一个实现IMobileStorage的SuperStorageAdapter，它聚合了一个SuperStorage，并将rd和wt适配为Read和Write，SuperStorageAdapter

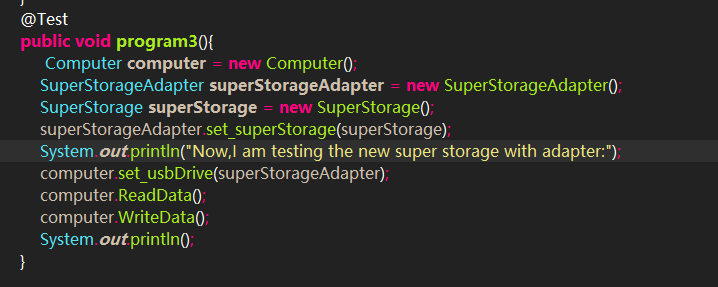
编写SuperStorageAdapter类



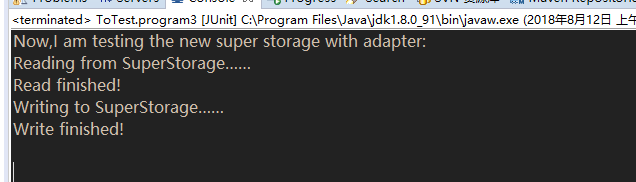
编写SuperStorage类



编写测试代码：



运行结果：



# 类

一组有相同属性和方法的**对象**。类实际上是一种数据类型，是我们根据需要创建的复合数据类型。

# 封装

指隐藏对象的属性和实现细节，仅对外提供公共访问方式，将代码及其处理的数据绑定在一起的一种编程机制，保证了程序和数据都不受外部干扰且不被误用。封装的基本单位是类。

好处：将变化隔离，便于使用，提高重用性，安全性。

原则：将不需要对外提供的内容都隐藏起来，把属性都隐藏，提供公共方法对其访问。

# 继承

一个对象获得另一个对象的属性和方法的过程。

## 好处

1：提高了代码的复用性。

2：让类与类之间产生了关系，提供了多态的前提。

# 多态

允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）实现多态的技术称为动态绑定，是指在执行期间判断所引用对象的实际类型，根据其实际类型调用其相应的方法。

## 作用

消除类型之间的耦合关系。

## 举例

比方说按下 F1 键这个动作，如果当前在 Flash 界面下弹出的就是AS3的帮助文档；如果当前在 Word 下弹出的就是 Word 帮助；在Windows下弹出的就是 Windows 帮助和支持。同一个事件发生在不同的对象上会产生不同的结果。

## 多态存在的三个必要条件

（1）要有继承

（2）要有重写

（3）父类引用指向子类对象

## 多态的好处

（1）可替换性。多态对已存在代码具有可替换性。

（2）可扩充性。多态对代码具有可扩充性。增加新的子类不影响已存在类的多态性、继承性，以及其他特性的运行和操作，实际上新增加子类更容易获得多态功能。

（3）接口性。多态是超类通过方法签名，向子类提供了一个共同接口，由子类来完善或者覆盖它而实现的。

（4）灵活性。它在应用中体现了灵活多样的操作，提高了使用效率。

（5）简化性：多态简化对应用软件的代码编写和修改过程，尤其在处理大量对象的运算和操作时，这个特点尤为突出和重要。

## 多态的实现方式

1. 接口实现
2. 子类继承父类进行方法重写
3. 同一个类中进行方法重载。

## 方法重载

在一个类中创建多个方法，它们具有相同的名字，但是具有不同的参数个数或类型，调用方法时通过传递给它们的不同参数个数和参数类型来决定具体使用哪个方法，这就是多态性。重载的时候，方法名要一样，但是参数类型和个数不一样，返回值类型可以相同也可以不相同，不能以返回类型作为重载函数的区分标准。方法重载是一个类中多态性的一种表现。

## 方法重写

父类与子类之间的多态性，对父类的函数进行重新定义，如果在子类中定义某方法与其父类有相同的名称和参数，我们说该方法被重写。在Java中，子类可继承父类中的方法，而不需要重新编写相同的方法。但是有时子类并不想原封不动的继承父类的方法，而是想作一定的修改，这就需要采用方法的重写。若子类中的方法与父类中的某一方法具有相同的方法名、返回类型和参数列表，则新方法将覆盖原有的方法，如果需要使用父类中原有的方法，可使用super关键字，该关键字引用了当前类的父类。子类函数的访问修饰权限不能小于父类。

多态性是面向对象编程的一种特性，和方法无关，简单的说，就是同样的一个方法根据输入数据的不同，做出不同的处理，即方法的重载--有不同的参数列表（静态多态性），而当子类继承自父类的相同方法，输入数据一样，但要做出有别于父类的响应时，你就要覆盖父类方法。即在子类中重写该方法-相同参数，不同实现（动态多态性）。重写的主要优点是能够定义某个子类特有的特征。

## 重写的规则

（1）方法名相同，参数列表必须完全与被重写的方法相同，否则不能称其为重写而是重载。

（2）返回类型必须与被重写的方法的返回类型相同，否则不能称为重写而是重载。

（3）访问修饰符的限制一定要大于被重写方法的访问修饰符（public>protected>default>private）

（4）重写方法一定不能抛出新的检查异常或者比被重写方法申明更加宽泛的检查型异常。例如：

父类的一个方法申明了一个检查异常IOException，在重写这个方法就不能抛出Exception,只能抛出IOException的子类异常，可以抛出非检查异常。

## 重载的规则

（1）方法名相同，必须具有不同的参数列表。

（2）可以有不同的返回类型，只要参数列表不同就可以了。

（3）可以有不同的访问修饰符。

（4）可以抛出不同的异常。

## 重写和重载的区别

调用被重载过的方法可以大大减少代码的输入量，同一个方法名只要往里面传递不同的参数就可以拥有不同的功能或返回值。用好重写和重载可以设计一个结构层次清晰而简洁的类，可以说重写和重载在编写代码过程中的作用非同一般。

# 变量

内存中的一个存储空间，用于存储常量数据。

作用：方便于运算。因为有些数据不确定，所以确定该数据的名称和存储空间。

特点：变量空间可以重复使用。

什么时候定义变量？只要是数据不确定的时候，就定义变量。

# 成员变量和局部变量的区别

（1）成员变量直接定义在类中。局部变量定义在方法中、参数上、语句中。

（2）成员变量在这个类中有效。局部变量只在自己所属的大括号内有效，大括号结束，局部变量失去作用域。

（3）成员变量存在于堆内存中，随着对象的产生而存在，消失而消失。局部变量存在于栈内存中，随着所属区域的运行而存在，结束而释放。

# 成员变量和静态变量的区别

（1）成员变量所属于对象，所以也称为实例变量。静态变量所属于类，所以也称为类变量。

（2）成员变量存在于堆内存中，静态变量存在于方法区中。

（3）成员变量随着对象创建而存在，随着对象回收而消失。静态变量随着类的加载而存在，随着类的消失而消失。

（4）成员变量只能被对象所调用 ，静态变量可以被对象调用，也可以被类名调用。所以，成员变量可以称为对象的特有数据，静态变量称为对象的共享数据。

# 数据类型

## 基本数据类型

byte、short、int、long、float、double、char、boolean

## 引用数据类型

数组、类、接口、集合

级别从低到高为：byte,char,short(这三个级)-->int-->float-->long-->double

自动类型转换：从低级别到高级别系统自动转换。（低--》高）

强制类型转换：把一个高级别的变量赋给一个比该数级别低的变量。（高--》低）

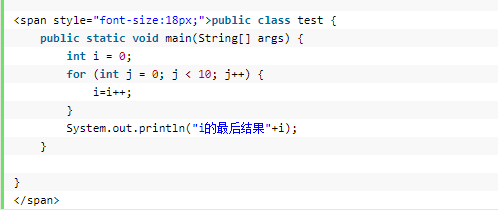
# 修饰符

public,protected,default,private这四种级别的修饰符都可以用来修饰类、方法和字段。

# 运算符

## i++和++i的区别

先看一个代码：



正确答案是0

区别在于：i=i++是先赋值在自增,所以无论循环多少次，左边的i始终是0，最后结果就是0，改成i=++i,是可以达到效果的，++i是先自增在赋值。

所以要达到自增就可以用i=++i，但是一般是直接用i++，这样更好;这也算是JAVA的一个自增陷阱。

## & 和 &&区别

& ：无论左边结果是什么，右边都参与运算。

&&：短路与，如果左边为false，那么右边不参与运算。

## | 和|| 区别

|：两边都运算。

||：短路或，如果左边为true，那么右边不参与运算。

Java中没有=+ 这个运算符

java中1表示true，0表示false

# 循环

## 语法

java中有三种主要的循环结构：

while 循环

do...while 循环

for 循环

在java5中引入一种主要用于数组的增强型for循环。

## while循环

while是最基本的循环，它的结构是：



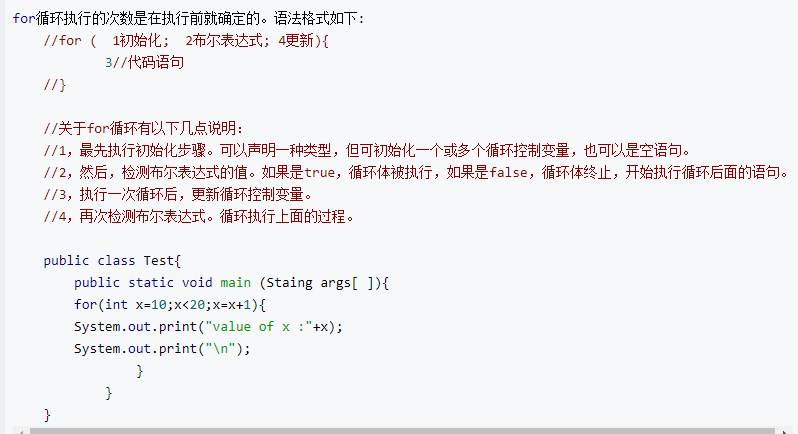
## do-while循环

对于while语句而言，如果不满足条件，则不能进入循环。但有时候我们需要即使不满足条件，也至少执行一次。 do…while循环和while循环相同，不同的是， do…while循环至少会执行一次。



