# JAVA语言概述和环境搭建

## JAVA程序设计概述

**Java是由Sun Microsystems公司于1995年5月推出的Java**程序设计语言和Java平台（即JavaEE, JavaME, JavaSE）的总称。Java自面世后就非常流行，发展迅速，对C++语言形成了有力冲击。**Java 技术具有卓越的通用性、高效性、平台移植性和安全性**，广泛应用于个人PC、数据中心、游戏控制台、科学超级计算机、移动电话和互联网，同时拥有全球最大的开发者专业社群。

Java 语言是对软件开发技术有深远影响、应用前景广泛、具有丰富的类库、继承了C++传统（摈弃了某些不足）广泛使用的网络编程语言。Java语言的特性使它可以最大限度地利用网络。

Java并不是一种语言，Java是一个完整的平台，有一个庞大的库，其中包含了很多可重用的代码和一个提供诸如**安全性、跨操作系统的的可移植以及自动垃圾回收等服务的执行环境**。

[Java语言](http://www.so.com/s?q=Java%E8%AF%AD%E8%A8%80&ie=utf-8&src=internal_wenda_recommend_textn)的主要特点：  
　　1. 跨平台性  
　　所谓的跨平台性，是指[软件](http://www.so.com/s?q=%E8%BD%AF%E4%BB%B6&ie=utf-8&src=internal_wenda_recommend_textn)可以不受[计算机硬件](http://www.so.com/s?q=%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A1%AC%E4%BB%B6&ie=utf-8&src=internal_wenda_recommend_textn)和[操作系统](http://www.so.com/s?q=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&ie=utf-8&src=internal_wenda_recommend_textn)的约束而在任意[计算机](http://www.so.com/s?q=%E8%AE%A1%E7%AE%97%E6%9C%BA&ie=utf-8&src=internal_wenda_recommend_textn)环境下正常运行。这是软件发展的趋势和编程人员追求的目标。之所以这样说，是因为计算机硬件的种类繁多，操作系统也各不相同，不同的用户和公司有自己不同的计算机环境偏好，而软件为了能在这些不同的环境里正常运行，就需要独立于这些[平台](http://www.so.com/s?q=%E5%B9%B3%E5%8F%B0&ie=utf-8&src=internal_wenda_recommend_textn)。  
　　而在Java语言中， Java自带的[虚拟机](http://www.so.com/s?q=%E8%99%9A%E6%8B%9F%E6%9C%BA&ie=utf-8&src=internal_wenda_recommend_textn)很好地实现了跨平台性。 Java源程序[代码](http://www.so.com/s?q=%E4%BB%A3%E7%A0%81&ie=utf-8&src=internal_wenda_recommend_textn)经过编译后生成[二进制](http://www.so.com/s?q=%E4%BA%8C%E8%BF%9B%E5%88%B6&ie=utf-8&src=internal_wenda_recommend_textn)的[字节](http://www.so.com/s?q=%E5%AD%97%E8%8A%82&ie=utf-8&src=internal_wenda_recommend_textn)码是与平台无关的，但是可被Java虚拟机识别的一种[机器码](http://www.so.com/s?q=%E6%9C%BA%E5%99%A8%E7%A0%81&ie=utf-8&src=internal_wenda_recommend_textn)指令。 Java虚拟机提供了一个字节码到[底层](http://www.so.com/s?q=%E5%BA%95%E5%B1%82&ie=utf-8&src=internal_wenda_recommend_textn)硬件平台及操作系统的屏障，使得Java语言具备跨平台性。  
  
　　2. 面向对象  
　　面向对象是指以对象为基本[粒度](http://www.so.com/s?q=%E7%B2%92%E5%BA%A6&ie=utf-8&src=internal_wenda_recommend_textn)，其下包含属性和方法。对象的说明用属性表达，而通过使用方法来操作这个对象。[面向对象技术](http://www.so.com/s?q=%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1%E6%8A%80%E6%9C%AF&ie=utf-8&src=internal_wenda_recommend_textn)使得[应用程序](http://www.so.com/s?q=%E5%BA%94%E7%94%A8%E7%A8%8B%E5%BA%8F&ie=utf-8&src=internal_wenda_recommend_textn)的开发变得简单易用，节省代码。Java是一种面向对象的语言，也继承了面向对象的诸多好处，如代码扩展、代码复用等。  
  
　　3. 安全性  
　　安全性可以分为四个层面，即语言级安全性、编译时安全性、[运行时](http://www.so.com/s?q=%E8%BF%90%E8%A1%8C%E6%97%B6&ie=utf-8&src=internal_wenda_recommend_textn)安全性、[可执行代码](http://www.so.com/s?q=%E5%8F%AF%E6%89%A7%E8%A1%8C%E4%BB%A3%E7%A0%81&ie=utf-8&src=internal_wenda_recommend_textn)安全性。语言级安全性指Java的[数据结构](http://www.so.com/s?q=%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84&ie=utf-8&src=internal_wenda_recommend_textn)是完整的对象，这些封装过的[数据类型](http://www.so.com/s?q=%E6%95%B0%E6%8D%AE%E7%B1%BB%E5%9E%8B&ie=utf-8&src=internal_wenda_recommend_textn)具有安全性。编译时要进行Java语言和[语义](http://www.so.com/s?q=%E8%AF%AD%E4%B9%89&ie=utf-8&src=internal_wenda_recommend_textn)的检查，保证每个[变量](http://www.so.com/s?q=%E5%8F%98%E9%87%8F&ie=utf-8&src=internal_wenda_recommend_textn)对应一个相应的值，编译后生成Java类。运行时Java类需要类加载器载入，并经由字节码校验器校验之后才可以运行。 Java类在网络上使用时，对它的[权限](http://www.so.com/s?q=%E6%9D%83%E9%99%90&ie=utf-8&src=internal_wenda_recommend_textn)进行了设置，保证了被访问用户的安全性。  
  
　　4. 多线程  
　　多线程在操作系统中已得到了最成功的应用。多线程是指允许一个应用程序同时存在两个或两个以上的[线程](http://www.so.com/s?q=%E7%BA%BF%E7%A8%8B&ie=utf-8&src=internal_wenda_recommend_textn)，用于支持事务并发和[多任务](http://www.so.com/s?q=%E5%A4%9A%E4%BB%BB%E5%8A%A1&ie=utf-8&src=internal_wenda_recommend_textn)处理。 Java除了内置的[多线程技术](http://www.so.com/s?q=%E5%A4%9A%E7%BA%BF%E7%A8%8B%E6%8A%80%E6%9C%AF&ie=utf-8&src=internal_wenda_recommend_textn)之外，还定义了一些类、方法等来建立和管理用户定义的多线程。  
  
　　5. 简单易用  
　　Java[源代码](http://www.so.com/s?q=%E6%BA%90%E4%BB%A3%E7%A0%81&ie=utf-8&src=internal_wenda_recommend_textn)的书写不拘泥于特定的环境，可以用记事本、[文本编辑器](http://www.so.com/s?q=%E6%96%87%E6%9C%AC%E7%BC%96%E8%BE%91%E5%99%A8&ie=utf-8&src=internal_wenda_recommend_textn)等编辑软件来实现，然后将[源文件](http://www.so.com/s?q=%E6%BA%90%E6%96%87%E4%BB%B6&ie=utf-8&src=internal_wenda_recommend_textn)进行编译，编译通过后可直接运行，通过调试则可得到想要的结果。

## Java的发展史

* 1996年初sun发布了第一个版本jdk1.0。这个版本包括两个部分：运行环境(即JRE)和开发环境（即JDK）。运行环境包括**核心API、集成API、用户界面API、发布技术、Java虚拟机(JVM)**五个部分。开发环境包括编译java程序的编译器(即javac)。
* 1997年2月18发布了发布了jdk1.1版本**增加了JIT(既时编译器)。**JIT和传统编译器不同，传统编译器是编译一条，运行完成后废弃，而JIT会将经常用到的指令保存到内存中，当下次调用不需要重新编译。这种方式让jdk效率得到很大的提升。
* 1998年12月发布了jdk1.2，伴随着jdk1.2发布的还有jsp/servlet、EJB等规范。并将java分成了j2EE、j2SE、j2ME三个版本。
* J2ME:主要用控制移动设备和信息家电等有限存储的设备。
* J2SE:整个Java技术的核心和基础，它是J2ME和J2EE编程的基础，也是这本书主要介绍的内容。
* J2EE: Java技术屮应用最广泛的部分，J2EE提供了企业应用开发相关的完整解决方案。

这标志Java己经吹响了向企业、桌面和移动三个领域进军的号角，标志着Java己经进入Java 2时代，这个时期也是Java飞速发展的时期

* 2002年2月sun发布了jdk历史上最为成熟的版本：JDK1.4.此时由于Compaq、Fujitsu、SAS、Symbian、IBM等公司的参与，使jdk1.4成为发展最快的一个jdk版本。J**dk1.4有已经可以使用java实现大多数应用了**。
* 2004年10月sun发布了万众期待的Jdk1.5.同时sun将jdk改名为JavaSE5.0，J2EE、J2ME、也改名为JavaEE、JavaSE。**JDK1.5增加了泛型、增强的for语句，可变数量形参，注解，自动拆箱和装箱等功能**，同时发布了企业级平台规范。
* 2005年6月，在Java One大会上，Sun公司发布了Java SE 6。
* **2009年，甲骨文公司宣布收购Sun** 。2010年，Java编程语言的共同创始人之一詹姆斯·高斯林从Oracle公司辞职。
* 2011年7月28日 oracle公司发布了**JavaSE7---引入了二进制整数、支持字符串的Switch语句，菱形语法，多异常捕获，自动关闭资源的try语**句等特性
* 2014年3月18日oracle发布了J**avaSe8**这次版本升级带来了全新的**Lambda表达式**，除此以外Java8还增加了大量的新特性。

## Java的运行机制

* Java语法比较特殊，由于java语言编写程序需要经过编译步奏，但这个编译步奏并不会生成特定平台的机器码，而是生**成一种与平台无关的字节码(也就是.class)文件。**这种字节码必须使用java解释器来解释执行。也就是说Java程序的执行必须经过**先编译后运行两个步骤。**
* Java语言里**负责解释执行字节码文件的是Java虚拟机**，即JVM (Java Virtual Machine)。JVM是可运行Java字节码文件的虚拟计算机。所有平台上的JVM向编译器提供相同的编程接口，而编译器只需要面向虚拟机，生成虚拟机能理解的代码，然后由虚拟机来解释执行。在一些虚拟机的实现中，还会将虚拟机代码转换成特定系统的机器码执行，从而提高执行效率。
* JVM是一个抽象的计算机，和实际计算机一样，它具有指令集并使用不同的存储区域，它负责执行指令，还有管理数据、内存和寄存器

## 开发环境的搭建

在开发java程序之前需要先准备一些工作，就是在计算机上按照开发java的环境，需要安装和配置JDK。

### 下载和安装Java8的JDK

JDK的全称是Java SE Devlopmem Kit,即Java标准版开发包，是Sun提供的一套用于开发Java 应用程序的开发包，它提供了编译、运行Java程序所需的各种工具和资源，包括java编译器、java运行时环境，以及常用的Java类库等。这里又涉及一个概念：Java运行时环境，它的全称是Java Runtime Environment，因此也被称为JRE, 它是运行Java程序的必须条件。

**JDK和JRE的区别？**

简单的说，JRE包含了JVM。JVM是运行java程序的核心虚拟机，而运行java程序不仅需要核心虚拟机还需要其他的类加载器、字节码校验器、以及大量的基础类库，JRE除了包含jvm之外还包含了java程序的其他环境的支持。

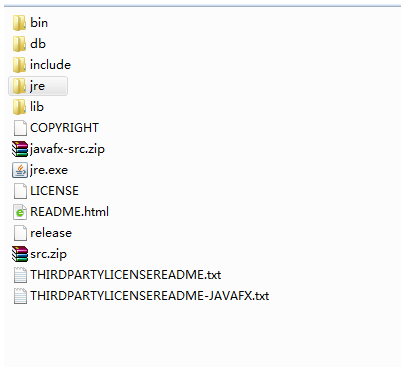
1. 官网下载Java8的jdk

http://www.oracle.com/technetwork/java/javase/downloads/index.html

2、傻瓜式的安装 点击下一步即可

安装路径不要有中文、空格或者特殊符号。

1. 查看安装目录结构



* bin:该路径下存放了 JDK的各种工具命令，常用的javac、java等命令就放在该路径下
* db:该路径足安装JavaDB的路径。
* include:—些平台特定的头文件。
* jre:该路径下安装的就是运行Java程序所必需的JRE环境
* lib:该路径存放的JDK 工具命令的实际执行程序，可以使用WinRAR打开lib路径的 tools.jar文件
* Javafx-src.zio:该压缩文件存放的就是JavaFX所以的核心类库的源代码。
* src.zip该压缩包下存放的就是**Java所有核心类库的源代码**
* Readme和license等说明文档

4、配置环境变量

我的电脑--->右键属性--->高级系统设置--->高级—-->环境变量

**在一台电脑上配置java环境，path起什么作用？如何配置？**

答：a）path的作用是在DOS环境下，能在任意位置使用JDK目录中bin文件夹中的 可执行程序，来编译执行java程序。

点击path选择编辑 在最后添加一个英文的; 添加如下内容

Path=C:\guigu\Java\jdk1.8.0\_151\bin

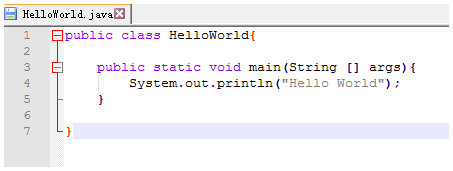
点击新建 JAVA\_HOME =C:\guigu\Java\jdk1.8.0\_151

5、验证是否安装正确

Java javac java –version 验证是否安装正确

## 编写第一个Java程序HelloWorld

### 编辑java源代码



### 编译运行程序

* 编译Java程序 需要javac命令

语法规则如下 ：

Javac –d destdir srcFile

可以简化为javac srcFile 举例javac HelloWorld.java

* 运行java程序

Java 类名 举例 java HelloWorld

## Java程序的基本规则

### Java程序的组织形式

* Java程序是一种纯粹的面向对象的程序设计语言，因此Java程序必须以类（class)的形式存在， 类（class)是Java程序的最小程序单位。Java程序不允许可执行性语句、方法等成分独立存在，**所存的程序部分都必须放在类定义里。**

上面的HelloWorld.java程序是一个简单的程序，但还不是最简单的Java程序，最简单的Java程序是只包含一个空类定义的程序。

class Test { }

这是一个最简单的java程序，这个程序定义了一个Test类，这个类里没有任何的类成分，是一个空类，但这个Java程序是绝对正确的，如果使用javac命令来编译这个程序，就知道这个程序可以通过编译，没有任何问题。

但如果使用java命令来运行上而的Test类，则会得到如下错误提示：

错误：在类Test中找不到main方法，请将main方法定义为： public static void main(String[) args)

上面的错误提示仅仅表明：这个类不能被java命令解释执行，并不表示这个类是错误的。实际上， Java解释器规定：如需某个类能被解释器直接解释执行，则这个类里必须包含main方法，而且main 方法必须使用public static void来修饰，且main方法的形参必须是字符串数组类型（String[] args是字 符串数组的形式）。也就是说，main方法的写法几乎是固定的。Java虚拟机就从这个main方法开始解释执行，因此，main方法是Java程序的入口。对于那些不包含main方法的类，也是有用的类。对于一个大型的Java程序而言，往往只需要一个 入口，也就是只有一个类包含main方法，而其他类都是用于被main方法直接或间接调用

### Java源文件的命名规则

Java源文件命名不可以随意命名，必须遵循如下规则:

* Java源文件的后缀名必须是.java
* 必须是一个合法的标识符 以英文的形式体现
* 是**以驼峰命名法**  单词首字母都是大写的

## 初学者容易犯的错误

### Classpath环境变量的问题

由于历史原因，几乎所有的图书和资料中都介绍必须设置这个环境变量。实际上，正如前面所介绍 的，如果使用1.5以上版本的JDK,完全可以不用设置这个环境变摄。如果不设置这个环境变量，将可以正常编译和运行Java程序。

相反，如果有的读者看过其他Java入门书籍，或者参考过网上的各种资料（网络是一个最大的资源库，但网络上的资料又是鱼龙混杂、良莠不齐的。网络上的资料很多都是转载的，只要一个人提出一个错误的说法，这个错误的说法可能被成千上万的人转载，从而看到成千上万的错误说法），可能总是习惯设置CLASSPATH环境变量。

设置CLASSPATH环境变量没有错，关键是设置错了就比较麻烦了。正如前面所介绍的，如果没有 设置CLASSPATH环境变量，Java解释器将会在当前路径下搜索Java类，因此在HelloWorld.class文件 所在的路径运行java HelloWorld将没有任何问题；但如果设置了 CLASSPATH环境变量，Java解释器 将只在CLASSPATH环境变量所指定的系列路径中搜索Java类，这样就容易出现问题了。

由于很多资料上提到CLASSPATH环境变量中应该添加dt.jai和tools.jar两个文件，因此很多读者 会设置 CLASSPATH 环境变量的值为：D:\Java\jdkl.8.0\_05\lib\dt.jar;D:\Java\jdkl.8.0\_05\lib\tools.jar，这 将导致Java解释器不在当前路径下搜索Java类。如果此时在HelloWorld.class文件所在的路径运行java

上面的错误是一个典型错误：找不到类定义的错误，通常都是由CLASSPATH环境变量设置不正确 造成的。因此，如果读者要设置CLASSPATH环境变量，一定不要忘记在**CLASSPATH环境变量中增加 一点（.），强制Java解释器在当前路径下搜索Java类。**

### 大小写问题

前面已经提到：**Java语言是严格区分大小写的语言。**但由于大部分读者都是Windows操作系统的忠实拥护者，因此对大小写问题往往都不够重视（Linux平台是区分大小写的）。

例如，有的编写的Java程序里的类是HelloWorld,但当他运行Java程序时，运行的则是java helloworld这种形式——这种错误的形式有很多种(对的道路只有一条，但错误的道路则有成千上万条）。 总之，就是java命令后的类名没有严格按Java程序中编写的来写，可能引起系统的错误。

因此必须注意：在Java程序里，HelloWorld和helloworld是完全不同的，必须严格注意 Java程序里的大小写问题。

不仅如此，编写Java程序时，必须严格注意Java程序中每个单词的大小写， 不要随意编写。例如class和Class是不同的两个词，class是正确的，但如果写成Class，则程序无法编译通过。实际上，Java程序中的关键字全部是小写的，无须大写任何字母。

### 何时开始使用IDE工具

对于 Java语言的初学者而言，这里给出一个忠告:不要使用任何IDE工具来学习Java编程,Windows 平台上可以选择记事本，Linux平台上可以选择使用VI工具。如果嫌Windows上的记事本的颜色太单调，可以选择使用EditPlus或者Notepad++。

在多年的程序幵发生涯中，常常见到一些所谓的Java程序员，他们怀揣一本Edipse从入门到精通， 只会单击几个“下一步”按钮就敢说自己精通Java ，实际上他们连动手建一个Web应用都不会，连 Java的Web应用的文件结构都搞不清楚,他们习惯了在Edipse或者NetBeans 工具里通过单击鼠标来新建Web应用，而从来不去看这些工具为我们做了什么。

曾经看到一个在某培训机构己经学习了 2个月的学生，连extends这个关键字都拼不出来，不禁令人哑然，这就是依赖IDE工具的后果。

还见过许多所谓的技术经理，他们来应聘时往往滔滔不绝，口若悬河。他们知道很多新名词、新概念，但机试往往很不乐观：说没有IDE工具，提供了 IDE 工具后，又说没文挡，提供了文档又说不能上网，提供上网乂说不是在自己的电脑上，没有代码参考……他们的理由比他们的技术强！

可能有读者会说，程序员是不需要记那些简单语法的！关于这一点也有一定的道理。但问题是：没有一个人会在遇到1+1 = ?的问题时说，我要査一下文档！对于一个真正的程序员而言，大部分代码就在手边，还需要记忆？

当然IDE工具也有其优势，在项目管理、团队开发方面都有不可比拟的优势。但并不是每个人都可以使用IDE工具的。

那么何时开始使用IDE工具呢？标准是：如果你还离不开这个IDE 这个工具，那么你就不能使用这个 IDE工具：只有当你分清楚在IDE工具里单击每一个菜单，单击每一个按钮……IDE工具在底层为你做的每个细节时，才可以使用IDE工具！

如果读者有志成为一名优秀的Java程序员，那么，到了更高层次后，就不可避免地需要自己开发IDE 7工具的插件（例如开发Eclipse插件），定制自己的IDE工具，甚至负责开发整个团队的开发平台，这些都要求开发者对Java开发的细节非常熟悉。因此，不要从IDE工具开始学习。

### md命令

cd \进入到根目录

D: 进入到指定磁盘目录

cd WorkSpace 进入到指定的目录中

dir 列出当前文件夹下所有的文件以及文件夹

cd ..退回上一级目录

md 创建目录

rd 删除目录 目录必须为空

del a.txt 删除指定的文件

del \*.txt 删除所有的txt文件

exit 退出

# Java语法基础

## Java语法基础

### 分隔符

Java语言里的分号（;)、花括号（{}）、方括号([])、圆括号（（））、空格、圆点（.）都有特殊的分隔作用，因此被统称为分隔符

1.分号

Java语言里对语句的分隔不是使用回车来完成的，Java语言采用分号（;）作为语句的分隔，因此 每个Java语句必须使用分号作为结Java程序允许一行书写多个语句，每个语句之间以分号隔开即可：一个语句也可以跨多行，只要在最后结束的地方使用分号结束即可。

不仅如此，虽然Java语法允许一行书写多个语句，但从程序可读性角度来看，应该避免在一行书 写多个语句

2.花括号

花括号的作用就是定义一个代码块，一个代码块指的就足“{和“}所包含的一段代码，代码块在逻辑上是一个整体。对Java语言而言，类定义部分必须放在一个代码块中，方法体部分也必须放在一个代码块里。除此之外，条件语句中的条件执行体和循环语句中的循环体通常也放在代码块里。 花括号一般是成对出现的，有一个“{则必然有一个“}，反之亦然。

### 标识符和关键字

Java语言也和其他编程语言一样，使用标识符作为变量，对象的名字，也提供了系列关键字用以实现特别的功能。

#### 标识符

标识符就是用于给程序中变量、类、方法命名的符号。Java 语言的标识符必须以字母、下画线（\_)、 美元符（$)开头，后面可以跟任意数目的字母、数字、下画线（\_）和美元符（$)。此处的字母并不局限于 26个英文字母，而且可以包含中文字符、日文字符等。

由于Java语言支持Unicode 6.2.0字符集，因此Java的标识符可以使用Unicode 6.0.0所能表示的多种语言的字符。 java语言是区分大小写的，因此abc和Abc是两个不同的标识符。

使用标识符时，需要注意如下规则

>标识符可以由字母、数字、下画线（\_)和美元符（$)组成，其中数字不能打头。

>标识符不能是Java关键字和保留字，但可以包含关键字和保留字。

>标识符不能包含空格。

>标识符只能包含美元符（$),不能包含@、#等其他特殊字符。

下面那些是合法的标识符：

UserName √

3dmax ×

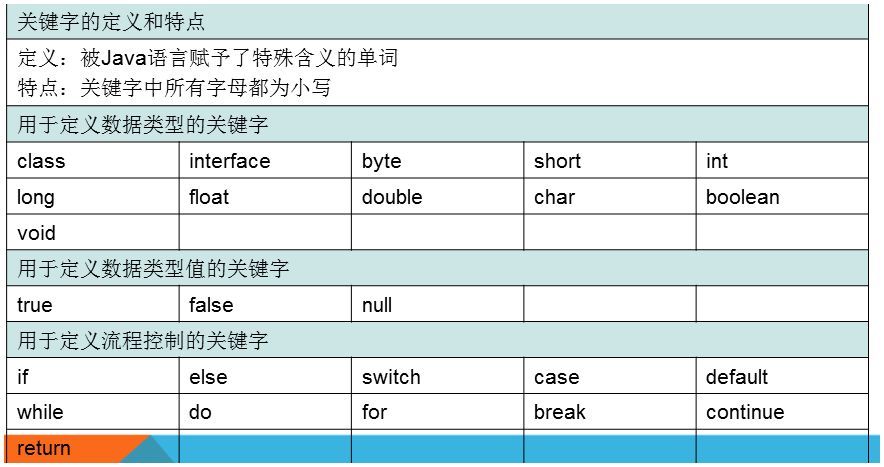
room# ×

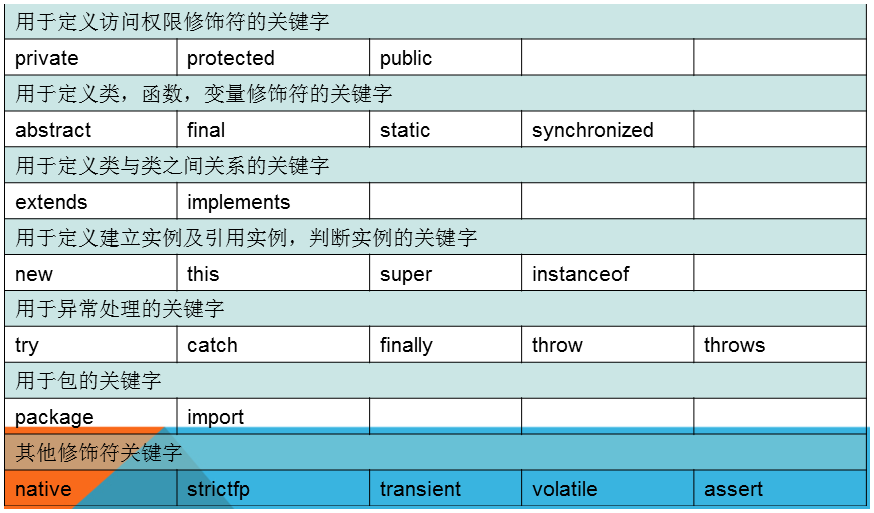
class ×

classA √

#### 关键字和保留字

Java语言里有一些特殊用途的单词被称为关键字，当定义标识符的时候不要使用系统的关键字即可。常见的关键字如下:**关键词中所有字母都是小写**





### 注释

永远不要过于相信自己的理解力！当你思路通杨，进入编程境界时，你可以很流程地实现某个功能.但这种流畅时可能因为你当时处于这种开发思路中，为了在再次阅读这段代码时。还能找回当初编写这段代码的思路，建议添加注释！

**可读性第一，效率第二！**在那些“古老”的岁月里.编程是少数人的专利他们随心所欲地写程序，他们以追逐程序执行效率为目的.但随者软件行业的发展，人们发现仅有技术极客编程满足不了日益增长的软件需求，越来越多的人加入了编程队伍，并引入了工程化的方式来管理软件开发，这个时候，软件开发变成团队协同作战，团队成员的沟通变得很重要，因此， —个人写的代码，想要被整个团队的其他人所理解：而且，随着硬件设备的飞速发展，程序的 可读性取代执行效率变成了第一考虑的要素。

代码即文档！很多刚刚学完学校软件工程课程的学生会以为：文档就是Word文捫！实际上源代码就是程序文档的重要组成部分，在想着把各种软件相关文档写规范的同时不要忘了把软件里最重要的文档——源代码写规范！

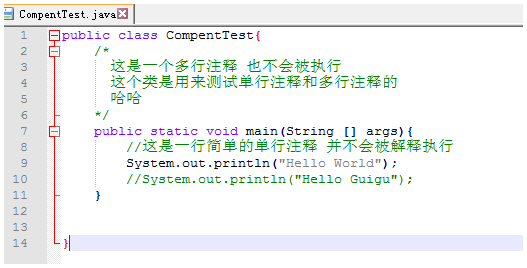
程序注释源代码的一个重要部分.对于一份规范的程序源代码而言，注释应该占到总代码的1/3 以上.几乎所有的编程语言都提供了添加注释的方法。一般的编程讲言都提供了基本的单行注释和多行 注释，Java语言也不例外，除此之外，Java语言还提供了一种文档注释。Java语言的注释一共分为三种 类型。

单行注释

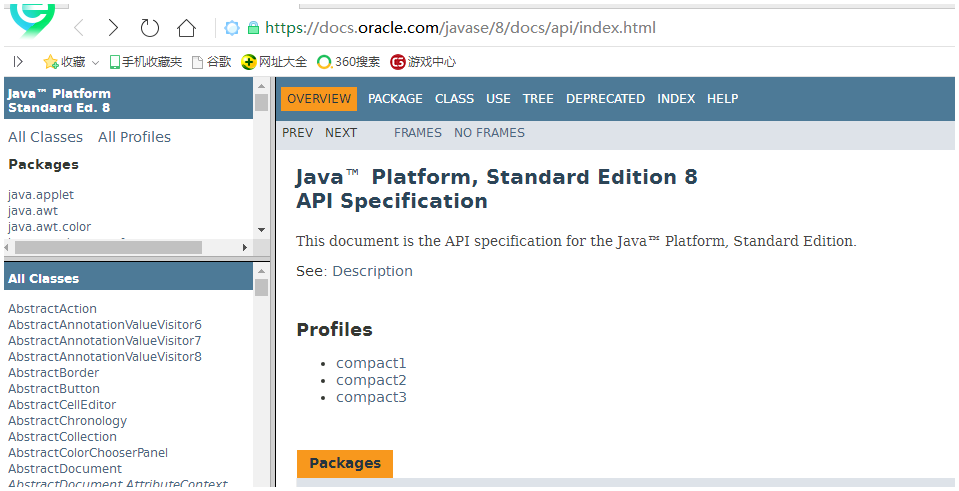
多行注释

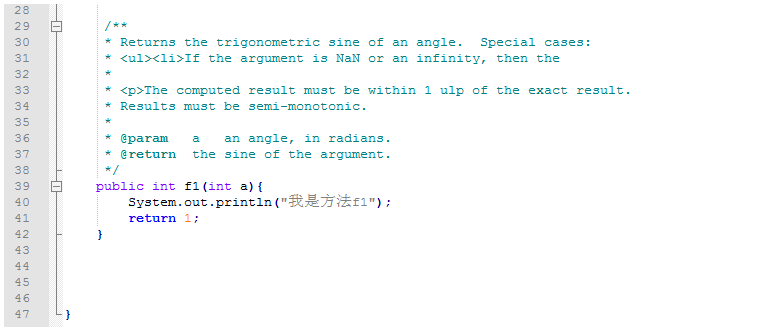
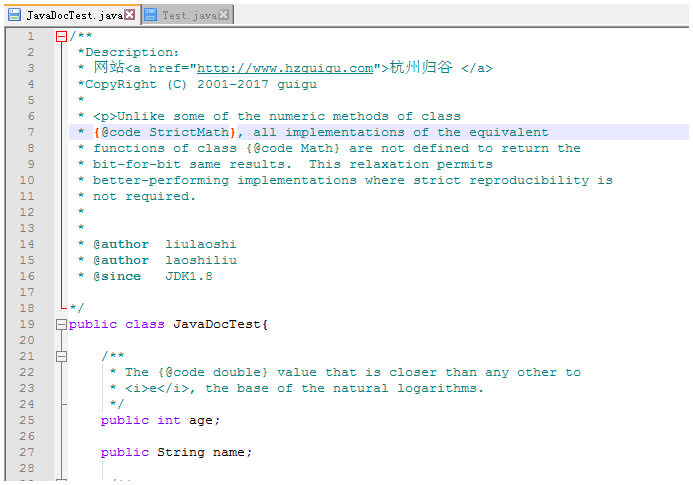
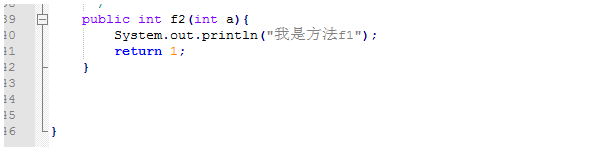
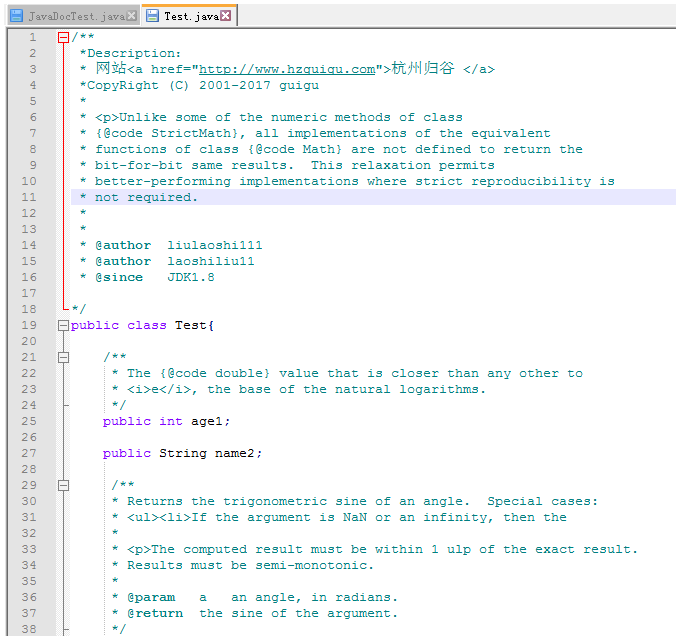
文档注释

#### 单行注释和多行注释



#### 文档注释





生成javadoc的命令规则如下：

**javadoc -d apidoc \*Test.java**

参数-d 是指定生成api的目录

## 基本数据类型

Java语言是**强类型 (strongly typed)语言**，强类型包含两个方面的含义：**①**所有的变量必须先声明、后使用;**②**指定类型的变量只能接受类型之匹配的值。

这意味着每个变量和每个表达式都有一个在**编译时就确定的类型**。类型限定了一个变量能被陚的值.限制了一个表达式可以产生的值，限制了在这些值上可以进行的操作，并确定了这值操作的含义强类型语言可以在编译时进行严格的语法检査.从而减少编程错误。

Java语言支持的类型分为两类**：基本类型和引用类型。**

* 基本类型包含8种类型byte、short 、int、 long、float、double、boolean、char
* 引用类型包括类、接口、数组类型

byte 1个字节 8位 2的7次方减一

short 2个字节 16位 2的15次方减一

整形类型

int 4个字节 32位

long 8个字节 64位

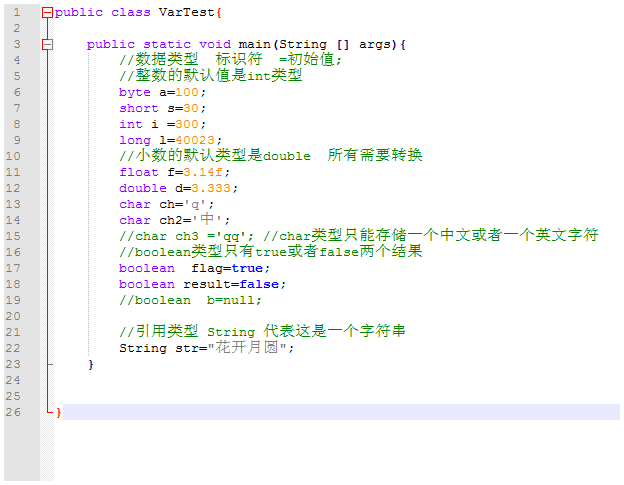
基本类型 字符型 char 2个字节 16位

float 4个字节 32位

浮点型

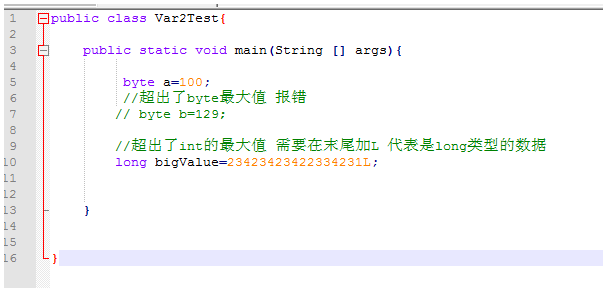
double 8个字节 64位

布尔型 Boolean 1位 true或者false



### 整形

* Byte 一个byte类型在内存中占有8位 大小范围是 -128(-2的7方) ~ 127（2的7次方减一）





### 字符型

字符型通常表示单个的字符，字符必须使用单引号。

A对应的ASSCII码是65 a--->97

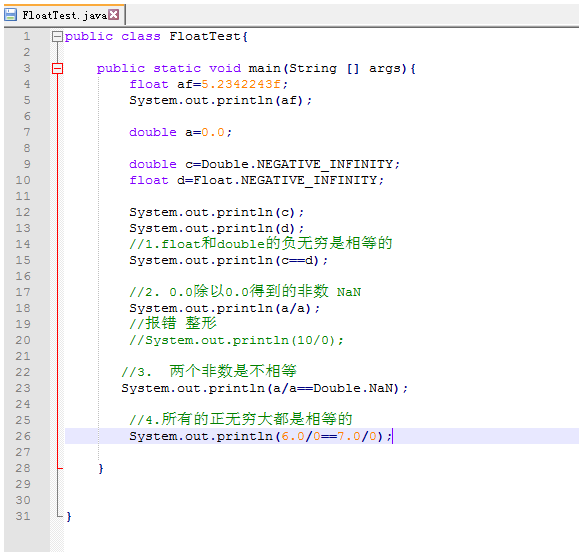


### 浮点型

Java语言里浮点型默认是double类型，如果希望把一个浮点型当前float处理需要在字符的值之后添加f或者F。

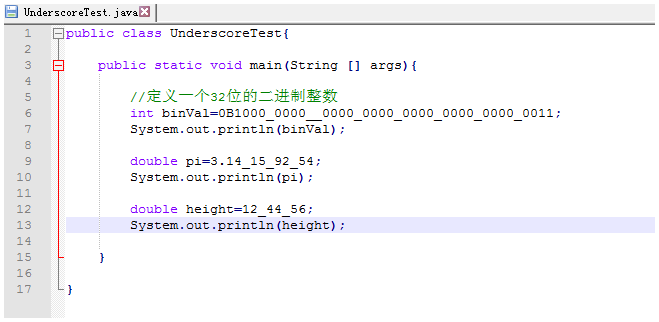
Java还提供了三个特殊的浮点值。 正无穷大，负无穷大,非数

double和float用NEGATIVE\_INFINITY负无穷大，用POSITIVE\_INFINITY表示正无穷大。

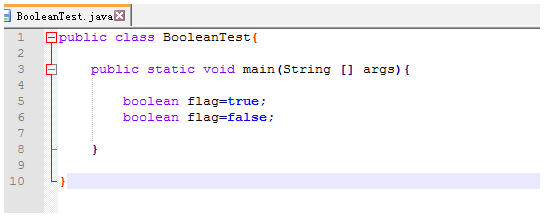


#### 数值中使用下划线分割

当程序中数值位数特别多，程序员容易看错，为了解决这种问题，在jdk1.7之后引入了一个新功能。程序中允许使用下划线分割。



### 布尔类型

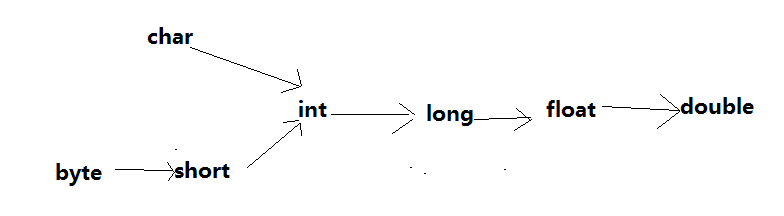


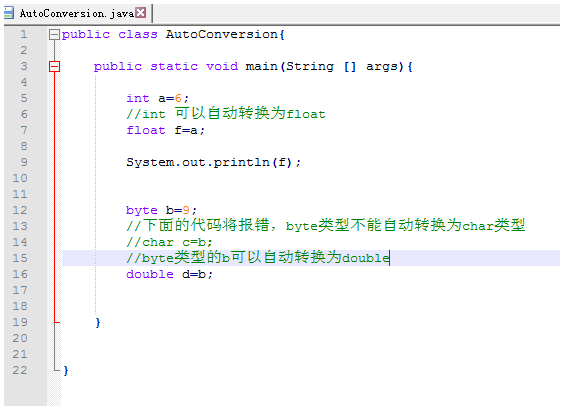
## 基本类型的类型转换

在java程序中不同的基本数据类型需要经常进行转换，java语言提供了7种数值类型之间是可以进行转换。有两种转换方式，自动类型转换和强制类型转换。

### 自动类型转换

Java所有的数值变量是可以相互转换，如果系统支持把某种基本类型的数值直接赋值给另外一种基本类型，那么这种方式成为自动类型转换。当把一个较小的数据类型直接赋值给一个较大的数据类型可以直接转换。

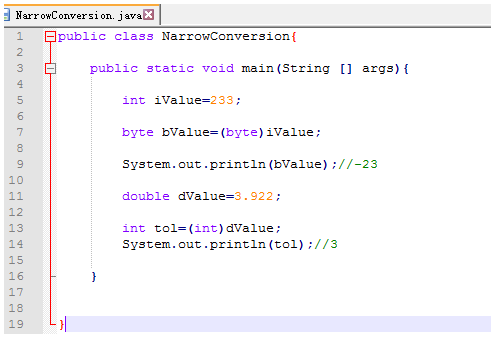


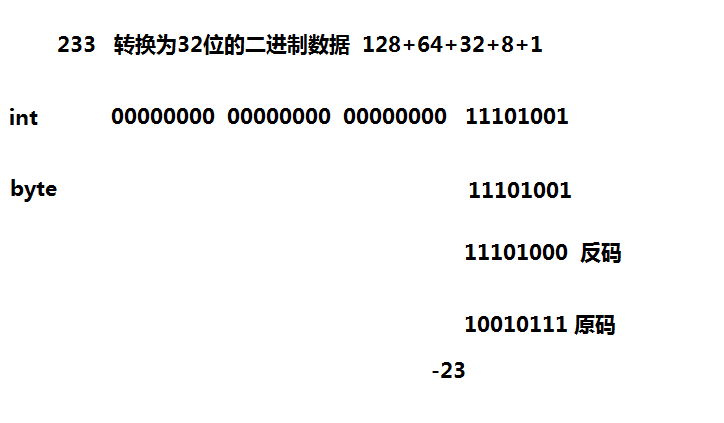




### 强制类型转换

把较大的数据类型赋值给较小的数据类型需要强制类型转换





所有的操作都是补码

正数的原码=补码

负数的 补码=原码 取反 +1

负数的 原码=补码-1 取反

### 表达式类型的自动提升

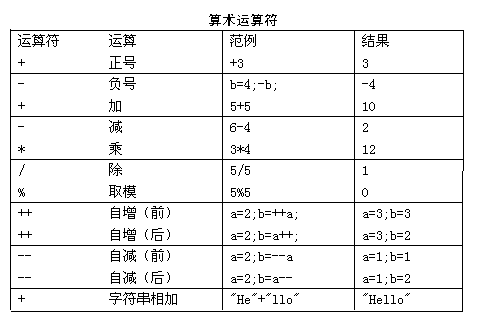
当一个算数表达式中包含多个基本类型的值时，整个算数表达式的数据类型自动发生提升，java中定义了以下提升规则:

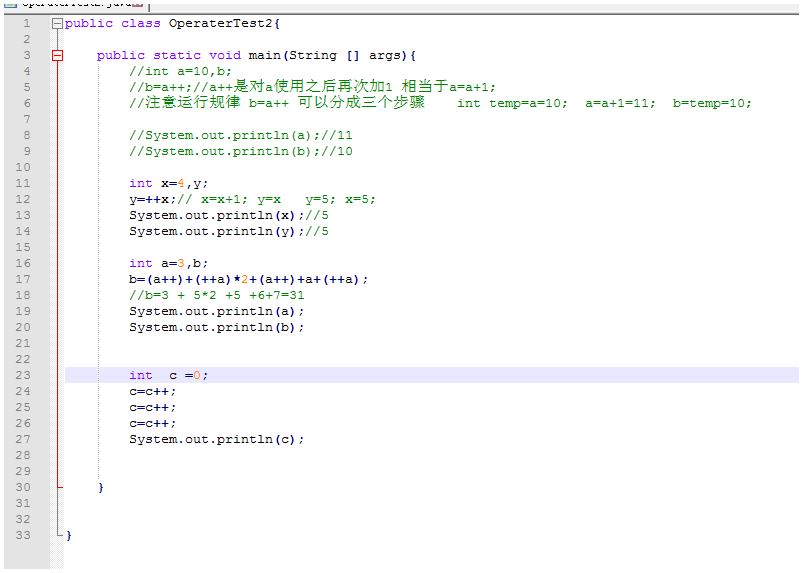
* 所有的byte、short和char自动提升到int
* 整个算数表达式的类型和最高类型保持一致。



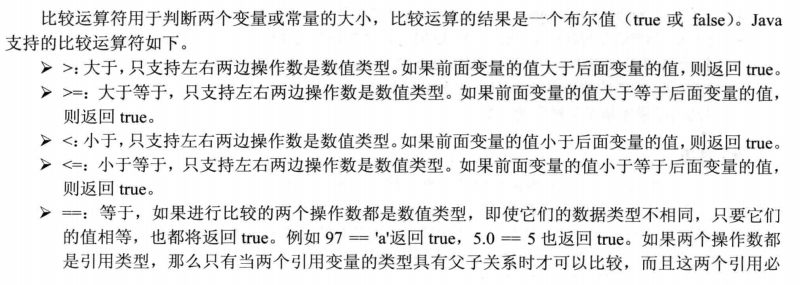
## 运算符

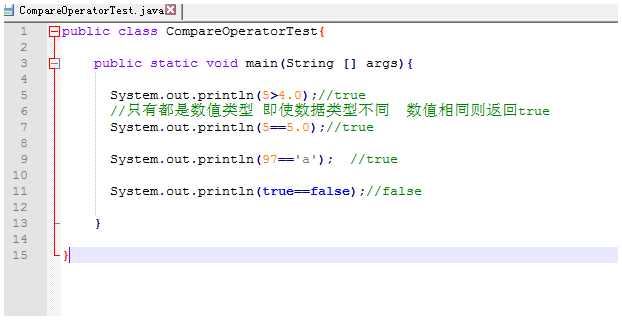
### 算术运算符





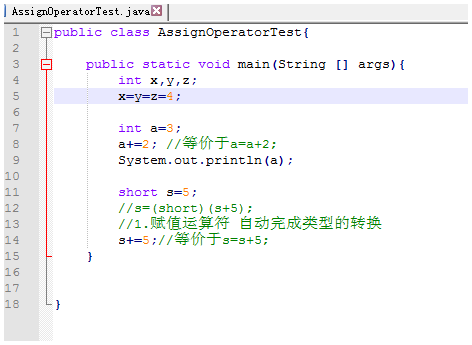
### 比较运算符



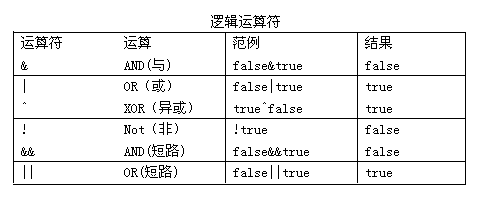


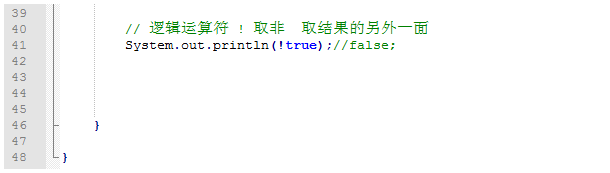
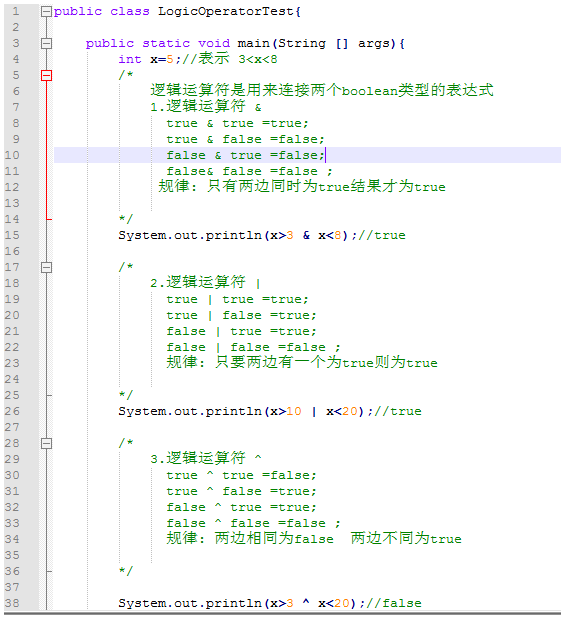
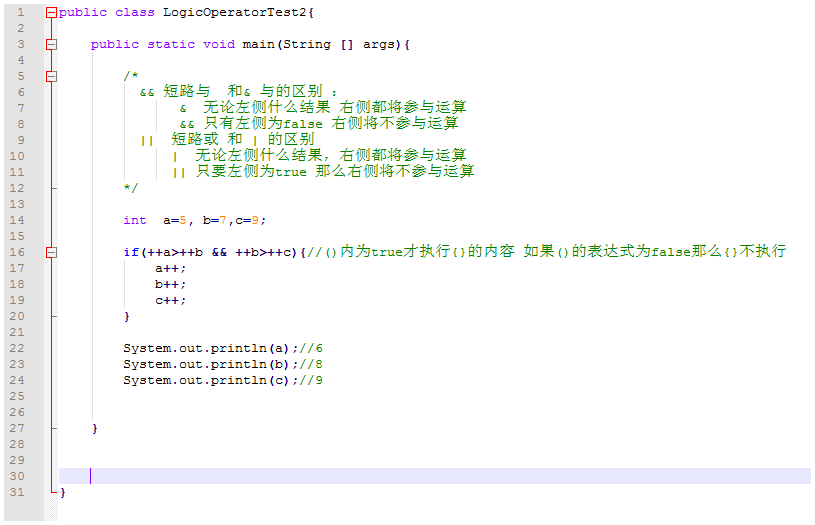
### 赋值运算符

|  |  |  |  |
| --- | --- | --- | --- |
| 运算符 | 举例 | 运算符 | 举例 |
| += | x+=a; 等价于 x=x+a | /= | x/=a;等价于x=x/a; |
| -= | x-=a; 等价于 x=x-a | %= | X%=a;等价于x=x%a; |
| \*= | x\*=a;等价于 x=x\*a; |  |  |



### 逻辑运算符



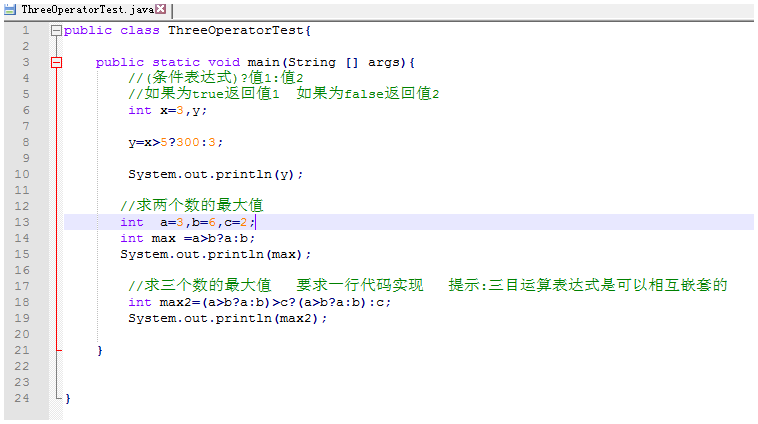
### 三元运算符

格式

(条件表达式)?表达式1：表达式2；

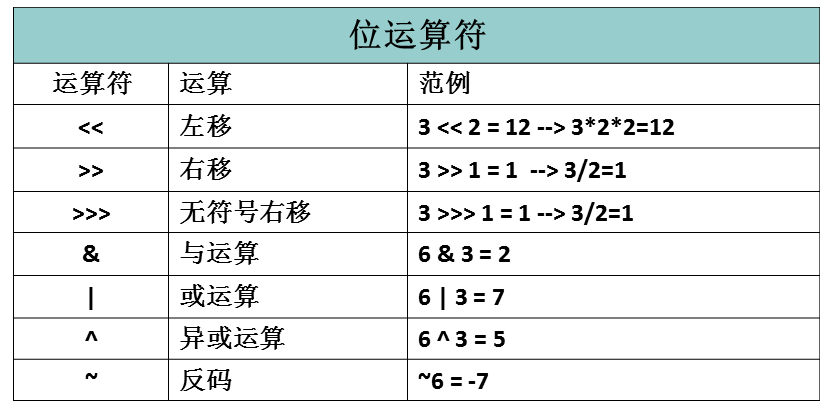
如果条件为true，运算后的结果是表达式1；

如果条件为false，运算后的结果是表达式2；

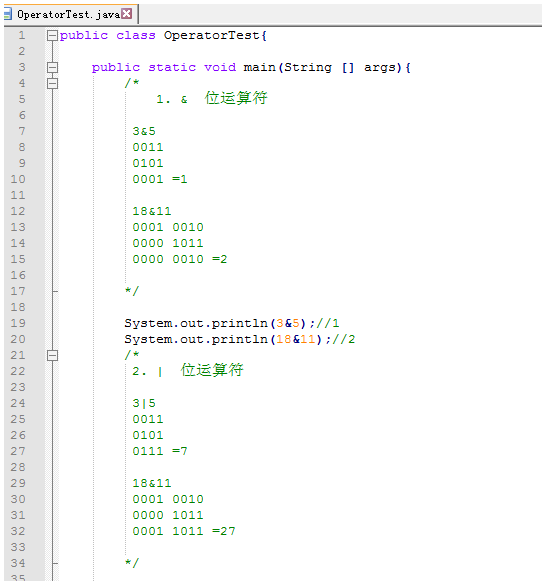


### 位运算符

位运算是直接对二进制进行运算。

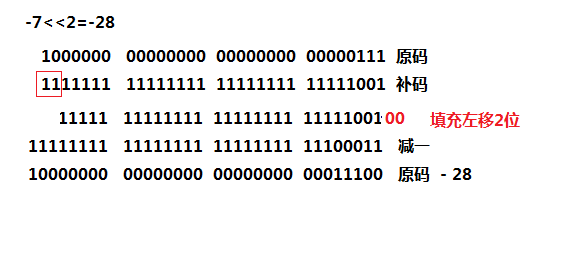


|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 第一个运算符 | 第二个运算符 | 按位& | 按位| | 按位^ |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

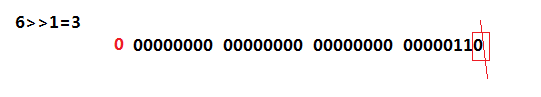


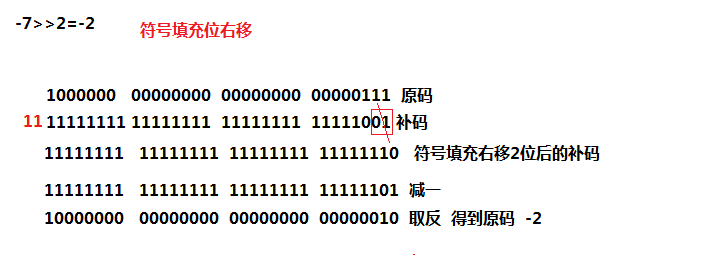


**1、<< 无符号左移**：相当于 乘以2的倍数 3<<3 等价于3\*2的3次方 = 24



**2、>> 符号填充位右移**  相当于 除以2的倍数 6>>1 =6/2的1次方法 =3 6>>2 =1





所有的位运算的操作都是在补码上进行的

正数的原码=补码

负数的 补码=原码 取反 +1

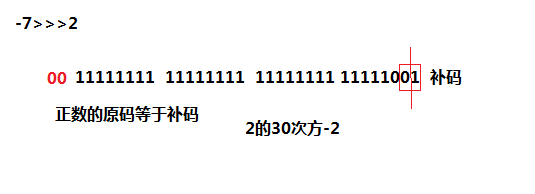
负数的 原码=补码-1 取反

取反符号位不变

取非符号位发生变化

**3、>>> 无符号右移**

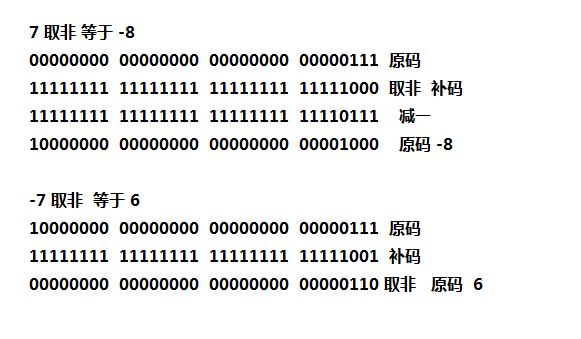
-7>>>2

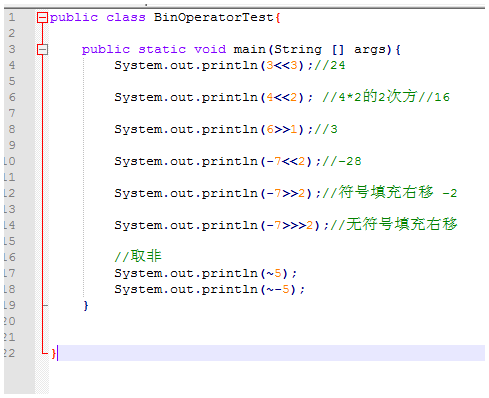


**4、取非**

7 取非 是 -8

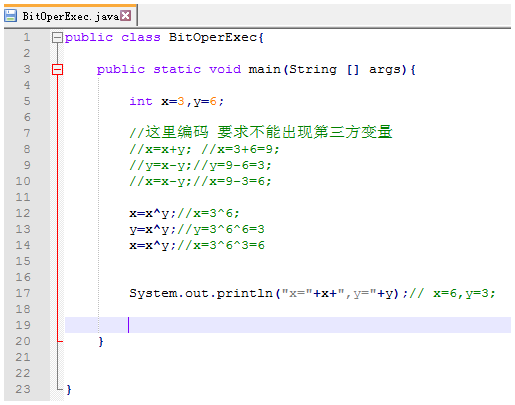
-7 取非 是 6



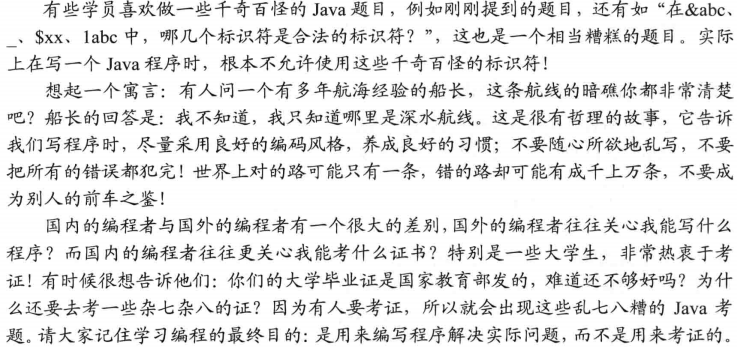


课堂练习：

1. 最有效率的方式 计算2乘以8 2<<3
2. 两个整数变量进行交互(不需要第三方的变量)



小提示

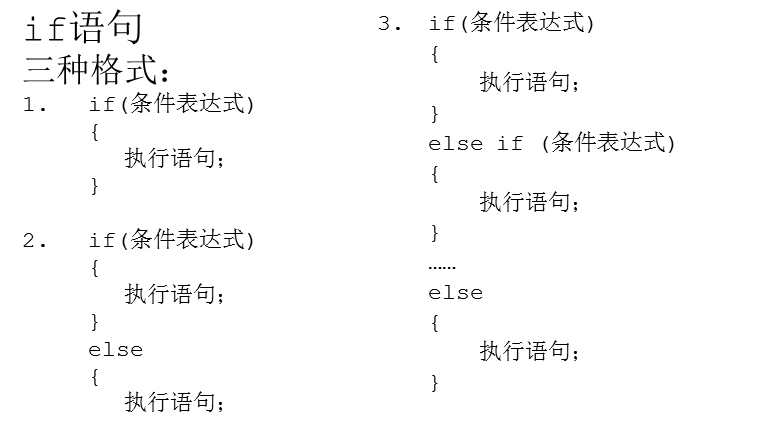


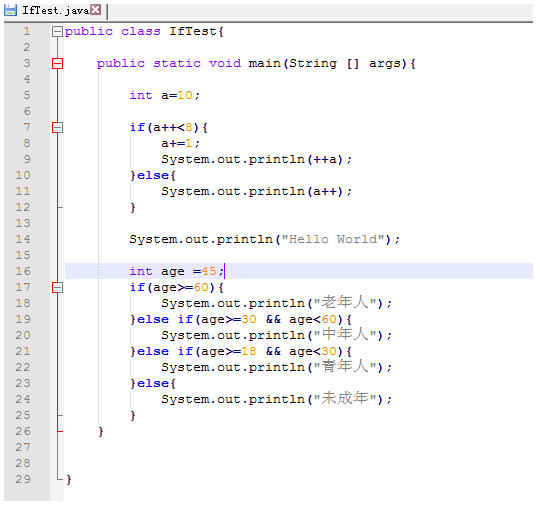
## 流程控制

不论哪一种编程语言，都会提供两种基本的流程控制结构：分支结构和循环结构。其中**分支结构用于实现根据条件来选择性地执行某段代码**，循环结构则用于实现**根据循环条件重复执行某段代码**。Java 同样提供了这两种流程控制结构的语法，Java提供了 if和switch两种分支语句，并提供了 while、do-while 和for三种循环语句。除此之外，JDK5还提供了一种新的循环：foreach循环，能以更简单的方式来遍 历集合、数组的元素。Java还提供了 break和continue来控制程序的循环结构。

### 分支结构

#### IF语句





#### Switch语句

Switch语句由一个控制表达式和多个case标签组成的。Switch后面的表达式的数据类型只能是byte 、short、int 、char 四种类型。在jdk1.7以后新增了String也可以

switch语句

格式：

switch(表达式)

{

case 取值1:

执行语句；

break；

case 取值2:

执行语句；

break；

…...

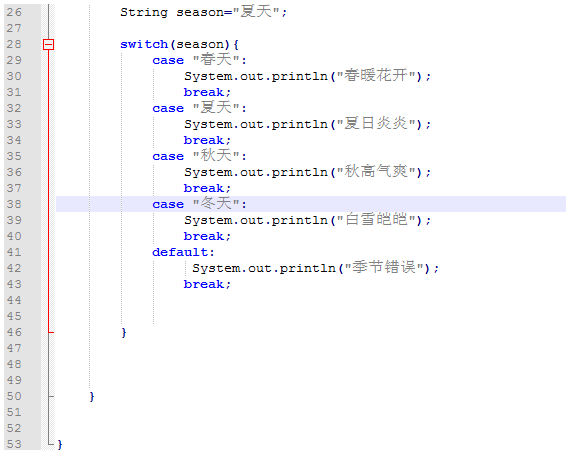
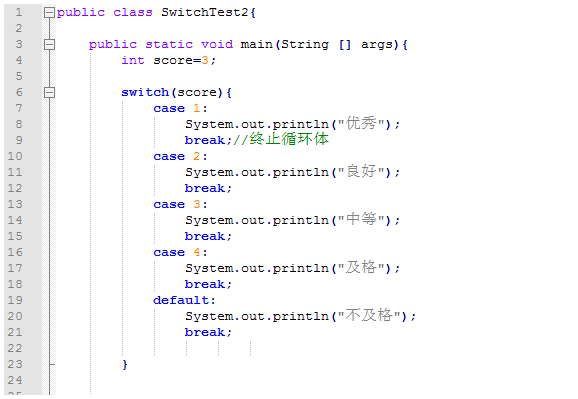
default:

执行语句；

break；

}





### 循环结构

循环语句可以在满足循环条件的情况下，反执行某一段代码，这段被重复行的代码被称为循环体。当反复执行这个循环体时，需要在合适的时候把循环条件改为假，从而结束循环，否则循环将一直 执行下去，形成死循环。循环语句可能包含如下4个部分。

>初始化语句（init\_statement):—条或多条语句，这些语句用于完成一些初始化工作，初始化语句在循环开始之前执行。

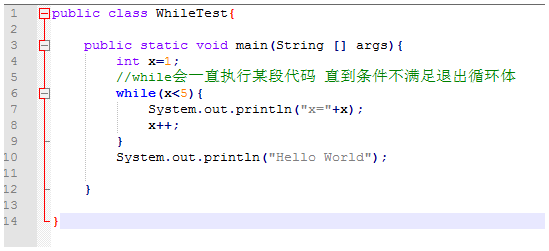
>循环条件（test\_expression):这是一个boolean表达式，这个表达式能决定是否执行循环体。

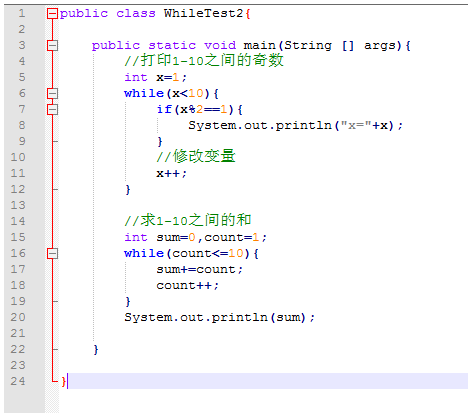
>循环体（body\_tatement):这个部分是循环的主体，如果循环条件允许，这个代码块将被重复 执行。如果这个代码块只有一行语句，则这个代码块的花括号是可以锊略的。

>迭代语句（iteration\_statement):这个部分在一次循环体执行结束后，对循环条件求值之前执行， 通常用于控制循环基件中的变量，使得循环在合适的时候结束。

上面4个部分只是一般性的分类，并不是每个循环中都非常清晰地分出了这4个部分。

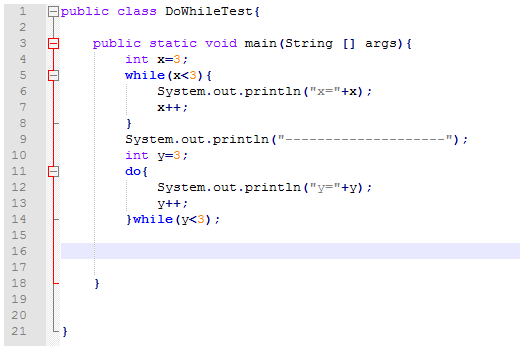
#### While循环





#### Do-while循环

Do-while循环和while循环基本类似最大的不同在于while循环是先判断条件在执行循环体，而do-while是先执行一次循环体在判断条件，也就是说do-while至少执行一次循环体。



#### For循环语句

For语句是三个循环语句中功能最强大的，使用最广泛的。For的语句格式如下:

for(表达式1 ; 表达式2 ; 表达式3){

循环体4;

}

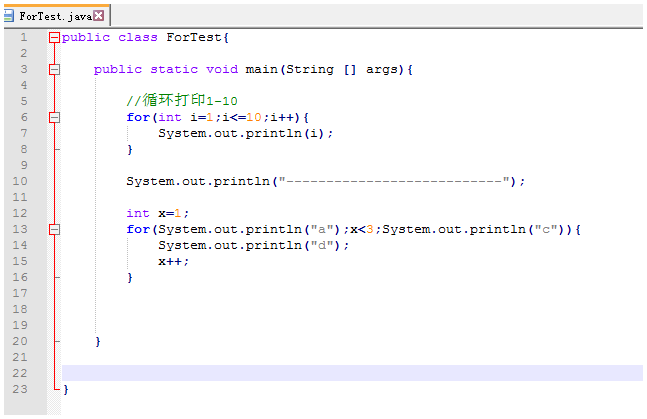
表达式1:一般是一个赋值语句，它是用来给循控制变量赋初值。

表达式2:一般是一个布尔类型的表达式，它决定什么时候推出循环

表达式3：一般是用来修改变量 控制每一次循环按照什么样的方式进行改变

For语句的执行过程：

* 1.先计算表达式1 完成对变量的初始化
* 2. 继续执行表达式2 判断是否满足条件 如果不满足直接结束整个循环体，如果满足执行下一步
* 如果表达式2满足条件则执行 循环体4
* 执行完循环体 继续执行表达式3用来修改变量的变化方式
* 继续执行表达式2 依次执行



\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

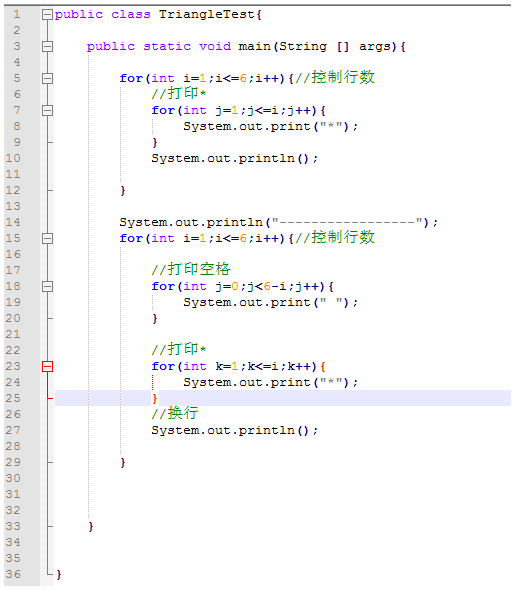
\*

\*\*

\*\*\*

\*\*\*\*

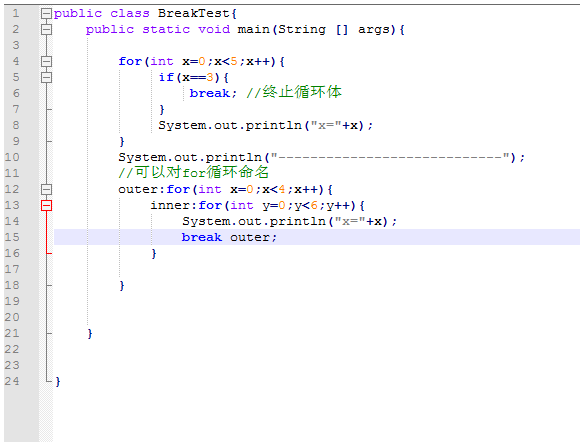
\*\*\*\*\*



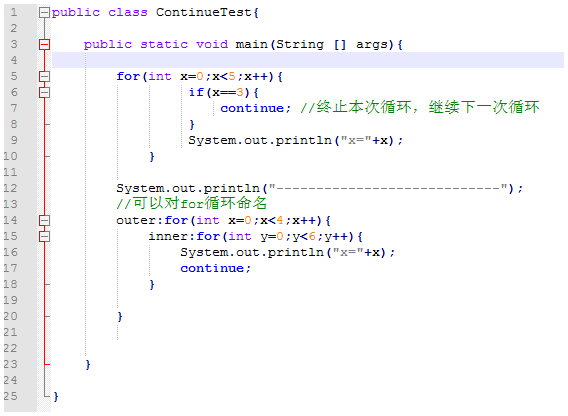
### 控制循环体

Java语言没有goto语句控制跳转，goto在java中是最为保留字存在的。这种做法提高了程序流程的可读性，但是降低了程序的灵活性。为了弥补不足Java提供了Continue和break来控制循环。除此以外还有return结束整个方法。

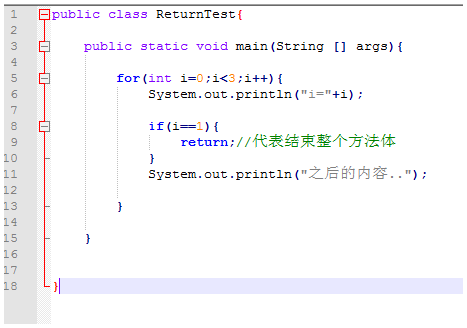
#### Break



#### Continue



#### Return



## 数组

### 数组的基本概念

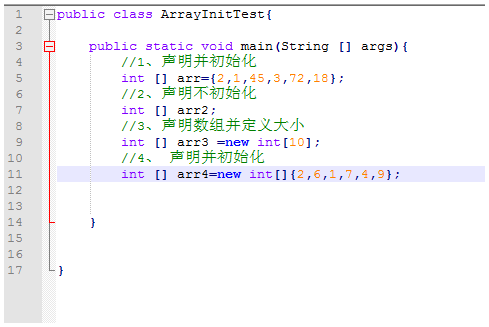
数组是用于存储多个数据，每个数组元素存放一个数据，所有的数据元素具有相同的数据类型，因此在一个数组中，数组元素的类型的唯一的，即一个数组里只能存储一种数据类型的数据，而不能存储多种数据类型的数据。

—旦数组的初始化完成，数组在内存中所占的空间将被固定下来，因此数组长度将不可改变。即使把某个数组元素的数据清空，但它所占的空间依然被保留，依然属于该数组，数组的长度依然不变。

Java的数组既可以存储基本类型的数据，也可以存储引用类型的数据，只要所有的数组元素具有相同的类型即可。

值得指出的是，数组也是一种数据类型，它本身是一种引用类型。例如int是一个基本类型，但int[] (这是定义数组的一种方式）就是一种引用类型了。

### 定义数组和初始化数组



执行数组初始化的时候，程序员只需要指定长度即可。系统会根据定义的数据类型为每个数组进行初始化：

* 数组中的类型是基本类型的整形数据(byte/short/int /long)则数组元素默认初始值是0
* 数组中的类型是基本类型的浮点型(float/double)则数组元素的默认初始值是0.0
* 数组中的类型是基本类型的字符型(char)则数组元素的默认初始值是’\u0000’;
* 数组中的类型是基本类型的布尔型(boolean)则数组元素的默认初始值是false
* 数组中的类型是引用类型则数组元素的默认初始值是null



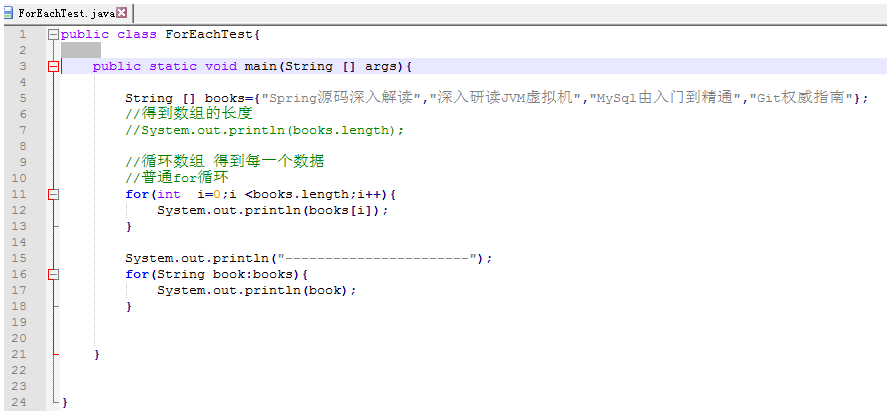
### foreach循环

在jdk1.5之后java提供了一种更加简单的循环。Foreach循环，这种循环遍历数组和集合更加简洁，使用foreach循环遍历数组无需获得数组的和集合的长度。

语法规则：

for(集合中的数据类型 合法的标识符 : 要迭代的集合){

}



### 深入数组

#### 内存中的数组

数组引用变量只是一个引用，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的内存才可以通过数组的下标进行访问数组元素。