## for循环

虽然所有循环结构都可以用while或者do…while表示，但java提供了另一种语句（for循环），使一些循环结构变得更简单。



## 三种循环的区别

（1）do…while循环至少执行一次循环体，而for,while循环必须先判断条件是否成立，然后决定是否执行循环体语句。

（2）如果你想在循环结束后，继续使用控制条件的那个变量，用while循环，   
否则用for循环。

（3）不知道用谁就用for循环，因为变量及早的从内存中消失，可以提高内存的使用效率。

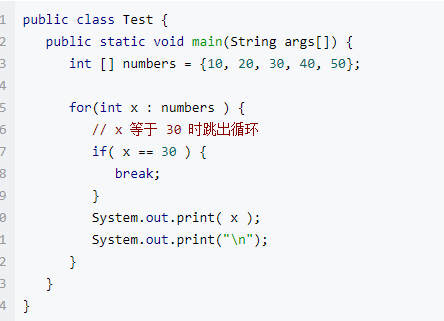
## 增强for循环

Java5引入一种主要用于数组的增强型循环。 java增强for循环语法格式如下：



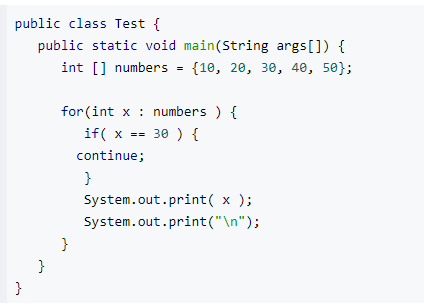
## break关键字

break主要用在循环语句或者switch语句中，用来跳出整个语句块。break跳出最里面的循环，并且继续执行该循环下面的语句。



## continue关键字

continue 适用于任何循环控制结构中。作用是让程序立刻跳到下一次循环的迭代。在for循环中continue语句使程序立即跳转到更新语句。在while或者do...while循环中，程序立即跳转到布尔表达式的判断语句。



## return关键字

它的作用不是结束循环的，而是结束方法的。

## return、break以及continue的区别

（1）return是结束方法。

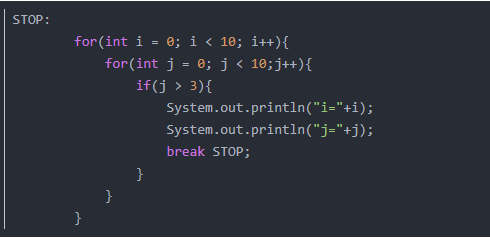
（2）break是跳出循环，应该是跳出本次循环，对于多循环的话。

（3）continue是终止本次循环，继续下次循环。

## 跳出多重循环的方法

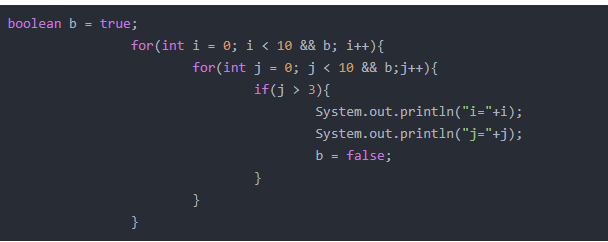
JAVA中，多重循环（多个FOR），如果只在其中一个循环中break那么只能跳出当前循环。这里有3个办法可以解决。

### （1）使用类似C语言中的GOTO

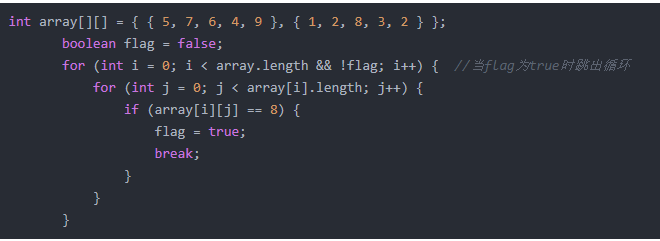


说明：给程序加标记

### （2）使用判断的方式加条件



### （3）实现方式类似于第二种



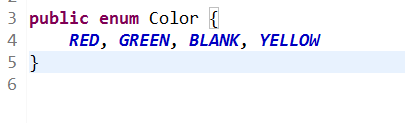
# 枚举

枚举不能实例化。

## 用法一：常量

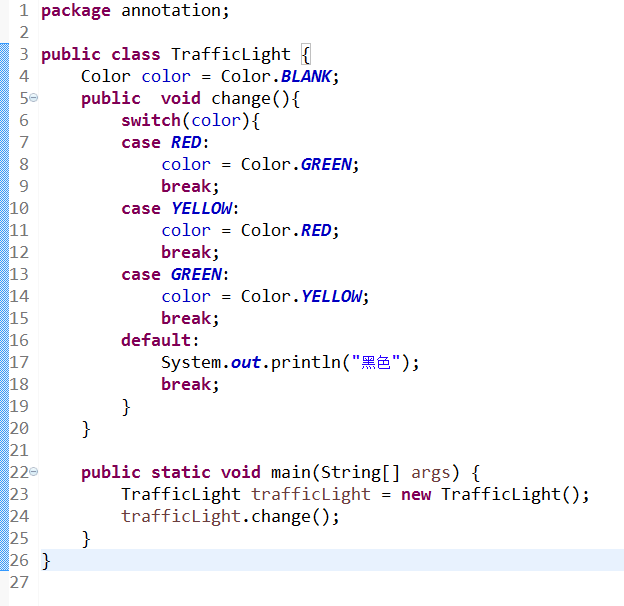
在JDK1.5 之前，我们定义常量都是： public static fianl.... 。现在好了，有了枚举，可以把相关的常量分组到一个枚举类型里，而且枚举提供了比常量更多的方法。

代码示例：



## 用法二：switch

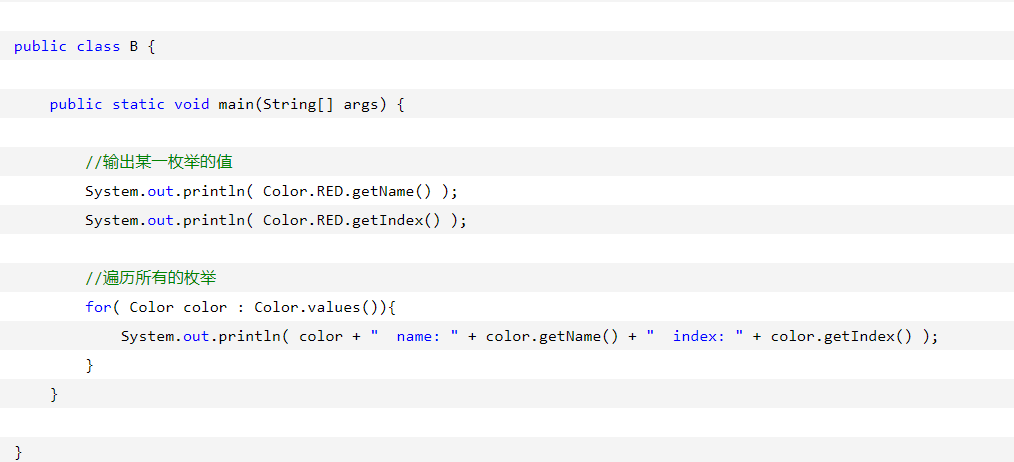
JDK1.6之前的switch语句只支持int,char,enum类型，使用枚举，能让我们的代码可读性更强。



## 用法三：向枚举中添加新方法



使用：



总结：

1、枚举的本质是类，在没有枚举之前，仍然可以按照java最基本的编程手段来解决需要用到枚举的地方。枚举屏蔽了枚举值的类型信息，不像在用public static final定义变量必须指定类型。枚举是用来构建常量数据结构的模板，这个模板可扩展。枚举的使用增强了程序的健壮性，比如在引用一个不存在的枚举值的时候，编译器会报错。

2、枚举在Java家族中只占了很小的一块比重，所以我在项目中用枚举的地方不是很多，所以常量大都是用public static final 来定义的。

# 函数的作用

（1）用于定义功能。

（2）用于封装代码，提高代码的复用性。

注意：函数中只能调用函数，不能定义函数。

# 主函数main的作用

（1）保证该类的独立运行。

（2）它是程序的入口。

（3）它在被jvm调用。

# 面向对象的特点

（1）将复杂的事情简单化。

（2）面向对象将以前过程中的执行者变成了指挥者。

（3）面向对象这种思想是符合现在人们思考问题的一种习惯。

# 堆内存和栈内存（重点）

Java把内存分成两种，一种叫做栈内存，一种叫做堆内存。

在函数中定义的**一些基本数据类型的变量和对象的引用变量都是在栈内存中分配**。当在一段代码块中定义一个变量时，Java就在栈中为这个变量分配内存空间，当超过变量的作用域后，Java会自动释放掉该变量分配的内存空间，该内存空间可以立刻被另作他用。

**堆内存用于存放由new创建的对象和数组**。在堆中分配的内存，由**Java虚拟机自动垃圾回收器来管理**。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者对象在堆内存中的首地址，在栈中的这个特殊的变量就变成了数组或者对象的引用变量，以后就可以在程序中使用栈内存中的引用变量来访问堆中的数组或者对象，引用变量相当于为数组或者对象起的一个别名或者代号。

引用变量是普通变量，定义时在栈中分配内存，引用变量在程序运行到作用域外释放，而数组和对象在堆内存中分配。

# 构造函数

用于给对象进行初始化，是给与之对应的对象进行初始化，它具有针对性，是函数中的一种。

特点：

（1）该函数的名称和所在类的名称相同。

（2）不需要定义返回类型。

（3）函数没有具体的返回值。

记住：所有对象创建时都需要初始化才可以使用。

# 构造函数和一般函数的区别

（1）两个函数定义格式不同。

（2）构造函数是在对象创建时就被调用，用于初始化，而且初始化动作只执行一次。一般函数是对象创建后，需要调用才执行，可以被对象调用多次。

# 构造代码块和构造函数的区别

## 构造代码块

是给所有的对象进行初始化，也就是说，所有的对象都会调用一个代码块，只要对象一建立就会调用这个构造代码块。

## 构造函数

是给与之对应的对象进行初始化，它具有针对性。

示例：

{

System.out.println("这是构造代码块！");

}

public Test10(){

System.out.println("这是构造函数！");

}

# static关键字

## 基本概念

static是一个修饰符，用于修饰成员(成员变量和成员函数)，不能修饰类。在Java中，我们不能用static修饰顶级类（top level class），只有内部类可以用static修饰。

示例：

public class Child {

public static class Demo{

public static void main(String[] args) {

}

}

}

## static变量

声明为static的变量实质上就是全局变量。当声明一个对象时并不产生static变量的拷贝，而是该类所有的实例变量共用同一个static变量。静态变量与静态方法类似，所有此类实例共享此静态变量，也就是说在类装载时，只分配一块存储空间，所有此类的对象都可以操控此块存储空间，当然对于final则另当别论了。