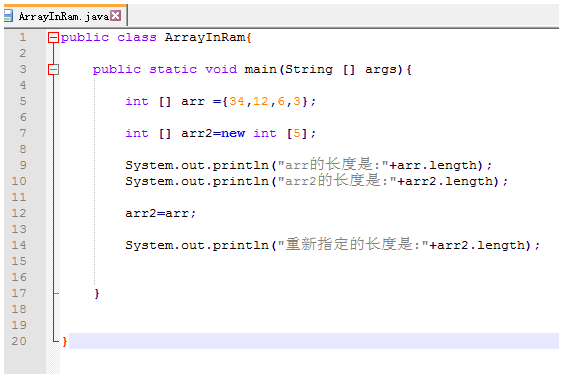
内存分为五大区域：

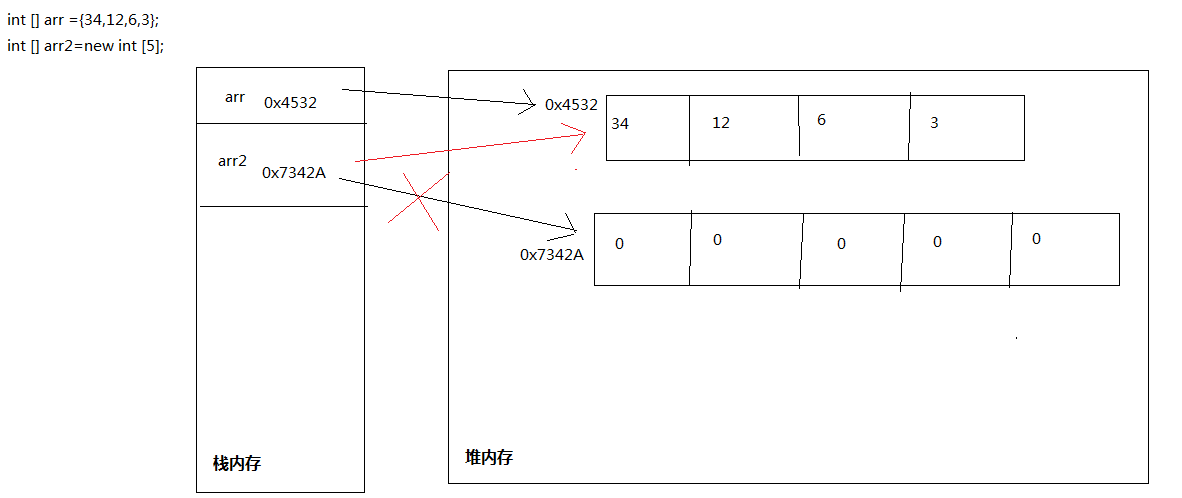
寄存器，栈内存，堆内存，方法区，本地方法区

栈内存：存储的都是局部变量，一旦使用完毕，立即释放。

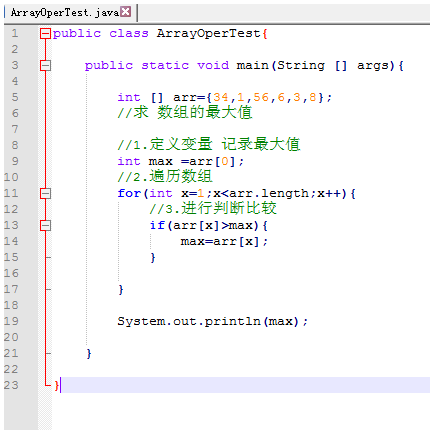
堆内存： 存储的是实体(对象)。

* 堆内存中每一个实体都有自己的内存地址的值
* 堆内存中每一个变量都有默认的初始值
* 当实体被系统认为是垃圾时，系统会在不定期的时间执行垃圾回收。





### 数组的使用



#### 数组的排序

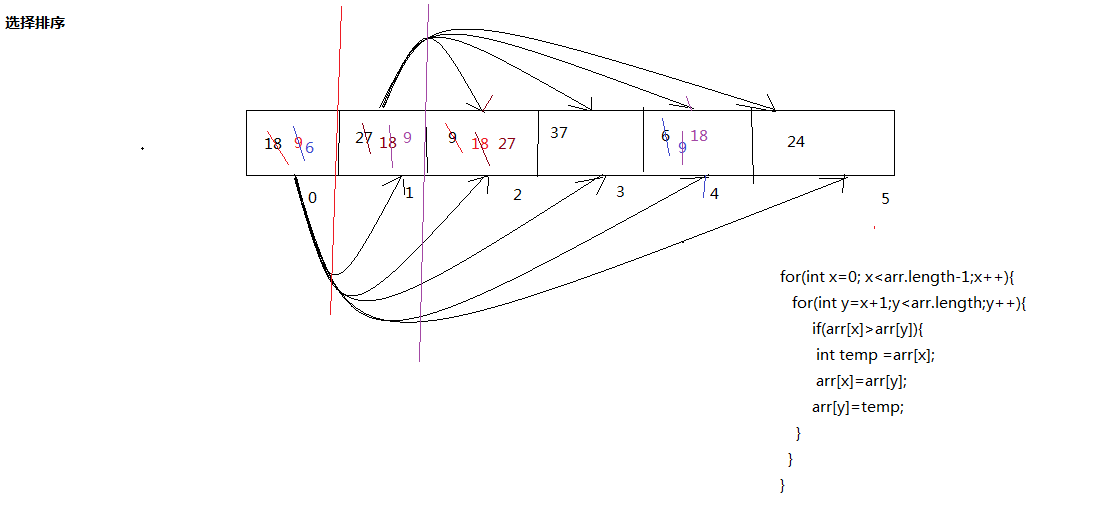
归并排序

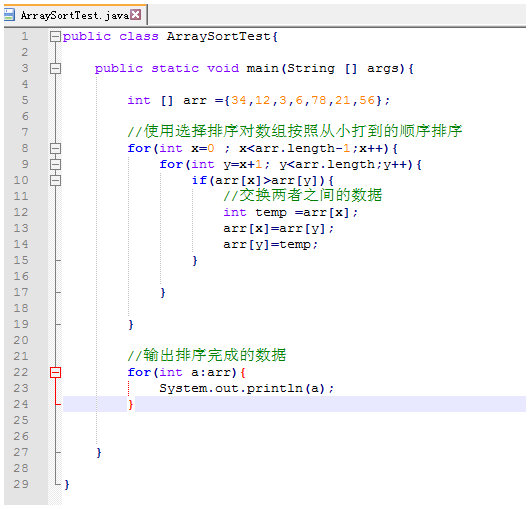
1. public static int[] sort(int[] a,int low,int high){
2. int mid = (low+high)/2;
3. if(low<high){
4. sort(a,low,mid);
5. sort(a,mid+1,high);
6. //左右归并
7. merge(a,low,mid,high);
8. }
9. return a;
10. }
12. public static void merge(int[] a, int low, int mid, int high) {
13. int[] temp = new int[high-low+1];
14. int i= low;
15. int j = mid+1;
16. int k=0;
17. // 把较小的数先移到新数组中
18. while(i<=mid && j<=high){
19. if(a[i]<a[j]){
20. temp[k++] = a[i++];
21. }else{
22. temp[k++] = a[j++];
23. }
24. }
25. // 把左边剩余的数移入数组
26. while(i<=mid){
27. temp[k++] = a[i++];
28. }
29. // 把右边边剩余的数移入数组
30. while(j<=high){
31. temp[k++] = a[j++];
32. }
33. // 把新数组中的数覆盖nums数组
34. for(int x=0;x<temp.length;x++){
35. a[x+low] = temp[x];
36. }
37. }

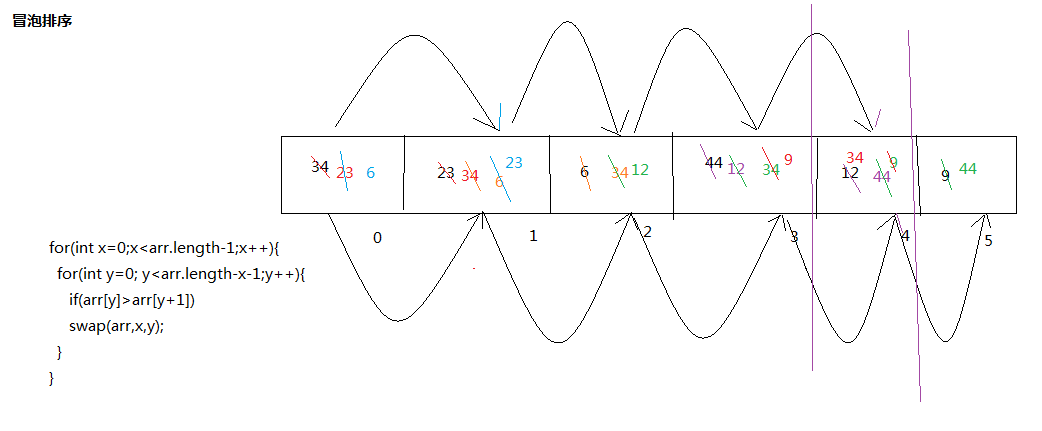
二分排序

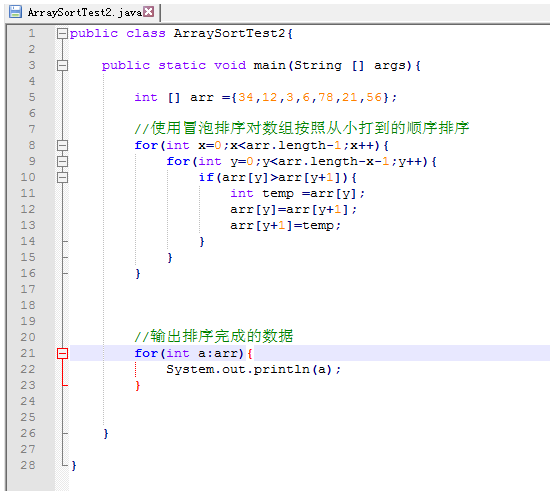
1. **public** **class** Test {
2. **public** **static** **void** main(String[] args) {
3. **int** []a={4,2,1,6,3,6,0,-5,1,1};
4. **int** i,j;
5. **int** low,high,mid;
6. **int** temp;
7. **for**(i=1;i<10;i++){
8. temp=a[i];
9. low=0;
10. high=i-1;
11. **while**(low<=high){
12. mid=(low+high)/2;
13. **if**(a[mid]>temp)
14. high=mid-1;
15. **else**
16. low=mid+1;
18. }
19. **for**(j=i-1;j>high;j--)
20. a[j+1]=a[j];
21. a[high+1]=temp;
22. }
23. **for**(i=0;i<10;i++){
24. System.out.printf("%d",a[i]);
25. }
26. }
28. }

冒泡排序，选择排序





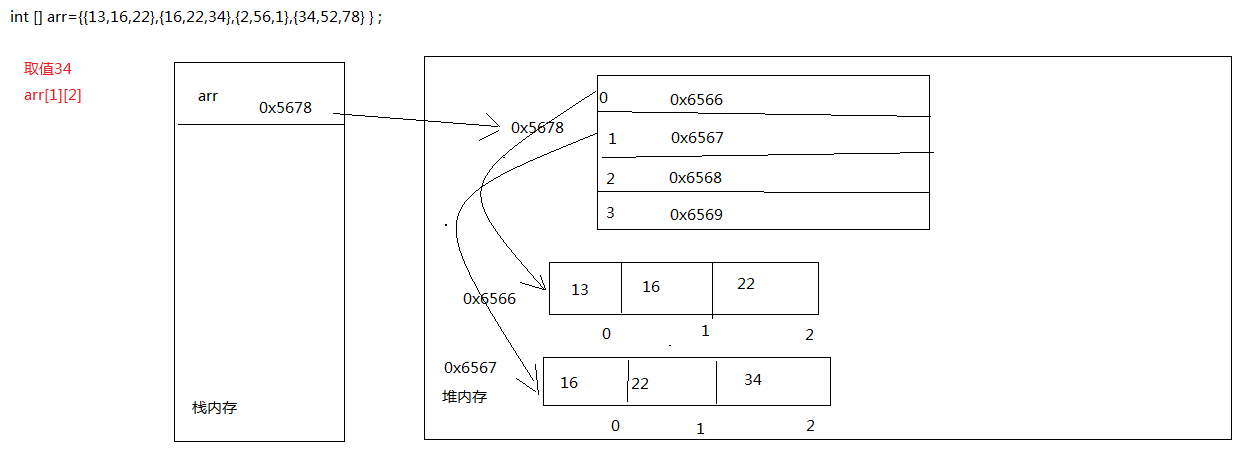


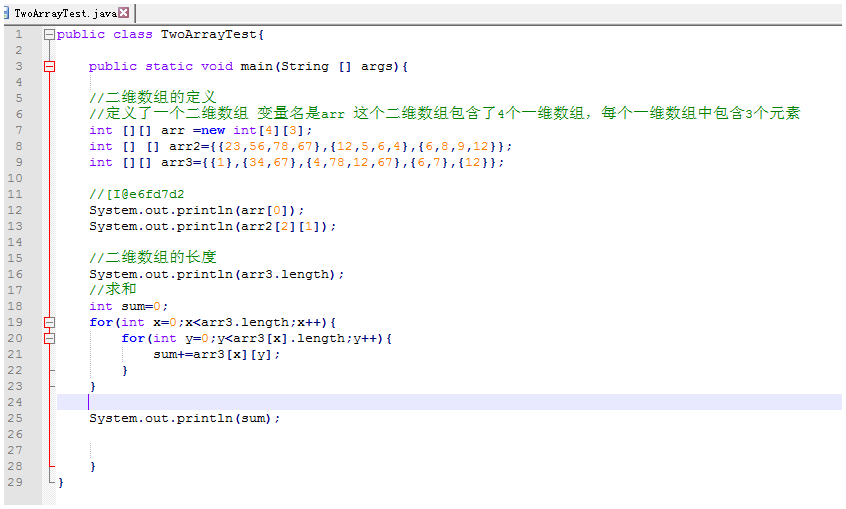


#### 数组的错误操作



### 二维数组





# 面向对象

## 面向对象的基础知识

### 面向对象的概念

面向过程是一种以过程为中心的编程思想，他们不支持丰富的面向对象的思想和特性(比如继承，多态)吗，面向过程不允许混合使用持久化状态和域逻辑，面向过程就是分析问题然后提出解决问题的步骤，然后用函数把这些步骤一步一步实现即可。

面向对象是把设计到的对象抽离出来，然后考虑对象之间的关系，然后构建整个项目，面向过程是提供一种符号设计系统的面向对象的实现。

举例分析： 把大象装进冰箱

面向过程：

1. 打开冰箱
2. 装进冰箱
3. 关上冰箱

面向对象的思想：

1. 冰箱打开
2. 冰箱存储
3. 冰箱关闭

万物皆对象

### 面向对象的特征

在java语言中出了8种基本数据类型以外，一切都是对象，万物皆对象，面向对象就是面向程序编程的核心。可以把设置到对象的一切属性和方法进行提取，然后完成各个模块的构建。

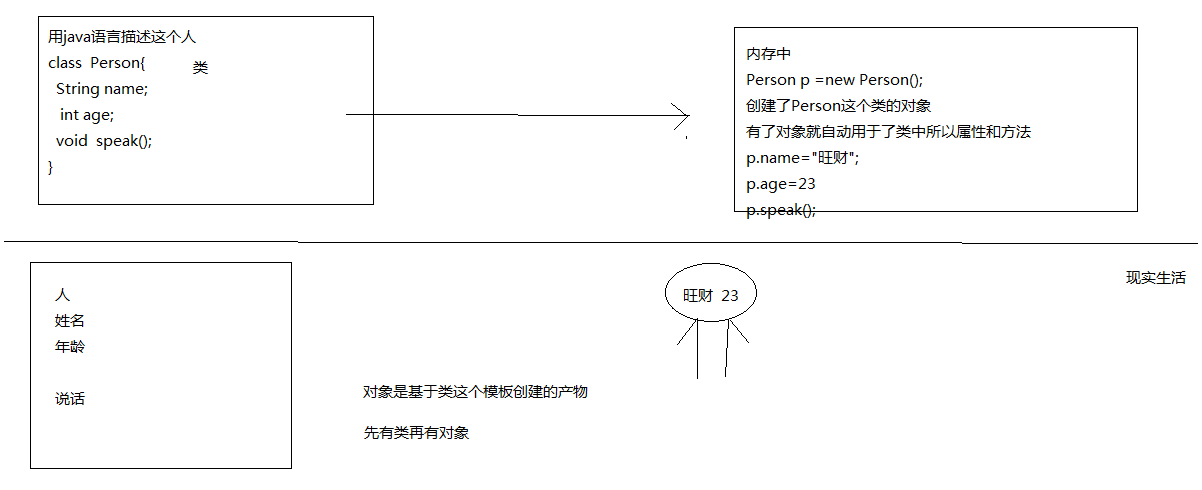
对象具有状态，一个对象用数据值描述他的状态，java对通过为对象定义成员变量来描述对象的状态。

对象是Java的核心，所有的java对象具有唯一性，每一个对象都有一个唯一的标识符来代表这个对象，如果某个对象失去了这个标识符，那么将会被当做垃圾进行回收。

### 类和对象的关系

类是面向对象一个概念，一个类定义了一组对象，类具有行为，它描述这个类能做什么，以及做什么的方法。

具有相同或相似的一组对象就是称为类，类是对一类事物的描述。



在Java中所有的代码必须在类中。

语句不能直接出现在类中 类中可以出现以下内容:

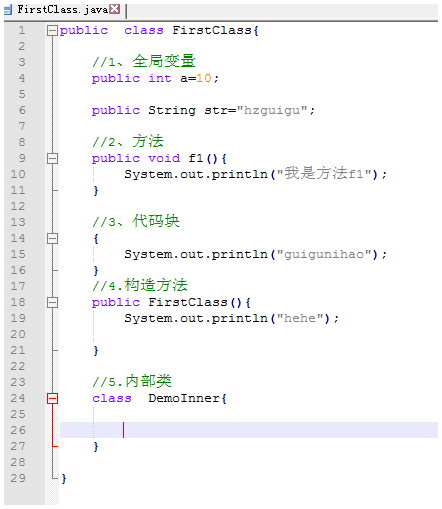
方法

成员变量(全局变量)

类 构造方法

代码块

内部类



## 方法

方法就是类或者对象的抽象，方法是类或者对象的重要的组成部分，从功能上看方法和函数是完全一致的。在java中方法是不能独立存在的，必须在类中进行定义。

### 方法的概念

方法就是定义在类中具有特定功能的一段独立小程序。

方法的语法格式

修饰符 返回值 函数名 (参数类型 形式参数1,参数类型 形式参数2….){

执行语句;

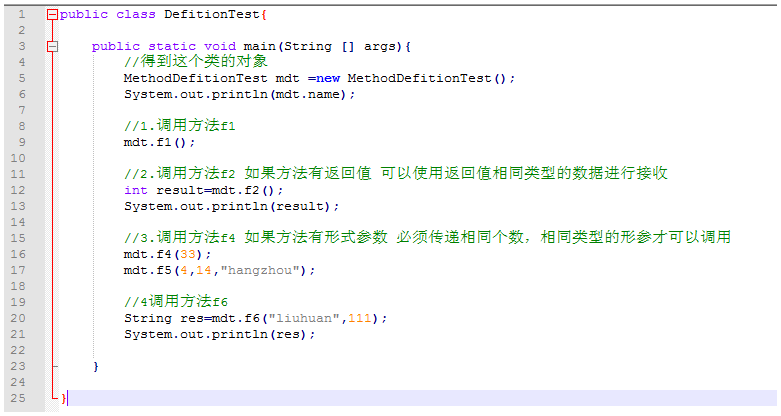
return 返回值类型

}

### 方法的定义

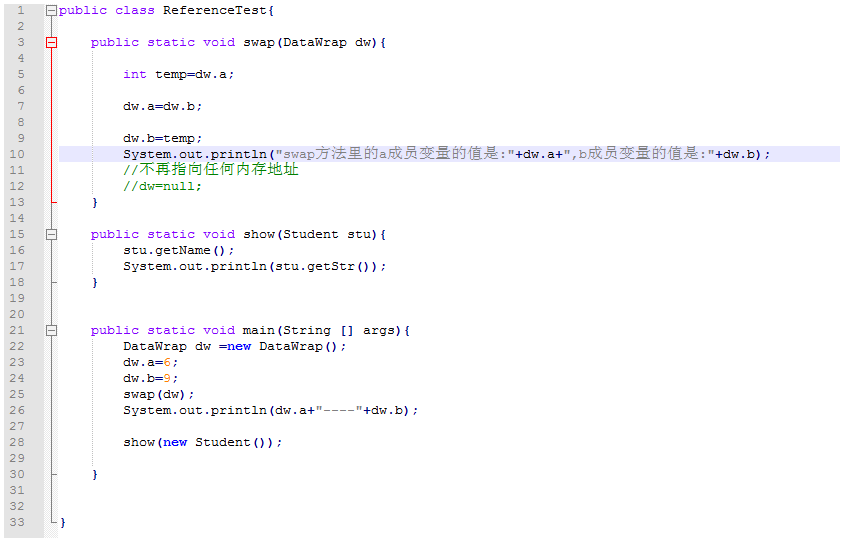
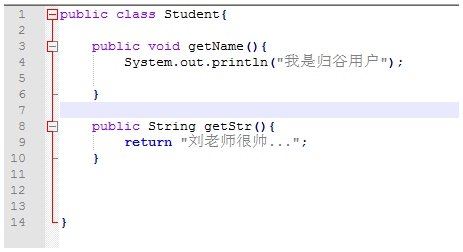
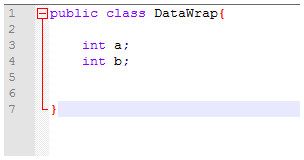
#### 基本类型的形参





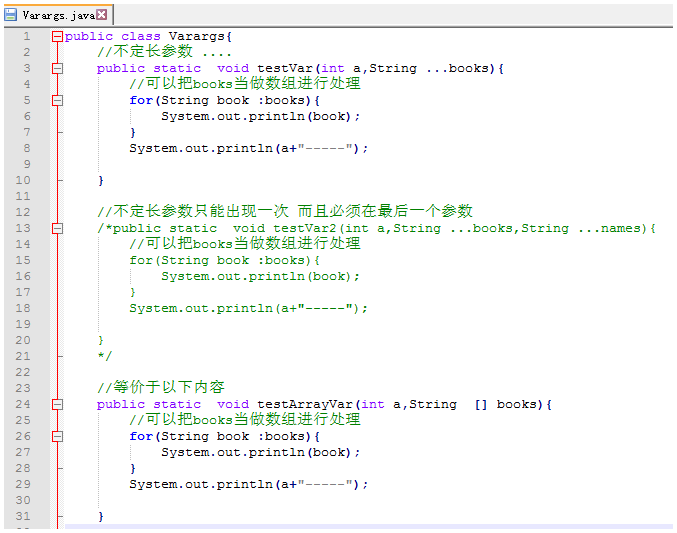


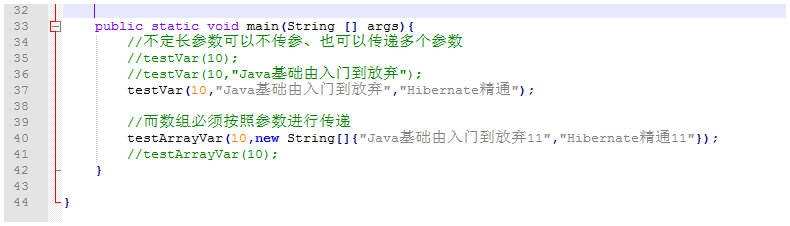
#### 引用类型的形参



#### 形参个数可变的方法(不定长参数)

在jdk1.5之后java允许定义可变的参数，从而允许方法的形参为不确定的形参。不确定的参数是(…),只能出现在形参的最后一个，并且只能出现一次。

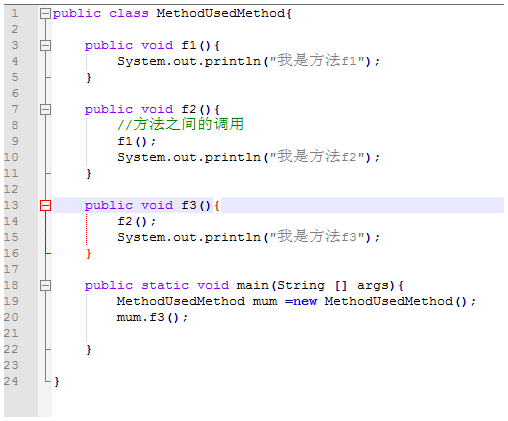


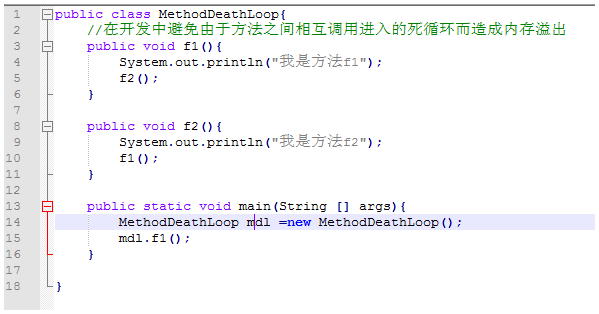


### 方法的递归

一个方法体内自身调用自身称为方法的递归。方法的递归包含了一种隐式的循环，它会重复执行某段代码。但是这种重复不需要循环控制。

#### 方法之间的调用

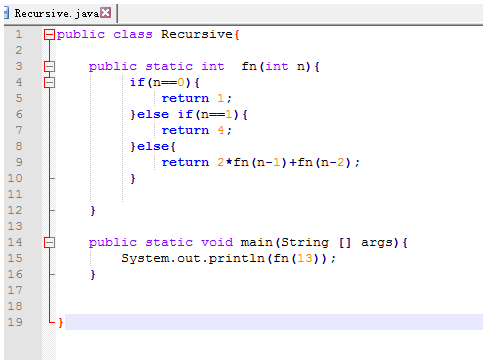




#### 方法的递归

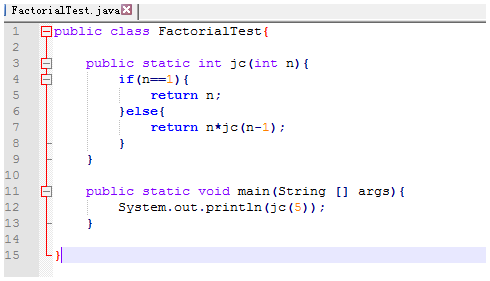
数学题目

f(0)=1 ; f(1)=4 ; f(n+2)=2\*(fn+1)+f(n); 求第n个数是多少



求n的阶乘

5！=5\*4\*3\*2\*1



### 方法的重载

Java允许一个类定义多个同名的方法，只要形式参数列表不同即可。如果同一个类中包含了两个以上方法名相同，但是形参列表不同，则成为方法的重载。

方法的重载就是要求两同一不同，**同一个类方法名相同，参数列表不同（1、个数不同2、类型不同3、位置不同）**，至于方法的其他部分和方法的返回值，修饰符无关。





#### 不定长参数的重载



## 成员变量和局部变量

实例变量(不用static修饰的变量)

成员变量(全局变量)

类变量(用static修饰的变量)

所有的变量

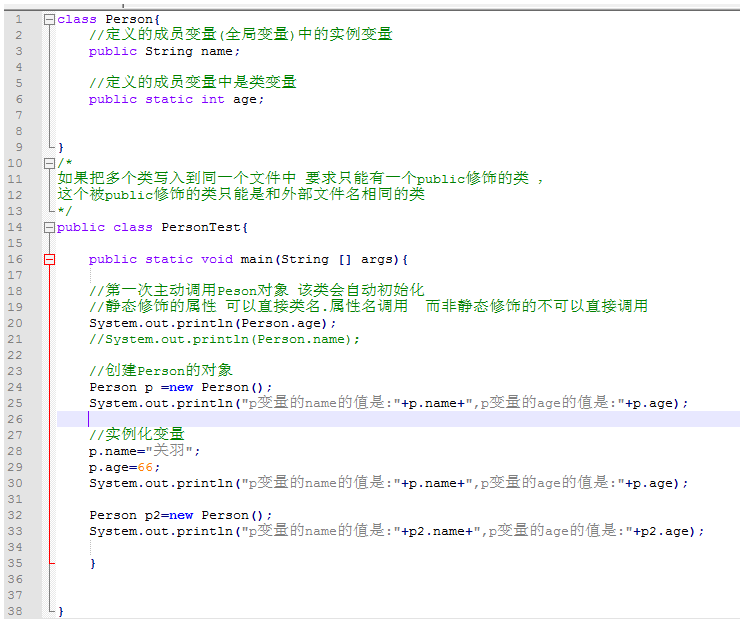
形参(在方法签名中定义的变量)

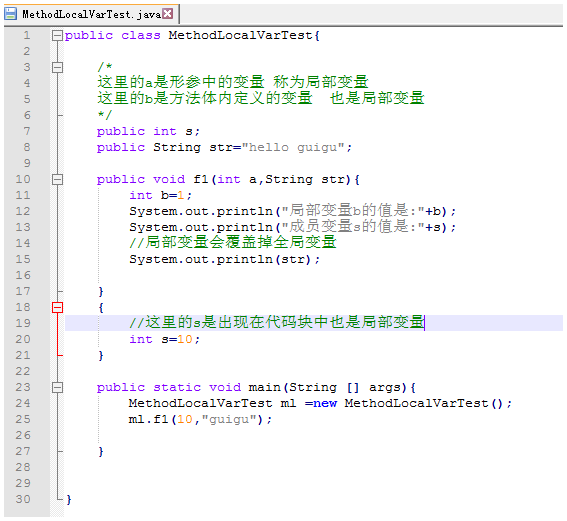
局部变量 方法局部变量(在方法体内定义的变量)

代码块中的变量(在代码块中定义的)

对比成员变量和局部变量的区别

|  |  |  |
| --- | --- | --- |
| 对比项 | 成员变量 | 局部变量 |
| 定义的位置 | 成员变量是定义在类中 | 定义在函数中，语句中 |
| 内存中的位置 | 在堆内存中 | 栈内存中 |
| 初始值 | 成员变量有默认的初始值可以不用赋初值依然可以使用 | 必须赋初值才能使用，否则报错 |
| 生命周期 | 成员变量是随着对象的出现而出现，随着对象的回收而消失 | 局部变量是一旦作用域结束立马被回收 |





## 封装和隐藏

我们说封装面向对象的三大特征之一，另外两个是继承和多态。所谓的封装他指的是把对象的状态信息封装在对象内部不允许外部程序，直接对对象内部的信息加以修改和访问。

封装可以实现以下目标:

* 第一点是说隐藏类的细节。
* 让使用者通过事先预定的方法来访问数据，从而可以在方法里加入逻辑的控制。
* 第三点，是说我们可以进行数据的检查，从而保证数据的完整性。
* 我们将对象的成员变量和实现的细节隐藏起来，不允许我们外部直接访问来保护程序的一个隐私。

封装需要考虑两个方面的问题：

* 将对象的成员变量和实现细节隐藏起来，不允许外部访问。
* 把公共方法暴露出来，让方法控制成员变量的数据。

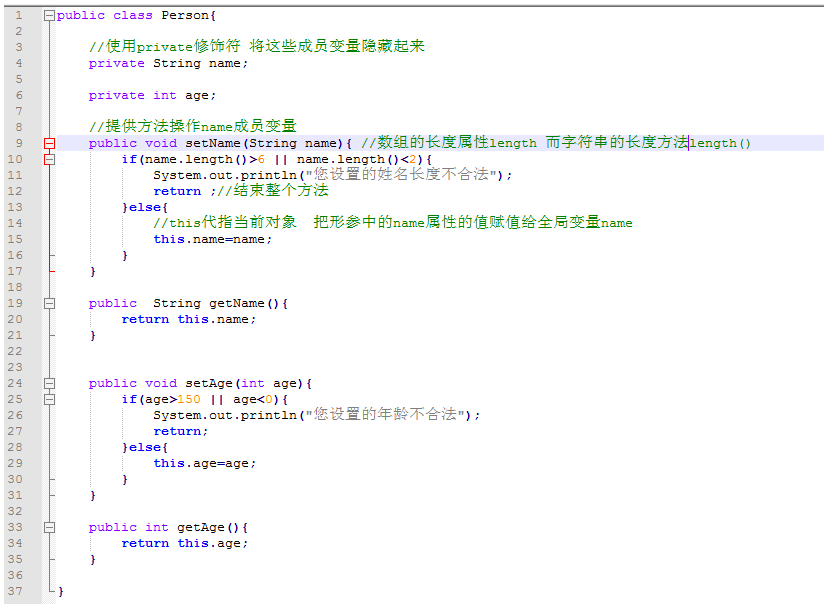
### 访问控制修饰符

Java提供了三个访问修饰符。private/public/protected分别代表了三种访问控制级别，另外还有一个不加任何访问修饰符的访问控制级别。默认的default、所以一共有四种访问修饰符。

四种访问修饰符从小到大的顺序

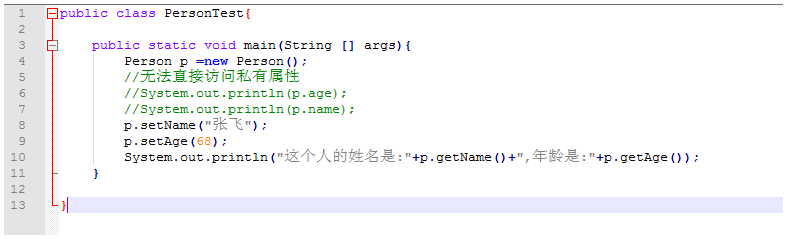
private ---->default---->protected---->public

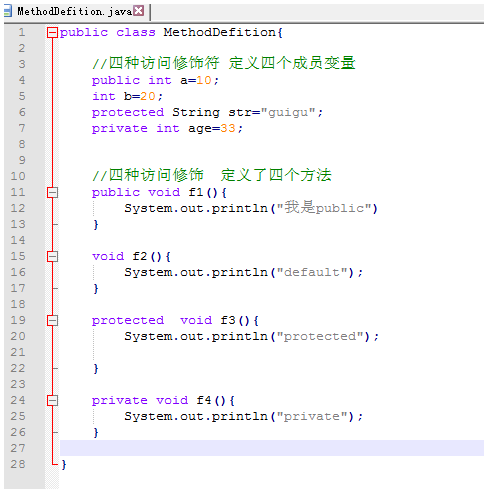
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | private | defalut | protected | public |
| 同一个类中 | √ | √ | √ | √ |
| 同一个包中 |  | √ | √ | √ |
| 不同包下的子类中 |  |  | √ | √ |
| 全局范围 |  |  |  | √ |



提示：Java里实例变量的setter和getter方法有着重要的意义。规则是setXXX和getXXX要首字母大写即可.比如对name属性赋值和取值setName和getName

可以更好的进封装和操作。





### 包

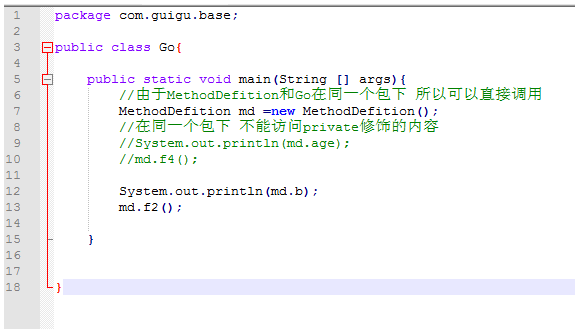
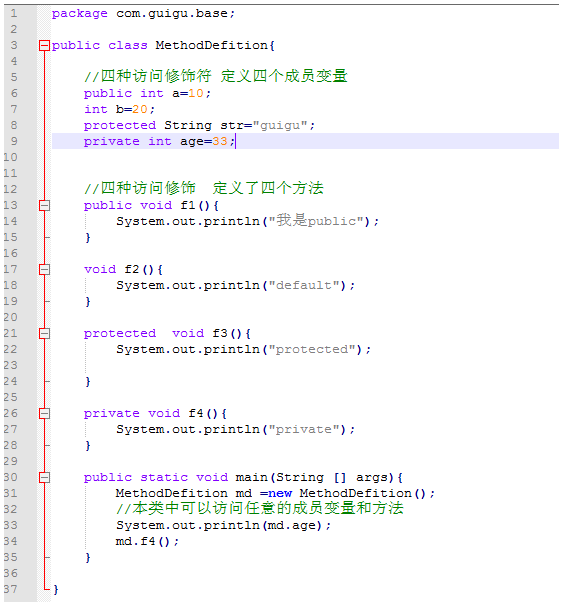
包是用来区分类的，java允许将一组相同功能的类放在同一个包下(package),**主要用于解决命名冲突的问题。**

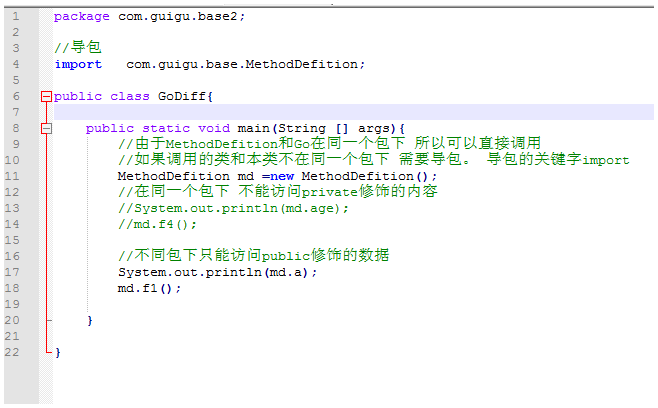
限定符 通过包名的前缀明确使用哪个类

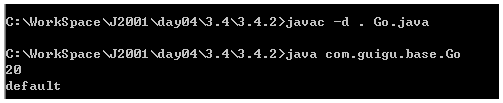
包有自己的命名规则: 采取倒域名的方式 包名都是小写

组织机构名.公司名.项目名.模块名.子模块名

com.guigu.shopping.user.dao







#### 普通导包

Import

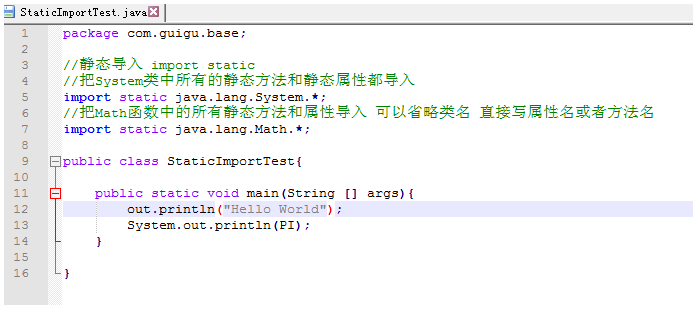
可以使用\*简化导包

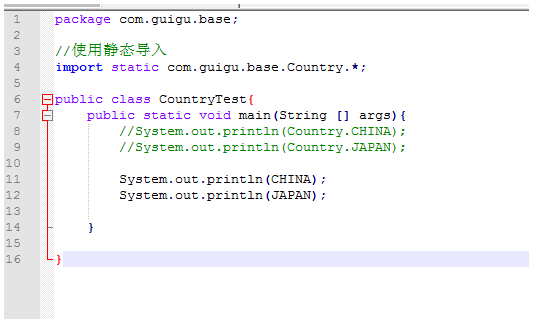
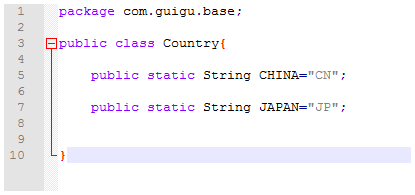


Java里所有的类都默认导入Java.lang.\*包下所有的类。不需要import再次导入。

#### 静态导入包

在jdk1.5以后增加了一种静态导入包，他用于导入指定的某个静态成员变量和方法。





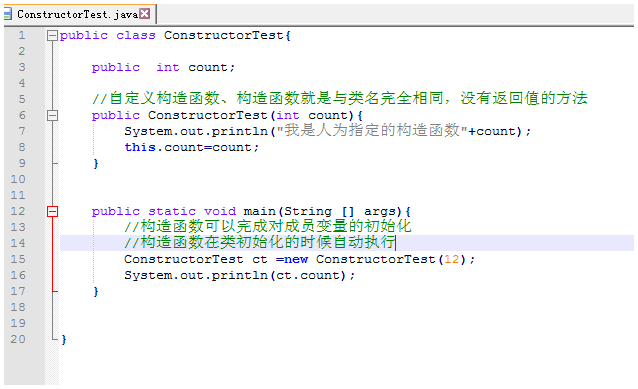
## 构造函数

构造函数就是一个特殊的方法，这个特殊的方法用于创建实例的时候进行初始化。构造函数是创建对象的一个重要途径。即使使用工厂模式，反射等方式创建对象，依然是依赖于构造函数。

在一个Java类中可以有一个或者多个构造函数。

### 构造函数的初始化

**构造函数最大的用途是创建类的时候进行初始化**。每个类都有默认的构造函数默认的就是无参的构造函数。如果一旦人为指定系统默认的构造函数将不再执行。



### 构造函数的重载

同一个类中可以有多个构造函数，而自多个构造函数是形式参数列表不同，被称为构造函数的重载。构造器的重载和方法的重载基本相同。要求构造函数的名字相同，然后是在同一个类中，并且形式参数不同。



构造函数之间的调用

**This调用构造函数必须在方法体内的最开始**



### this关键字

Java提供了一个this关键字，关键是总是指向调用该方法的对象。根据this出现位置的不同，**this作为为对象的默认用有两种情况。**

* **构造器中引用构造器正在初始化的对象。**
* **在方法中引用调用该方法的对象。**



this可以代表任何对象，当this出现在某个方法体中，它所代表的对象是不确定的。只有当该方法被调用的时候才能知道该this代指哪个对象？



### Static关键字

静态的关键字是static

Static是修饰符，能修饰成员(成员变量和成员方法)

* 特点:

1. 被静态修饰的内容可以被所有对象共享。
2. 静态数据随着类的加载而加载，也就意味着随着类的消失而消失。由于生命周期过长，所以他优先于对象处。
3. 由于静态数据属于累，所以多了一种调用方式。类名.属性名 类名.方法名

* 静态使用的注意事项

1. 静态方法只能访问静态成员，不访问非静态成员。也就是说static修饰的方法只能调用static修饰的方法和属性，而不能调用非static修饰的。
2. 静态方法中不可以使用this关键字，也不可以使用super关键字。



### 单例模式

首先设计模式是做什么？所谓的设计模式就是解决问题行之有效的办法之一是一种思想。单例模式是解决什么问题呢，主要解决。保证一个类在内存中只产生一个对象。

如何保证一个类在内存中对象的唯一性？

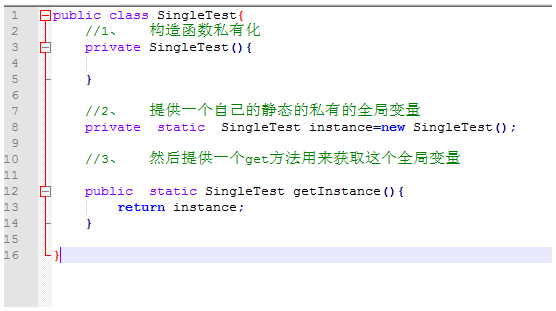
1. 不让其他程序创建该类的对象。
2. 在本类中自己创建一个本类对象。

实现的步骤:

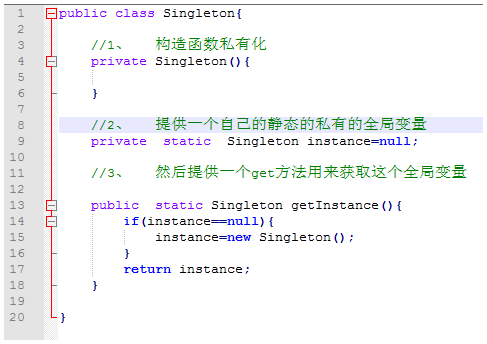
1. 构造函数私有化
2. 提供一个自己的静态的私有的全局变量
3. 然后提供一个get方法用来获取这个全局变

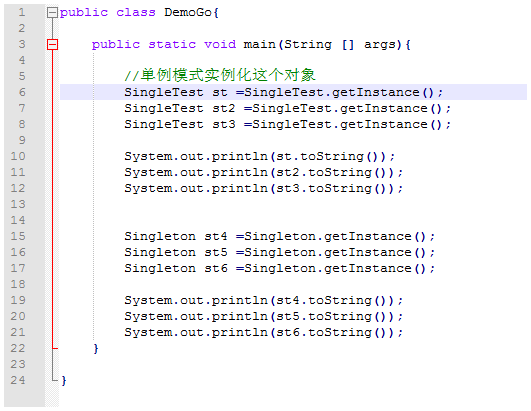
单例模式分为懒汉式和饿汉式两种模式

#### 饿汉式



#### 懒汉式





## 继承

继承面向对象的三大特征之一。继承也是实现代码复用的重要手段。继承是具有传递性的，并且每个子类只有一个父类。

### 继承的特点

Java的继承是通过extends关键字来实现的。实现继承的类被称为子类，被继承的类称为父类。父类在有些教程中也称为基类，超类。

Java继承的语法规则如下:

修饰符 class subclass extends superclass{

//定义内容

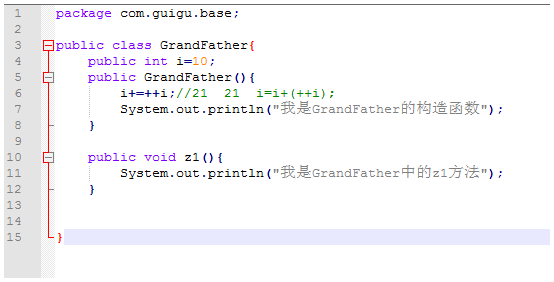
}

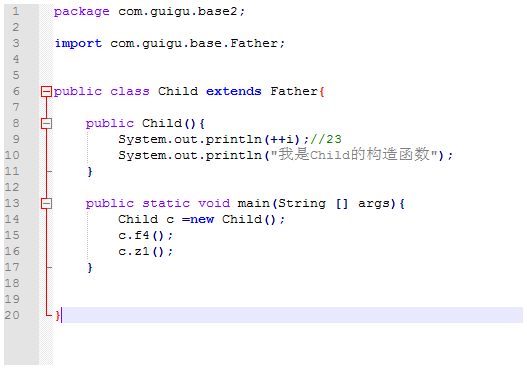
我们说Java使用extends作为关键字，但是extends在英文里是扩展的意思，而不是继承。如果从严格意义上来看子类继承父类还是一种错误的说法，应该严格来说是字类扩展父类的特性。

我们翻译为子类继承父类，由于此类能够获得父类，全部的成员变量和方法和汉语中继承有着相同的含义。

继承的特性：

* 在java中 所有的类都默认继承了java.lang.Object类
* 继承具有传递性
* 继承只能是单继承的 只能继承一个父类
* 构造函数是不能被继承的 但是可以在子类中调用
* 当子类初始化的时候 先初始化父类



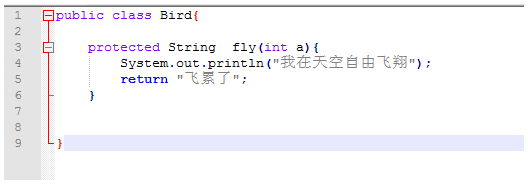
 

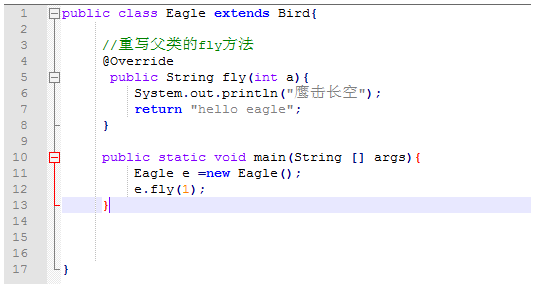
### 方法的覆盖

子类继承父类并且重写了父类的方法的这种行为被称为方法的覆盖。

方法的覆盖遵循以下原则：

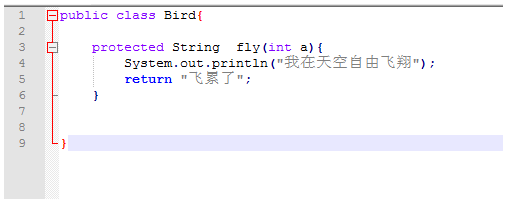
“两同两小一大的原则”:即方法名相同，形式参数列表相同、 两小是指子类的方法的返回值类型应该比父类更小或者相等，子类抛出的异常只能比父类更少，更小才可以。一大是指子类的作用权限应该比父类的权限更大或相等。

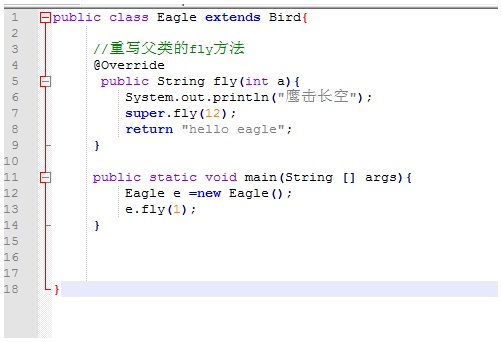




### Super关键字

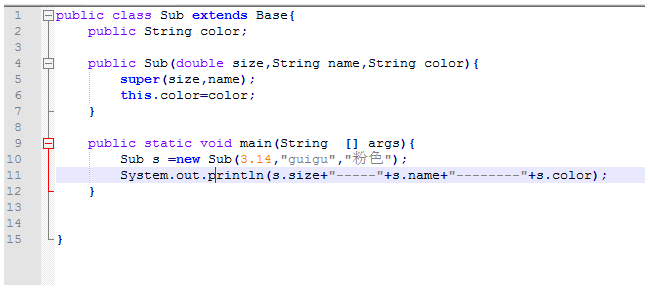
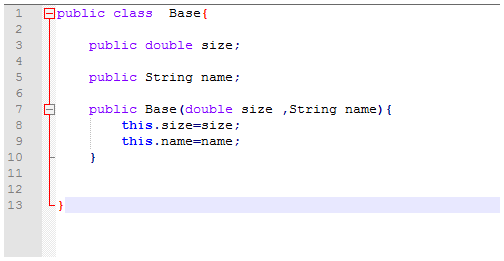
如果需要在子类的方法中调用父类被覆盖的实例方法，这需要super限定来调用父类覆盖的实例方法。





this不能和static关键字共存，super也不能出现在static修饰的方法中。Static修饰的方法是属于类，该方法的调用者可能是一个类，而不能是对象，所以super就没有意义了。

### 调用父类的构造函数

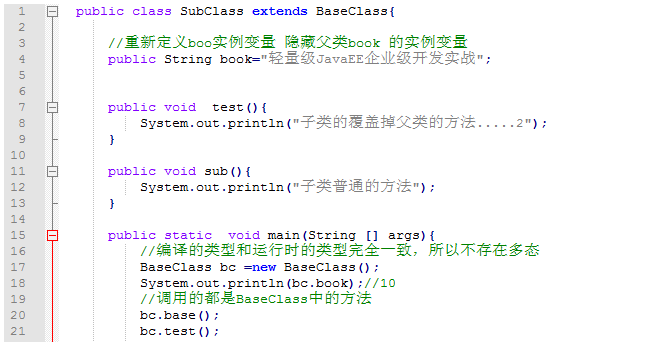
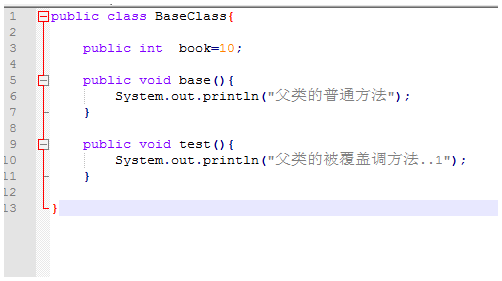


使用super调用和使用this调用很像。区别别在于super调用的是父类的构造器，而this调用的是同一个类中重载的构造器。使用super调用父类构造器也必须出现在子类构造器的第一行。

## 多态

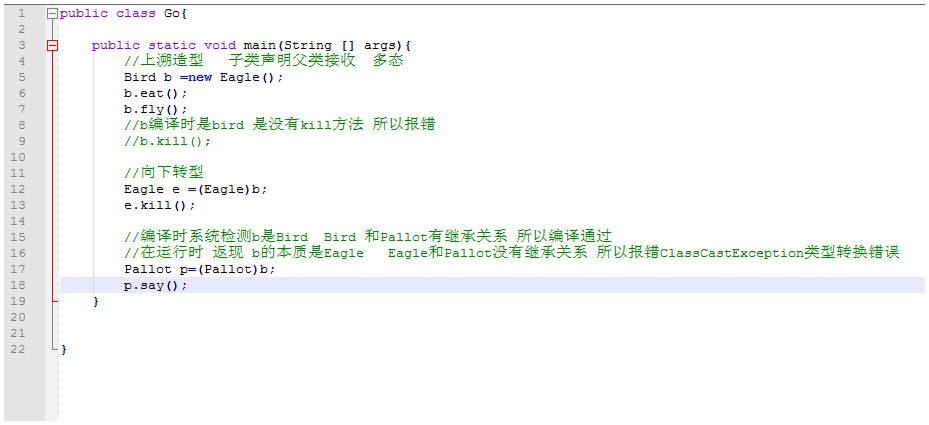
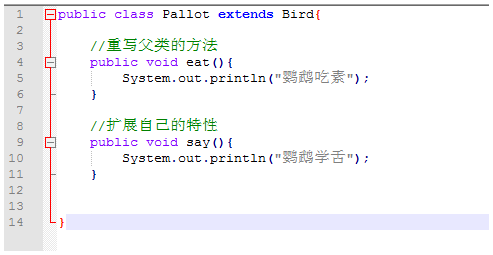
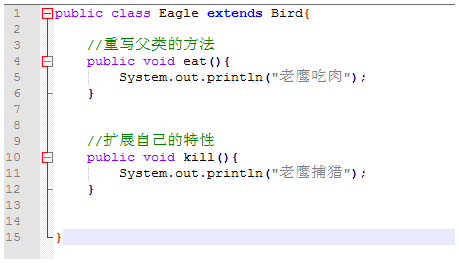
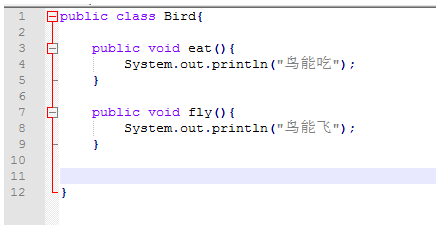
### 多态的特性

Java引用类型的变量有两个类型 ，一个是编译时的类型，一个是运行时的类型。编译时的类型由声明该变量时使用的类型决定，运行时的类型由实际赋值给该变量的对象决定。如果编译的类型和运行的类型不一致，就可能出现多态。

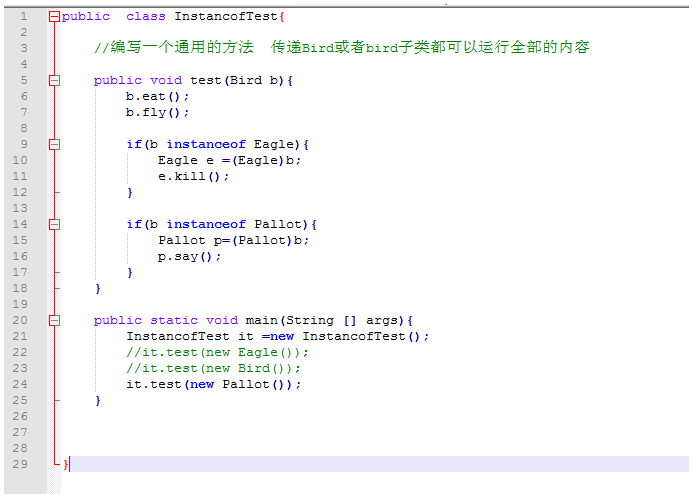


### 引用类型的强制类型转换

基本类型的类型转换分为 自动类型转换和强制类型转换，而引用类型的类型转换分为 上溯造型和向下转型。 必须有继承关系才能完成引用类型的类型转换。



### Instanceof



## 代码块

Java使用构造器对单个对象进行初始化操作，使用构造器先完成整个java对象的初始化，然后将java对象返回程序，这样操作能让java对象的信息更加完整，代码块的作用域构造器的作用非常类似，都是完成对类进行初始化。

修饰符{

//初始化内容

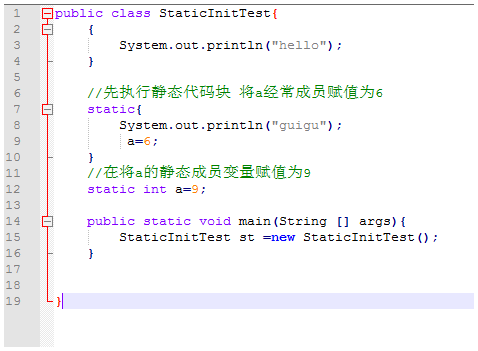
}

这里的修饰符只能是static。 也可以不可写。

### 普通代码块

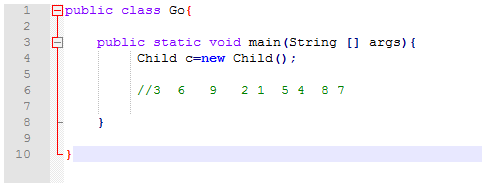
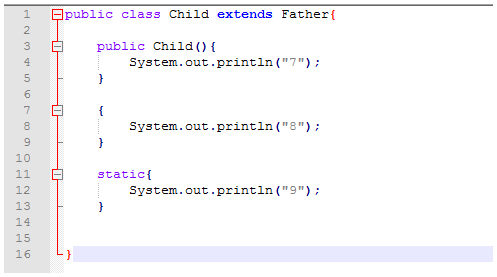
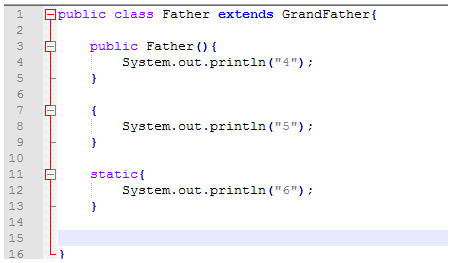
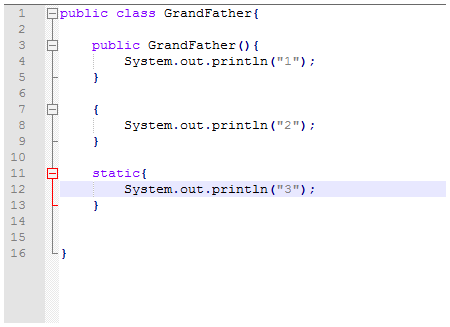


### 静态代码块



面试题

子类初始的时候先初始化父类，类是优先于对象存在的先执行属于类的，然后在执行属性对象的。代码块优先于构造函数执行。



## 基本数据类型的包装类

### 包装类型的基础

Java是面向对象的编程语言。它包含了八种基本数据类型，这八种基本数据类型不支持面向对象的编程机制。基本数据类型不具备对象的特征。他是没有成员变量方法可以被调用。

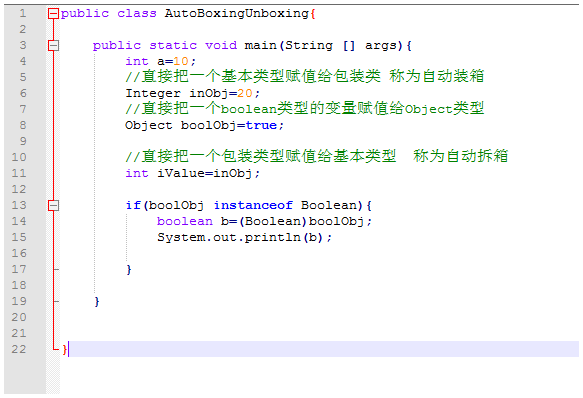
这八种基本类型带来了一些方便能更加简单有效的处理数据。但是同时也有一些限制。

为了解决八种基本数据类型不能当Object的类型使用的问题，定义了八种引用类型。

八种基本数据类型对应的8种包装类型

|  |  |
| --- | --- |
| 基本数据类型 | 包装类型 |
| byte | Byte |
| short | Short |
| int | Interger |
| long | Long |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |

JDK1.5以后提供了自动装箱和自动拆箱功能。所谓的自动装箱就是把一个基本数据类型直接赋值给包装类的变量，或者赋值给Object变量；自动拆箱就与之相反，允许直接把包装类对象直接赋值给一个基本类型的变量。



### 基本类型和字符串之间的转换

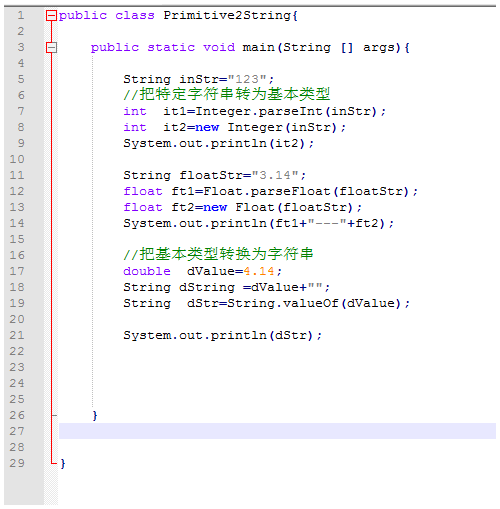
除此以外，包装类还提供了实现基本类型和字符串之间的转换，把字符串类型也能转换为基本类型。

1.把字符串转换为基本类型

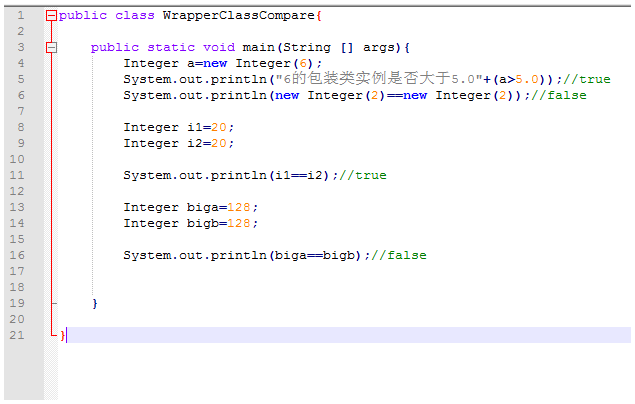
* 利用包装类提供的parsXxxx(String )静态方法把字符串转换为指定的基本类型
* 利用包装类自身提供的构造函数自动完成转换

2.把基本类型转换为字符串

* +” ”
* String 提供的重载的方法.valueOf(…)转为对应的类型



### 包装类型和基本类型的比较



两个包装类对象比较比较情况比较复杂 ，查阅源码

这样定义int的

static final Integer cache[] = cache = new Integer[(high - low) + 1];;

static{

cache = new Integer[(high - low) + 1];

int j = low;

//执行初始化 创建-128到127的Integer实例，并且放入到cache缓存中。

for(int k = 0; k < cache.length; k++)

cache[k] = new Integer(j++);

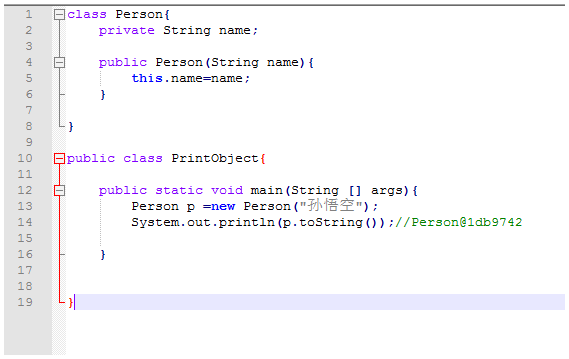
// range [-128, 127] must be interned (JLS7 5.1.7)

assert IntegerCache.high >= 127;

}

## 处理对象

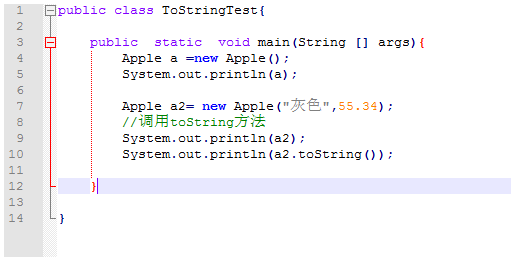
### Object中的toString



toString()方法是Object类中一个方法，所有的java类都默认继承了java.lang.Object这个类，所以所有的对象都可以钓鱼toString()这个方法。

toString()这个方法是一个非常特殊的方法，它是一个自我描述的方法，该方法通用用于实现这样一个功能，当程序员直接打印该对象时，系统会输出该对象的自我描述信息toString()方法，可以告诉外界这个对象具有的状态信息。





### ==和equals

1. Java程序中测试两个变量是否相等有两种方式：一种是利用==运算符，一种是利用Object对象中的equals方法。 当使用==判断两个对象是否相等时，如果基本是数据类型，并且都是数值类型，只要两个数变量的值相等那么，那么就返回true、
2. 但是对于两个引用类型，只有他们指同一个对象时，==判断才返回true，==不可用于比较类型上没有父子关系的两个对象(必须有继承关系)。

public boolean equals(Object obj)指示其他某个对象是否与此对象“相等”。

equals 方法在非空对象引用上实现相等关系：

自反性：对于任何非空引用值 x，x.equals(x) 都应返回 true。

对称性：对于任何非空引用值 x 和 y，当且仅当 y.equals(x) 返回 true 时，x.equals(y) 才应返回 true。

传递性：对于任何非空引用值 x、y 和 z，如果 x.equals(y) 返回 true，并且 y.equals(z) 返回 true，那么 x.equals(z) 应返回 true。

一致性：对于任何非空引用值 x 和 y，多次调用 x.equals(y) 始终返回 true 或始终返回 false，前提是对象上 equals 比较中所用的信息没有被修改。

对于任何非空引用值 x，x.equals(null) 都应返回 false。

Object 类的 equals 方法实现对象上差别可能性最大的相等关系；即，对于任何非空引用值 x 和 y，当且仅当 x 和 y 引用同一个对象时，此方法才返回 true（x == y 具有值 true）。

注意：当此方法被重写时，通常有必要重写 hashCode 方法，以维护 hashCode 方法的常规协定，该协定声明相等对象必须具有相等的哈希码。

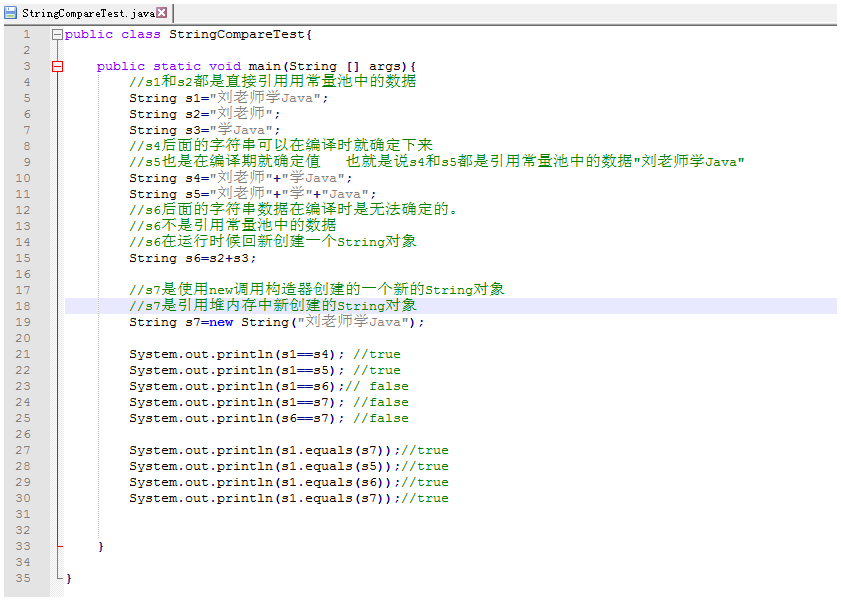
参数：

obj - 要与之比较的引用对象。

返回：

如果此对象与 obj 参数相同，则返回 true；否则返回 false。

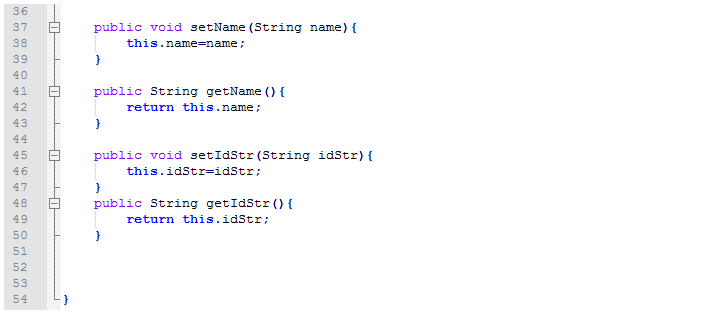
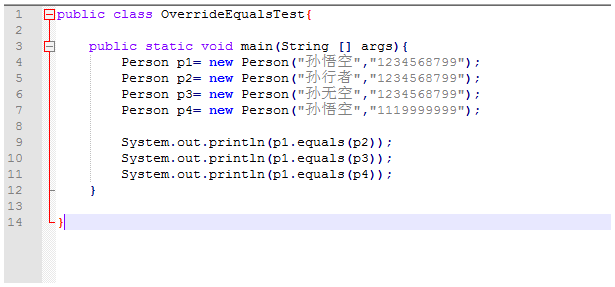




JVM保证常量池中相同的字符串只有一个，不能产生多个副本。 使用new产生的String字符串不是存储于常量池而是存储于堆内存中。

### 重写equals方法

在很多时候程序判断两个引用变量是否相等时，也希望类似于值相等的规则判断。

## final

final是修饰符，可以修饰变量，修饰方法，修饰类

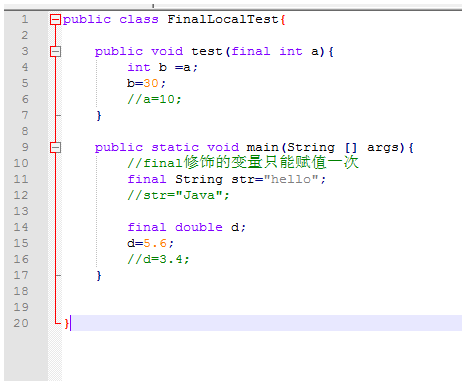
1、修饰类代表这个类不能被继承

2、修饰方法代表这个方法不能被覆盖

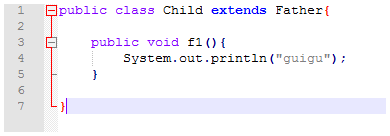
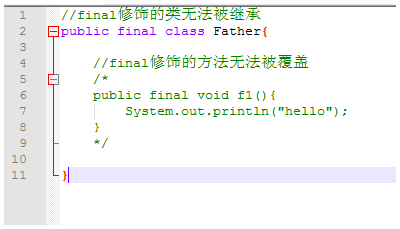
3、修饰变量代表这个变量的值不能改变

### Final修饰变量





### final修饰类和方法



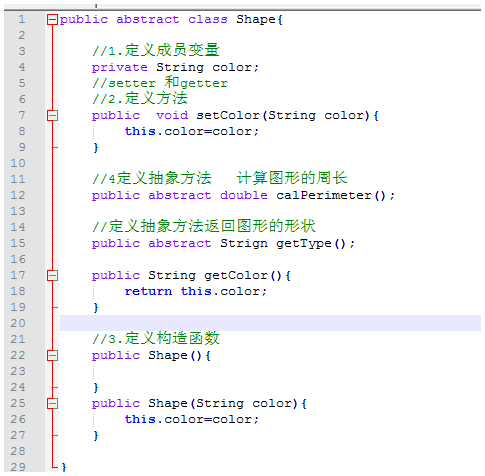
## 抽象类

### 抽象类的概念

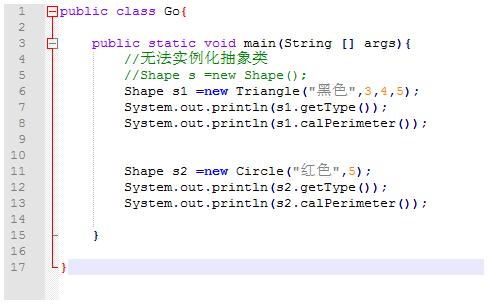
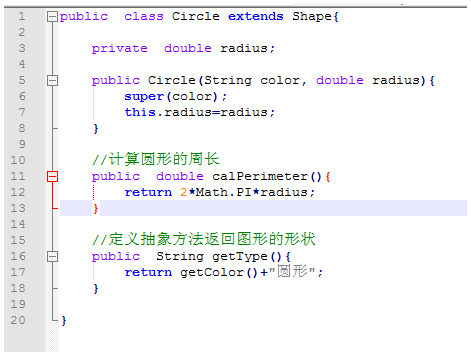
抽象类和抽象方法都必须使用abstract关键字修饰，有抽象方法的类只能被定义为抽象类。抽象类中可以没有抽象方法。

抽象方法和抽象类的使用规则如下：

* 抽象类 必须使用abstract关键字修饰，抽象方法也必须使用abstract修饰，抽象方法是指没有方法体的方法 Public void f1(){ }; public abstract void f2();
* 抽象类不能被实例化 也就是不能被new
* 抽象类中可以包含成员变量，方法，构造器，初始化代码块，内部类
* 抽象类中定义的抽象方法是依赖于子类实现，子类继承抽象类必须实现抽象类中全部的抽象方法





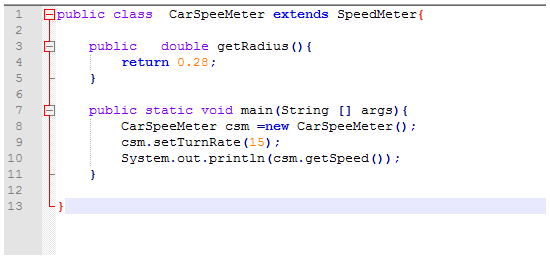


### 抽象类的作用

抽象类是从多个具体类中抽象出来的父类，从多个具有相同特征的类中抽象出一个抽象类，以这个抽象类为模板，从而避免子类的随意设计。

抽象类体现的是一种叫做模板策略。抽象类是多个子类通用的模板。





### 抽象类注意的细节

1、abstract可以修饰类、修饰方法、 但是不能修饰变量

2、有抽象方法的类 一定是抽象类

3、抽象类中 可以有构造函数 而且构造函数是可以重载的

4、抽象类不能和那些关键字共存

Final 不能和 abstract共存

Private 不能和abstract共存

Static 不能和abstract共存

## Java8改进的接口

抽象类从多个类抽象出来的模板，如果将这种抽象进行的更加彻底，则可以提炼出一种更加特殊的抽象类------接口。接口中不能有普通方法，也就是没有方法的实现Jdk8对接口进行了改进，允许在接口中定义默认方法和类方法

### 接口的定义

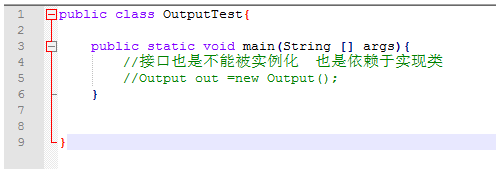
和类的定义不同，定义接口不再使用class关键字 而是使用interface

修饰符 interface 接口名 extends 父接口1, 父接口2{

}

1. 接口中定义的变量就是常量，并且系统会自动为成员变量添加 public static final
2. 接口是无法被实例化的





### 接口的使用

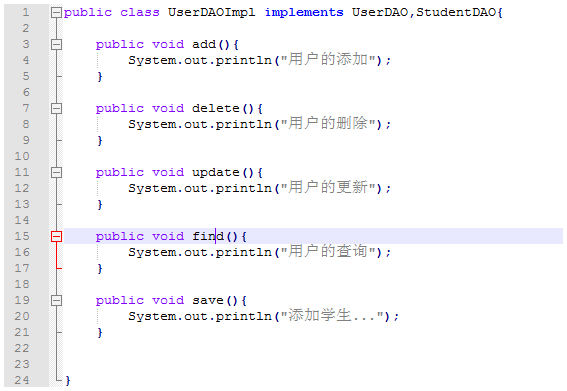
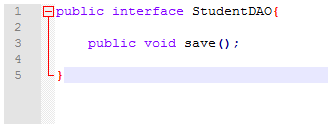
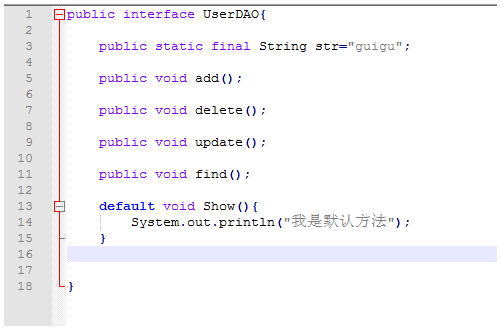
类与类之间的关系是继承 extends

类与接口之间的关系是实现 implements

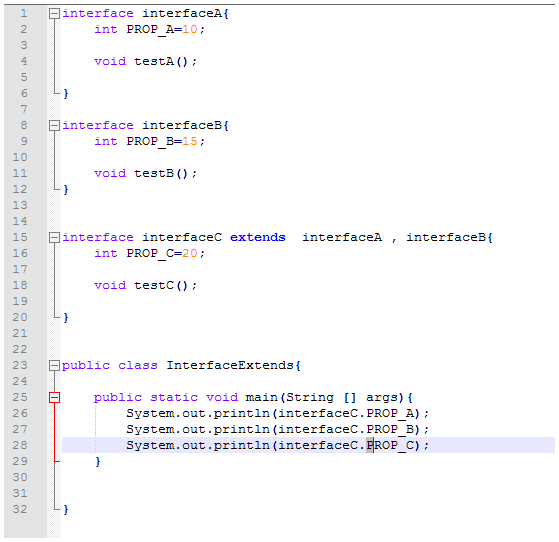
接口与接口之间的关系也是继承 extends

实现类实现了接口，必须实现接口中定义的全部的方法

一个类可以实现多个接口



### 接口的继承



### 接口和抽象类的比较

接口和抽象类很相似，他们具有如下特征:

* 接口和抽象类不能被实例化
* 接口和抽象类中都可以包含抽象方法，而子类也必须实现父类中定义的全部抽象方法

除此以外还有如下不同：

* 接口只能包含抽象方法和默认方法，不能有普通方法，而抽象类中可以有普通方法
* 接口中定义的变量就是常量（接口中的变量只能是public修饰或者不写 系统默认添加public） 抽象类中可以定义普通成员变量
* 一个类只能有一个直接的父类，而接口可以继承多个父接口，一个类也可以实现多个接口

## 内部类

大部分时候，类被定义为一个独立的单元，在某些情况下也会把一个类放入到另外一个类中，这个定义在其他类内部的类称为内部类，这个包含内部类的类称为外部类。

内部类分为四种情况，非静态内部类，静态内部类，局部内部类(了解),匿名内部类。

### 非静态内部类

1、非静态内部类 可以访问外部类任意成员吗?