### 示例代码

**class** Value {

**static** **int** *c* = 0;

**static** **void** inc() {

*c*++;

}

**public** **class** Count2 {

**public** **static** **void** prt(String s) {

       System.*out*.print(s);

    }

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

       Value v1, v2;

       v1 = **new** Value();

       v2 = **new** Value();

*prt*("v1.c=" + v1.*c* + "  v2.c=" + v2.*c*);

       v1.*inc*();

*prt*(" v1.c=" + v1.*c* + "  v2.c=" + v2.*c*);

    }

}

结果为：v1.c=0  v2.c=0 v1.c=1  v2.c=1

由此可以证明它们共享一块存储区，static变量有点类似于C中的全局变量的概念。

## static方法

就是没有this的方法，在static方法内部不能调用非静态方法，反过来是可以的。而且可以在没有创建任何对象的前提下，仅仅通过类本身来调用static方法，这实际上正是static方法的主要用途。

static关键字的主要作用是方便在没有创建对象的情况下调用方法和变量。很显然，被static关键字修饰的方法或者变量不需要依赖于对象来进行访问，只要类被加载了，就可以通过类名去进行访问。另外可以编写static代码块来优化程序性能。static方法一般称作静态方法，并且由于这个特性，在静态方法中不能访问类的非静态成员变量和非静态成员方法，因为非静态成员方法/变量都是必须依赖具体的对象才能够被调用。但是要注意的是，虽然在静态方法中不能访问非静态成员方法和非静态成员变量，但是在非静态成员方法中是可以访问静态成员方法/变量的。因此，如果说想在不创建对象的情况下调用某个方法，就可以将这个方法设置为static。我们最常见的static方法就是main方法，因为程序在执行main方法的时候没有创建任何对象，因此只有通过类名来访问。另外记住，即使没有显示地声明为static，类的构造器实际上也是静态方法。

## 特点

（1）想要实现对象中共性数据的对象共享，可以将这个数据进行静态修饰。

（2）被静态修饰的成员，可以直接被类名所调用，也就是说，静态的成员多了一种调用方式，类名.静态方式。

（3）静态随着类的加载而加载，而且优先于对象存在，静态方法中不能使用this，super关键字。

# final关键字

（1）这个关键字是一个修饰符，可以修饰类，方法，变量。

（2）被final修饰的类是一个最终类，不可以被继承。

（3）被final修饰的方法是一个最终方法，不可以被重写，但可以重载。

（4）被final修饰的变量是一个常量，只能赋值一次。

# 静态代码块

## 基本概念

就是一个有静态关键字static标识的一个代码块区域，定义在类中。

## 作用

可以完成类的初始化。静态代码块随着类的加载而执行，而且只执行一次（new 多个对象就只执行一次）。如果和主函数在同一类中，优先于主函数执行。

静态的注意：静态的生命周期很长。

静态代码块、构造代码块、构造函数同时存在时的执行顺序：静态代码块 构造代码块 构造函数。

# super()和this()是否可以同时出现在构造函数中

　　两个语句只能有一个定义在第一行，所以只能出现其中一个。super()或者this():为什么一定要定义在第一行？因为super()或者this()都是调用构造函数，构造函数用于初始化，所以初始化的动作要先完成。

# 接口和抽象类的理解

在面向对象的概念中，所有的对象都是通过类来描绘的，但是并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。抽象类往往用来表征我们在对问题领域进行分析、设计中得出的抽象概念，是对一系列看上去不同，但是本质上相同的具体概念的抽象，我们不能把它们实例化，所以称之为抽象。比如：我们要描述“水果”，它就是一个抽象，它有质量、体积等一些共性（水果有质量），但又缺乏特性（苹果、橘子都是水果，它们有自己的特性），我们拿不出唯一一种能代表水果的东西（因为苹果、橘子都不能代表水果），可用抽象类来描述它，所以抽象类是不能够实例化的。当我们用某个类来具体描述“苹果”时，这个类就可以继承描述“水果”的抽象类，我们都知道“苹果”是一种“水果”。在面向对象领域中，抽象类主要用来进行类型隐藏，我们可以构造出一个固定的一组行为的抽象描述，但是这组行为却能够有任意个可能的具体实现方式，这个抽象描述就是抽象类，而这一组任意个可能的具体实现则表现为这个抽象类的所有派生类。接口和抽象类中的所有抽象方法不能有具体的实现，而应在它们的子类中实现所有的抽象方法，java的设计者可能为抽象方法的灵活性考虑，每个子类可根据自己的需要来实现抽象方法，抽象类是为了继承而存在的。

简言之抽象类是一种功能不全的类，接口只是一个抽象方法声明和静态不能被修改的数据的集合，两者都不能被实例化。从某种意义上说，接口是一种特殊形式的抽象类。在许多情况下，接口确实可以代替抽象类，如果你不需要刻意表达属性上的继承的话。进一步理解，关于java引入抽象类、接口的目的，1、从类的层次结构上看，抽象类是在层次的顶端，但在实际的设计当中，一般来说抽象类应当是后面才会出现。为什么？实际上抽象类的获取有点像数学中的提取公因式：ax+bx，x就是抽象类，如果你没有前面的式子，你怎么知道x是不是公因式呢？在这点上，也符合人们认识世界的过程，先具体后抽象。因此在设计过程中如果你得到大量的具体概念并从当中找到其共性时，这个共性的集合就是抽象类应当是没错的。2、interface从表面上看，和抽象类很相似，但用法完全不同。它的基本功能就是把一些毫不相关的类（概念）集合在一起形成一个新的、可集中操作的“新类”。我给学生的一个典型例子就是“司机”。谁可以当司机？谁都可以，只要领取了驾照。所以我不管你是学生，白领、蓝领还是老板，只要有驾照就是司机。

interface DriverLicence {

Licence getLicence(); //获取驾照

}

class StudentDriver extends Student implements DriverLicence {

}

class WhtieCollarEmployeeDriver extends WhtieCollarEmployee implements DriverLicence {

}

class BlueCollarEmployeeDriver extends BlueCollarEmployee implements DriverLicence {

}

class BossDriver extends Boss implements DriverLicence{

}

当我定义了“汽车”类后，我就可以指定“司机”了。

class Car {

setDriver(DriverLicence driver); //指定开车的司机

}

这时候，Car的对象并不关心这个司机到底是干什么的，他们的唯一共同点是领取了驾照（都实现了DriverLicence接口）。这个应当是接口最强大的地方也是抽象类无法比拟的。

总结：抽象类是提取具体类的公因式，而接口是为了将一些不相关的类“杂凑”成一个共同的群体。通常我们平时养成良好的习惯就是多用接口，毕竟java是单继承，不像C++，但是在需要使用抽象类的时候一定还是要用的。

# 抽象类

抽象类（abstract class）的定义方式如下：

public abstract class AbstractClass //里面至少有一个抽象方法

{

　　public int t; //普通数据成员

　　 public abstract void method1(); //抽象方法，抽象类的子类在类中必须实现抽象类中的抽象方法

　　public abstract void method2();

　　publi int method3 (){

　　…… //抽象类中可以赋予非抽象方法的默认行为，即方法的具体实现

　　}

}

# 抽象类的特点

（1）抽象方法只能定义在抽象类中，抽象类和抽象方法必须由abstract关键字修饰（可以描述类和方法，不可以描述变量），抽象类中可以定义非抽象方法。

（2）抽象方法只定义方法声明，并不定义方法实现。

（3）抽象类不可以被实例化。

（4）只有通过子类继承抽象类并重写抽象类中的所有抽象方法后，该子类才可以实例化，否则，该子类还是一个抽象类。

（5）抽象关键字abstract和哪些关键字不可以共存？final ,private , static。

# 接口

接口中有抽象方法，说明接口不可以实例化。接口的子类必须实现接口中所有的抽象方法后，该子类才可以实例化，否则该子类还是一个抽象类。类与类之间存在着继承关系，类与接口之间存在的是实现关系。继承用extends，实现用implements，接口和类不一样的地方就是接口可以被多实现，这就是多继承改良后的结果，Java将多继承机制通过多实现来体现。 抽象类可以不用实现接口中的方法，抽象类不用全部实现接口中的所有方法，其余的方法实现可以交给该抽象类的子类去实现即可。

接口（interface）的定义方式如下：

public interface Interface

{

　　static final int i; //接口中不能有普通数据成员，只能够有静态的不能被修改的数据成员,static表示全局，final表示不可修改，可以不用static final 修饰，会隐式的声明为static和final

　　 public void method1(); //接口中的方法一定是抽象方法，所以不用abstract修饰。

　　public void method2(); //接口中不能赋予方法的默认行为，即不能有方法的具体实现。

}

# 接口的特点

接口都用于设计上，设计上的特点：（可以理解为主板上提供的接口）

（1）接口是对外提供的规则。

（2）接口是功能的扩展。

（3）接口的出现降低了耦合性。

耦合性也叫耦合度，是对模块间关联程度的度量。耦合的强弱取决于模块间接口的复杂性、调用模块的方式以及通过界面传送数据的多少。

模块间的耦合度是指模块之间的依赖关系，包括控制关系、调用关系、数据传递关系。模块间联系越多，其耦合性越强，同时表明其独立性越差，联系越少，耦合性越差，表明其独立性越强。软件设计中通常用耦合度和内聚度作为衡量模块独立程度的标准。划分模块的一个准则就是高内聚低耦合。

# 抽象类和接口的区别

（1）抽象类只能被继承，而且只能单继承。接口需要被实现，而且可以多实现。

（2）抽象类中可以定义非抽象方法，子类可以直接继承使用。接口中都是抽象方法，需要子类去实现。

（3）抽象类使用的是 is a 关系。接口使用的是 like a 关系。 （is a 表示的是继承关系）

（4）抽象类的成员修饰符可以自定义。接口中的成员修饰符都是固定的，全都是public。

# 内部类

## 基本概念

内部类是指在一个外部类的内部再定义一个类，类名不需要和文件夹相同。内部类可以是静态static的，也可用public，default，protected和private修饰。（而外部顶级类即类名和文件名相同，只能使用public和default）。