非静态内部类可以无条件访问任意外部类成员

2、内部类和可以和外部类同名吗?

内部类不可以和自己的外部类同名，但是可以和自己外部类同级目录的外部类同名

1. 外部类 能访问内部类的成员吗？

不能

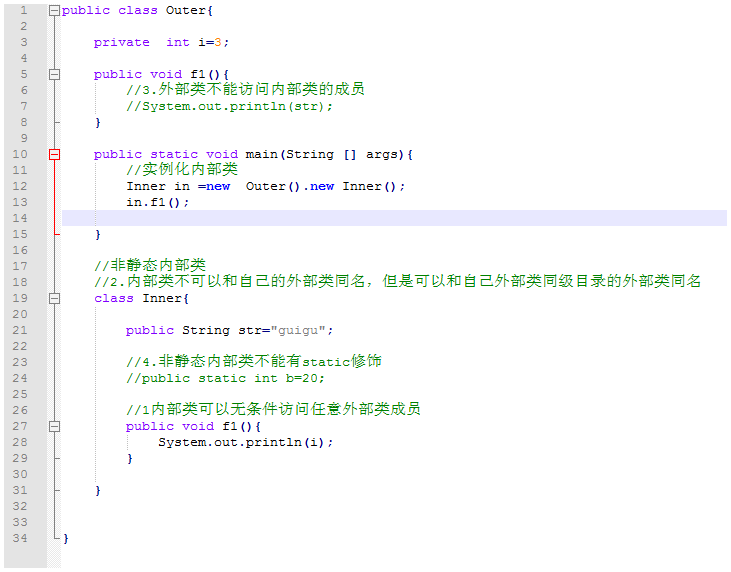
4、非静态内部类能否有static修饰

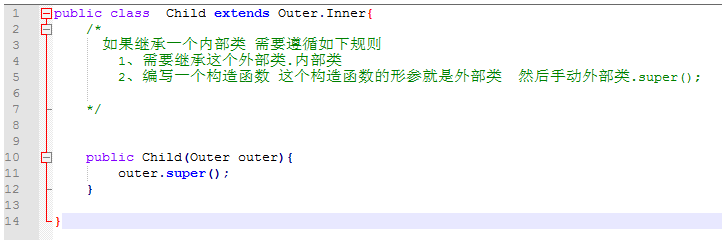
不能

5、内部类能否被实例化?

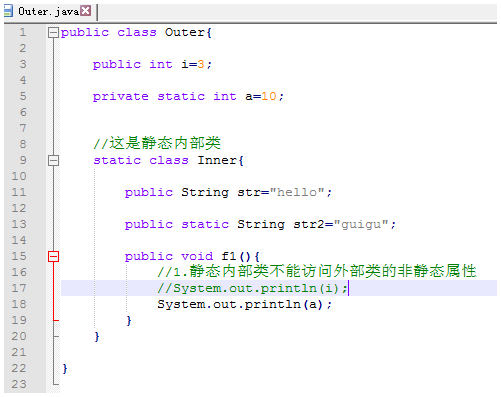
能在本类中实例化

6、内部类能否被其他类继承



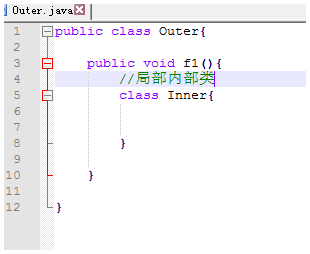


### 静态内部类



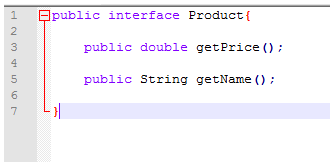
### 局部内部类(了解)

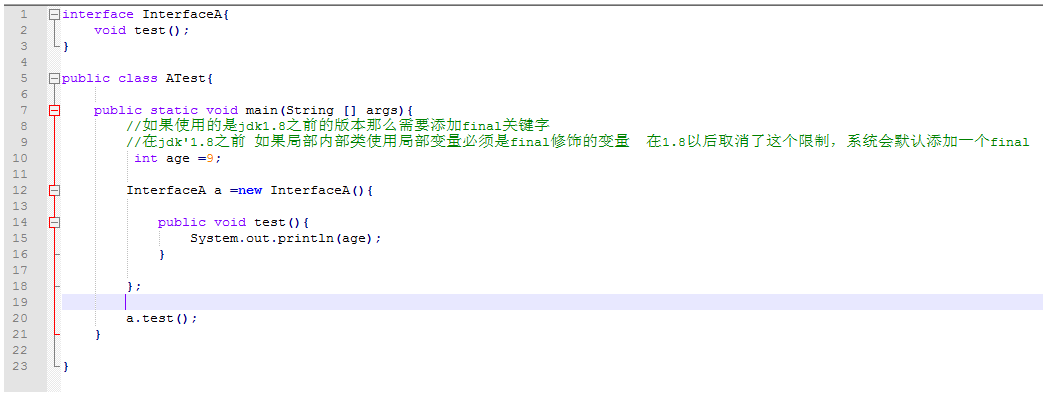
所谓的局部内部类就是把一个类定义在一个方法体内。



### Java8改进的匿名内部类

匿名内部类是说适合只创建一次使用的类。只能使用一次。

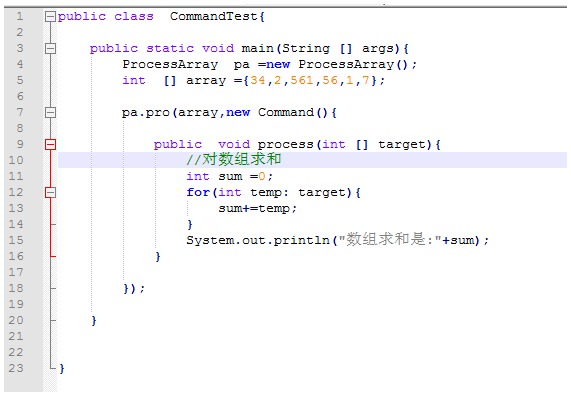
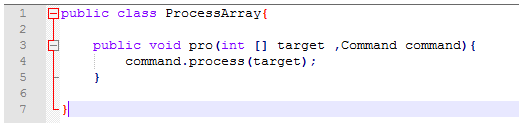
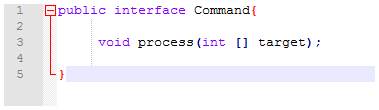




## Java8新增的Lambda表达式

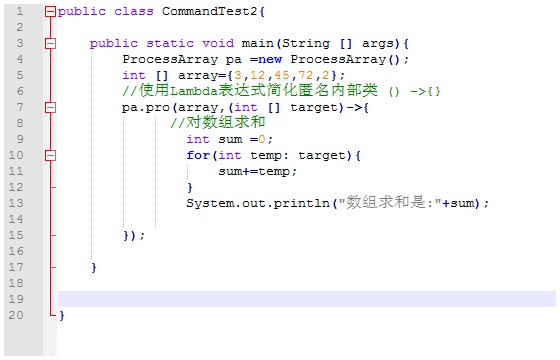
Lambda表达式是Java8重要的更新，Lambda表达式支持将代码作为方法的参数，允许使用更简洁的代码创建只有一个抽象方法的接口（这种接口也被称为函数式接口）。

### Labmda表达式入门



Lambda表达式完全用于简化创建匿名内部类对象。

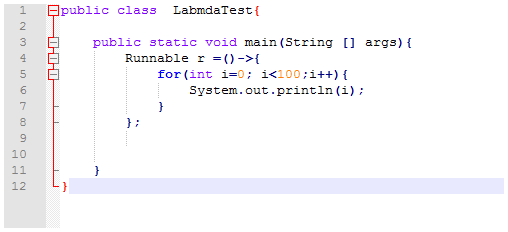
* ()里是形参，形参允许省略形参类型 如果形参只有一个参数 连小括号也可以省略
* ->
* 代码块



### Lambda表达式和函数式接口

Lambda表达式类型也被称为目标类型， Lambda表达式的目标类型必须是函数式接口。所谓的函数式接口是指只包含一个抽象方法的接口。函数式接口可以包含很多默认方法，或者类方法。只能声明一个抽象方法。

Java8专门为函数式接口定定义了一个注解@FunctionalInterface，该注解通常是放在接口定义的头部，该注解对程序没有任何作用，它仅仅是一个标识，它告诉编译器更严格的检测该接口是否是函数式接口，否则编译器报错。

Lambda表达式会有如下两个限制:

* Lambda表达式 目标类型必须是明确的函数式接口
* Lamda表达式 只能为函数式接口创建对象，Lambda表达式只能实现一个方法，所以只能有一个抽象方法

为了保证Lambda表达式目标类型是一个明确的函数式接口，可以有如下三种方式：

* 将Lambda表达式赋值给函数式接口的变量
* 将Lambda表达式作为函数式接口类型传递给某个方法。
* 使用函数式接口对Lambda表达式强制类型转换

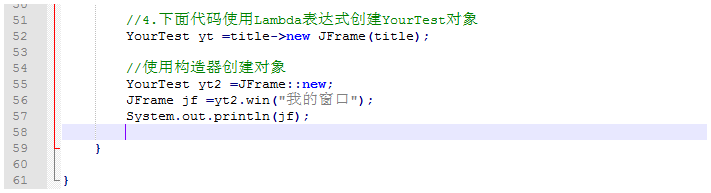
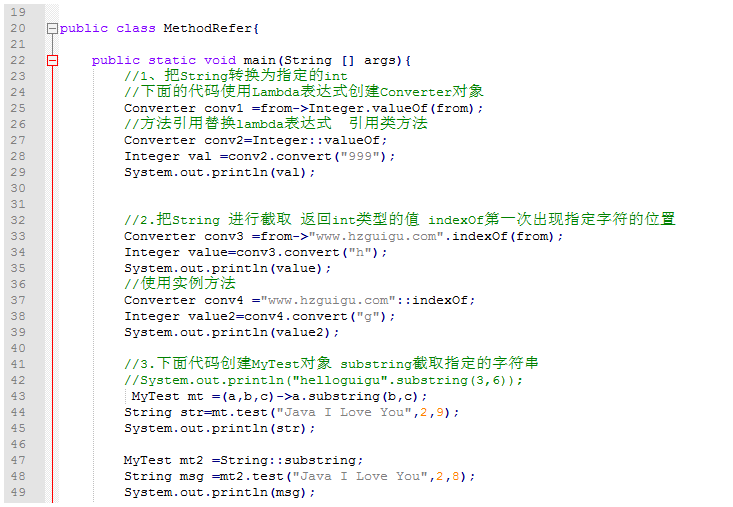
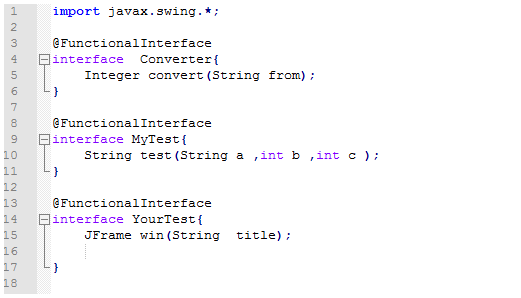


### 方法引用和构造器引用

如果Lambda表达式只有一条代码，程序可以省略{}。

方法引用和构造器引用可以让Lambda表达式更加简洁，方法引用和构造器引用都需要使用英文冒号Lambda表达式支持如下几种引用:

|  |  |  |  |
| --- | --- | --- | --- |
| 种类 | 示例 | 说明 | 对应的Lambda表达式 |
| 引用类方法 | 类名::类方法 | 函数式接口中被实现方法全部参数传递给该类方法作为参数 | (a,b…)->类名.方法名(a,b…) |
| 引用特定对象的实例方法 | 特定对象::实例方法 | 函数式接口中被实现全部参数传递给该方法作为参数 | (a,b…)->特定对象.实例方法(a,b…) |
| 引用某种类对象的实例方法 | 类名::实例方法 |  | (a,b…)->a.实例方法(b,..) |
| 引用构造器 | 类名::new |  | (a,b..)->new 类名(a,b..) |



### Lambda表达式和 匿名内部类的区别

Lambda表达式是匿名内部类的一种简化，它可以取代匿名内部类的作用，Lambda表达式和内部类有如下相同点:

* Lambda表达式和匿名内部类都可以直接访问 final修饰的局部变量，以及外部类的成员变量
* Lambda表达式创建的对象与匿名内部类生成的对象是一样的，都可以直接从接口中继承默认方法

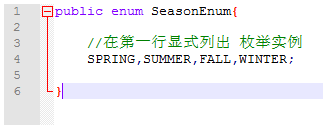
## 枚举类

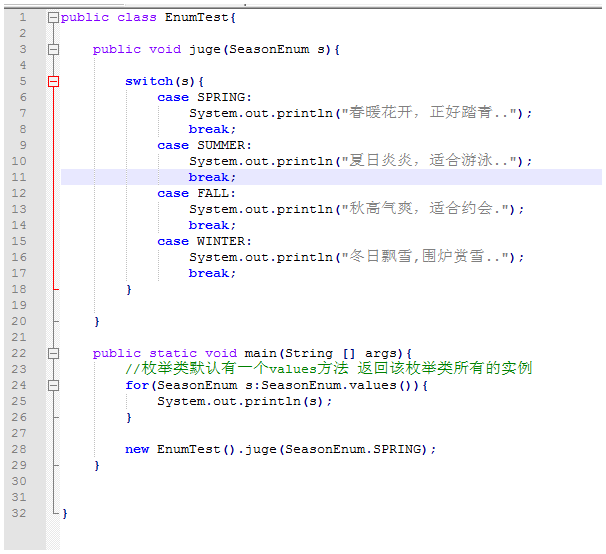
### 枚举入门

Java5新增了一个enum关键字，它和class 以及interface地位相同。 用enum定义枚举类，枚举类是一个特殊的类，它可以拥有自己的成员变量，方法，可以实现一个多个接口。

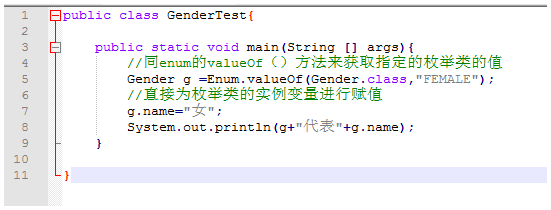
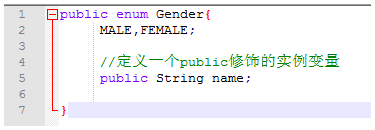
枚举类毕竟不是普通的类它和普通类有很多不同。

* 枚举类可以实现一个或者多个接口，使用Enum类默认继承了java.lang.Enum类，而其他类默认继承java.lang.Object类，所以枚举类不能显示继承其他类。
* 枚举类的构造器只能使用private修饰，如果省略了修饰符默认是private.如果强制指定也只能是private
* 使用enum定义。非抽象的枚举类默认使用final修饰，所以枚举类不能派生子类。
* 枚举类的所有实例弄下在枚举类第一行显式列出，否则这个类永远不能产生实例。

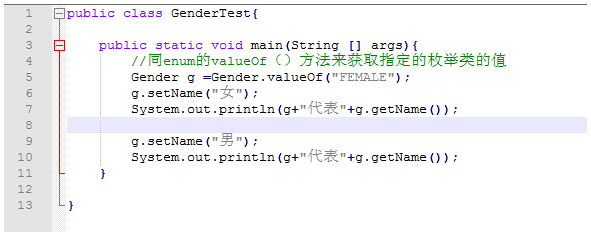




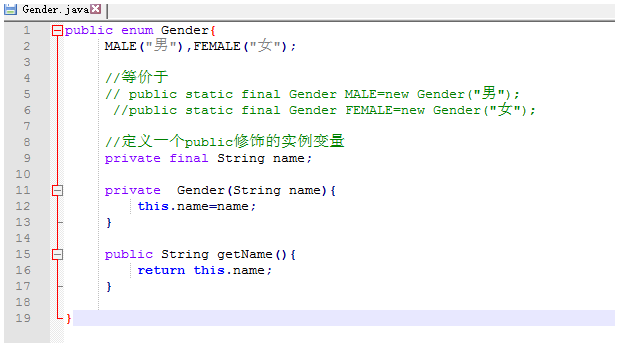
### 枚举类的成员变量和方法



Java应该有良好的封装，所以不允许直接方法Gender类中的name成员变量



这种做法还不是最好，枚举类应该设计为不可变类，也就是说 它的成员变量不应该允许改变，这样做会更安全，代码更加简洁，所以建议将枚举类的成员变量设计为private final修饰。



## 对象和垃圾回收

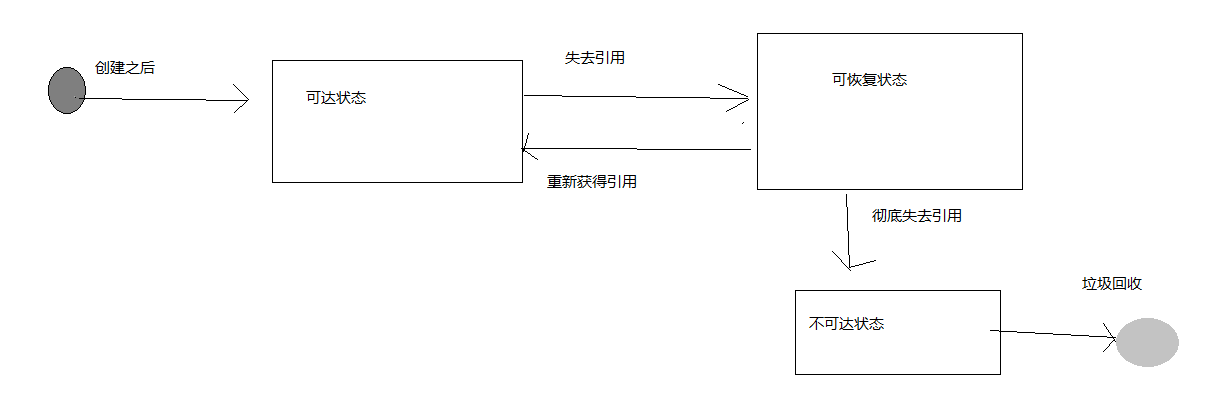
Java的垃圾回收是Java语言最重要的功能之一，当程序创建对象，数据等引用类型的实体，系统会在堆内存中分配一块区域，对象就是保存在这个内存区域中。当这块内存区域不再被任何引用变量引用的时候，这块内存就变成垃圾，等待垃圾回收机制进行回收。垃圾回收机制具有以下特点:

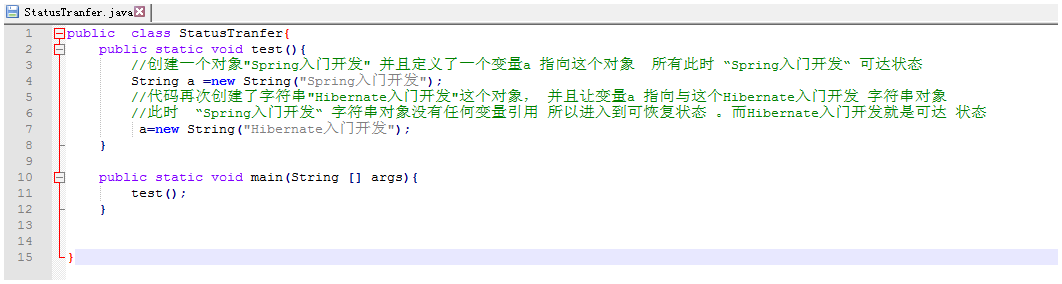
* 垃圾回收只负责回收堆内存中的对象，不会回收任何物理资源(比如数据库连接，网络io等资源)。
* 程序无法精准的控制垃圾回收的运行，垃圾回收会在合适的时候自动执行，当对象永久的失去引用，系统会在合适的时候进行回收它所占的内存。
* 在垃圾回收对象之前，总会调用它的finalize()方法，该方法可能使该对象重新复活，从而导致垃圾回收机制取消。

### 对象在内存中状态

当一个对象在内存中运行时候，根据他引用变量的状态，可以把他所处的状态分为三种：

* 可达状态：当一个对象被创建出来后，如果有一个以上的变量引用它，则这个对象处于可达状态，程序可以通过引用变量来调用该对象的是实例变量和方法。
* 可恢复状态：如果程序中某个对象不再被任何引用变量引用它，它就是处于可恢复状态，在这种状态下，系统的垃圾回收机制，准备回收该对象所占的资源，系统会调用finalize()方法判断该对象是否可以进入到可达状态，。如果可以达到则取消垃圾回收，否则进入到不可达状态。
* 不可达状态： 当对象与所有引用变量的关系都被切断，并且系统已经调该对象的finalize()方法依然无法到达可达状态，那么这个对象将永久性的失去引用，最后变为不可达状态。只有当一个对象变为不可达状态，系统才会真正的执行垃圾回收。





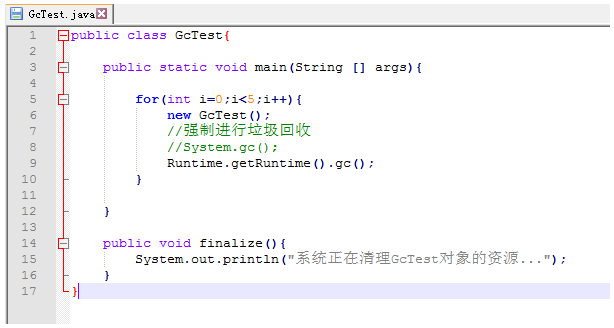
### 强制垃圾回收

当一个对象失去引用后，系统会调用它的finalize()方法对他进行资源的清理,何时它会变为不可达引用变量，在进行垃圾回收。 绝不能控制它何时被回收。

程序无法精准的控制java的垃圾回收机制，但是依然可以强制系统进行垃圾回收---这种强制垃圾回收，只是通知系统进行垃圾回收，但是是否执行垃圾回收依然不确定，大部分时候程序强制垃圾回收后总是有一些效果。

强制垃圾回收机有如下两种情况

* System类的gc静态方法
* 调用runntime对象的gc实例方法



java -verbose:gc GcTest

这个命令可以更加详细的查阅垃圾回收的提升信息

### Finalize方法

当垃圾回收机制对某个对象回收之前，通常调用适当的方法进行清理资源，在没有明确指定清理资源的情况下，java提供了默认机制清理这个对象的资源，这个机制就是finalize()方法.

Finalize方法的特点:

* 永远不要主动调用某个对象的finalizefangf，该方法应该交给垃圾回收机制进行调用
* Finalize方法何时被调用，是否被调用具有不确定，不要把finalize方法当成一个一定会执行的方法。
* 当jvm执行可恢复对象的finalize方法，可能使该对象会系统中的其他对象重新变为可达状态
* 当jvm执行finalize方法出现异常时，垃圾回收机制不会报告异常。程序可以继续执行。



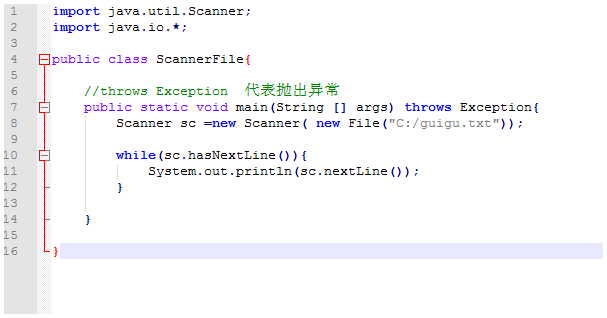
# Java基础类库

## 系统相关

### Scanner获取键盘输入



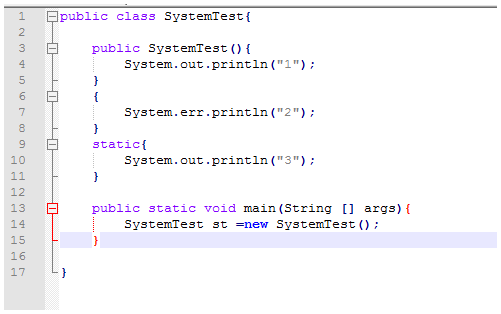
Scanner不仅能读取键盘的输入还能读取文件中的内容。

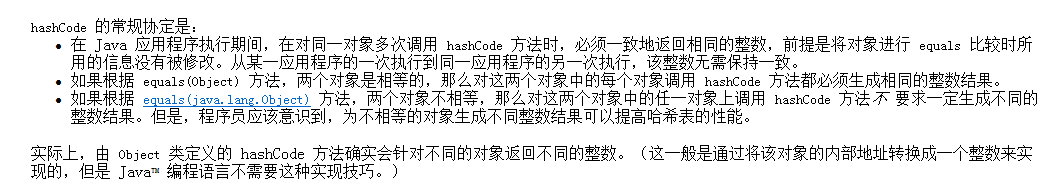


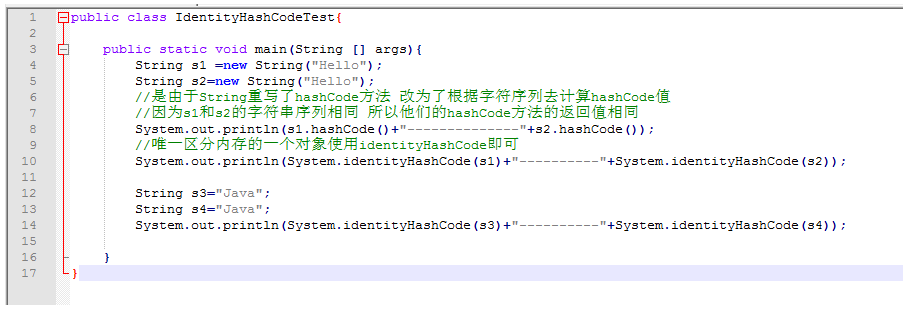
### System类

System代表当前java程序运行的平台，程序不能创建System类的对象， System类提供了一些类变量和类方法

其中System类中的err 是带有缓存的输出不是立即输出 而是缓存区域满了，或者程序执行完毕再执行输出。







### Runtime类

Runtime类也是代表Java程序运行时的环境，每个java程序都有一个与之对应的Runtime实例，它是应用程序与运行环境进行相连的类。



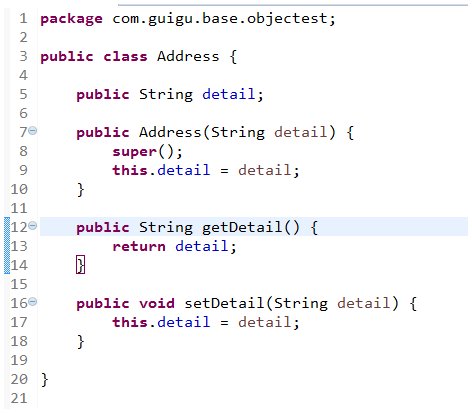
## 字符串

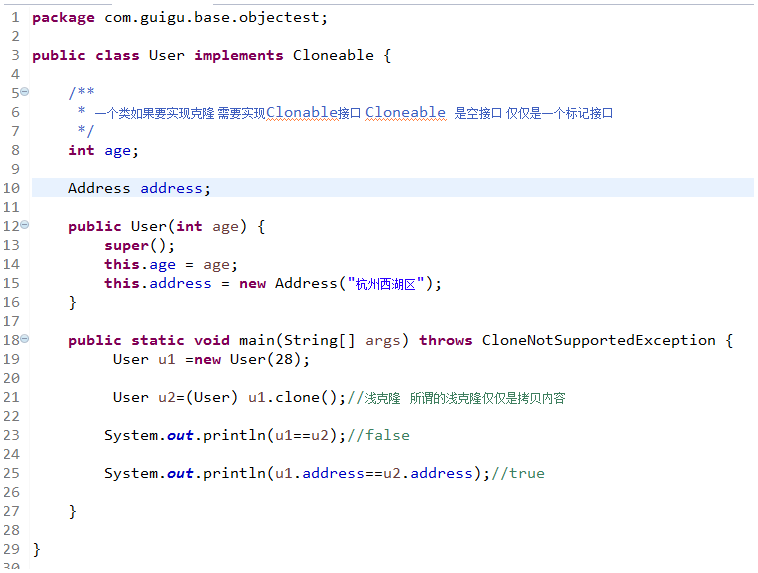
### Object类和Java7新增的Objects类

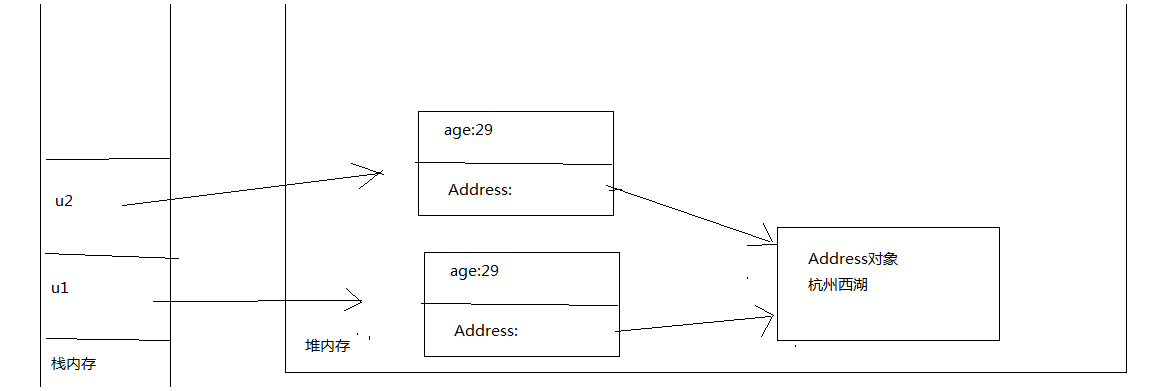
Object类是所有类、数组的父类，也就是说java允许任何类型的对象赋值给Object类型的变量 。在任何类中都可以调用Object的方法。

#### Object对象的浅克隆

Java还提供了一个protected修饰的clone的方法，该方法是用于帮助其他对象实现所谓的自我克隆。得到当前对象的一个副本。但是两者之间是完全隔离的，由于Object类提供的clone方法 使用Protected修饰所以只能在子类中重写或者调用。







#### Objects工具类

Java7新增了一个Objects工具类，它提供了一些工具方法操作对象，这些工具方法大多数是空指针安全的。比如你不确定一个引用变量是否为null，如果贸然使用toString方法则系统会抛出NullPointExcetion但是使用Objects类提供的toString方法 就不会引发空指针异常



### String类

**package** com.guigu.base.stringtest;

**public** **class** StringTest {

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

String s1 = "Hello world";

// charAt(int index) 返回指定索引处的char值

System.***out***.println(s1.charAt(3));// l

// compareTo

String s2 = "hello guigu";

System.***out***.println(s1.compareTo(s2));// 比较g103和w119的ASCII的差值

// 忽略大小写 继续比较

System.***out***.println(s1.compareToIgnoreCase(s2));

// concat 两个字符串连接在一起

System.***out***.println(s1.concat(s2));

System.***out***.println(s1 + s2);

// contains是否包含指定的内容

System.***out***.println(s1.contains("llo"));

System.***out***.println(s2.contains("ow"));

// 判断是否以指定的内容开始和结束

System.***out***.println(s1.endsWith("rld"));

System.***out***.println(s2.endsWith("uig"));

System.***out***.println(s1.startsWith("He"));

// 可以把一个String字符串转换为byte字节数组

**byte**[] buffer = s1.getBytes();

**for** (**byte** b : buffer) {

System.***out***.println((**char**)b);

}

//indexOf第一次出现指定字符的位置 lastIndexOf最后一次出现指定字符的位置

//s1 =Hello world

String s3="Hello guigu lh";

System.***out***.println(s3.indexOf("l"));

//从第4个字符以后开始出现l的位置

System.***out***.println(s3.indexOf("l", 4));

//最后一次出现指定字符的位置

System.***out***.println(s3.lastIndexOf("l"));

//isEmpty 判断字符串是否为空

System.***out***.println(s1.isEmpty());

//返回字符串的长度 字符串长度是length方法() 而数组的长度是length属性

System.***out***.println(s1.length());

**int** [] arr = {1,45,561,23,67};

System.***out***.println(arr.length);

//replace替换指定的内容

String s4="Hello World lloAbc ll";

System.***out***.println(s4);

System.***out***.println(s4.replace("llo", "liuhuan"));

//Split切割字符串

String str5="aaa,bbb,ccc";

//把Str5按照逗号进行分割

String [] splitStr=str5.split(",");

**for** (String ss : splitStr) {

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

}

//subString 截取字符串

String str6="Hello guigu,Hello Java";

//把前方4个字符截取掉

System.***out***.println(str6.substring(4));

System.***out***.println(str6.substring(2,9));

//toCharArray 把字符串转换为cha类型的数组

**char** [] ch=str6.toCharArray();

**for** (**char** c : ch) {

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

}

String str7="AbcDefGHIJ";

//转换为大写或者小写

System.***out***.println(str7.toUpperCase());

System.***out***.println(str7.toLowerCase());

//trim 去除首尾的空格

String str8=" Hello guigu ";

System.***out***.println(str8.length());

System.***out***.println(str8.trim().length());

// valueOf是把基本类型转换为字符串

}

}

1. 判断邮箱是否合法

Hzguigu@163.com

1. 必须有且只有一个@
2. @不能出现在头部或者尾部
3. @和. 不能同时出现
4. @之后不超过15位
5. @之后必须有.
6. @之后 是以com cn net 或者org结尾

**package** com.guigu.base.stringtest;

**public** **class** CheckEmail {

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

String email = "hzgg@163.com";

*printResult*(email);

}

**public** **static** **void** printResult(String email) {

**boolean** result = *checkEmail*(email);

**if** (result) {

System.***out***.println("邮箱是一个合法的邮箱");

} **else** {

System.***out***.println("邮箱格式不正确..");

}

}

**public** **static** **boolean** checkEmail(String email) {

**if** (!(email.contains("@") && email.indexOf("@") == email.lastIndexOf("@"))) {

**return** **false**;

}

**if** (email.startsWith("@") || email.endsWith("@")) {

**return** **false**;

}

**if** (email.contains("@.") || email.contains(".@")) {

**return** **false**;

}

String subStr = email.substring(email.indexOf("@") + 1);

**if** (subStr.length() > 15) {

**return** **false**;

}

**if** (!subStr.contains(".")) {

**return** **false**;

}

**if** (!(subStr.endsWith("com") || subStr.endsWith("cn") || subStr.endsWith("org") || subStr.endsWith("net"))) {

**return** **false**;

}

**return** **true**;

}

}

### StringBuffer 和StringBulider

StringBuilder 提供一个一系列的插入、追加、改变字符串包含的字符序列的方法，而StringBuffer用法和StringBulider完全相同， 只是StringBuffer是线程安全的。

StringBulider和StringBuffer是一个可变字符集，String是不可变字符集

**package** com.guigu.base.stringtest;

**public** **class** StringBuliderTest {

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

// StringBuilder sb =new StringBuilder();

StringBuffer sb =**new** StringBuffer();

//初始化容量是16

// System.out.println(sb.capacity());

//追加字符串

sb.append("java"); //sb="java"

//插入

sb.insert(0, "hello "); //sb= "hello java";

//替换 包含左侧 不包含右侧

sb.replace(5, 6, ",");//sb="hello,java"

//删除

sb.delete(5, 6);//sb="hellojava"

//反转

// sb.reverse();

//长度

System.***out***.println(sb.length());

System.***out***.println(sb.capacity());

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

//append可以拼接任意类型

System.***out***.println(sb.append(1).append(**true**).append("abc").append(**new** StringBuliderTest()).toString());

System.***out***.println(sb.length());

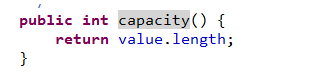
System.***out***.println(sb.capacity());

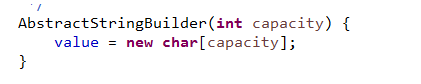
}

}

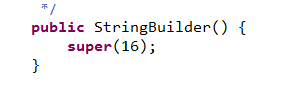
Capacity解析

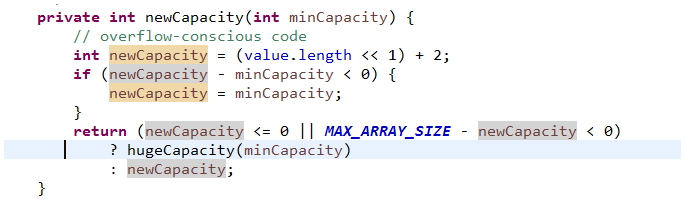
AbstractStringBuilder





StringBuilder类中 继承于AbstractStringBuilder

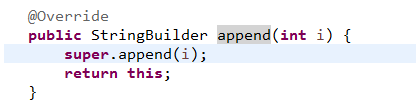




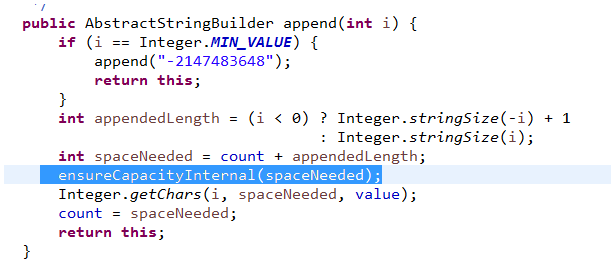
初始大小是16的容量

每次append 要先判断是否需要扩容

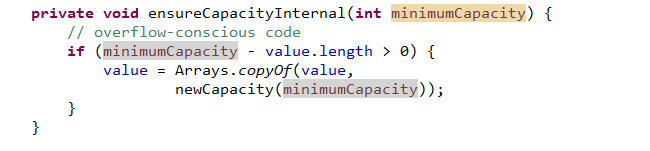
Stringbulider中的append

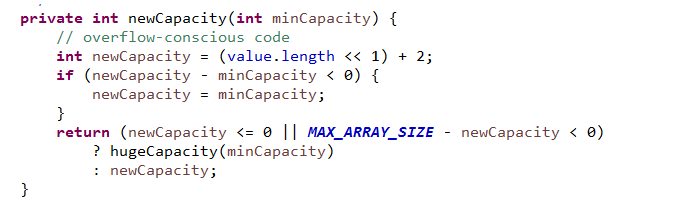


调用父类的append



判断是否需要扩容





#### StringTokenizer

进行字符串的切割 不需要满足正则表达式

**package** com.guigu.base.stringtest;

**import** java.util.StringTokenizer;

**public** **class** StringSplitTest {

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

String str = "aaa！@#￥%^&\*bbb！@#￥%^&\*ccc！@#￥%^&\*ddd";

// 切割要满足 正则表达式

// String[] splitStr = str.split("！@#￥%^&\*");

StringTokenizer st = **new** StringTokenizer(str, "！@#￥%^&\*");

**while** (st.hasMoreTokens()) {//如果切割以后 有下一个数据则取出下一个数据

System.***out***.println(st.nextToken());

}

}

}

## 数学数字相关的

### Math类

**package** com.guigu.base.mathtest;

**public** **class** MathTest {

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

**int** a = -10, b = 30;

**double** c = 3.14, d = 3.94;

System.***out***.println(Math.*floor*(c));

System.***out***.println(Math.*floor*(d));

System.***out***.println(Math.*round*(c));

System.***out***.println(Math.*round*(d));

System.***out***.println(Math.*abs*(a));

System.***out***.println(Math.*sin*(Math.*toRadians*(b)));

// 1-100之间的随机数

System.***out***.println((**int**) (Math.*random*() \* 100));

// 求 4的8次方

System.***out***.println(Math.*pow*(4, 8));

**int** max = Math.*max*(a, b);

**double** min = Math.*min*(c, d);

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

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

}

}

### Random类和Java7新增的ThreadLocalRandom

Random类生产一个随机的伪数。

**package** com.guigu.base.mathtest;

**import** java.util.Random;

**public** **class** RandomTest {

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

//得到当前系统的时间 以long类型的毫秒数表示

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

Random r =**new** Random();

System.***out***.println("取下一个Boolean:"+r.nextBoolean());

//取0-100之间的随机数

System.***out***.println("取下一个int:"+r.nextInt(100));

//Raodom是使用一个48位的种子，如果两个数字序列相同 可能产生相同的数据

// Random r2 =new Random(50);

Random r2 =**new** Random(System.*currentTimeMillis*());

System.***out***.println("第一个种子为50的random对象");

System.***out***.println("r2.nextBoolean:"+r2.nextBoolean());

System.***out***.println("r2.nextInt:"+r2.nextInt());

System.***out***.println("r2.nextDouble:"+r2.nextDouble());

Random r3 =**new** Random(System.*currentTimeMillis*());

System.***out***.println("第二个种子为50的random对象");

System.***out***.println("r3.nextBoolean:"+r3.nextBoolean());

System.***out***.println("r3.nextInt:"+r3.nextInt());

System.***out***.println("r3.nextDouble:"+r3.nextDouble());

}

}

ThreadLocalRandom功能和Radom类似 只是这个类是在多线程环境下使用，可以保证安全性和唯一性

//不是同new的方获取 而是 通过单例模式的方式

ThreadLocalRandom r4 =ThreadLocalRandom.*current*();

//随机生成4-20之间的随机数

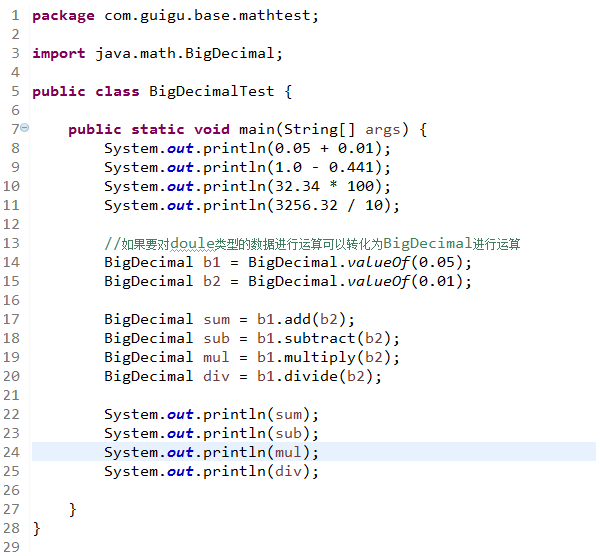
**int** va1 =r4.nextInt(4, 20);

System.***out***.println(va1);

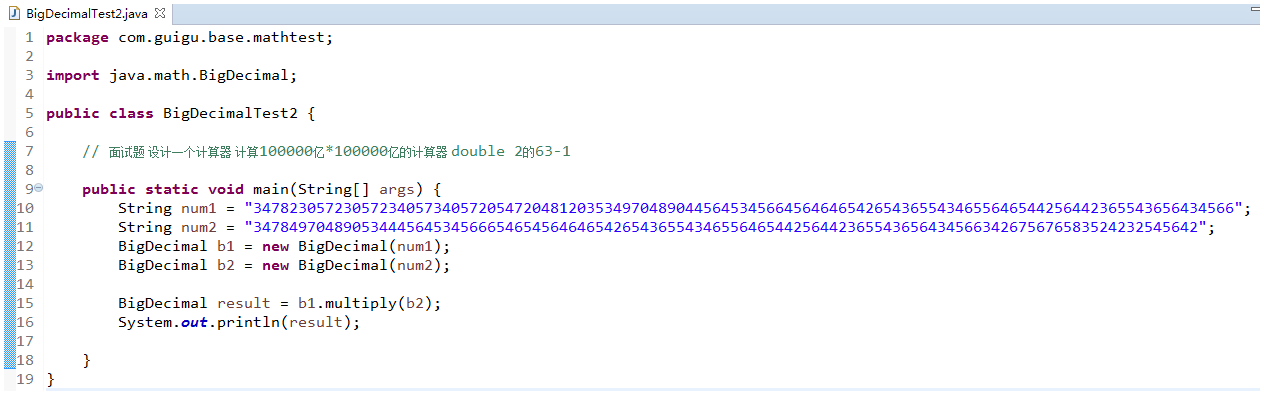
### BigDecimal

不可变的、任意精度的有符号十进制数。BigDecimal 由任意精度的整数非标度值 和 32 位的整数标度 (scale) 组成。如果为零或正数，则标度是小数点后的位数。如果为负数，则将该数的非标度值乘以 10 的负 scale 次幂。因此，BigDecimal 表示的数值是 (unscaledValue × 10-scale)。

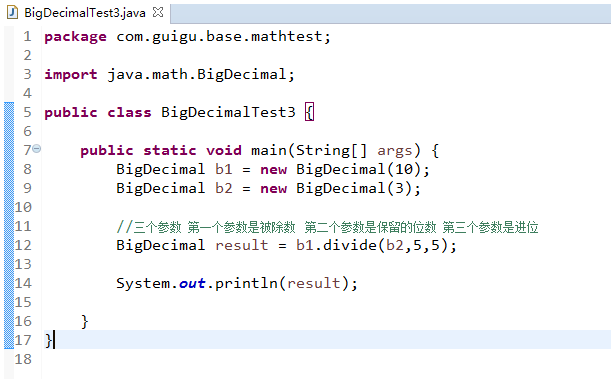
BigDecimal 类提供以下操作：算术、标度操作、舍入、比较、哈希算法和格式转换。toString() 方法提供 BigDecimal 的规范表示形式。

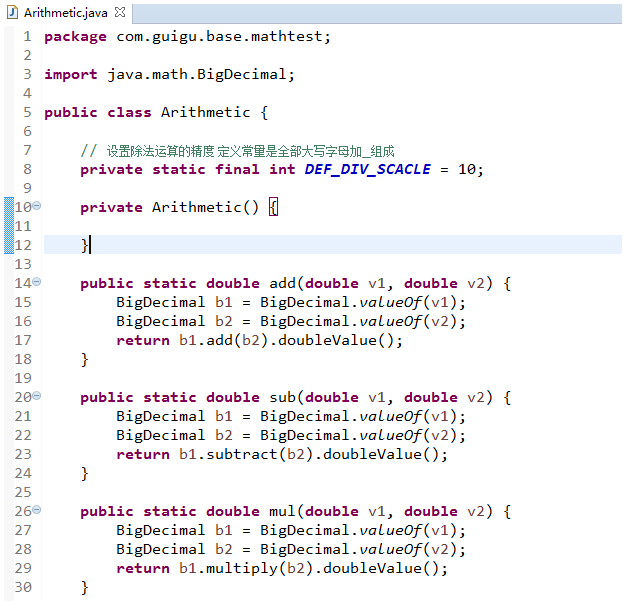


大数的运算



除法运算注意的细节问题

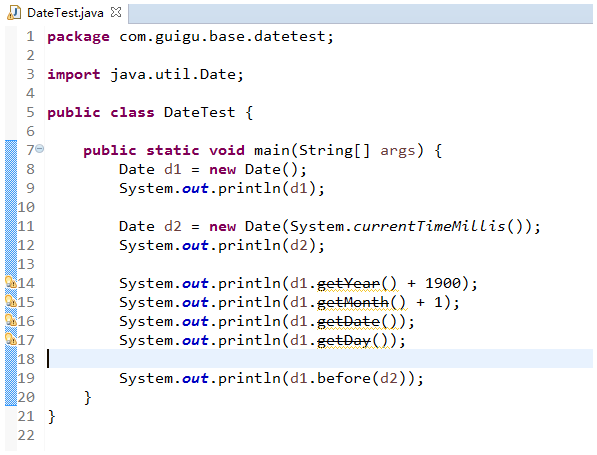




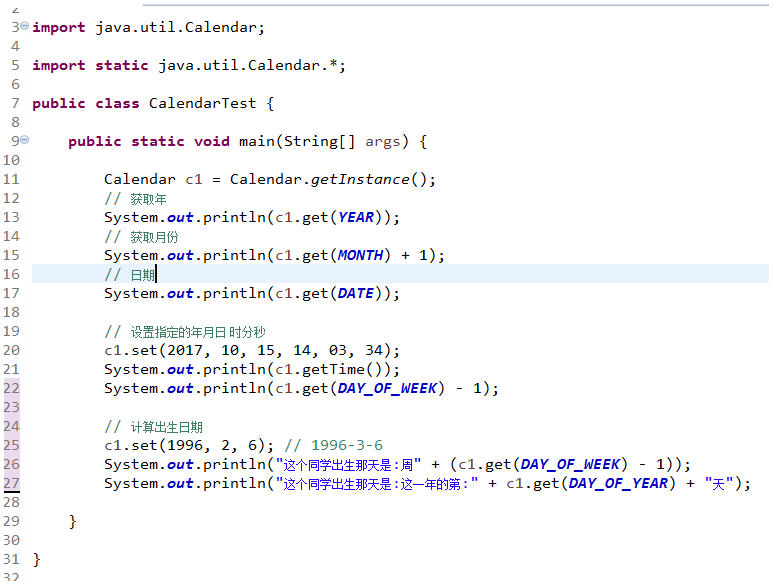
## Java8日期和时间类

Java原本提供了Date和Calendar用于处理日期和时间类，。包括创建日期，日期对象获取系统时间等。

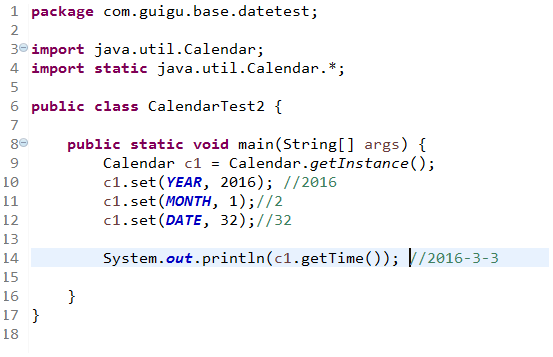
### Date



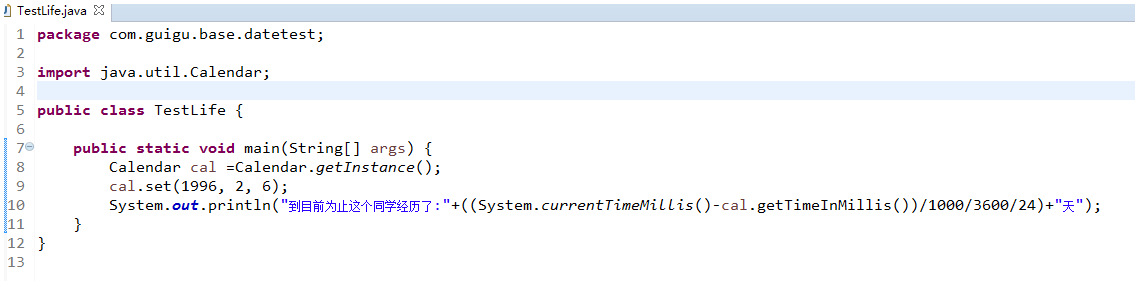
### Calendar



面试题



计算到目前为止自己活了多少天



### Java8新增的日期和时间包

Java8新增了一个java.time包，该包下有如下类：

* Clock 该类用于获取指定时区的当前时间，该类可以取代System类的currentTimeMillis方法，而且还提供了更多方法获取当前日期和时间 (课下阅读API文档)
* Duration该类代表持续时间，该类可以方便的获取一段时间，并完成转换
* Instant 该类代表一个具体的时刻 可以精确到纳秒 ，该类提供了静态的now方法来获取当前的时刻 ，除此以外还提供了now(Clock clock)这个方式获取Clock对象。提供了isBefore和isAfter判断日期的前后， 还提供了以一系列的minusXxx()方法 在当前时刻上减去指定的一段时间。 PlusXxx()，在当前时刻增加指定的一段时间。
* LocalDate 该类不带时区的 日期
* LocalTime 该类不带时区的 时间
* LocalDateTime不带失去的 日期时间

**package** com.guigu.base.datetest;

**import** java.time.Clock;

**import** java.time.Duration;

**import** java.time.Instant;

**import** java.time.LocalDate;

**import** java.time.LocalDateTime;

**import** java.time.LocalTime;

**import** java.time.Month;

**import** java.time.MonthDay;

**import** java.time.Year;

**import** java.time.YearMonth;

**public** **class** NewDatePackageTest {

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

// --colok 该类代表指定时区的时间

// 通过clock获取当前时间

Clock clock = Clock.*systemUTC*();

Clock clock2 = Clock.*systemDefaultZone*();

System.***out***.println(clock.instant());

// 获取Clock的毫秒数和System.currentTimeMillis()

System.***out***.println(clock.millis());

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

// ---Duration

Duration d = Duration.*ofSeconds*(6000);

System.***out***.println("6000秒相当于" + d.toMinutes() + "分");

System.***out***.println("6000秒相当于" + d.toHours() + "小时");

System.***out***.println("6000秒相当于" + d.toDays() + "天");

// 在colock的基础上 增加6000s然后返回一个新的Clock

Clock clock3 = Clock.*offset*(clock, d);

System.***out***.println("新增6000秒后 为:" + clock3.instant());

// Instance

// 获取当前时间

Instant instant = Instant.*now*();

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

// 使用Instance增加6000秒

Instant instant2 = instant.plusSeconds(6000);

System.***out***.println("新增6000秒后:" + instant2);

// 在当前日期上增加5小时4分钟

Instant instant3 = instant2.plus(Duration.*ofHours*(5).plusMinutes(4));

System.***out***.println(instant3);

// LocalDate 年月日的操作

LocalDate localDate = LocalDate.*now*();

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

// 获取2017年的第135天是哪一天

localDate = LocalDate.*ofYearDay*(2017, 135);

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

// 设置2017-11-11

localDate = LocalDate.*of*(2017, Month.***NOVEMBER***, 11);

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

// LocalTime 时分秒的操作

LocalTime localTime = LocalTime.*now*();

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

//设置时分秒

localTime=LocalTime.*of*(18, 16, 41);

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

LocalDateTime localDateTime =LocalDateTime.*now*();

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

//在当前日期加上25小时3分钟

LocalDateTime future=localDateTime.plusHours(25).plusMinutes(3);

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

//Year 年

Year year =Year.*now*();

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

year =year.plusYears(5);

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

YearMonth ym =YearMonth.*now*();

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

YearMonth ym2=year.atMonth(10);

System.***out***.println(ym2);

MonthDay monthDay =MonthDay.*now*();

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

MonthDay md2=monthDay.with(Month.***MAY***).withDayOfMonth(23);

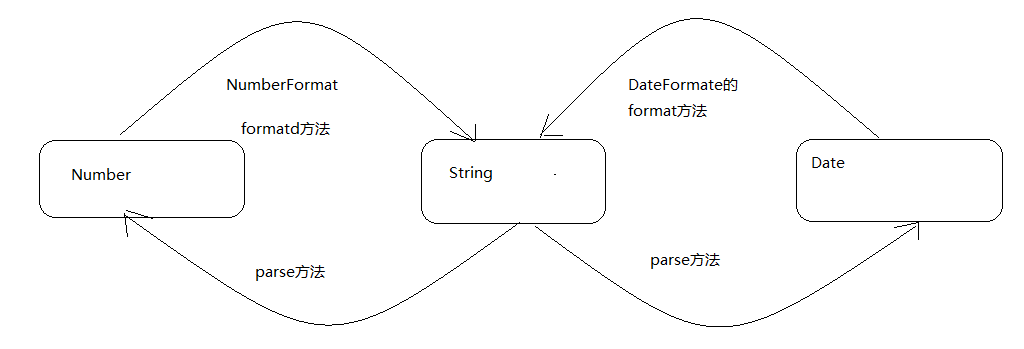
System.***out***.println(md2);

}

}

## 位符字符串

NumberFormat 、DateFormat、SimpleDateFormat



### NumberFormat

**package** com.guigu.base.format;

**import** java.text.NumberFormat;

**import** java.util.Locale;

**public** **class** NumberFormatTest {

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

// 格式化指定的数据

**double** db = 1234567.897;

// 把db解析为四个国家的数值格式

// 创建四个Local 分别代表 中国 日本 德国 美国

Locale[] locals = { Locale.***CHINA***, Locale.***JAPAN***, Locale.***GERMAN***, Locale.***US*** };

// 把上方的数值 分别转换为 数值格式 百分比格式 货币格式

NumberFormat[] nf = **new** NumberFormat[12];

// 为每个Local分别进行三种格式化

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

nf[i \* 3] = NumberFormat.*getNumberInstance*(locals[i]);

nf[i \* 3 + 1] = NumberFormat.*getPercentInstance*(locals[i]);

nf[i \* 3 + 2] = NumberFormat.*getCurrencyInstance*(locals[i]);

}

// 输出每个国家的三种格式化后的内容

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

String tip = i == 0 ? "------中国的格式-------": i == 1 ? "------日本的格式-------" : i == 2 ? "------德国的格式-------" : "------美国的格式-------";

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

System.***out***.println("通用的数值格式:"+nf[i\*3].format(db));

System.***out***.println("通用的百分比格式:"+nf[i\*3+1].format(db));

System.***out***.println("货币格式:"+nf[i\*3+2].format(db));

}

}

}

### DateFormat

**package** com.guigu.base.format;

**import** java.text.DateFormat;

**import** java.text.ParseException;

**import** java.util.Date;

**import** java.util.Locale;

**import** **static** java.text.DateFormat.\*;

**public** **class** DateFormatTest {

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

Date d = **new** Date();

// 创建两个Local 代表中国和美国

Locale[] locals = { Locale.***CHINA***, Locale.***US*** };

DateFormat[] df = **new** DateFormat[16];

// 为上面两个国家创建16个DateFormat对象进行日期的格式化

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

df[i \* 8] = DateFormat.*getDateInstance*(***SHORT***, locals[i]);

df[i \* 8 + 1] = DateFormat.*getDateInstance*(***MEDIUM***, locals[i]);

df[i \* 8 + 2] = DateFormat.*getDateInstance*(***LONG***, locals[i]);

df[i \* 8 + 3] = DateFormat.*getDateInstance*(***FULL***, locals[i]);

df[i \* 8 + 4] = DateFormat.*getTimeInstance*(***SHORT***, locals[i]);

df[i \* 8 + 5] = DateFormat.*getTimeInstance*(***MEDIUM***, locals[i]);

df[i \* 8 + 6] = DateFormat.*getTimeInstance*(***LONG***, locals[i]);

df[i \* 8 + 7] = DateFormat.*getTimeInstance*(***FULL***, locals[i]);

}

// 打印数据

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

String tip = i == 0 ? "---中国格式---" : "---美国格式---";

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

System.***out***.println("Short格式的日期格式:" + df[i \* 8].format(d));

System.***out***.println("Medium格式的日期格式:" + df[i \* 8 + 1].format(d));

System.***out***.println("Long格式的日期格式:" + df[i \* 8 + 2].format(d));

System.***out***.println("Full格式的日期格式:" + df[i \* 8 + 3].format(d));

System.***out***.println("Short格式的时间格式:" + df[i \* 8 + 4].format(d));

System.***out***.println("Medium格式的时间格式:" + df[i \* 8 + 5].format(d));

System.***out***.println("Long格式的时间格式:" + df[i \* 8 + 6].format(d));

System.***out***.println("Full格式的时间格式:" + df[i \* 8 + 7].format(d));

}

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

// 把字符串进行格式化为指定格式

String str1 = "2017-11-16";

String str2 = "2017年11月16日";

System.***out***.println(DateFormat.*getDateInstance*().parse(str1));

System.***out***.println(DateFormat.*getDateInstance*(***LONG***).parse(str2));

}

}

### SimpleDateFormat

**package** com.guigu.base.format;

**import** java.text.SimpleDateFormat;

**import** java.util.Date;

**public** **class** SimpleDateFormatTest {

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

Date d = **new** Date();

SimpleDateFormat sdf = **new** SimpleDateFormat("Gyyyy年中第D天");

// 将d格式化日期 输出 公元2017年中第几天

String datestr = sdf.format(d);

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

// SimpleDateFormat只解析关键词

SimpleDateFormat sdf2 = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

System.***out***.println(sdf2.format(d));

}

}

## 正则表达式

正则表达式是一个强大的字符串处理工具，可以对字符串进行查找，提取，分割，替换等操作。

Java提供的正则表达式的支持提供了Pattern和Matcher两个类进行处理。

**字符**

|  |  |
| --- | --- |
| *x* | 字符 *x* |
| \\ | 反斜线字符 |
| \0*n* | 带有八进制值 0 的字符 *n* (0 <= *n* <= 7) |
| \0*nn* | 带有八进制值 0 的字符 *nn* (0 <= *n* <= 7) |
| \0*mnn* | 带有八进制值 0 的字符 *mnn*（0 <= *m* <= 3、0 <= *n* <= 7） |
| \x*hh* | 带有十六进制值 0x 的字符 *hh* |
| \u*hhhh* | 带有十六进制值 0x 的字符 *hhhh* |
| \t | 制表符 ('\u0009') |
| \n | 新行（换行）符 ('\u000A') |
| \r | 回车符 ('\u000D') |
| \f | 换页符 ('\u000C') |
| \a | 报警 (bell) 符 ('\u0007') |
| \e | 转义符 ('\u001B') |
| \c*x* | 对应于 *x* 的控制符 |
|  |  |

正则表达式的特殊字符

$ 匹配一行的结束 要匹配$字符本身 \$

^ 匹配一行的开头

() 标记开始和结束的位置

[] 确定中括号的开始和结束

{} 可以用来标记前方表达式出现的次数(频率)

\* 指定的表达式可以出现零次或者多次

+ 指定的表达式 可以出现一次或者多次

? 指定的表达式可以出现零次或者一次

. 匹配除了换行符以外的任意单字符

\ 定义转移字符

| 指定两者之间选择一个

X? X，一次或一次也没有

X\* X，零次或多次

X+ X，一次或多次

X{n} X，恰好 n 次

X{n,} X，至少 n 次

X{n,m} X，至少 n 次，但是不超过 m 次

|  |  |
| --- | --- |
| [abc] | a、b 或 c（简单类） |
| [^abc] | 任何字符，除了 a、b 或 c（否定） |
| [a-zA-Z] | a 到 z 或 A 到 Z，两头的字母包括在内（范围） |
| [a-d[m-p]] | a 到 d 或 m 到 p：[a-dm-p]（并集） |
| [a-z&&[def]] | d、e 或 f（交集） |
| [a-z&&[^bc]] | a 到 z，除了 b 和 c：[ad-z]（减去） |
| [a-z&&[^m-p]] | a 到 z，而非 m 到 p：[a-lq-z]（减去） |

预定义字符类

|  |  |
| --- | --- |
| . | 任何字符（与行结束符可能匹配也可能不匹配） |
| \d | 数字：[0-9] |
| \D | 非数字： [^0-9] |
| \s | 空白字符：[ \t\n\x0B\f\r] |
| \S | 非空白字符：[^\s] |
| \w | 单词字符：[a-zA-Z\_0-9] |
| \W | 非单词字符：[^\w] |

**package** com.guigu.base.regtest;

**public** **class** RegDemo {

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

/\*\*

\* 正则表达式 是针对字符串进行操作的

\*

\*/

String QQ = "12345";

String reg = "[1-9][0-9]{4,14}";

**boolean** b = QQ.matches(reg);

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

String temp = "booooooooooooooooooooooooooooook";

String reg2 = "bo+k";

System.***out***.println(temp.matches(reg2));

// checkQQ(QQ);

}

// 要求QQ长度5-15之间 不能以0开头 不能有非数字

**public** **static** **boolean** checkQQ(String QQ) {

**boolean** temp = **false**;

**int** len = QQ.length();

**if** (len >= 5 && len <= 15) {

**if** (!QQ.startsWith("0")) {

// 如果能正确转换则代表正常 ，如果不能转换代表有非数字 进入到异常

**try** {

**long** l = Long.*parseLong*(QQ);

System.***out***.println("QQ正确:" + QQ);

temp = **true**;

} **catch** (Exception e) {

System.***out***.println("QQ中有非数字");

}

} **else** {

System.***out***.println("不能以0开头");

}

} **else** {

System.***out***.println("长度是5-15之间");

}

**return** temp;

}

}

案例二：

**package** com.guigu.base.regtest;

**import** java.util.regex.Matcher;

**import** java.util.regex.Pattern;

**public** **class** RegDemo2 {

/\*\*

\* 正则表达式是对字符串操作 1. 匹配 使用的就是字符串的matches(String reg);

\*

\* 2.切割 使用的是字符串中的split()方法

\*

\* 3.替换 使用的是字符串中的replaceAll()方法

\*

\* 4.获取 字符串中没有提供该功能 只能使用正则表达式取匹配

\* Pattern p=Pattern.compile("a\*b");

\* Matcher m =p.matcher("aaaaaab");

\* boolean b=m.matches();

\*

\*/

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

// checkPhone();

// splitDemo();

// replaceAllDemo();

*getString*();

}

**public** **static** **void** getString() {

String str="da jia zhu yi laa , ma shang xia ke la ";

//定义规则 匹配任意三个字符组成的 \b 是单词的边界

String reg="\\b[a-z]{3}\\b";

//将规则编译为正则表达式对象

Pattern p =Pattern.*compile*(reg);

//通过正则表达式 获取匹配对象

Matcher m =p.matcher(str);

**while**(m.find()) { //取出满足条件的内容

System.***out***.println(m.start()+"....."+m.group()+"......."+m.end());

}

}

**public** **static** **void** replaceAllDemo() {

String str="wer#####taayuiop\*\*\*fghkkqwer";

// str=str.replaceAll("(.)\\1+", "#");

//去除字符串中的重复 数据只保留一个 $1 是匹配前方第一组的数据

str=str.replaceAll("(.)\\1+", "$1");

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

//手机号 158\*\*\*\*1234

str="15845672584";

str=str.replaceAll("(\\d{3})\\d{4}(\\d{4})", "$1\*\*\*\*$2");

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

}

**public** **static** **void** splitDemo() {

String str="12 3 45 56 7 19 34 567";

String [] arr=str.split(" +");

**for** (String ss : arr) {

// System.out.println(ss);

}

// String str2="hangzhou.guigu.liuhuan";

// String reg2="\\.";

String str2="wer#####taayuiop\*\*\*fghkkqwer";

//对出现两次以的字符 为切割位置

//(.) 第一个任意字符 \\1代表获取第一组的数据 +是代表\\1可以出现一次或者多次 也就是说整体至少出现两次以上重复的字符

String reg2="(.)\\1+";

String [] arr2=str2.split(reg2);

**for** (String ss : arr2) {

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

}

}

// 校验手机号 13 14 15 17 18

**public** **static** **void** checkPhone() {

String tel = "18159643174";

String reg = "1[34578]\\d{9}";

**boolean** b = tel.matches(reg);

System.***out***.println(tel + ":" + b);

}

}

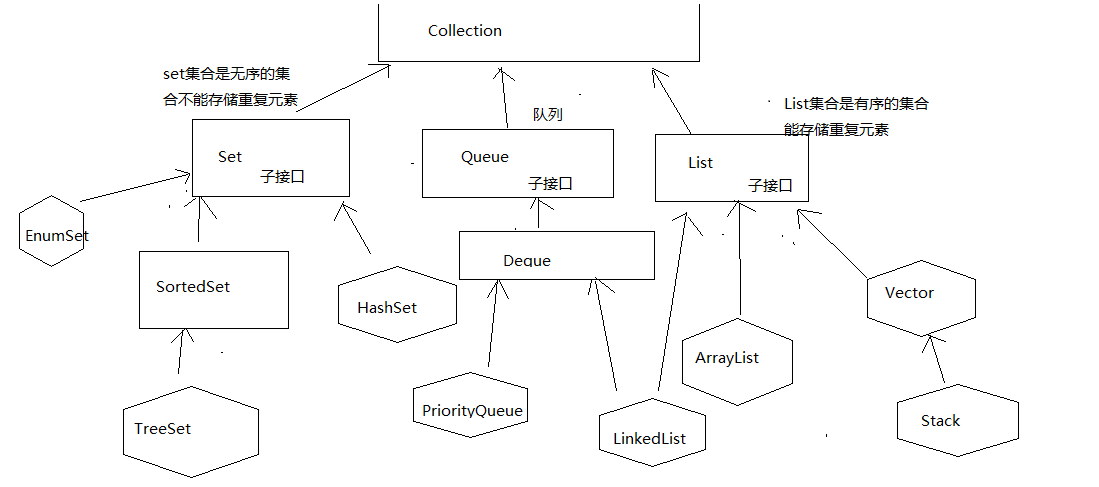
# 集合

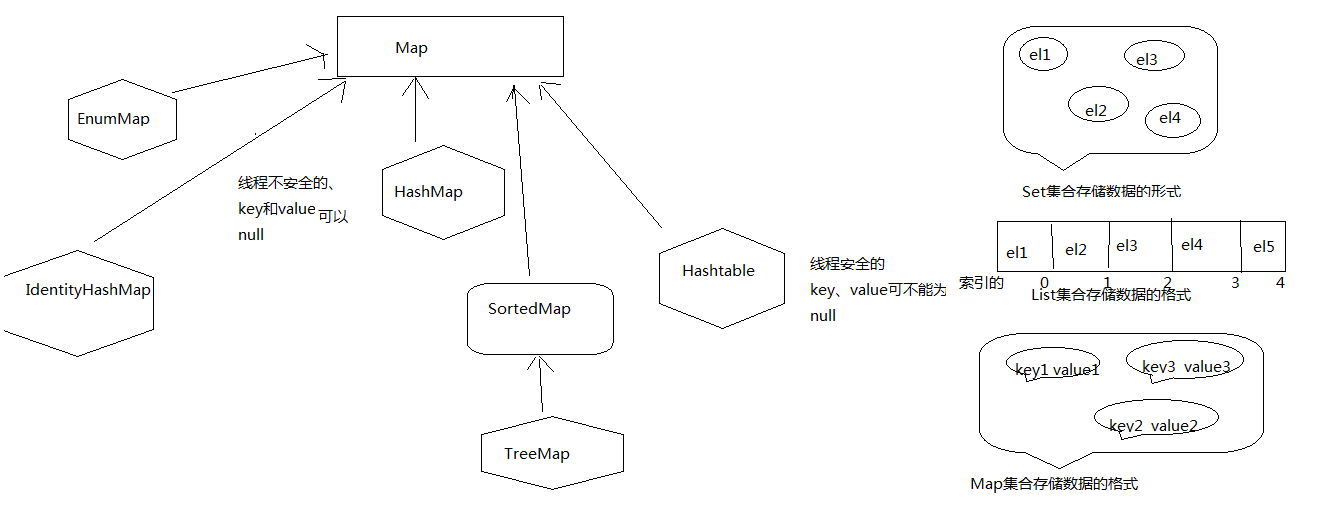
为了保存不确定的数据，以及保存具有映射关系(也被称为关联数组)的数据。Java提供为了集合类。集合类的主要的作用是存取其他数据。所以称为集合为容器类。所有的集合都是位于java.util包下，后来为了处理多线程环境下的安全问题，java5以后提供了java.util.concurrent包提供了集合对多线程的支持。

集合和数组不一样，数组元素既可以是基本类型也可以是对象(实际上保存的是对象的引用变量)，而集合只能保持对象(实际上只是保存对象的引用变量)。

Java集合类主要有两个接口派生。 Collection和Map接口。Collection和Map接口是java集合框架的根接口。这个两个接口包含了一些子接口和实现类。

Collection 层次结构 中的根接口。Collection 表示一组对象，这些对象也称为 collection 的元素。一些 collection 允许有重复的元素，而另一些则不允许。一些 collection 是有序的，而另一些则是无序的。JDK 不提供此接口的任何直接 实现：它提供更具体的子接口（如 Set 和 List）实现。此接口通常用来传递 collection，并在需要最大普遍性的地方操作这些 collection。





## Collection和Iterator接口

Collection接口是List和Set以及Queue接口的父接口，该接口里定义的方法既可以用于操作List集合 也可以操作Set集合和Queue集合。

**package** com.guigu.collection;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.HashSet;

**public** **class** CollectionTest {

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

Collection collection =**new** ArrayList();

//添加元素

collection.add("张三丰");

collection.add("张三丰");

//1.虽然集合里不能存储基本类型的值 但是java支持 自动拆箱和自动装箱

//2.虽然一个集合里能存储不同类型的元素 但是 不要在一个集合里存储不同类型的元素 要保证一个集合里元素类型是同一个类型

collection.add(6);

collection.add('a');

//得到集合的大小

System.***out***.println(collection.size());

//删除指定元素

//移除的元素 如果在集合中不存在 不会报错

System.***out***.println(collection.remove(16));

System.***out***.println(collection.remove(6));

System.***out***.println(collection.size());

//判断集合中是否包含指定的内容

System.***out***.println(collection.contains("张三丰"));

System.***out***.println(collection.contains('b'));

Collection books =**new** HashSet();

//set集合不能存储重复元素

books.add("Spring入门基础");

books.add("Spring入门基础");

books.add("Java入门基础");

System.***out***.println(books.size()+"-------------");

//清空当前集合中所有的元素

// books.clear();

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

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

//控制books集合里只剩下collection集合里 也包含的元素

books.retainAll(collection);

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

//addAll 是把集合拼接在一起

collection.addAll(books);

// books.addAll(collection);

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

}

}

### 使用Foreach遍历集合

**package** com.guigu.collection;

**import** java.util.Collection;

**import** java.util.HashSet;

**public** **class** ForeachTest {

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

Collection books = **new** HashSet();

books.add("Linux入门基础");

books.add("Redis入门基础");

books.add("Hibernate入门基础");

books.add("MyBatis入门基础");

// 对集合进行遍历

**for** (Object obj : books) {

String book = (String) obj;

//引发ConcurrentModificationException错误

//foreach循环中迭代集合 然后又在改变集合的大小 也就是不能再迭代中删除

**if** (book.equals("Redis入门基础")) {

books.remove(book);

}

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

}

}

}

### 使用Lambda表达式遍历集合

Java8 为迭代增加了Lambda表达式的功能 新增的foreach方法 可以传递一个函数式接口。

**package** com.guigu.collection;

**import** java.util.Collection;

**import** java.util.HashSet;

**public** **class** CollectionEachTest {

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

Collection books = **new** HashSet();

books.add("Linux入门基础");

books.add("Redis入门基础");

books.add("Hibernate入门基础");

books.add("MyBatis入门基础");

//使用foreach进行迭代 使用lambda表达式

books.forEach( obj-> System.***out***.println(obj));

}

}

### 使用Java8增强的Iterator遍历集合元素

**package** com.guigu.collection;

**import** java.util.Collection;

**import** java.util.HashSet;

**import** java.util.Iterator;

**public** **class** IteratorEachTest {

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

Collection books = **new** HashSet();

books.add("Linux入门基础");

books.add("Redis入门基础");

books.add("Hibernate入门基础");

books.add("MyBatis入门基础");

//把Collection集合转换为Iterator迭代器

/\*Iterator iter =books.iterator();

while(iter.hasNext()) {

System.out.println(iter.next());

}\*/

//使用Java8增长的Iterator迭代器 使用Lambda表达式实现

Iterator it =books.iterator();

it.forEachRemaining(obj->System.***out***.println(obj));

}

}

### 使用Java8新增的Predicate

Java8为Collection集合新增了一个removeIf(Predicate filter)方法，该方法将会批量删除符合filter 条件的所有的元素。 但是该方法需要一个Predicate 作为参数，Perdicate是一个函数式接口。所以可以使用Lambda表达式。

**package** com.guigu.collection;

**import** java.util.Collection;

**import** java.util.HashSet;

**import** java.util.Iterator;

**public** **class** PredicateTest {

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

Collection books = **new** HashSet();

books.add("Linux入门基础");

books.add("Redis入门基础");

books.add("Hibernate入门基础");

books.add("MyBatis入门基础");

books.removeIf(ele->((String)ele).length()<10);