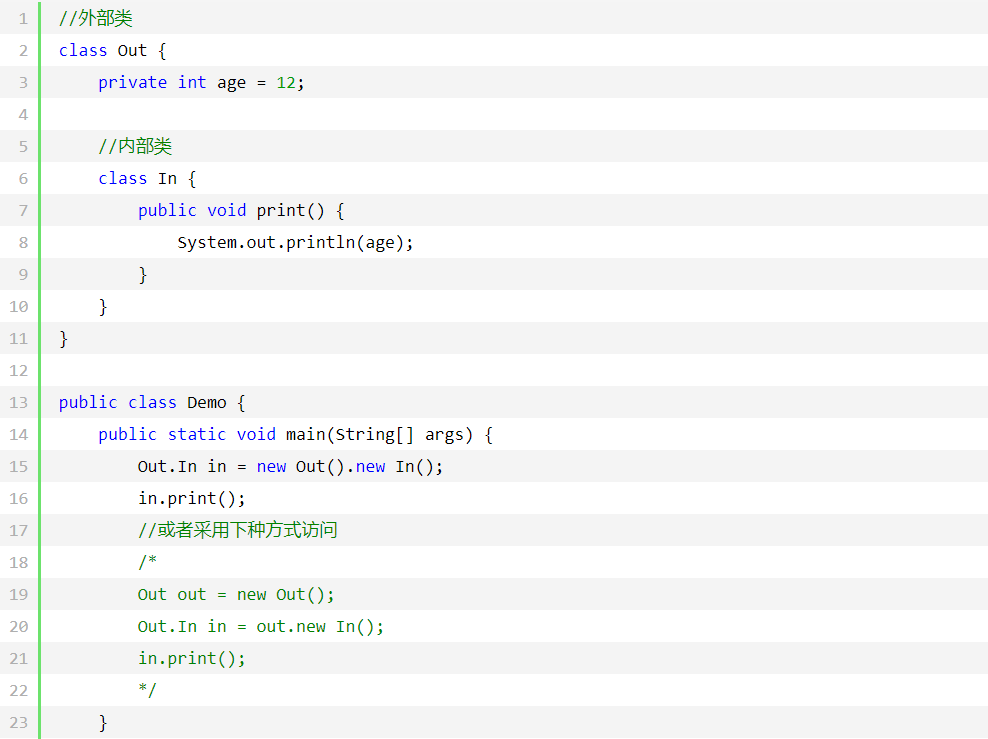
注意：内部类是一个编译时的概念，一旦编译成功，就会成为完全不同的两个类，对于一个名为outer的外部类和其内部定义的名为inner的内部类。编译完成后出现outer.class和outer$inner.class两类，所以内部类的成员变量/方法名可以和外部类相同。

成员内部类就是作为外部类的成员，可以直接使用外部类的所有成员和方法，即使是private的。同时外部类要访问内部类的所有成员变量/方法，则需要通过内部类的对象来获取。要注意的是，成员内部类不能含有static的变量和方法。因为成员内部类需要先创建了外部类，才能创建它自己的。在成员内部类要引用外部类对象时，使用outer.this来表示外部类对象。

示例：



## 内部类的基本结构



运行结果是12，从上面的例子中不难看出，内部类其实严重破坏了良好的代码结构，但为什么还要使用内部类呢？

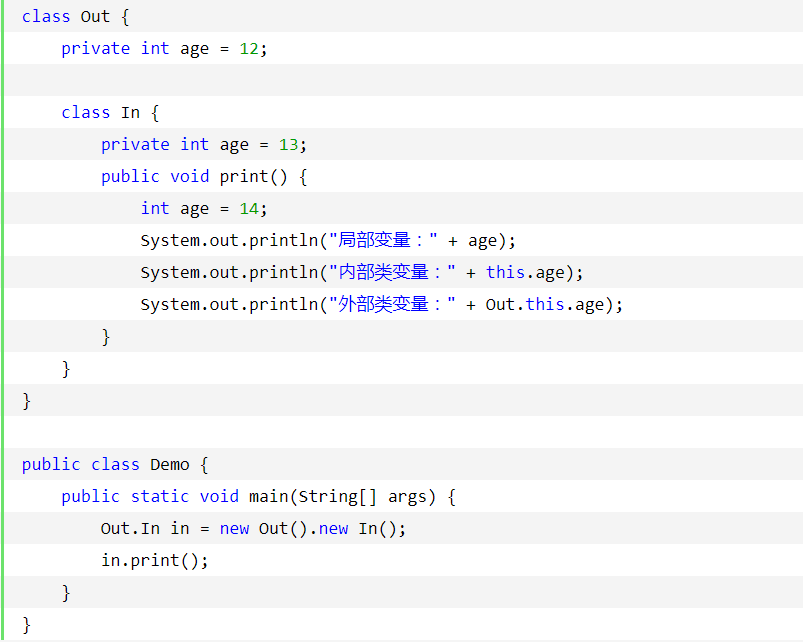
因为内部类可以随意使用外部类的成员变量（包括私有）而不用生成外部类的对象，这也是内部类的唯一优点。程序编译通过后会产生两个.class文件，分别是Out.class和Out$In.class，其中$代表了上面程序中Out.In中的那个 .

Out.In in = new Out().new In()可以用来生成内部类的对象，这种方法存在两个小知识点需要注意：

（1）开头的Out是为了标明需要生成的内部类对象在哪个外部类当中。

（2）必须先有外部类的对象才能生成内部类的对象，因为内部类的作用就是为了访问外部类中的成员变量。

## 内部类中的变量访问形式



运行结果：

局部变量：14

内部类变量：13

外部类变量：12

从实例1中可以发现，内部类在没有同名成员变量和局部变量的情况下，内部类会直接访问外部类的成员变量，而无需指定Out.this.属性名，否则，内部类中的局部变量会覆盖外部类的成员变量，而访问内部类本身的成员变量可用this.属性名，访问外部类的成员变量需要使用Out.this.属性名。

# 匿名内部类

## 基本概念

没有名字的内部类，就是内部类的简化形式，一般只用一次就可以用这种形式。匿名内部类其实就是一个匿名子类对象。想要定义匿名内部类需要前提，内部类必须继承一个类或者实现接口。

示例：

匿名内部类的基本实现

abstract class Person {

public abstract void eat();

}

public class Demo {

public static void main(String[] args) {

Person p = new Person() {

public void eat() {

System.out.println("eat something");

}

};

p.eat();

}

}

在接口上使用匿名内部类

interface Person {

public void eat();

}

public class Demo {

public static void main(String[] args) {

Person p = new Person() {

public void eat() {

System.out.println("eat something");

}

};

p.eat();

}

}

由上面的例子可以看出，只要一个类是抽象的或是一个接口，那么其子类中的方法都可以使用匿名内部类来实现，最常用的情况就是在多线程的实现上，因为要实现多线程必须继承Thread类或是实现Runnable接口。

## Thread类的匿名内部类实现

public class Demo {

public static void main(String[] args) {

Thread t = new Thread() {

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.print(i + " ");

}

}

};

t.start();

}

}

## Runnable接口的匿名内部类实现

public class Demo {

public static void main(String[] args) {

Runnable r = new Runnable() {

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.print(i + " ");

}

}

};

Thread t = new Thread(r);

t.start();

}

}

# 数组

## 概念

同一种数据类型的集合，其中数组就是一个容器。

## 数组的好处

可以自动给数组中的元素从0开始编号，方便操作这些元素。

## 语法

type arrayName[];  
type[] arrayName;

type 为Java中的任意数据类型，包括基本类型和组合类型，arrayName为数组名，必须是一个合法的标识符，[ ] 指明该变量是一个数组类型变量。例如：

int demoArray[];

int[] demoArray;

这两种形式没有区别，使用效果完全一样，读者可根据自己的编程习惯选择。

与C、C++不同，Java在定义数组时并不为数组元素分配内存，因此[ ]中无需指定数组元素的个数，即数组长度。而且对于如上定义的一个数组是不能访问它的任何元素的，我们必须要为它分配内存空间，这时要用到运算符new，其格式如下：

arrayName=new type[arraySize];

其中，arraySize 为数组的长度，type 为数组的类型。如：

demoArray=new int[3];

为一个整型数组分配3个int 型整数所占据的内存空间。通常你可以在定义的同时分配空间，语法为：  
    type arrayName[] = new type[arraySize];  
例如：

int demoArray[] = new int[3];

## 数组的初始化

你可以在声明数组的同时进行初始化（静态初始化），也可以在声明以后进行初始化（动态初始化）。例如：

// 静态初始化

// 静态初始化的同时就为数组元素分配空间并赋值

int intArray[] = {1,2,3,4};

String stringArray[] = {"微学苑", "http://www.weixueyuan.net", "一切编程语言都是纸老虎"};

// 动态初始化

float floatArray[] = new float[3];

floatArray[0] = 1.0f;

floatArray[1] = 132.63f;

floatArray[2] = 100F;

## 数组引用

可以通过下标来引用数组

arrayName[index];  
与C、C++不同，Java对数组元素要进行越界检查以保证安全性。每个数组都有一个length属性来指明它的长度，例如 intArray.length 指明数组 intArray 的长度。

代码示例：写一段代码，要求输入任意5个整数，输出它们的和。

int[] demoArray = new int[5];

int len = demoArray.length;

long sum = 0;

System.out.println("请输入"+len+"个数字，已空格分隔：");

Scanner scanner = new Scanner(System.in);

for(int i=0;i<len;i++){

demoArray[i] = scanner.nextInt();

}

//计算和

for(int j=0;j<len;j++){

sum+=demoArray[j];

}

System.out.println("总数是："+sum);

scanner.close();

## 数组的遍历

实际开发中，经常需要遍历数组以获取数组中的每一个元素。最容易想到的方法是for循环，例如：

int arrayDemo[] = {1, 2, 4, 7, 9, 192, 100};

for(int i=0,len=arrayDemo.length; i<len; i++){

System.out.println(arrayDemo[i] + ", ");

}

输出结果：  
1, 2, 4, 7, 9, 192, 100,

不过，Java提供了”增强版“的for循环，专门用来遍历数组，语法为：

for( arrayType varName: arrayName ){

// Some Code

}

arrayType 为数组类型（也是数组元素的类型）；varName 是用来保存当前元素的变量，每次循环它的值都会改变；arrayName 为数组名称，每循环一次，就会获取数组中下一个元素的值，保存到 varName 变量，直到数组结束。即，第一次循环 varName 的值为第0个元素，第二次循环为第1个元素......例如：

int arrayDemo[] = {1, 2, 4, 7, 9, 192, 100};

for(int x: arrayDemo){

System.out.println(x + ", ");

}

这种增强版的for循环也被称为”foreach循环“，它是普通for循环语句的特殊简化版。所有的foreach循环都可以被改写成for循环。但是，如果你希望使用数组的索引，那么增强版的 for 循环无法做到。

## 二维数组

二维数组的声明、初始化和引用与一维数组相似，数组的数组---二维数组的每一个元素都是一个一维数组。

### 定义格式

数据类型[][] 数组名 = new 数据类型[二维数组的长度/包含的一维数组的个数][每个一维数组的长度];

int[][] arr = new int[3][5];---定义了一个整型的二维数组，其中包含3个一维数组，每个一维数组可以存储5个整数

int intArray[ ][ ] = { {1,2}, {2,3}, {4,5} };

int a[ ][ ] = new int[2][3];

a[0][0] = 12;

a[0][1] = 34;

// ......

a[1][2] = 93;

Java语言中，由于把二维数组看作是数组的数组，数组空间不是连续分配的，所以不要求二维数组每一维的大小相同。例如：

int intArray[ ][ ] = { {1,2}, {2,3}, {3,4,5} };

int a[ ][ ] = new int[2][ ];

a[0] = new int[3];

a[1] = new int[5];

说明：二维数组一般很少用到。

# 泛型

## 概念

jdk1.5版本以后出现的一个安全机制，表现格式：< >。所谓泛型，就是允许在定义类、接口、方法时使用类型形参，这个类型形参将在声明变量、创建对象、调用方法时动态的指定（即传入实际的类型参数，也可称为类型实参）。

Java5改写了集合框架中的全部接口和类，为这些接口、类增加了泛型支持，从而可以在声明集合变量、创建集合对象时传入类型实参，比如看到的List<String>和ArrayList<String>两种类型。

包含泛型声明的类型可以在定义变量、创建对象时传入一个类型实参，从而可以动态的生成无数多个逻辑上的子类，但这种子类在物理上并不存在。

我们可以为任何类、接口增加泛型声明（并不是只有集合类才可以使用泛型声明，虽然集合类是泛型的重要使用场所）。

说明：T ？都表示泛型。一般K表示[key](https://www.baidu.com/s?wd=key&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)（键）；T表示type（类型）；V表示value（值）；E表示entry （实体）

## 定义一个泛型类

//定义泛型类

public class Apple<T> {

private T info;

public Apple(){}//构造函数

public Apple(T info){

this.info = info;

}

public T getInfo() {

return info;

}

public void setInfo(T info) {

this.info = info;

}

public static void main(String[] args) {

Apple<String> a = new Apple<String>("苹果");

System.out.println(a.getInfo());

Apple<Double> a1 = new Apple<Double>(3.14);

System.out.println(a1.getInfo());

}

}

当创建带泛型声明的自定义类，为该类定义构造器，构造器名还是原来的类名，不要增加泛型声明。例如：为Apple<T>类定义构造器，其构造器名依然是Apple,而不是Apple<T>，调用该构造器时却可以使用Apple<T>的形式，当然应该为T形参传入实际的类型参数，Java7提供了菱形语法，允许省略<>中的类型实参。

Apple<T>意思是Apple这个类里面需要用到另外一个类，但是又不能确定要用到的那个类的具体类型，所以暂时用T来代替，当具体的程序知道要用到的那个类型是什么的时候就用那个类型来代替T就ok了。例如我需要实例化一个Apple类，需要用到的那个类是String，那么我就可以这么来实例化Apple<String>();

## 好处

（1）将运行时期的ClassCastException问题转换成了编译失败，体现在编译时期，程序员就可以解决问题。

（2）避免了强制转换的麻烦。

只要带有<>的类或者接口，都属于带有类型参数的类或者接口，在使用这些类或者接口时，必须给<>中传递一个具体的引用数据类型。

（3）泛型技术：其实应用在编译时期，是给编译器使用的技术，到了运行时期，泛型就不存在了。为什么? 因为泛型的擦除：也就是说，编辑器检查了泛型的类型正确后，在生成的类文件中是没有泛型的。

（4）在运行时，如何知道获取的元素类型而不用强转呢？

泛型的补偿：因为存储的时候，类型已经确定了是同一个类型的元素，所以在运行时，只要获取到该元素的类型，在内部进行一次转换即可，所以使用者不用再做转换动作了。

（5）什么时候使用泛型类呢？

　　　当类中操作的引用数据类型不确定的时候，以前用的Object来进行扩展的，现在可以用泛型来表示。这样可以避免强转的麻烦，而且将运行问题转移到编译时期的问题。

# instanceof

## 基本概念

instanceof是一个运算符，运算符是双目运算符，左面的操作元是一个对象，右面是一个类或接口，当左面的对象是右面的类创建的对象时，该运算符运算的结果是true，否则是false。

说明：一个类的实例包括本身的实例,以及所有直接或间接子类的实例，instanceof左边操作元显示声明的类型与右边操作元必须是同种类或右边是左边父类的继承关系。不同的继承关系下，编译会出错。

## 代码示例

**package** mavenproject;

**class** Person{

}

**class** Student **extends** Person{

}

**class** Postgraduate **extends** Student{

}

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

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

*instanceofTest*(**new** Student());

}

**public** **static** **void** instanceofTest(Person p){

**if**(p **instanceof** Person){

System.***out***.println("p是Person类的实例");

}

**if**(p **instanceof** Student){

System.***out***.println("p是Student类的实例");

}

**if**(p **instanceof** Postgraduate){

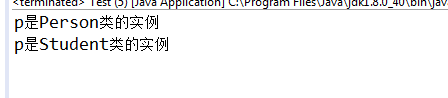
System.***out***.println("p是Postgraduate类的实例");

}

}

}

输出的结果：



# 注解（重点）

## 概念

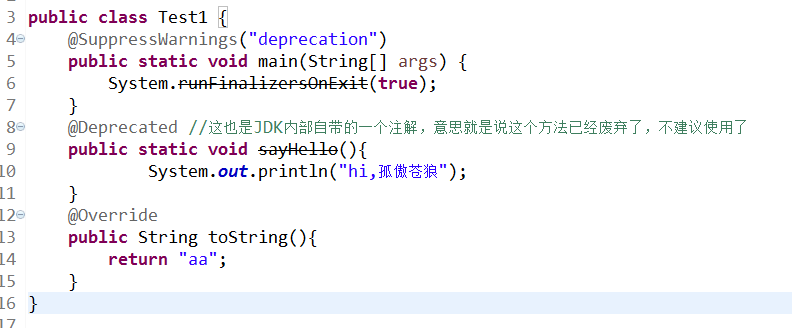
注解是JDK1.5之后才有的新特性，JDK1.5之后内部提供的三个注解：

（1）@Deprecated：废弃的、过时的。

（2）@Override：重写、覆盖

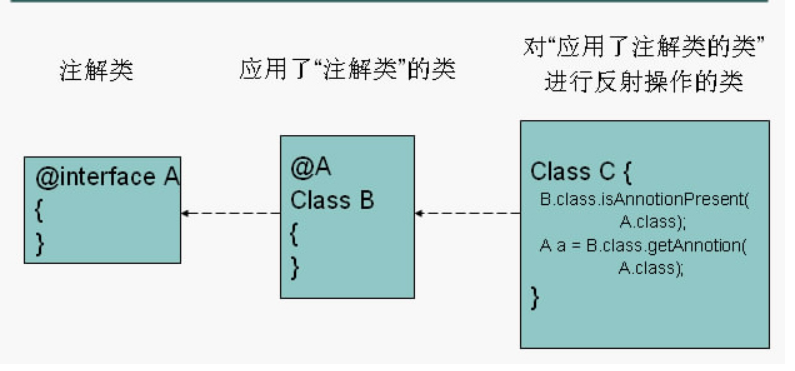
（3）@SuppressWarnings：压缩、警告

注解的应用：



总结：**注解(Annotation)相当于一种标记，在程序中加入注解就等于为程序打上某种标记**，没有加则等于没有任何标记，以后，javac编译器、开发工具和其他程序可以通过反射来了解你的类及各种元素上有无何种标记，看你的程序有什么标记，就去干相应的事，**标记可以加在包、类，属性、方法，方法的参数以及局部变量上。**

注解的应用结构图：



注解就相当于一个你的源程序要调用一个类，在源程序中应用某个注解，得事先准备好这个注解类。就像你要调用某个类，得事先开发好这个类。包 java.lang.annotation 中包含所有自定义注解所需用到的原注解和接口。如接口 java.lang.annotation.Annotation 是所有注解继承的接口,并且是自动继承，不需要定义时指定，类似于所有类都自动继承Object。

## 作用

（1）生成文档，这是最常见的，也是java 最早提供的注解。常用的有@see @param @return 等。

（2）跟踪代码依赖性，实现替代配置文件的功能。比较常见的是spring 2.5 开始的基于注解配置，作用就是减少配置。现在的框架基本都使用了这种配置来减少配置文件的数量。

（3）在编译时进行格式检查。如@Override 放在方法前，如果你这个方法并不是覆盖了超类方法，则编译时就能检查出。

## 使用注解的一些注意事项

（1） 要用好注解，必须熟悉Java 的反射机制，注解的解析完全依赖于反射。

（2）不要滥用注解。平常我们编程过程很少接触和使用注解，只有做设计，且不想让设计有过多的配置时。

## 常用的注解

### JDK自带的注解

#### （1）@Override

重写， 标识覆盖它的父类的方法

#### （2）@Deprecated

已过期，表示方法是不被建议使用的

#### （3）@SuppressWarnings

压制警告，抑制警告

#### （4）@version

指定版本信息

#### （5）@since

指定最早出现在哪个版本中

#### （6）@author

指定作者

#### （7）@see

生成参考其他的JavaDoc文档的链接

#### （8）@link

生成参考其他的JavaDoc文档，它和@see标记的区别在于，@link标记能够嵌入到注释语句中，为注释语句中的特殊词汇生成连接。  eg.[{@link](mailto:%7B@link)Hello}

#### （9）@param

描述方法的参数

#### （10）@return

描述方法的返回值

#### （11）@throws

描述方法抛出的异常，指明抛出异常的条件

### 常见的第三方注解

#### （1）@Repository

用于标注数据访问组件，即DAO组件

#### （2）@Service

用于标注业务层组件

#### （3）@Transactional

声明所有的方法都需要事务管理。每一个业务方法开始时都会打开一个事务

#### （4）@Controller

控制层

#### （5）@Component

把中立的类交给spring管理

#### （6）@Autowired

自动装配，将bean容器里的值自动注入到bean

#### （7）@Path

处理REST请求,接口路径

#### （8）@Method

常用的HTTP动词有下面五个（括号里是对应的SQL命令）

GET（SELECT）：从服务器取出资源（一项或多项）。

POST（CREATE）：在服务器新建一个资源。

PUT（UPDATE）：在服务器更新资源（客户端提供改变后的完整资源）。

PATCH（UPDATE）：在服务器更新资源（客户端提供改变的属性）。

DELETE（DELETE）：从服务器删除资源

还有两个不常用的HTTP动词

HEAD：获取资源的元数据。

OPTIONS：获取信息，关于资源的哪些属性是客户端可以改变的

#### （9）@Accept

表示接口要返回给客户端的数据格式

#### （10）@Content-Type

表示客户端发送给服务器端的数据格式，这个是写REST接口时候定义的

#### （11）@Produces

表示类或者方法返回的MIME数据类型

有几种格式如下：

（1）@Produces("text/plain") 文本类型

（2）@Produces("text/html")  Html类型

（3）@Produces({"application/xml"}) Xml类型

（4）@Produces({ "application/json"}) Json类型

可以一次注解两种或多种的MIME类型，格式如：{"application/xml", "application/json"}这表示两者都可以使用，但是选择的时候一般会选择前者，即application/xml,因为它第一次出现

#### （12）@Consumes

代表的是一个资源可以接受的 MIME 类型

#### （13）@Queryparam

指定的是URL中的参数是以键值对的形式出现的,而在程序中 @QueryParam("from")  int from则读出URL中from的值.