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

}

}

使用Predicate可以简化集合的运算。有如下三个需求

* 统计书名中带有入门 字符串的数量
* 统计 书名中出现 java 字符串的图书的数量
* 统计书名长度大于10的图书的数量

**package** com.guigu.collection;

**import** java.util.Collection;

**import** java.util.HashSet;

**import** java.util.function.Predicate;

**public** **class** PredicateTest2 {

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

Collection books = **new** HashSet();

books.add("Linux入门基础");

books.add("Redis入门基础");

books.add("Hibernate入门基础");

books.add("MyBatis入门基础");

books.add("Hadoop精通");

books.add("MySql精通");

books.add("Java入门基础");

books.add("Java高级进阶");

//1.统计带有入门案 字符串的图书数量

System.***out***.println(*sumAll*(books, ele->((String)ele).contains("入门")));

//2.统计书名包含java字符串的数量

System.***out***.println(*sumAll*(books, ele->((String)ele).contains("Java")));

//3. 统计书名长度大于10的图书的数量

System.***out***.println(*sumAll*(books, ele->((String)ele).length()>10));

}

//自定义方法 根据Predicate 进行计算数量

**public** **static** **int** sumAll(Collection books, Predicate p) {

**int** total =0;

**for** (Object object : books) {

**if**(p.test(object)) {

total++;

}

}

**return** total;

}

}

### 使用Java8新增的Stream

Java8还新增了Stream、IntStream、LongStream、DoubleStream等流使的API这些API代表多个支持串行和并行聚集操作的元素。

独立使用Stream的步骤:

1. 使用Stream或XxxStream 的bulider方法 创建Stream对应的Bulider类
2. 重复调用Bulider类的add方法向流中添加元素
3. 调用Bulider的bulider方法获取对应的Stream
4. 调用Stream的聚集方法

**package** com.guigu.collection;

**import** java.util.stream.IntStream;

**public** **class** IntStreamTest {

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

IntStream is =IntStream.*builder*().add(20).add(30).add(40).add(22).build();

//下面的代码调用聚集方法 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所有元素的总和是:"+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().getAsDouble());

//

// System.out.println("is中是否有元素大于20:"+is.anyMatch(ele-> ele > 20));

// System.out.println("is中是否所有元素都大于20:"+is.allMatch(ele-> ele > 20));

//将is映射为一个新的Stream ，新的Stream的每个元素是原Stream的2倍+1

IntStream newIs=is.map(ele->ele\*2+1);

//打印集合

newIs.forEach(System.***out***::println);

}

}

**package** com.guigu.collection;

**import** java.util.Collection;

**import** java.util.HashSet;

**public** **class** CollectionStream {

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

Collection books = **new** HashSet();

books.add("Linux入门基础");

books.add("Redis入门基础");

books.add("Hibernate入门基础");

books.add("MyBatis入门基础");

books.add("Hadoop精通");

books.add("MySql精通");

books.add("Java入门基础");

books.add("Java高级进阶");

//1.统计带有入门案 字符串的图书数量

System.***out***.println(books.stream().filter(ele->((String)ele).contains("入门")).count());

//2.统计书名包含java字符串的数量

System.***out***.println(books.stream().filter(ele->((String)ele).contains("Java")).count());

//3.统计书名长度大于10的图书的数量

System.***out***.println(books.stream().filter(ele->((String)ele).length()>10).count());

//输出集合中的每个length的长度

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

books.stream().mapToInt(ele->((String)ele).length()).forEach(System.***out***::println);

}

}

## List集合

List集合代表一个元素有序，可重复的集合，集合中每个元素有其对应的顺序索引。

### ArrayList基础用法

**package** com.guigu.list;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** ArrayListTest {

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

// Collection col =new ArrayList();

// Collection集合所有的方法 在这里都可以使用

List books = **new** ArrayList();

books.add(**new** String("Ajax精通"));

books.add(**new** String("Ajax精通"));

books.add(**new** String("SpringBoot入门管理"));

books.add(**new** String("Git权威指南"));

// 在指定位置添加元素

books.add(1, **new** String("Maven项目管理工具"));

// books.forEach(ele -> System.out.println(ele));

// 将第二个元素 替换

books.set(2, **new** String("ELK最佳实践"));

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

System.***out***.println(books.subList(1, 4));

}

}

### ArrayList判断对象相同

实际上List集合中并未包含字符串对象，添加的两个字符串都是通过new创建的新的字符串对象，通过new创建的两个对象 明显不是同一个对象， 但是List集合依然可以添加进来。 List是如何判断两个对象的呢? List是通过对象的equals方法进行判断的 如果返回true则认为是同一个对象

**package** com.guigu.list;

**import** java.util.ArrayList;

**import** java.util.List;

**class** DemoA {

@Override

**public** **boolean** equals(Object obj) {

**return** **true**;

}

}

**public** **class** ArrayListTest2 {

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

List books = **new** ArrayList();

books.add(**new** String("Ajax精通"));

books.add(**new** String("Ajax精通"));

books.add(**new** String("SpringBoot入门管理"));

books.add(**new** String("Git权威指南"));

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

// books.remove("SpringBoot入门管理");

// System.out.println(books);

books.remove(**new** DemoA());

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

books.remove(**new** DemoA());

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

books.remove(**new** DemoA());

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

}

}

程序试图删除一个A对象，List会调用该A对象的equals方法，依次与集合元素进行比较，如果equals方法以某个集合元素作为参数返回true。

### Java8为List新增的方法

Java8为List集合增加了sort 和replaceALL两个常用的默认方法， 其中sort需要一个Comparator对象来控制元素的排序，程序可以使用Lambda表达式作为参数。 而replaceAll需要一个UnaryOperator 来替换集合中所有的集合元素。 UnaryOperator也是一个函数式接口。

**package** com.guigu.list;

**import** java.util.ArrayList;

**import** java.util.Comparator;

**import** java.util.List;

**public** **class** ArrayListTest3 {

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

List books = **new** ArrayList();

books.add(**new** String("Ajax精通"));

books.add(**new** String("Ajax精通12"));

books.add(**new** String("SpringBoot入门管理"));

books.add(**new** String("Git权威指南"));

// 使用目标类型为Compator的Lambda表达式对List集合排序

books.sort((a1, a2) -> ((String) a1).length() - ((String) a2).length());

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

//使用Lambda表达式替换集合中的元素

//把上方集合替换为自身的长度 然后输出

books.replaceAll(ele ->((String)ele).length());

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

}

}

与Set集合不同，List集合还提供了一listIterator方法，该方法返回一个ListIterator对象。

**package** com.guigu.list;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.ListIterator;

**public** **class** ListIteatorTest {

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

String [] books = {"BoostStrap入门案例","EasyUI入门案例","AngleJS入门"};

List bookList =**new** ArrayList();

//循环数组 然后把数组的内容赋值给集合

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

bookList.add(books[i]);

}

ListIterator lit =bookList.listIterator();

**while**(lit.hasNext()) {

System.***out***.println(lit.next());

}

System.***out***.println("下面的内容可以反向迭代");

**while**(lit.hasPrevious()) {

System.***out***.println(lit.previous());

}

}

}

### 实现自己的MyArrayList

**package** com.guigu.list;

**public** **class** MyArrayList {

**private** Object[] data;

**private** **int** size;

**public** MyArrayList() {

data = **new** Object[10];

}

// 实现 add

**public** **void** add(Object obj) {

// 判断是否需要扩容

ensureCapatity();

data[size++] = obj;

}

**public** **void** ensureCapatity() {

// 1,判断是否越界

**if** (size >= data.length) {

//创建一个新的的数组

Object [] newData =**new** Object[data.length<<1];

//把原有的数组 拷贝进新的数组

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

newData[i]=data[i];

}

//把data指向新的 数组

data=newData;

}

}

// 实现 size

**public** **int** size() {

**return** **this**.size;

}

// 实现get

**public** Object get(**int** index) {

**return** data[index];

}

}

**package** com.guigu.list;

**public** **class** MyTest {

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

MyArrayList mal =**new** MyArrayList();

mal.add("aaa");

mal.add("bbb");

mal.add("ccc");

**for**(**int** i=0; i<mal.size();i++) {

System.***out***.println(mal.get(i));

}

}

}

### ArrayList和Vector

ArrayList和Vector都是List类两个典型的实现类 。

ArrayList和Vector在用法上完全相同，但是由于Vector是一个古老的集合，在jdk1.0开始就存在，Java在1.0还没有提供系统的集合框架， 所有Vector中很多方法名特别长 。所以List是作为vector的一个改版，从而导致Vector中很多方法是重复的。

除此之外 ArrayList和Vector最大的区别是ArrayList是线程不安全的，而Vector是线程安全的。

任何时候都不推荐使用Vector。

ArrayList可以通过Colllections工具类转换为线程安全的。 所以 ArrayList可以在线程安全的环境下使用。

**package** com.guigu.list;

**import** java.util.Iterator;

**import** java.util.Vector;

**public** **class** VectorTest {

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

Vector vector =**new** Vector();

vector.add("abc");

vector.add("abcd");

vector.add("abce");

vector.add("ab");

vector.add("a");

Iterator iter =vector.iterator();

// it.forEachRemaining(obj->System.out.println(obj));

**while**(iter.hasNext()) {

System.***out***.println(iter.next());

}

}

}

### LinkedList

**package** com.guigu.list;

**import** java.util.LinkedList;

**public** **class** LinkedListTest {

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

LinkedList list =**new** LinkedList();

list.add("aaa");

list.add("bbb");

list.add("ccc");

list.addFirst("ddd");

list.addLast("eee");

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

//取出 元素 但是不移除元素

System.***out***.println(list.getFirst());

System.***out***.println(list.getLast());

//取出 元素 但是不移除元素

System.***out***.println(list.peekFirst());

System.***out***.println(list.peekLast());

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

//取出元素 并移除集合中的元素

System.***out***.println(list.pollFirst());

System.***out***.println(list.pollLast());

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

//pop 取出集合中的第一个元素并移除

System.***out***.println(list.pop());

//存储于集合中第一个元素的位置

list.push("qq");

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

}

}

### 各种线性表性能分析

Java提供的List就是一个线性接口，而ArrayList/LinkedList又是线性表两种典型的实现，基于数组的线性表和基于链表的线性表。

ArrayList底层是基于数组的 线性表

LinkedList底层是基于链表的线性表

使用List集合有如下建议：

* 如果需要遍历List集合元素 关于ArrayList Vector 集合应该使用随机访问的方法get来遍历集合元素，这样性能会更好一些。
* 如果需要经常执行插入 删除 操作 来改变大量数据的List集合的大小，可以考虑使用LinkedList集合 ，使用ArrayList vector集合可能经常需要重新分配内存效果可能相对于LinkedListJ较差。
* 如果有多个线程同时访问List集合元素，可以考虑使用Collections将集合包装为线程安全的集合

## Set集合

Set集合是一个无序的，不能存储重复元素。 程序可以依次把对象存储在set集合中，而set通常是无法记住添加的顺序，set集合和Collection基本相关，没有提供额外的方法， 实际上set集合就是Collection。 只是略微不同。

Set集合不允许包含相同的元素，如果试图添加相同的元素，则添加失败。

### HashSet

HashSet是set接口的典型实现，大多数时候使用这个实现类完成功能，HashSet是按Hash算法来存储集合中的元素，所以他的存取和查找性能良好。

HashSet具有以下特点:

* 不能保证元素的排列顺序，顺序可能与添加的不同， 顺序可能发生变化。
* HashSet不是同步的(不是线程安全的)， 如果多个线程访问一个HashSet 必须通过其他方式保证这个方法的同步。Collections完成。
* 集合中的元素可以是null

**package** com.guigu.set;

**import** java.util.HashSet;

**class** A {

@Override

**public** **boolean** equals(Object obj) {

**return** **true**;

}

}

**class** B {

@Override

**public** **int** hashCode() {

**return** 1;

}

}

**class** C {

@Override

**public** **boolean** equals(Object obj) {

**return** **true**;

}

@Override

**public** **int** hashCode() {

**return** 2;

}

}

**public** **class** HashSetTest {

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

HashSet set = **new** HashSet();

set.add("aaa");

set.add("bbb");

set.add("ccc");

set.add("ddd");

set.add("aaa");

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

HashSet set2 = **new** HashSet();

set2.add(**new** A());

set2.add(**new** A());

set2.add(**new** B());

set2.add(**new** B());

set2.add(**new** C());

set2.add(**new** C());

System.***out***.println(set2);

}

}

从上方的程序可以看成 有两个A 两个B 一个C

HashSet判断两个对象是否相等是根据equals和hashCode两个进行判断的。

**package** com.guigu.set;

**public** **class** Person {

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

**private** String name;

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

**public** Person() {

**super**();

}

**public** Person(**int** id, String name, **int** age) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.age = age;

}

@Override

**public** **int** hashCode() {

**return** id+name.hashCode();

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj) {

**return** **true**;

}

**if** (obj != **null** && obj.getClass() == Person.**class**) {

Person otherPerson = (Person) obj;

// 只有id和姓名相同则认为是同一个人

**if** (otherPerson.getId() == **this**.getId() && otherPerson.getName().equals(**this**.getName())) {

**return** **true**;

}

}

**return** **false**;

}

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

**return** id;

}

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

**this**.id = id;

}

**public** String getName() {

**return** name;

}

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

**this**.name = name;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

@Override

**public** String toString() {

**return** "Person [id=" + id + ", name=" + name + ", age=" + age + "]";

}

}

**package** com.guigu.set;

**import** java.util.HashSet;

**public** **class** HashSetTest2 {

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

Person p1 = **new** Person(1, "张三", 34);

Person p2 = **new** Person(1, "张三", 35);

HashSet set = **new** HashSet();

set.add(p1);

set.add(p2);

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

}

}

### LinkedHashSet

HashSet还有一个子类LinkedHashSet,这个集合是根据元素的HashCode值进行决定元素的存储位置的。但是它同时使用链表维护元素的次序。

LinkedHashSet将元素按添加顺序访问集合中的元素，LinkedHashSet需要维元素的插入顺序，所以性能相对于HashSet较为低一些。

**package** com.guigu.set;

**import** java.util.LinkedHashSet;

**public** **class** LinkedHashSetTest {

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

LinkedHashSet lhs =**new** LinkedHashSet();

lhs.add("SpringBoot入门基础");

lhs.add("Kafka入门基础");

lhs.add("Zookper入门基础");

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

lhs.remove("Kafka入门基础");

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

lhs.add("Kafka入门基础");

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

}

}

### TreeSet

TreeSet是SortedSet接口的实现类。正如名字一样是对set集合完成排序的

**package** com.guigu.set;

**import** java.util.TreeSet;

**public** **class** TreeSetTest {

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

// 默认按照自然顺序排序

TreeSet set = **new** TreeSet();

set.add(34);

set.add(12);

set.add(18);

set.add(22);

set.add(29);

set.add(-10);

set.add(-2);

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

//取出集合中第一个和最后一个元素

System.***out***.println(set.first());

System.***out***.println(set.last());

//返回小于15的集合

System.***out***.println(set.headSet(15));

//返回大于15的集合

System.***out***.println(set.tailSet(15));

//取范围

System.***out***.println(set.subSet(2, 20));

}

}

**1、自然排序**

TreeSet会调用元素compareTo方法(Object obj)来比较元素的大小。 这种方式是进行自然排序、

Java 还提供了一个Compareable接口，该接口中定义了一个compareTo的方法该方法返回一个数值，根据这个数值可以进行比较顺序。

**package** com.guigu.set;

**import** java.util.TreeSet;

**class** User {

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

**private** String name;

**public** User(**int** id, String name) {

**super**();

**this**.id = id;

**this**.name = name;

}

}

**public** **class** TreeSetErrorTest {

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

TreeSet set = **new** TreeSet();

set.add(**new** User(1, "张三"));

set.add(**new** User(2, "李四"));

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

}

}

如果一个对象要实现排序 这个对象 必须实现Compable接口

**package** com.guigu.set;

**public** **class** User **implements** Comparable<User> {

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

**private** String name;

**public** User(**int** id, String name) {

**super**();

**this**.id = id;

**this**.name = name;

}

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

**return** id;

}

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

**this**.id = id;

}

**public** String getName() {

**return** name;

}

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

**this**.name = name;

}

@Override

**public** String toString() {

**return** "User [id=" + id + ", name=" + name + "]";

}

@Override

**public** **int** compareTo(User o) {

**int** temp = **this**.id - o.id;

**return** temp == 0 ? **this**.name.compareTo(o.name) : temp;

}

}

**package** com.guigu.set;

**import** java.util.TreeSet;

**public** **class** TreeSetErrorTest {

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

TreeSet set = **new** TreeSet();

set.add(**new** User(3, "赵六"));

set.add(**new** User(1, "张三"));

set.add(**new** User(2, "李四"));

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

}

}

**2.自定义排序**

**package** com.guigu.set;

**import** java.util.TreeSet;

**public** **class** TreeSetByUsed {

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

TreeSet set = **new** TreeSet((o1, o2) -> {

**int** a1 = (Integer) o1;

**int** a2 = (Integer) o2;

**return** a1 > a2 ? -1 : a1 < a2 ? 1 : 0;

});

set.add(34);

set.add(12);

set.add(18);

set.add(22);

set.add(29);

set.add(-10);

set.add(-2);

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

}

}

### Set各实现类的性能分析

HashSet和TreeSet是set的两个典型实现类，HashSet和TreeSet该如何选择和使用呢?

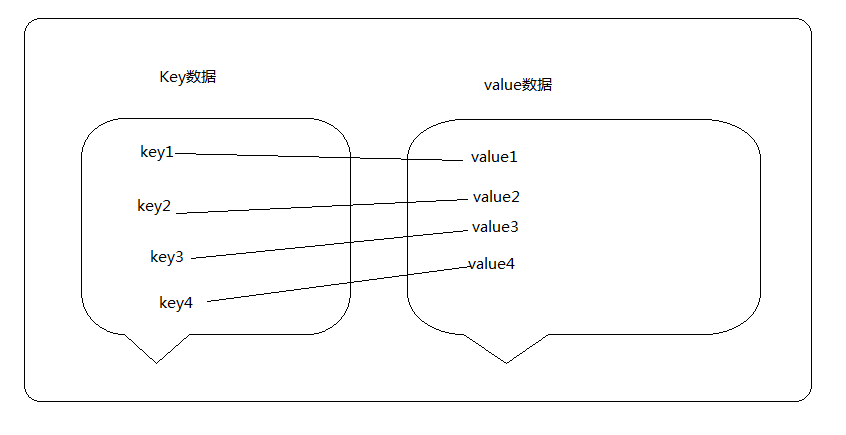
HashSet的性能总是比TreeSet好，特别是常用添加，查询等元素的操作，推荐使用HashSet,因为TreeSet还要额外的红黑树算法维护元素的次序。当需要对一个集合排序 需要使用TreeSet会比HashSet更好。 HashSet还要一个子类是LinkedHashSet 比HashSet性能低一些，但是由于有链表所以LinkedHashSet是有序的。

HashSet 、 TreeSet、LinkedHashSet三个都是线程不安全的，但是可以通过Collections工具类转换为线程安全的。

## Map集合

Map集合是用于保存具有映射关系的数据，Map集合中保存两组的值， 一组是用于保存Map里的key，一组是用于保存Map的value.

其中key和value可以是任意引用类型的数据类型，其中Map的key是不允许出现重复数据，即同一个map对象任何两个key值是不同的 是通过equals方法进行判断的。



### Map的基础用法

**package** com.guigu.map;

**import** java.util.HashMap;

**import** java.util.Iterator;

**import** java.util.Map;

**import** java.util.Set;

**public** **class** MapTest {

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

Map<String,Integer> map = **new** HashMap<String,Integer> ();

// 存储数据

map.put("Spring基础", 100);

map.put("Ajax基础", 200);

map.put("Hibernate基础", 300);

map.put("IOS入门基础", 400);

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

// 如果key值重复 将发生覆盖

// 这个返回的是覆盖掉的key对应的value的值

System.***out***.println(map.put("Ajax基础", 500));

//根据key值移除数据

map.remove("Spring基础");

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

// 测试是否包含指定的key值和value值

System.***out***.println(map.containsKey("IOS入门基础1"));

System.***out***.println(map.containsValue("200"));

// 根据key值得到对应的value值

System.***out***.println(map.get("Hibernate基础"));

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

//对map集合迭代

Set keySet=map.keySet();

// keySet.forEach(System.out::println);

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

keySet.forEach(obj->System.***out***.println(obj+"---"+map.get(obj)));

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

//第二种迭代方式

Set<Map.Entry<String, Integer>> setReal= map.entrySet();

Iterator<Map.Entry<String, Integer>> it =setReal.iterator();

**while**(it.hasNext()) {

Map.Entry<String, Integer> mm =it.next();

//取key

String key=mm.getKey();

//取value

Integer value=mm.getValue();

System.***out***.println(key+"----"+value);

}

}

}

### Java8为Map新增的方法

**package** com.guigu.map;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** MapNewTest {

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

Map map = **new** HashMap();

map.put("Spring基础", 100);

map.put("Ajax基础", 200);

map.put("Hibernate基础", 300);

map.put("IOS入门基础", 400);

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

// 当key值不存在的时候 map集合没有变化

// map.replace("Redis入门基础", 600);

map.replace("IOS入门基础", 600);

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

// 使用原来的value参与计算 然后得到新的结果 覆盖掉以前的值

map.merge("Hibernate基础", 20, (oldValue, newParam) -> (Integer) oldValue \* (Integer) newParam);

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

// 当key为Spring的时候，对应的value为null，使用计算的结果作为新的value 然后添加到集合中

map.computeIfAbsent("Java", (key) -> ((String) key).length());

// 当key为Java 对应的value存在的时候， 使用新的计算的结果覆盖掉之前的内容

map.computeIfPresent("Java", (key, value) -> (Integer) value \* (Integer) value);

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

// 迭代map集合

map.forEach((key, value) -> System.***out***.println(key + "---" + value));

}

}

Map集合是不保证迭代的顺序 特别是不保证该顺序恒久不变

**package** com.guigu.map;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** HomeView {

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

Map<String, String> map = **new** HashMap<String, String>();

map.put(String.*valueOf*(System.*currentTimeMillis*()) + "a", "1");

map.put(String.*valueOf*(System.*currentTimeMillis*()) + "b", "2");

map.put(String.*valueOf*(System.*currentTimeMillis*()) + "c", "3");

map.put(String.*valueOf*(System.*currentTimeMillis*()) + "d", "4");

map.forEach((key, value) -> System.***out***.println(value));

}

}

### Java8改进的HashMap和Hashtable

HashMap 和Hashtable都是Map接口的实现类。他们之间的关系类似于ArrayList和Vector之间的关系 。

Java8改进了HashMap 的实现(分析HashMap的底层实现 ConcurrentHashMap)

使 HashMap如果存在key值冲突依然会有很好的性能。

除此以外 HashMap和Hashtable主要体现在以下两点的不同:

* Hashtable是一个线程安全的类，但是HashMap不是线程安全的类，HashMap的性能高于Hashtable。如果有多个线程访问统一Map集合时，Hashtable 的实现会更加一些，但是还是不推荐使用。
* Hashtable 不允许使用null作为key值和value值，而HashMap允许key值和value值为null，但是key值也只能是一个为null否则发生覆盖，value可以多个null.

**package** com.guigu.map;

**import** java.util.HashMap;

**import** java.util.Hashtable;

**public** **class** NullHashMap {

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

HashMap hm = **new** HashMap();

hm.put(**null**, **null**);

hm.put(**null**, 1);

hm.put(1, **null**);

hm.put(2, **null**);

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

//Hashtable 是不允许key和value为null

Hashtable ht = **new** Hashtable();

// ht.put(null, null);

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

}

}

HashSet集合不能保证元素的迭代顺序，而且集合中是不能有重复元素的，HashMap和Hashtable也不能保证迭代顺序，其中key值也不能有重复元素。

判断key相等的标准是 两个key通过equals方法返回true ，两个key 的hashCode也相同。

**package** com.guigu.map;

**import** java.util.Hashtable;

**import** org.omg.PortableInterceptor.USER\_EXCEPTION;

**class** User {

**private** **int** money;

**public** User(**int** money) {

**this**.money = money;

}

@Override

**public** **boolean** equals(Object obj) {

**if** (obj == **this**) {

**return** **true**;

}

**if** (obj != **null** && obj.getClass() == User.**class**) {

User otherUser = (User) obj;

**return** **this**.money == otherUser.money;

}

**return** **false**;

}

@Override

**public** **int** hashCode() {

**return** **this**.money;

}

}

**class** Person {

@Override

**public** **boolean** equals(Object obj) {

**return** **true**;

}

}

**public** **class** HashtableTest {

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

Hashtable ht = **new** Hashtable();

ht.put(**new** User(600000), "Java入门基础");

ht.put(**new** User(98765), "Kafka分布式");

ht.put(**new** User(98765), "Hadoop分布式");

ht.put(**new** User(23456), **new** Person());

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

System.***out***.println(ht.containsValue("刘老师讲了这么长时间你们累了吧"));

}

}

### LinkedHashMap

HashMap有一个子类LinkedHashMap。 其中LinkedHashMap也是使用双向链表 维护key-value 的次序。(其实只有key的次序)。该链表负责维护map的迭代顺序，迭代顺序与key-value对应的插入顺序相同

**package** com.guigu.map;

**import** java.util.LinkedHashMap;

**public** **class** LinkedHashMapTest {

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

LinkedHashMap score = **new** LinkedHashMap();

score.put("Java", 99);

score.put("MySql", 97);

score.put("Orace", 92);

score.put("高数", 88);

score.forEach((key, value) -> System.***out***.println(key + "--" + value));

}

}

### Properties

Properties是Hashtable的子类，该对象是处理属性文件。

Properties是一个双向集合既有集合的特性 也拥有IO流的特性。

**package** com.guigu.map2;

**import** java.io.FileReader;

**import** java.io.FileWriter;

**import** java.util.Properties;

**public** **class** PropertiesTest {

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

Properties pro = **new** Properties();

// 想pro中增加属性

pro.setProperty("username", "guigu");

pro.setProperty("password", "123456");

// 将Properties中的key-value保存到guigu.ini 配置文件中

// 第一个参数是输出流 第二个参数是注释

pro.store(**new** FileWriter("guigu.ini"), "this is test properties");

// 新建Properties

Properties pro2 = **new** Properties();

pro2.setProperty("gender", "male");

// 加载guigu.ini中的内容

pro2.load(**new** FileReader("guigu.ini"));

System.***out***.println(pro2);

}

}

### TreeMap

**package** com.guigu.map2;

**import** java.util.TreeMap;

**public** **class** TreeMapTest {

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

TreeMap tm =**new** TreeMap();

tm.put(3, "Java入门基础");

tm.put(12, "Redis基础");

tm.put(6, "SpringBoot基础");

tm.put(8, "深入研读jvm虚拟机");

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

System.***out***.println(tm.firstKey());

System.***out***.println(tm.firstEntry());

System.***out***.println(tm.lastKey());

System.***out***.println(tm.lastEntry());

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

//只是返回一个值

System.***out***.println(tm.higherKey(5));

System.***out***.println(tm.lowerKey(5));

System.***out***.println(tm.subMap(1, 7));

}

}

### WeakHashMap

WeakHashMap与HashMap用法基本相同，与HashMap的区别在于，HashMap的key保留了对实际对象的强引用，也就意味着只要hashMap对象不被销毁，该HashMap所有的key引用的对象就不会被垃圾回收，但是WeakHashMap的key只保留了对实际对象的弱引用，也就意味着如果WeakHashMap中的对象的key所引用的对象没有被其他强引用变量所引用，那么这些key可能被垃圾回收。

**package** com.guigu.map2;

**import** java.util.HashMap;

**import** java.util.WeakHashMap;

**public** **class** WeakHashMapTest {

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

WeakHashMap whm = **new** WeakHashMap();

// HashMap whm=new HashMap();

whm.put(**new** String("Java"), **new** String("优秀"));

whm.put(**new** String("Oracle"), **new** String("良好"));

whm.put(**new** String("MySql"), **new** String("良好"));

whm.put(**new** String("Servlet"), **new** String("中等"));

whm.put("Spring", **new** String("优秀"));

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

// 通知系统立马进行垃圾回收

System.*gc*();

System.*runFinalization*();

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

}

}

### IdentityHashMap

**package** com.guigu.map2;

**import** java.util.IdentityHashMap;

**public** **class** IdentityHashMapTest {

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

IdentityHashMap ihm = **new** IdentityHashMap();

// HashMap ihm = new HashMap();

// IdentityHashMap将对象中创建的新对象都认为是不同的 都可以存储

ihm.put(**new** String("Java"), 80);

ihm.put(**new** String("Java"), 90);

ihm.put("Spring", 99);

ihm.put("Spring", 98);

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

}

}

## 工具类Collections和Arrays

Java提供了一个操作set、list和map等集合的工具类，Collections该工具类提供了大量的方法对集合进行排序，查询和修改等操作。

### Collections

#### 排序操作

**package** com.guigu.utils;

**import** java.util.ArrayList;

**import** java.util.Collections;

**public** **class** CollectionsSortTest {

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

ArrayList list = **new** ArrayList();

list.add(3);

list.add(10);

list.add(9);

list.add(32);

list.add(18);

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

// 反转

Collections.*reverse*(list);

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

// 排序 按照自然顺序排序

// Collections.sort(list);

// Collections.sort(list,(o1, o2) -> {

// int a1 = (Integer) o1;

// int a2 = (Integer) o2;

//

// return a1 > a2 ? -1 : a1 < a2 ? 1 : 0;

// } );

// System.out.println(list);

// 对list集合按照随机顺序排序

Collections.*shuffle*(list);

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

}

}

#### 查找替换操作

**package** com.guigu.utils;

**import** java.util.ArrayList;

**import** java.util.Collections;

**public** **class** CollectionsSearchTest {

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

ArrayList nums = **new** ArrayList();

nums.add(12);

nums.add(45);

nums.add(12);

nums.add(143);

nums.add(16);

nums.add(1);

nums.add(9);

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

System.***out***.println(Collections.*max*(nums));

System.***out***.println(Collections.*min*(nums));

//替换

Collections.*replaceAll*(nums, 1, 99);

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

System.***out***.println(Collections.*frequency*(nums, 12));

Collections.*sort*(nums);

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

//使用二分查询 查询数据 如果使用二分查询数据 必须是对集合排序的

System.***out***.println(Collections.*binarySearch*(nums, 16));

}

}

#### 同步控制

**package** com.guigu.utils;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.Collections;

**import** java.util.HashMap;

**import** java.util.HashSet;

**import** java.util.List;

**import** java.util.Map;

**import** java.util.Set;

**public** **class** SynchronizedTest {

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

// 此时Collection就是一个线程安全的类

Collection c = Collections.*synchronizedCollection*(**new** ArrayList());

// List集合也是一个线程安全的类

List list = Collections.*synchronizedList*(**new** ArrayList());

Set set = Collections.*synchronizedSet*(**new** HashSet());

Map map = Collections.*synchronizedMap*(**new** HashMap());

}

}

### Arrays

Java提供了Arrays工具类，包含了一些static修饰的方法，可以直接操作数组。

**package** com.guigu.utils;

**import** java.util.Arrays;

**import** java.util.List;

**public** **class** ArraysTest {

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

// 定义一个数组

**int**[] arr1 = **new** **int**[] { 23, 12, 54, 51, 67, 1, 6 };

**int**[] arr2 = **new** **int**[] { 23, 12, 54, 51, 67, 1, 6 };

// 比较两个数组是否相同

System.***out***.println(Arrays.*equals*(arr1, arr2));

// 实现数组的复制和扩容

**int**[] arr3 = Arrays.*copyOf*(arr1, 10);

// for (int i : arr3) {

// System.out.println(i);

// }

// 返回指定数组内容的字符串表示形式。

System.***out***.println(Arrays.*toString*(arr3));

// 对数组进行排序

Arrays.*sort*(arr1);

System.***out***.println(Arrays.*toString*(arr1));

//返回一个受指定数组支持的固定大小的列表。

List list = Arrays.*asList*(12, 56, 123, 89, 1, 67, 234);

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

System.***out***.println(Arrays.*toString*(arr1));

Arrays.*fill*(arr1,3,6,111);

System.***out***.println(Arrays.*toString*(arr1));

}

}

Java8增强了Arrays的功能， 为Arrays增加了一些工具方法，这些工具方法可以充分利用多CPU并发的能力提高设置值，排序的性能。

**package** com.guigu.utils;

**import** java.util.Arrays;

**public** **class** ArraysNewTest {

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

**int**[] arr1 = **new** **int**[] { 34, 12, 7, 2, 91, 467, 346 };

// 对数组arr1进行并发排序

Arrays.*parallelSort*(arr1);

System.***out***.println(Arrays.*toString*(arr1));

**int**[] arr2 = **new** **int**[5];

Arrays.*parallelSetAll*(arr2, operand->operand\*5);

System.***out***.println(Arrays.*toString*(arr2));

}

}

**package** com.guigu.utils;

**import** java.util.Arrays;

**public** **class** ArraysNewTest {

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

**int**[] arr1 = **new** **int**[] { 34, 12, 7, 2, 91, 467, 346 };

// 对数组arr1进行并发排序

Arrays.*parallelSort*(arr1);

System.***out***.println(Arrays.*toString*(arr1));

**int**[] arr2 = **new** **int**[5];

Arrays.*parallelSetAll*(arr2, operand -> operand \* 5);

System.***out***.println(Arrays.*toString*(arr2));

String[] arr3 = **new** String[] { "hello", "java", "abc", "a", "guigu" };

Arrays.*parallelSort*(arr3, (o1, o2) -> o1.length() - o2.length());

System.***out***.println(Arrays.*toString*(arr3));

}

}

# 泛型

## 泛型入门

Java集合有一个缺点，把一个对象存储进集合，集合就会忘记这个对象的数据类型，该对象的编译类型就编程了Object类型。

Java集合之所以这样设计，是因为设计者不知道我们会用集合存储什么类型的数据，所以设计为能保存任意的类型做到通用性，虽然做到了通用性但是有两个问题，

* 集合对元素类型没有任何限制。这样可能会导致一系列的问题。比如想创建一个只保存Dog的集合对象，那么保存Cat一样可以保存。
* 由于存储的对象 会丢失对象的信息，也就是存储任意类型的对象取出都是Object类型。

泛型就是在定义集合的时候限定数据的类型，从而减少出错和提高效率。

### 编译时不检查类型的异常

**package** com.guigu.test;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** ListErr {

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

List strList = **new** ArrayList();

strList.add("Spring入门基础");

strList.add("Hibnernate入门基础");

strList.add("Struts2入门基础");

strList.add(5);

strList.add('a');

strList.forEach(str->System.***out***.println(((String)str).length()));

}

}

### 使用泛型进行限定

**package** com.guigu.test;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** GenericTest {

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

List<String> strList = **new** ArrayList<String>();

strList.add("Spring入门基础");

strList.add("Hibnernate入门基础");

strList.add("Struts2入门基础");

//明确strList集合中存储的是String类型

// strList.add(5);

// strList.add('a');

strList.forEach(obj->System.***out***.println(obj.length()));

}

}

### Java7泛型的菱形语法

**package** com.guigu.test;

**import** java.util.ArrayList;

**import** java.util.HashMap;

**import** java.util.List;

**import** java.util.Map;

**public** **class** DiamondTest {

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

List<String> list = **new** ArrayList<String>();

Map<String, Integer> map = **new** HashMap<String, Integer>();

// jdk1.7之后支持菱形语句 系统会自动匹配类型

List<String> list2 = **new** ArrayList<>();

Map<String, Integer> map2 = **new** HashMap<>();

List<String> books = **new** ArrayList<>();

books.add("a");

books.add("adc");

books.add("qwer");

books.forEach(obj -> System.***out***.println(obj.length()));

Map<String, Integer> schoolInfo = **new** HashMap<>();

schoolInfo.put("杭电", 23);

schoolInfo.put("理工", 21);

schoolInfo.put("计量", 22);

schoolInfo.forEach((key, value) -> System.***out***.println(key + "---" + value));

}

}

## 深入泛型

所谓泛型就是指允许在定义类，接口，方法的时候使用类型的形参，这个类型的形参将在声明变量，创建对象，调用方法的时候动态的指定。

### 定义泛型接口和类

**package** com.guigu.test;

//定义了Apple类使用了泛型的声明

**public** **class** Apple<T> {

**private** T info;

**public** Apple() {

}

**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> al =**new** Apple<>("苹果");

System.***out***.println(al.getInfo());

Apple<Double> a2=**new** Apple<>(1.23);

System.***out***.println(a2.getInfo());

}

}

### 从泛型类中派生子类

**package** com.guigu.test;

**public** **class** HongFuShiApple **extends** Apple<String> {

@Override

**public** String getInfo() {

**return** "子类"+**super**.getInfo();

}

}

**package** com.guigu.test;

**public** **class** SheGuoApple **extends** Apple {

@Override

**public** String getInfo() {

**return** **super**.getInfo().toString();

}

}

## 泛型通配符

### 使用类型通配符

为了表示各种泛型List的父类，可以使用泛型通配符，通配符就是一个(?) 将一个问号作为类型的实参传递给List集合。

**package** com.guigu.test2;

**import** java.util.ArrayList;

**public** **class** GenericTest {

**public** **void** f1(ArrayList<Object> list) {

list.forEach(System.***out***::println);

}

**public** **void** f2(ArrayList<String> list) {

list.forEach(System.***out***::println);

}

**public** **void** f3(ArrayList<?> list) {

list.forEach(System.***out***::println);

}

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

GenericTest gt = **new** GenericTest();

ArrayList<String> list =**new** ArrayList<>();

list.add("guigu");

list.add("hangzhou");

list.add("zhejiang");

ArrayList<Integer> list2 =**new** ArrayList<>();

list2.add(12);

list2.add(21);

list2.add(123);

list2.add(321);

// gt.f1(list);

// gt.f2(list);

// gt.f3(list);

gt.f3(list2);

}

}

ArrayList<Object>不是ArrayList<String>的父类

ArrayList<?> 是ArrayList<String>的父类

### 设定通配符的上限和下限

**package** com.guigu.test2;

**import** java.util.ArrayList;

**import** java.util.Collection;

**public** **class** GenericTest2 {

/\*\*

\*

\* 泛型的限定

\* ? extends E 接收E类型或者E的子类

\*

\* ? super E 接收的E类型或者E的父类型

\*

\*/

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

ArrayList<Person> al = **new** ArrayList<>();

al.add(**new** Person(1, "张三1"));

al.add(**new** Person(2, "张三2"));

al.add(**new** Person(3, "张三3"));