例如：URL输入为:users?from=100&to=200&orderBy=age&orderBy=name

#### （14）@Pathparam

URL中只出现参数的值,不出现键值对

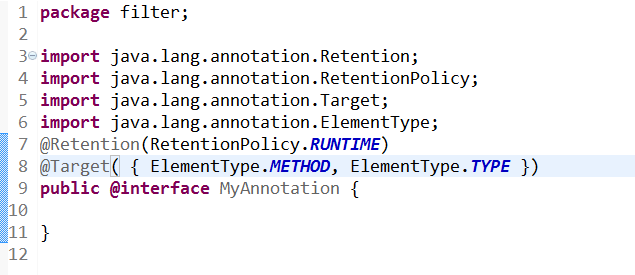
例如：/users/100

## 自定义注解及其应用

语法：

public @interface MyAnnotation{}

代码示例：

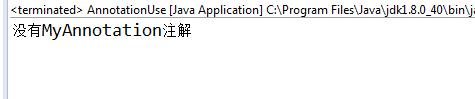


把自定义的注解加到某个类上



用反射进行测试AnnotationUse类的定义上是否有@MyAnnotation注解。一旦在某个类上使用了@MyAnnotation，那么这个MyAnnotation类的实例对象annotation就会被创建出来了。

结果如图：



## 元注解

### 基本概念

元注解的作用就是负责注解其他注解。Java5.0定义了4个标准的meta-annotation类型，它们被用来提供对其它 annotation类型作说明。这些类型和它们所支持的类在java.lang.annotation包中可以找到。

### @Retention（注解的生命周期）

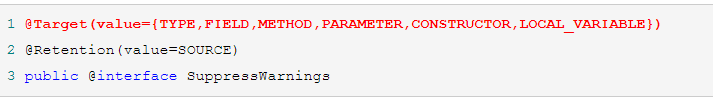
其三种取值：RetentionPolicy.SOURCE、RetentionPolicy.CLASS、RetentionPolicy.RUNTIME分别对应：Java源文件(.java文件)---->.class文件---->内存中的字节码。

说明：当在Java源程序上加了一个注解，这个Java源程序要由javac去编译，javac把java源文件编译成.class文件，在编译成class时可能会把Java源程序上的一些注解给去掉，java编译器(javac)在处理java源程序时，可能会认为这个注解没有用了，于是就把这个注解去掉了，那么此时在编译好的class中就找不到这个注解了， 这是编译器编译java源程序时对注解进行处理的第一种可能情况。假设java编译器在把java源程序编译成class时，没有把java源程序中的注解去掉，那么此时在编译好的class中就可以找到注解，当程序使用编译好的class文件时，需要用类加载器把class文件加载到内存中，class文件中的东西不是字节码，class文件里面的东西由类加载器加载到内存中去，类加载器在加载class文件时，会对class文件里面的东西进行处理，如安全检查，处理完以后得到的最终在内存中的二进制的东西才是字节码，类加载器在把class文件加载到内存中时也有转换，转换时是否把class文件中的注解保留下来，这也有说法，所以说一个注解的生命周期有三个阶段：java源文件是一个阶段，class文件是一个阶段，内存中的字节码是一个阶段**，**javac把java源文件编译成.class文件时，有可能去掉里面的注解，类加载器把.class文件加载到内存时也有可能去掉里面的注解，因此在自定义注解时就可以使用Retention注解指明自定义注解的生命周期，自定义注解的生命周期是在RetentionPolicy.SOURCE阶段(java源文件阶段)，还是在RetentionPolicy.CLASS阶段(class文件阶段)，或者是在RetentionPolicy.RUNTIME阶段(内存中的字节码运行时阶段)，根据JDK提供的API可以知道默认是在RetentionPolicy.CLASS阶段 (JDK的API写到：the retention policy defaults to RetentionPolicy.CLASS.)。

### @ Target（注解的目标）

@Target元注解决定了一个注解可以标识到哪些成分上，如标识在类身上，或者属性身上，或者方法身上等成分，@Target默认值为任何元素(成分)。

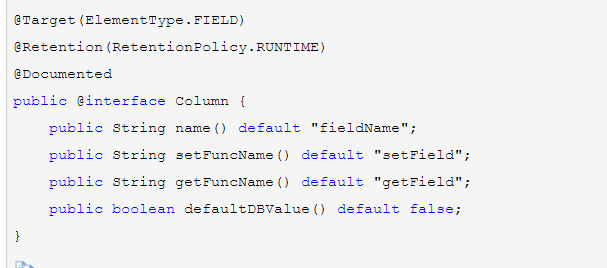
示例：



### @Documented（标记注解）

@Documented用于描述其它类型的annotation，应该被作为被标注的程序成员的公共API，因此可以被例如javadoc此类的工具文档化。Documented是一个标记注解，没有成员。说明：该注解将被包含在javadoc中。

示例代码：



### @Inherited（标记注解）

@Inherited 元注解是一个标记注解，@Inherited阐述了某个被标注的类型是被继承的。如果一个使用了@Inherited修饰的annotation类型被用于一个class，则这个annotation将被用于该class的子类。说明子类可以继承父类中的该注解。

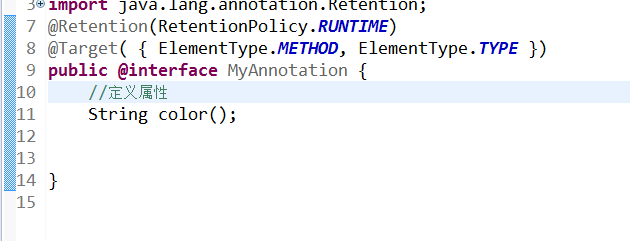
## 为注解增加属性

注解可以看成是一种特殊的类，既然是类，那自然可以为类添加属性。

### 添加属性

语法：类型 属性名（）;

示例：



其实从代码的写法上来看，注解更像是一种特殊的接口，注解的属性定义方式就和接口中定义方法的方式一样，而应用了注解的类可以认为是实现了这个特殊的接口。

### 应用属性

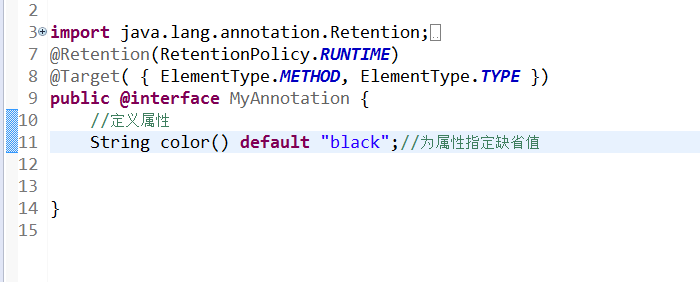
示例代码：



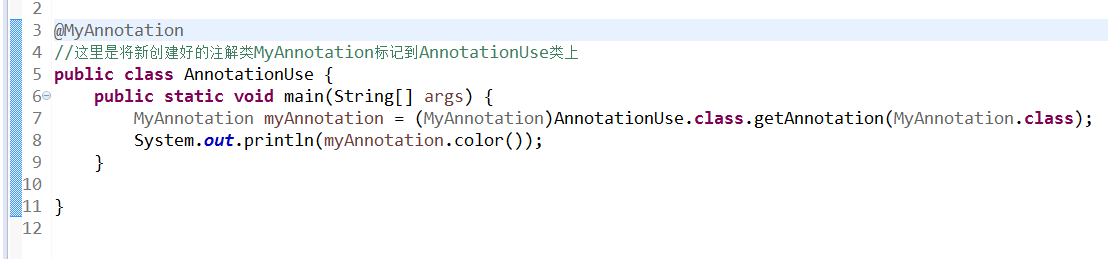
### 为属性指定缺省值（默认值）

语法：类型 属性名() default 默认值；

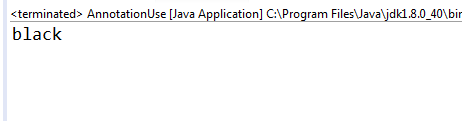
示例代码：



编写测试类：



输出结果：

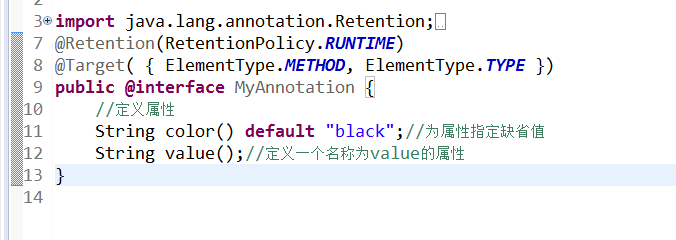


### value属性

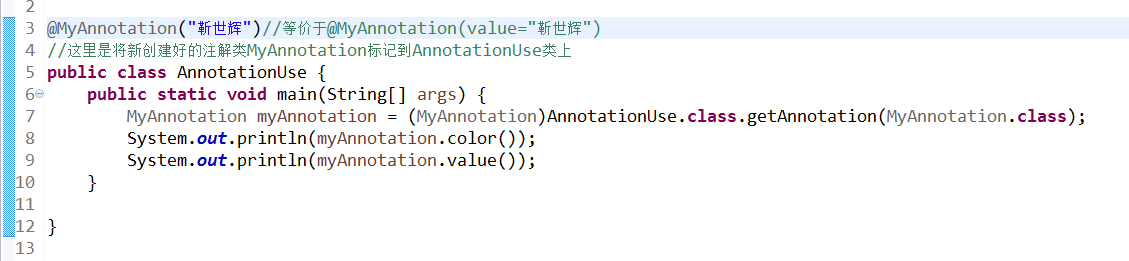
如果一个注解中有一个名称为value的属性，且你只想设置value属性(即其他属性都采用默认值或者你只有一个value属性)，那么可以省略掉“value=”部分。

例如：@SuppressWarnings("deprecation")

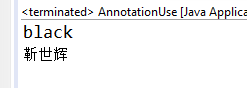
代码示例：



编写测试类：



运行结果：



## 为注解增加高级属性

### 数组类型的属性

增加数组类型的属性：int[] arrayAttr() default {1,2,4};

应用数组类型的属性：@MyAnnotation(arrayAttr={2,4,5})

如果数组属性只有一个值，这时候属性值部分可以省略大括号，如：@MyAnnotation(arrayAttr=2)，这就表示数组属性只有一个值，值为2。

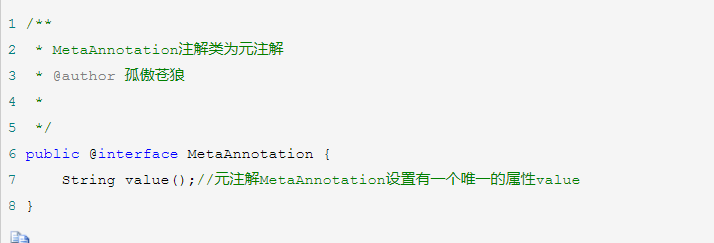
### 枚举类型的属性

增加枚举类型的属性：EumTrafficLamp lamp() default EumTrafficLamp.RED;

应用枚举类型的属性：@MyAnnotation(lamp=EumTrafficLamp.GREEN)

### 注解类型的属性

示例：

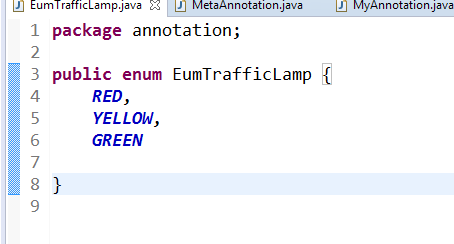


为注解添加一个注解类型的属性,并指定注解属性的缺省值：MetaAnnotation annotationAttr() default @MetaAnnotation("xdp");

## 注解综合测试

代码示例：

编写枚举类



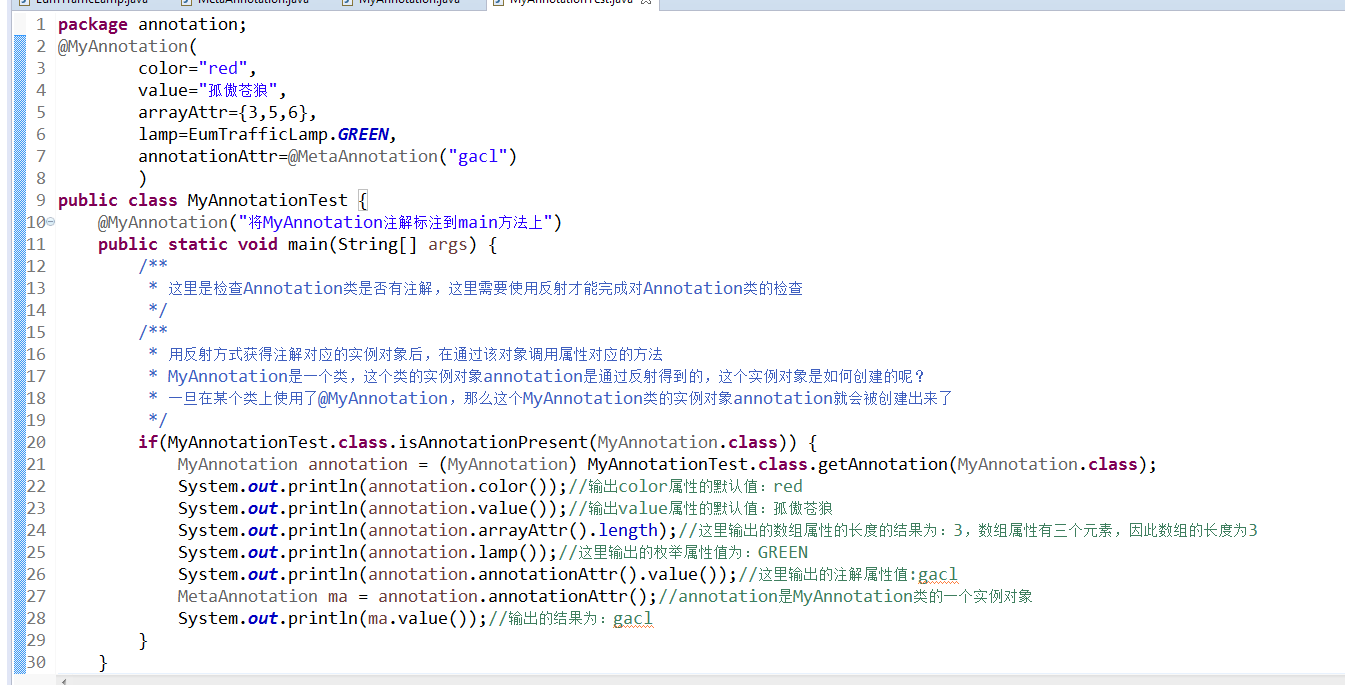
编写元注解类



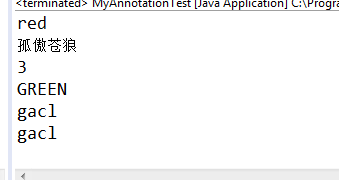
编写注解类



编写测试类：



运行结果：



# ThreadLocal类

## 定义

这个类在java.lang这个包中，它不是一个线程，而是线程的一个本地化对象。ThreadLocal是为了解决多个线程同时访问一个变量时的并发问题，在多线程环境下，当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程分配一个独立的变量副本，所以每个线程都可以独立的改变自己的副本，而不会影响其他线程所对应的副本，从线程的角度看，这个变量就像是线程的本地变量，这个变量里面的值是和其他线程分割开来的，变量的值只有当前线程能访问到，保证了多线程环境下当前线程中变量的安全性。

## 实现原理

ThreadLocal为每一个线程维护一个变量副本，在ThreadLocal类中有一个静态内部类ThreadLocalMap，用键值对的形式存储每一个线程的变量副本，ThreadLocalMap中元素的key为当前ThreadLocal对象，而value对应线程的变量副本，每个线程可能存在多个ThreadLocal。

## 方法

（1）void set(Object value)：用来设置当前线程中变量的副本。

（2） public Object get()：用来获取ThreadLocal在当前线程中保存的变量副本

（3）public void remove()：用来删除当前线程中变量的副本，该方法是JDK 5.0新增的方法，目的是为了减少内存的占用，防止内存泄漏。虽然当线程结束后，线程的变量副本会被自动垃圾回收，但是在线程结束前会有造成内存泄漏的情况，可以调用该方法来避免。

（4）protected Object initialValue()：返回该线程局部变量的初始值，该方法一般在子类中重写用来初始化变量副本。如果不在子类中重写，这个方法会延迟调用，在线程第1次调用get()或set(Object)时才执行，并且仅执行1次，在ThreadLocal中的实现直接返回null。

## 使用场景

### （1）数据库连接管理

同一事务多DAO共享同一Connection，必须在一个共同的外部类使用ThreadLocal保存Connection。代码如下：



这样就保证了一个线程对应一个数据库连接，保证了事务。因为事务是依赖一个连接来控制的，如commit,rollback,都是数据库连接的方法。

### （2）ThreadLocal实例



## 内存泄露问题

（1）Threadlocal实例为null

每个线程中都存在一个map, map的类型是ThreadLocal.ThreadLocalMap. Map中的key为一个threadlocal实例. 这个Map的确使用了弱引用,不过弱引用只是针对key. 每个key都弱引用指向threadlocal. 当把threadlocal实例置为null以后,没有任何强引用指向threadlocal实例,所以threadlocal将会被gc回收. 但是,我们的value却不能回收,因为存在一条从current thread连接过来的强引用. 只有当前thread结束以后, current thread就不会存在栈中,强引用断开, CurrentThread, Map, value将全部被GC回收。 所以只要这个线程对象被gc回收，就不会出现内存泄露，但在threadLocal设为null和线程结束这段时间不会被回收的，就发生了我们认为的内存泄露。

（2）线程对象没有被回收

线程对象不被回收的情况，这就发生了真正意义上的内存泄露。比如使用线程池的时候，线程池中对线程管理都是采用线程复用的方法。在线程池中线程非常难结束甚至于永远不会结束。就可能出现内存泄露。

## 与同步机制的比较

相同点：ThreadLocal和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。

不同点：同步机制中通过对象的锁机制保证同一时间只有一个线程来访问变量，这时该变量是多个线程共享的，使用同步机制要缜密的分析什么时候对变量进行读写，什么时候需要锁定某个对象，什么时候释放掉锁等复杂的问题，程序设计和编写难度大。

ThreadLocal则为每一个线程提供了一个独立的变量副本，从而隔离了多个线程对访问数据的冲突。因为每一个线程都有自己的变量副本，从而也就没有必要进行同步了。ThreadLocal提供了线程安全的对象的封装，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal中。

总的来说，对于多线程资源共享的问题，同步机制采用了“以时间换空间的”方式——访问串行化，对象共享化。而ThreadLocal则采用了“以空间换时间”的方式：访问并行化，对象独享化。前者仅提供一份变量，让不同的线程队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响。

ThreadLocal和线程同步机制是多线程访问的两种不同的方式，分别在不同的应用场景中发挥作用。

# 正则表达式

## 基本概念

正则表达式使用单个字符串来描述、匹配一系列符合某个句法规则的字符串搜索模式。

正则表达式是由一个字符序列形成的搜索模式。当你在文本中搜索数据时，你可以用搜索模式来描述你要查询的内容。正则表达式可以是一个简单的字符，或一个更复杂的模式。正则表达式可用于所有文本搜索和文本替换的操作。

## 语法

**/正则表达式主体/修饰符(可选)**

其中修饰符是可选的。

示例：

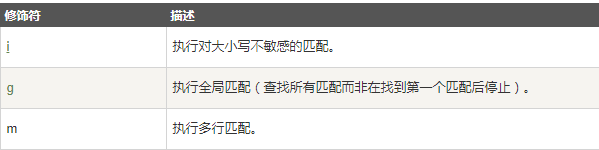
var patt = /runoob/i

解析：

/runoob/i  是一个正则表达式，runoob  是一个正则表达式主体 (用于检索)，i  是一个修饰符 (搜索不区分大小写)

### 修饰符

修饰符用于执行区分大小写和全局匹配。



### 方括号

方括号用于查找某个范围内的字符



### 元字符

元字符（Metacharacter）是拥有特殊含义的字符。



### 量词



\ 转义字符

^ 匹配输入字符串的开始位置

$ 匹配输入字符串的结束位置

\* 匹配前面的子表达式任意次，0次或者多次。

+ 匹配前面的子表达式一次或多次(大于等于1次）

？ 匹配前面的子表达式零次或一次

. 查找单个字符，除了换行和行结束符

## 在线正则表达式测试的网址

http://tool.oschina.net/regex/

## 常用的正则表达式

### （1）邮政编码的正则表达式

var pattern = /[1-9][0-9]{5}/; //必须是6位 第一位不能是0 必须是数字

var str = "100000";

alert(pattern.test(str));

### （2）检查文件的压缩包

var pattern = /^[\w]+\.(zip|rar|gz)$/; //|选择符要用分组符号包含起来

var str = "123.zip";

alert(pattern.exec(str));

### （3）删除首尾的空格

var pattern = /\s/g; //\s匹配空格

var str = "11 22 33";

alert(str.replace(pattern, ""));//112233

var pattern = /^\s+(.+?)\s+$/; //(.+)是贪婪模式，惰性模式

var str = " goo gle ";

var result = pattern.exec(str)[1];

var res = str.replace(pattern, "$1");//使用了分组模式

alert(result);

alert(res);

### （4）电子邮件的验证

var pattern = /^([\w]+)@([\w\-]+)\.([a-zA-Z]{2,4})$/;

var str = "992569344@qq.com";

alert(pattern.test(str));

### （5）汉字的正则表达式

^[\u4e00-\u9fa5]+$

### （6）简单匹配中文方法

/[^\u0000-\u00FF]/ (匹配非单字节字符 )

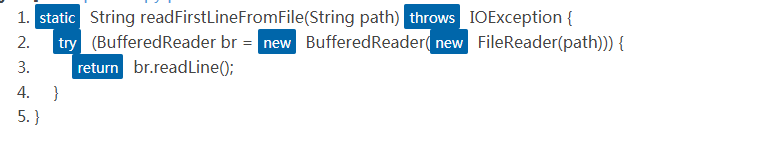
### （7）包含中文、大小写字母和数字

/^[\u4e00-\u9fa5a-zA-Z0-9]+$/ 中文和大小写英文字母数字都包括

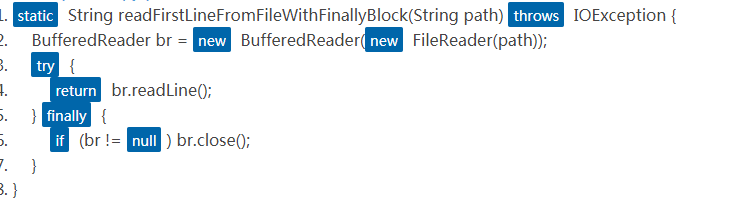
# Java7里try-with-resources

这个所谓的try-with-resources，是个语法糖。实际上就是自动调用资源的close()函数。这个语法是在jdk1.7之后出现的，包含jdk1.7。

示例：



可以看到try语句多了个括号，而在括号里初始化了一个BufferedReader。这种在try后面加个括号，再初始化对象的语法就叫try-with-resources。实际上，相当于下面的代码（其实略有不同，下面会说明）：



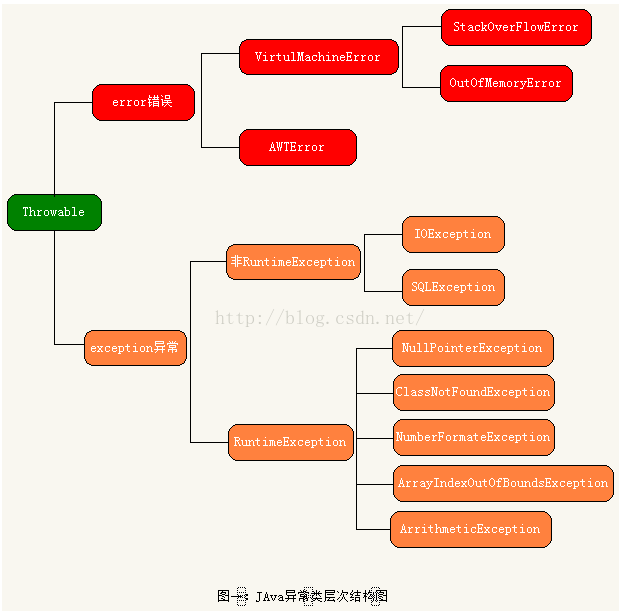
很容易可以猜想到，这是编绎器自动在try-with-resources后面增加了判断对象是否为null，如果不为null，则调用close()函数的的字节码。

说明：只有实现了java.lang.AutoCloseable接口，或者java.io.Closable（实际上继承自java.lang.AutoCloseable）接口的对象，才会自动调用其close()函数。有点不同的是java.io.Closable要求一实现者保证close函数可以被重复调用。而AutoCloseable的close()函数则不要求是幂等的。使用try-with-resources的语法可以实现资源的自动回收处理，大大提高了代码的便利性。用编绎器生成的字节码的角度来看，try-with-resources语法更加高效点。

# 异常

## 基本概念

1、Java中异常均继承自Throwable,其有两个重要的直接子类Error与Exception。下图是异常结构层次类。



2、Java错误error，大部分是由虚拟机爆出来的错误，是程序无法处理的错误,如OutOfMemoryError,当JVM需要更多内存空间而得不到满足时,就会爆出OutOfMemoryError。

3、Exception异常，其下分类很多，如可查异常与不可查异常，运行时异常与非运行时异常，基本概念一样，只是说法不同罢了。其有个重要的子类即RuntimeException运行时异常，其它直接子类都归为非RuntimeException，如IOException,SQLException等。

4、非RuntimeException是在代码书写时，编译器给你检查提示你要进行try catch或throws处理。如下异常所示：

IOException和SQLException

RuntimeException编译器不会帮你自动检查，当你运行程序时，虚拟机才会给你爆出错误让你去处理，这个往往是我们编码逻辑或不规范导致的。

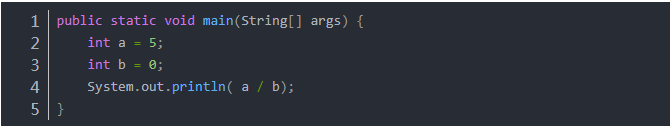
## 自定义异常

## 抛出异常的形式

### 系统自动抛出异常

当程序语句出现一些逻辑错误、主义错误或者类型转换错误时，系统会自动抛出异常。

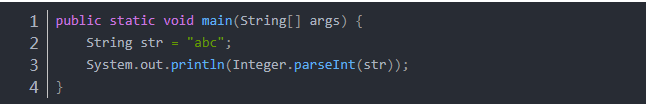
例一：



运行结果，系统会自动抛出ArithmeticException异常



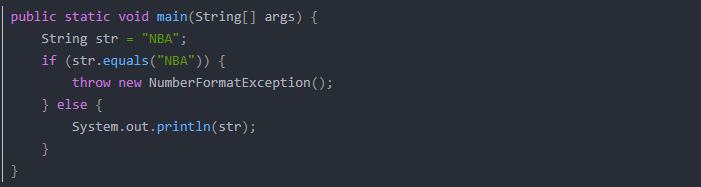
例二：



运行结果，系统会抛出NumberFormatException异常

### throw

throw语句是抛出一个异常，一般是在代码的内部，当程序出现某种逻辑错误时同程序主动抛出某种特定类型的异常。



运行结果，系统会抛出NumberFormatException异常

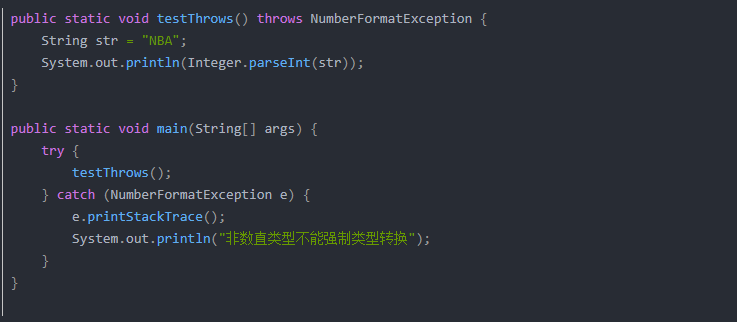


### throws

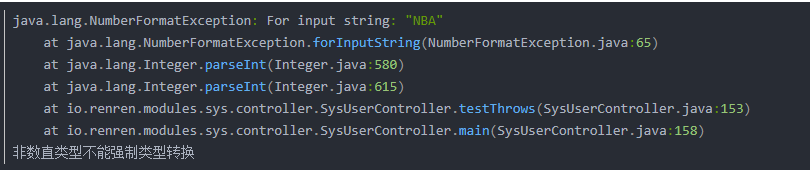
throws是方法可能会抛出一个异常(用在声明方法时，表示该方法可能要抛出异常)

public void function() throws Exception{......}

当某个方法可能会抛出某种异常时,用throws 声明可能抛出的异常，然后交给上层调用它的方法程序处理。



运行结果：



## throw与throws的比较

1、throws出现在方法函数头，而throw出现在函数体。

2、throws表示出现异常的一种可能性，并不一定会发生这些异常，throw则是抛出了异常，执行throw则一定抛出了某种异常对象。

3、两者都是消极处理异常的方式（这里的消极并不是说这种方式不好），只是抛出或者可能抛出异常，但不会由函数去处理异常，真正的处理异常由函数的上层调用处理。

## 编程习惯

1、在写程序时对可能会出现异常的部分通常要用try{…}catch{…}去捕捉它并对它进行处理。

2、用try{…}catch{…}捕捉了异常之后一定要在catch{…}中对其进行处理，那怕是最简单的一句输出语句或输入e.printStackTrace();

3、如果是捕捉IO输入输出流中的异常，一定要在try{…}catch{…}后加finally{…}把输入输出流关闭。

4、如果在函数体内用throw抛出了某种异常，最好要在函数名上加throws抛异常声明，然后交给调用它的上层函数进行处理。