ArrayList<Student> al2 = **new** ArrayList<>();

al2.add(**new** Student(1, "李四"));

al2.add(**new** Student(2, "李四2"));

al2.add(**new** Student(3, "李四3"));

ArrayList<Worker> al3 = **new** ArrayList<>();

al3.add(**new** Worker(1, "王五1"));

al3.add(**new** Worker(2, "王五2"));

al3.add(**new** Worker(3, "王五3"));

ArrayList<ComputerWorker> al4 = **new** ArrayList<>();

al4.add(**new** ComputerWorker(1, "赵六1"));

al4.add(**new** ComputerWorker(2, "赵六2"));

al4.add(**new** ComputerWorker(3, "赵六3"));

*showDetail*(al4);

*showDetail*(al3);

*showDetail2*(al);

*showDetail2*(al3);

}

// 只能接收Worker本身或者Worker子类

**public** **static** **void** showDetail(Collection<? **extends** Worker> collection) {

collection.forEach(System.***out***::println);

}

// 只能接收Worker本身或者Worker子类

**public** **static** **void** showDetail2(Collection<? **super** Worker> collection) {

collection.forEach(System.***out***::println);

}

}

## 泛型方法

**package** com.guigu.test3;

**import** java.util.ArrayList;

**import** java.util.Collection;

**public** **class** GenericMethodTest {

**public** **static** **void** fromArrayToCollection(String[] arr, Collection<String> col) {

**for** (String a : arr) {

col.add(a);

}

}

// 声明一个泛型方法 是泛型完成由数组到集合的转换

**public** **static** <T> **void** fromArrayToCollection2(T[] arr, Collection<T> col) {

**for** (T a : arr) {

col.add(a);

}

}

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

Object[] oa = **new** Object[100];

Collection<Object> co = **new** ArrayList<>();

// 下面的T是Object类型

*fromArrayToCollection2*(oa, co);

String[] sa = **new** String[100];

Collection<String> cs = **new** ArrayList<>();

// 下面的T是String类型

*fromArrayToCollection2*(sa, cs);

Integer[] ia = **new** Integer[100];

Float[] fa = **new** Float[100];

Number[] na = **new** Number[100];

Collection<Number> cn = **new** ArrayList<>();

// 下面的T是 Numnber类型 cn是ia的父类

*fromArrayToCollection2*(ia, cn);

// 下面的T是 Numnber类型 cn是fa的父类

*fromArrayToCollection2*(fa, cn);

// 下面的T是 Numnber类型

*fromArrayToCollection2*(na, cn);

// 因为Number不是String类 也吧亅它的子类 所以编译错误

// fromArrayToCollection2(na,cs);

}

}

**package** com.guigu.test3;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.List;

**public** **class** GenericMethodTest2 {

// 把一个Collection集合转换为另外一个Collection集合

**static** <T> **void** test(Collection<? **extends** T> from, Collection<T> to) {

**for** (T t : from) {

to.add(t);

}

}

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

List<Object> ao = **new** ArrayList<>();

List<String> as = **new** ArrayList<>();

*test*(as, ao);

}

}

## 泛型擦除

当把一个具有泛型信息的对象赋值给另外一个没有泛型新的变量时，所在类的尖括号都将被扔掉。

**package** com.guigu.test3;

**public** **class** ErasureTest {

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

Apple<Integer> a = **new** Apple<>(6);

Integer as = a.getSize(); // 6

// 把a对象赋值给Apple变量 丢失尖括号的类型信息

Apple b = a;

//下面代码报错 丢失了类型信息 需要强制类型转换

// Integer as2 = b.getSize();

Number as3 = b.getSize();

}

}

# 异常

## 异常的概述

异常已经成为判断一门编程语言是否成熟的标标准，除了传统的c语言没有提供异常机制，目前主流的编程语言Java,C#,Ruby,Python都提供了成熟的异常体制。异常机制可以使程序中的异常处理代码和正常的业务逻辑分离，保证程序代码更加简洁，并提高程序的健壮性。

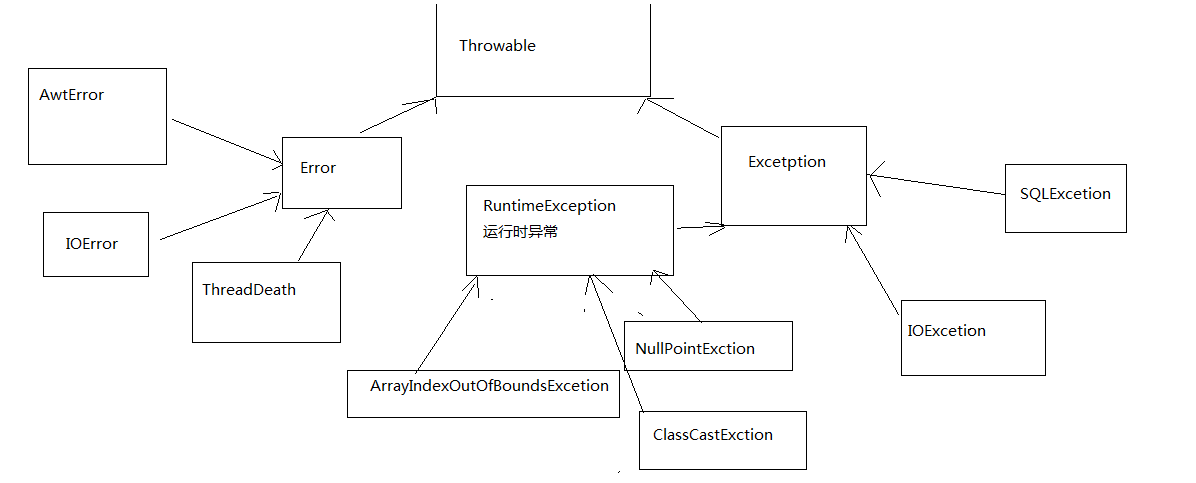
Java的异常机制主要依赖五个关键字。try, catch,finally ,throw,throws.

Java7进一步增强了异常处理机制的功能。包括带有try语句，捕获多异常的catch两个新功能。

## 异常处理机制

Java异常处理机制可以让程序拥有更好的容错性，更加健壮, 当程序运行出现意外情况时，系统会自动生成一个Exception通知程序。从而实现业务功能代码和错误处理代码进行分离，拥有更好的可读性。

### 异常的继承体系



### 使用try-catch捕获异常

**package** com.guigu.exception;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**public** **class** DivTest {

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

**try** {

// System.out.println(9 / 0);

FileInputStream fis = **new** FileInputStream("C:/guigu.txt");

// 加载某个指定的类

Class.*forName*("java.util.ArrayList1");

} **catch** (ArithmeticException e) {

System.***out***.println("捕获到了除法为0的错误");

} **catch** (FileNotFoundException e) {

System.***out***.println("捕获到了文件不存在的异常");

} **catch** (Exception e) { // 罗列所有的异常后 把未知的异常用最大的捕获， 而且最大的异常必须在最后

System.***out***.println("捕获到了未知的异常..");

}

}

}

注意异常捕获时候，一定要先捕获小的异常，再捕获大的异常。

### Java7提供的多异常捕获

在Java7之前每catch块，只能捕获一种类型的异常，从java7以后，一catch块可以捕获多个异常。

使用一个Catch块捕获多个异常需要注意以下细节：

* 捕获多类型的异常时，多异常类型之间用(|) 隔开
* 捕获的多种类型的异常，异常的变量有隐式的fianl修饰，所以不能为异常变量重新赋值

**package** com.guigu.exception;

**public** **class** MultiExcetionTest {

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

**try** {

**int** a = Integer.*parseInt*(args[0]);

**int** b = Integer.*parseInt*(args[1]);

**int** c = a / b;

} **catch** (IndexOutOfBoundsException | NumberFormatException | ArithmeticException ie) {

System.***out***.println("捕获到了异常");

// 下面的代表报错 这里的多异常中的形参 默认是final修饰的

// ie=new ArithmeticException();

} **catch** (Exception e) {

System.***out***.println("捕获到了未知的异常");

// 下面的代码正常

// e=new RuntimeException();

}

}

}

### 访问异常信息

如果catch块中访问异常信息看可以通过访问catch后异常的形参来获取相关信息。所有的异常都包含以下几个方法：

getMessage 返回异常的详细描述字符串

printStackTrance 将异常信息输出到标准错误输出

getStackTrance返回该异常的跟踪栈信息

**package** com.guigu.exception2;

**import** java.io.FileInputStream;

**import** java.io.IOException;

**public** **class** AccessException {

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

**try** {

FileInputStream fis = **new** FileInputStream("a.txt");

} **catch** (IOException ioe) {

System.***out***.println(ioe.getMessage());

// System.out.println(ioe.getCause());

ioe.printStackTrace();

}

}

}

### 使用finally回收资源

有些时候，程序在try语句里打开的一些物理资源，比如数据库的连接，网络连接，磁盘文件等等这些物理资源必须显式的回收。

Java的垃圾回收机制不会回收任何物理资源，垃圾回收机制只回收堆内存对象所占的内存。

**package** com.guigu.exception2;

**import** java.io.FileInputStream;

**import** java.io.IOException;

**public** **class** FinallyTest {

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

FileInputStream fis = **null**;

**try** {

fis = **new** FileInputStream("a.txt");

} **catch** (IOException ioe) {

System.***out***.println(ioe.getMessage());

**return**;// 结束这个方法体

// System.exit(0);// 退出整个jvm虚拟机

} **finally** {

// finally 权限大于return 必须等finally执行完毕 也就是无论如何都会执行的代码 ，除非是人为退出虚拟机

System.***out***.println("执行finally关闭资源");

// 关闭磁盘文件 回收资源都是在finally代码块中完成

**if** (fis != **null**) {

**try** {

fis.close();

} **catch** (IOException ioe) {

ioe.printStackTrace();

}

}

}

}

}

**package** com.guigu.exception2;

**public** **class** FinallyTest2 {

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

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

}

**public** **static** **int** test() {

**int** i = 1;

**try** {

System.***out***.println(9 / 0);

**return** i += 3;

} **catch** (Exception e) {

System.***out***.println("捕获到了异常");

**return** i += 5;

} **finally** {

System.***out***.println("无论如何都会执行的一段代码");

**return** i += 7;

}

}

}

**package** com.guigu.exception2;

**public** **class** ExceptionTest {

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

**try** {

System.***out***.println(9 / 0);

} **finally** {

System.***out***.println("我会执行..");

}

}

}

异常的处理机制 第一种方式是使用try-catch-finally代码块完成。

Try 是试图捕获异常，当捕获到异常进入到catch代码块， 捕获到异常try代码块中下方的代码将不再执行。

Catch是捕获异常 进行异常处理，catch可以有多个， catch也可以一次捕获多个异常。

Finally是无论如何都会执行的一段代码 一般是用来释放资源。

## Check异常和Runtime异常

Java异常被分为两大类，Check异常和Runtime异常。所有的RuntimeException以及其子类都被称为Runtime异常，不是RuntimeException以及子类的都是check异常。 Runtime异常 不处理也可以编译通过，而Check异常必须处理否则编译不通过。

关于check异常注意的问题：

* 当前方法明确知道如何处理该异常，程序中可以使用try-catch块捕获异常，然后在对应的代码块中修复该异常。或者提示用户异常的信息。
* 当前方法可能不知道怎么处理这种异常，。应该在方法定义时，抛出异常，抛给调用者。使用throws抛出异常。

### 使用throws声明抛出异常

使用throw声明抛出异常， 当前方法不知道如何处理这种类型的异常，需要抛给调用者。然后由调用者处理，如果调用者也无法处理，继续往上抛出，一直到jvm虚拟机。Jvm会直在系统中抛出异常，也就是打印异常栈信息。

**package** com.guigu.exception3;

**import** java.io.FileInputStream;

**import** java.io.IOException;

**public** **class** ThrowsTest {

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

FileInputStream fis = **new** FileInputStream("a.txt");

}

}

案例二

如果被调用的方法存在异常 ，需要调用者处理异常，或者继续抛出异常

**package** com.guigu.exception3;

**import** java.io.FileInputStream;

**import** java.io.IOException;

**public** **class** ThrowsTest2 {

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

// 调用者 调用的方法如果有抛出的异常 那么 调用者有两种处理方式 要么在当前方法处理 ，要么继续抛出异常

*test*();

// try {

// test();

// } catch (IOException e) {

// System.out.println("捕获到了文件不存在的异常..");

// }

}

**public** **static** **void** test() **throws** IOException {

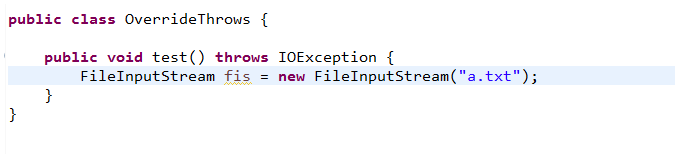
FileInputStream fis = **new** FileInputStream("a.txt");

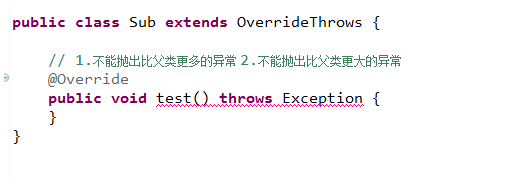
}

}

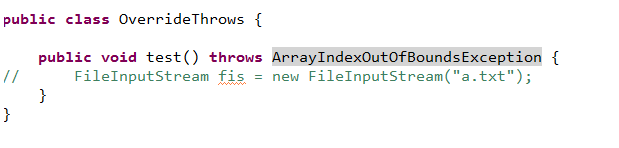
案例三：

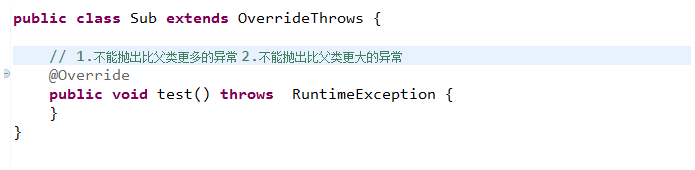
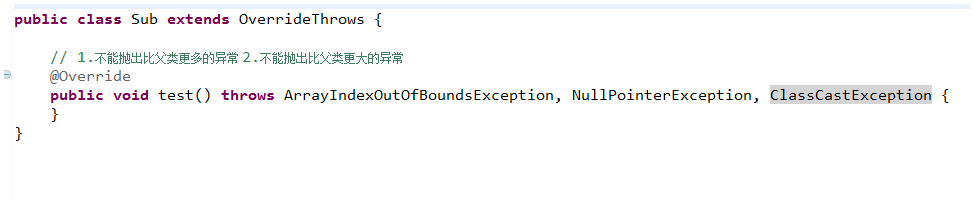
使用throws抛出异常 有一个限制，就是重写方法的两小原则，子类抛出的异常必须比父类少，比父类小。 是针对check异常





Runtime异常不遵循这个原则可以随便抛出runtime异常





### 使用throw抛出异常

使用throw异常抛出异常是check异常。是在方法体内进行抛出异常。是根据具体的业务逻辑进行处理。

**package** com.guigu.exception4;

**public** **class** ThrowTest {

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

**try** {

*throwChecked*(10);

} **catch** (Exception e) {

e.printStackTrace();

}

*throwRuntime*(10);

}

/\*\*

\* 如果在方法体内抛出的是check异常 则需要在方法声明出继续抛出异常 如果在方法体内抛出的是Runtime异常 则不需要在方法声明出继续抛出异常

\*

\* throw是 人为的根据业务逻辑进行抛出异常

\*/

**public** **static** **void** throwChecked(**int** a) **throws** Exception {

**if** (a > 0) {

**throw** **new** Exception("a的值大于0不符合要求");

}

}

**public** **static** **void** throwRuntime(**int** a) {

**if** (a > 0) {

**throw** **new** RuntimeException("a的值大于0不符合要求");

}

}

}

## 自定义异常

在通常情况下，程序很少会自行抛出系统的异常，因为系统的异常会包含该异常的有用信息，所以选择抛出异常的时候，根据业务选择合适的异常类。但是某些情况下系统无法体现业务逻辑的异常， 需要用户自定义异常。

自定义异常遵循以下步骤:

* 1、继承RuntimeException 或者Exception
* 2、写一个无参的构造函数
* 3、写一个String字符串类型的构造函数

**package** com.guigu.exception4;

**public** **class** AuctionException **extends** RuntimeException {

**public** AuctionException() {

}

**public** AuctionException(String msg) {

**super**(msg);

}

}

**package** com.guigu.exception4;

**public** **class** ArithmeticException **extends** RuntimeException {

**public** ArithmeticException() {

}

**public** ArithmeticException(String msg) {

**super**(msg);

}

}

**package** com.guigu.exception4;

**public** **class** ExceptionTest {

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

*test*(-2);

}

**public** **static** **void** test(**int** a) {

**if** (a > 0) {

System.***out***.print("hello" + a);

} **else** {

**throw** **new** ArithmeticException("a不能小于0");

}

}

}

## Java7增强的throw语句

**package** com.guigu.exception4;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.IOException;

**public** **class** ThrowTest2 {

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

//Java6 认为throw抛出的是 IOException 所以在方法声明出只能抛出 IOException 此时抛出的是FileNotFoundException 会报错

//Java7 以后 会检查一处抛出的 实际类型是哪一个 检查当前抛出的类型是FileNotFoundException 而方法声明出也是FileNotFoundException 所以不会报错

**try** {

FileInputStream fis = **new** FileInputStream("a.txt");

} **catch** (IOException ioe) {

ioe.printStackTrace();

**throw** ioe;//①

}

}

}

## Java的异常跟踪栈

**package** com.guigu.exception5;

**public** **class** PrintStackTranceTest {

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

*firstMethod*();

}

**public** **static** **void** firstMethod() {

*secondMethod*();

}

**public** **static** **void** secondMethod() {

*thirdMethod*();

}

**public** **static** **void** thirdMethod() {

*fourMethod*();

}

**public** **static** **void** fourMethod() {

*fiveMethod*();

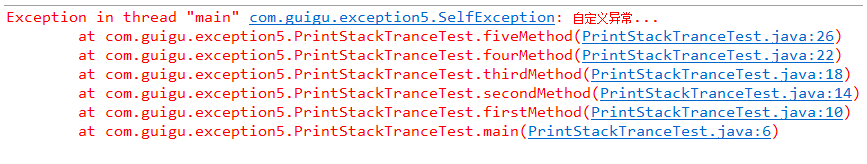
}

**public** **static** **void** fiveMethod() {

**throw** **new** SelfException("自定义异常...");

}

}



## 异常的使用规则

### 不要过度使用异常

Java的异常机制很方便去处理异常问题，但是滥用异常会造成一些负面的影响，过度使用异常会造成以下两个问题

* 把异常和普通的错误混淆在一起，不要在编写任何错误处理代码，而是简单的抛出异常替代错误处理。
* 使用异常控制替代业务逻辑流程也是不允许的。

### 不要使用过于庞大的try块

### 避免使用catch all语句

### 不要忽略捕获到的异常

# IO输入/输出

IO(输入流和输出流)，使用输入流允许程序读取外部的数据(包括磁盘，光盘等存储设备的数据)，使用输出机制允许程序记录运行的状态，将程序输出到磁盘或者光盘等存储设备上。Java的IO是通过java.io包下的类和接口支持的。 NIO.

## File类

File类是java.io包下 代表与平台无关的文件和目录，如果希望程序中操作文件和目录则可以通过file类完成。

### 访问文件和目录

**package** com.guigu.filetest;

**import** java.io.File;

**import** java.io.IOException;

**public** **class** FileTest {

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

File file = **new** File("C:/guigu.txt");

System.***out***.println(file.getName());

// 文件属性相关

System.***out***.println(file.canRead());

System.***out***.println(file.canExecute());

System.***out***.println(file.canWrite());

System.***out***.println(file.length());

// 文件路径

// 得到当前的父路径

System.***out***.println(file.getParent());// String类型

System.***out***.println(file.getParentFile());// File类型

// 得到路径

System.***out***.println(file.getPath());

System.***out***.println(file.getAbsolutePath());

System.***out***.println(file.getAbsoluteFile().getParent());

// 创建文件和目录

// 创建临时文件

File tempFile = File.*createTempFile*("aaa.txt", ".txt");

// 删除临时文件

tempFile.deleteOnExit(); // 当jvm退出的时候 删除临时文件

File file2 = **new** File("C:/hello.txt");

file2.createNewFile();

// 创建文件目录

File file3 = **new** File("C:/guigutest");

file3.mkdir();

File file4 = **new** File("C:/guiguabc/hello/abc/guigu");

file4.mkdirs();

file2.delete();

file3.delete();

// 得到当前系统的所有的磁盘信息

File[] roots = File.*listRoots*();

**for** (File ff : roots) {

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

}

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

// 得到指定目录下 有哪些文件和目录

File file6 = **new** File("C:/Users/Administrator/Desktop/Java/api/jdk/jdk-8u151-docs-all/docs/api");

File[] fileList = file6.listFiles();

**for** (File ff : fileList) {

System.***out***.println(ff.getName());

}

}

}

### 文件过滤器

**package** com.guigu.filetest;

**import** java.io.File;

**public** **class** FileNameFilterTest {

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

//只需要指定文件夹下的.html文件

File file = **new** File("C:/Users/Administrator/Desktop/Java/api/jdk/jdk-8u151-docs-all/docs/api");

File[] files = file.listFiles((dir,name)->name.endsWith(".js"));

**for** (File ff : files) {

System.***out***.println(ff.getName());

}

}

}

## 理解Java的IO流

Java的IO流是实现输入和输出流，它可以方便的实现数据的输入输出操作，在java中把不同的输入源和输出源抽象为流。

### 流的分类

按照不同的形式划分，可以把流分为不同的类型，从不同角度分类有不同的划分方式。

1：输入流和输出流

按照流的流向 可以分为输入流和输出流。

* 输入流：只能从中读取数据，而不能向其写入数据
* 输出流： 只能写入数据，无法从中读取数据。

Java的输入流主要由InputStream和Reader作为基类，输出流主要是由OutputStream和Writer作为基类，他们都是抽象类，无法直接创建对象，都是依赖子类。

1. 字节流和字符流

字节流和字符流用法基本相同，区别在于字节流和字符串操作的数据单元不同，----字节流操作的数据单元是8为的字节，字符流操作的数据单元是16位字符。

字节流主要是由InputStram和OutputStream作为基类， 而字符流则主要是Reader和Writer最为基类

Java的IO流共涉及到40多个类，这些类看起来是杂乱无章的，但是实际上他们之间存在着紧密的联系，Java的这些类都是由4个基类组成。

## 字节流和字符流

### InputStream和Reader

字符输入流

**package** com.guigu.readertest;

**import** java.io.File;

**import** java.io.FileReader;

**import** java.io.IOException;

**public** **class** FileReaderTest {

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

// 1.创建FileReader对象 并通过构造函数声明数据源

FileReader fr = **new** FileReader(**new** File("C:/guigu.txt"));

//2.每次读取一个字符

/\*

int ch1 = fr.read();

System.out.println((char)ch1);

int ch2 = fr.read();

System.out.println((char)ch2);

int ch3 = fr.read();

System.out.println((char)ch3);

\*/

//3.使用循环解决问题

/\*int ch=0;

while((ch=fr.read())!=-1) {

System.out.println((char)ch);

}

\*/

// 4.创建一个字符数组， 用于存储督读到的字符数据

**char**[] buffer = **new** **char**[1024];// 这里的数据一般是1024的整数倍

**int** hasRead = 0;

**while** ((hasRead = fr.read(buffer)) != -1) {

// 把读取到的内容拼接为String字符串

String str = **new** String(buffer, 0, hasRead);

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

}

//最后别忘记关闭资源

fr.close();

}

}

字节输入流读取数据

**package** com.guigu.readertest;

**import** java.io.File;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.IOException;

**public** **class** FileInputStreamTest {

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

File file = **new** File("C:/guigu.txt");

FileInputStream fis = **new** FileInputStream(file);

**byte**[] buffer = **new** **byte**[1024];

**int** hasRead = 0;

**while** ((hasRead = fis.read(buffer)) != -1) {

String s = **new** String(buffer, 0, hasRead);

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

}

fis.close();

}

}

关于如何选择字节流和字符流操作，如果是纯文本文件则使用字符流操作，如果是二进制文件则使用字节流读取。

### OutputStream和Writer

**package** com.guigu.outputtest;

**import** java.io.FileOutputStream;

**import** java.io.FileWriter;

**public** **class** OutputStreamTest {

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

// write();

*write2*();

}

**public** **static** **void** write() **throws** Exception {

// 向指定的文件追加内容

FileOutputStream fos = **new** FileOutputStream("C:/guigu.txt", **true**);

String str = "杭州归谷十二周年\r\n";

fos.write(str.getBytes());

fos.flush();// 立即写出

fos.close();

}

**public** **static** **void** write2() **throws** Exception {

FileWriter fw = **new** FileWriter("C:/guigu.txt", **true**);

String str = "hello guigu";

fw.write(str);

fw.flush();

fw.close();

}

}

通过输入流和输出流实现文件的拷贝

**package** com.guigu.outputtest;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**public** **class** CopyTest {

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

*copy*("C:\\Users\\Administrator\\Desktop\\Java\\J2001\\JavaSE\\第三周\\2017-11-20集合.rar", "D:/2017-11-20集合.rar");

}

**public** **static** **void** copy(String from, String to) {

FileInputStream fis = **null**;

FileOutputStream fos = **null**;

**try** {

fis = **new** FileInputStream(from);

fos = **new** FileOutputStream(to);

**byte**[] buffer = **new** **byte**[1024 \* 1024];

**int** hasRead = 0;

**while** ((hasRead = fis.read(buffer)) != -1) {

fos.write(buffer, 0, hasRead);

}

} **catch** (FileNotFoundException e) {

e.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

} **finally** {

**if** (fis != **null**) {

**try** {

fis.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

**if** (fos != **null**) {

**try** {

fos.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

}

}

## 输入流和输出流体系

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 分类 | 字节输入流 | 字节输出流 | 字符输入流 | 字符输出流 |
| 抽象基类 | InputStream | OutputStream | Reader | Writer |
| 访问文件 | FileInputStream | FileOutputStream | FileReader | FileWriter |
| 访问数组 | ByteArrayInputStream | ByteArrayOutputStream | CharArrayReader | CharArrayWriter |
| 访问字符串 |  |  | StringReader | StringWriter |
| 缓冲流 | BufferedInputStream | BufferedOutputStream | BufferedReader | BufferedWriter |
| 转换流 |  |  | InputStreamReader | OutputStreamWriter |
| 对象流 | ObjectInputStream | ObjectOutputStream |  |  |
| 打印流 |  | PrintStream |  | PrintWriter |
| 特殊流数据流 | DataInputStream | DataoutputStream |  |  |
| 切割流 | SequenceInputStream |  |  |  |

### 访问字符串

**package** com.guigu.stringio;

**import** java.io.IOException;

**import** java.io.StringReader;

**import** java.io.StringWriter;

**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,劈材，周游世界\n"

+ "从明天起,关心粮食和蔬菜\n"

+ "我有一个房子，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);

**while** ((hasRead = sr.read(buffer)) != -1) {

System.***out***.println(**new** String(buffer, 0, hasRead));

}

} **catch** (IOException e) {

e.printStackTrace();

}

System.***out***.println("-----------------分隔符------------------");

// 写出String数据

StringWriter sw = **new** StringWriter();

sw.write("有一个美丽的新世界,\n");

sw.write("她在远方等我,\n");

sw.write("哪天有天真的孩子,\n");

sw.write("还有姑娘的酒窝,\n");

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

}

}

### 缓冲流

从字符输入流中读取文本，缓冲各个字符，从而实现字符、数组和行的高效读取。

可以指定缓冲区的大小，或者可使用默认的大小。大多数情况下，默认值就足够大了。

字符缓冲流

**package** com.guigu.stringio;

**import** java.io.BufferedReader;

**import** java.io.FileReader;

**public** **class** BufferedReaderDemo {

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

FileReader fr = **new** FileReader("C:/guigu.txt");

BufferedReader br = **new** BufferedReader(fr);

//BufferedReader有一个读取行的方法

String line=**null**;

**while**((line=br.readLine())!=**null**) {

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

}

//只需要关闭缓冲流即可 而 fr会在缓冲流关闭后自动关闭

br.close();

}

}

**package** com.guigu.stringio;

**import** java.io.BufferedWriter;

**import** java.io.FileWriter;

**public** **class** BufferedWritertDemo {

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

FileWriter fw = **new** FileWriter("C:/guigu.txt", **true**);

BufferedWriter bw = **new** BufferedWriter(fw);

**for** (**int** i = 0; i < 10; i++) {

bw.write(i + "---hehe");

bw.newLine();

bw.flush();

}

bw.close();

}

}

字节缓冲流

**package** com.guigu.stringio;

**import** java.io.BufferedInputStream;

**import** java.io.BufferedOutputStream;

**import** java.io.FileInputStream;

**import** java.io.FileOutputStream;

**public** **class** CopyPic {

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

*copy*("C:\\soft\\26个字母\\I.jpg", "D:/I.jpg");

}

**public** **static** **void** copy(String from, String to) **throws** Exception {

BufferedInputStream bis = **new** BufferedInputStream(**new** FileInputStream(from));

BufferedOutputStream bos = **new** BufferedOutputStream(**new** FileOutputStream(to));

**byte**[] buffer = **new** **byte**[1024];

**int** hasRead = 0;

**while** ((hasRead = bis.read(buffer)) != -1) {

bos.write(buffer);

}

bos.close();

bis.close();

}

}

#### 装饰者模式

所谓的装饰者模式是将职责附加到对象上，如果要扩展新的功能，装饰者提供了比继承更加具有弹性的替代方案 。

设计原则：

多用组合少用继承

类的设计应该是对扩展开放，对修改关闭 OCP原则

**package** com.guigu.wrapper;

**public** **class** WrapperDemo {

/\*\*

\* 装饰者模式 为了解决给类的功能进行增强而出现的

\*

\* Writer

\* |---TextWriter

\* |---MediaWriter

\*

\* 需要对Writer进行二次迭代， 对功能进行增强，比如提高写入的效率。

\*

\* Writer

\* |---TextWriter

\* |---BufferedTextWriter

\* |---MediaWriter

\* |---BufferedMediaWriter

\*

\* 继续增强 代码 继续优化其他的问题 比如传输效率 或转换问题

\*

\* Writer

\* |---TextWriter

\* |---BufferedTextWriter、

\* |-- XxxBufferedTextWriter

\* |---MediaWriter

\* |---BufferedMediaWriter

\* |--XxxBufferedMediaWriter

\*

\* 每一个子类这样实现是可以的。但是会导致继承体系的臃肿。

\* 发现无论哪个子类都需要高效， 使用的都是缓冲技术

\* 干脆将缓冲技术单独进行描述和封装 谁用缓冲我就把缓冲传递给谁

\*

\*

\* Writer

\* |----TextWriter

\* |----MediaWriter

\* |----BufferedWriter

\*

\* 装饰者模式 的特点:

\* 1.装饰者类中的构造函数 需要接受被装饰的类

\* 2.装饰的类和被装饰的类应该属于同一个体系

\*

\*

\*/

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

Person p= **new** Person();

p.eat();

NewPerson np =**new** NewPerson();

np.eat();

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

NewWrapperPerson nwp =**new** NewWrapperPerson(**new** Person());

nwp.eat();

}

}

**class** Person {

**void** eat() {

System.***out***.println("吃饭");

}

}

**class** NewWrapperPerson{

**private** Person p;

NewWrapperPerson(Person p){

**this**.p=p;

}

**public** **void** eat() {

System.***out***.println("开胃酒...1");

p.eat();

System.***out***.println("在吃点吧....");

}

}

**class** NewPerson **extends** Person{

@Override

**void** eat() {

System.***out***.println("开胃酒");

**super**.eat();

System.***out***.println("饭后甜点");

}

}

1、列出指定目录下中的内容，包含子目录中的内容。

2、删除一个指定的目录的所有内容

3、给定一个目录，列出指定目录下所有指定的文件，并输出到指定的文件中。

练习题1

**package** com.guigu.homework;

**import** java.io.File;

**public** **class** HomeWorkTest1 {

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

File file = **new** File("C:\\Users\\Administrator\\Desktop\\Java\\J2001");

*listAll*(file, 0);

}

**public** **static** **void** listAll(File dir, **int** level) {

// 打印第一级目录

System.***out***.println(*getLevel*(level) + dir.getName());

File[] files = dir.listFiles();

level++;

**for** (File file : files) {

**if** (file.isDirectory()) {

*listAll*(file, level);

} **else** {

System.***out***.println(*getLevel*(level) + file.getName());

}

}

}

/\*\*

\* 根据目录等级进行空格的拼接

\*

\* **@param** level

\* **@return**

\*/

**public** **static** String getLevel(**int** level) {

StringBuilder sb = **new** StringBuilder();

sb.append("|--");

**for** (**int** i = 0; i < level; i++) {

sb.insert(0, " ");

}

**return** sb.toString();

}

}

2、删除一个指定的目录的所有内容

**package** com.guigu.homework;

**import** java.io.File;

**public** **class** HomeWorkTest2 {

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

*removeDir*(**new** File("D:\\guigu"));

}

**public** **static** **void** removeDir(File dir) {

File[] files = dir.listFiles();

**for** (File file : files) {

**if** (file.isDirectory()) {

*removeDir*(file);

} **else** {

file.delete();

}

}

// 删除根目录

dir.delete();

}

}

3、给定一个目录，列出指定目录下所有指定的文件，并输出到指定的文件中。

**package** com.guigu.homework;

**import** java.io.BufferedWriter;

**import** java.io.File;

**import** java.io.FileFilter;

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** HomeWorkTest3 {

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

File dir =**new** File("C:\\WorkSpace\\J2001");

//创建文件过滤器

List<File> list =**new** ArrayList<>();

*listAllFiles*(dir, pathname->pathname.getName().endsWith(".java"), list);

//创建一个文件用于输出 迭代后的内容 dir\javalist.txt

File file =**new** File(dir,"javalist.txt");

*writeToFile*(list, file);

}

// 遍历所有的文件

/\*\*

\*

\* **@param** dir

\* 要迭代的文件夹

\* **@param** fileFilter

\* 要过滤的内容

\* **@param** list

\* 迭代后满足条件的List<File>集合

\*/

**public** **static** **void** listAllFiles(File dir, FileFilter fileFilter, List<File> list) {

File[] files = dir.listFiles();

**for** (File file : files) {

**if** (file.isDirectory()) {

*listAllFiles*(file, fileFilter, list);

} **else** {

**if** (fileFilter.accept(file)) {// 如果迭代的文件满足 指定的fileFilter的条件则把数据保存起来

list.add(file);

}

}

}

}

/\*\*

\* 把满足条件的List<File>写入到指定的文件中

\*

\* **@param** list

\* 存储的是满足条件的所有的List<File>文件

\* **@param** file

\* 要存储数据的文件

\*/

**public** **static** **void** writeToFile(List<File> list, File file) {

BufferedWriter bufw = **null**;

**try** {

bufw = **new** BufferedWriter(**new** FileWriter(file));

// 对list集合迭代

**for** (File f : list) {

bufw.write(f.getAbsolutePath());

bufw.newLine();

bufw.flush();

}

} **catch** (IOException e) {

e.printStackTrace();

} **finally** {

**try** {

bufw.close();

} **catch** (IOException e) {

**throw** **new** RuntimeException("关闭失败");

}

}

}

}

### 转换流

输入流体系还提供了两个转换流，用于实现把字节流转换为字符流

**package** com.guigu.transfer;

**import** java.io.BufferedReader;

**import** java.io.BufferedWriter;

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**public** **class** KeyinTest {

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

// 把读取到的键盘输入的内容直接写出到指定的文件中

InputStreamReader isr = **new** InputStreamReader(System.***in***);

BufferedReader br = **new** BufferedReader(isr);

String line = **null**;

BufferedWriter writer = **new** BufferedWriter(**new** FileWriter("D:/guigu.txt", **true**));

**while** ((line = br.readLine()) != **null**) {

**if** ("exit".equals(line)) {

System.*exit*(0);

}

writer.write(line);

writer.newLine();

writer.flush();

}

writer.close();

br.close();

}

}

### 打印流

**package** com.guigu.transfer;

**import** java.io.IOException;

**import** java.io.PrintStream;

**public** **class** PrintStreamTest {

/\*\*

\* 打印流 PrintStream 和PrintWriter

\* PrintStram

\* 1.给字节输出流提供打印方法

\* 2.方便打印数值表示的形式

\* 3.给构造函数接收File对象

\* **@throws** IOException

\*

\*/

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

PrintStream ps =**new** PrintStream("D:/guigu.txt");

//write 写出一个int 只将最低字节写出

ps.write(97);

ps.write(353);

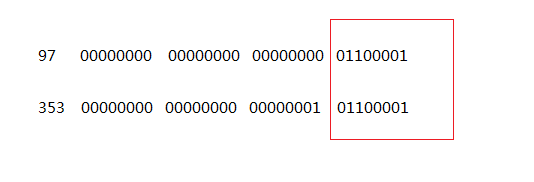
//print 将数值的形式保持不变输出

ps.print(97);

ps.print(353);

}

}

PrintWriter是字符打印流 打印流是用于大多数的输出。

**package** com.guigu.transfer;

**import** java.io.BufferedReader;

**import** java.io.BufferedWriter;

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.io.PrintWriter;

**public** **class** PrintWriterDemo {

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

BufferedReader br = **new** BufferedReader(**new** InputStreamReader(System.***in***));

PrintWriter pw = **new** PrintWriter(**new** BufferedWriter(**new** FileWriter("D:/guigu.txt")),**true**);

String line = **null**;

**while** ((line = br.readLine()) != **null**) {

**if** ("exit".equals(line)) {

System.*exit*(0);

}

pw.println(line.toLowerCase());

// pw.flush();

}

pw.close();

br.close();

}

}

重定向输出 输入

**package** com.guigu.transfer;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.IOException;

**import** java.util.Scanner;

**public** **class** RedirectIn {

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

FileInputStream fis = **new** FileInputStream("D:/guigu.txt");

//重定向输入

System.*setIn*(fis);

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

**while** (sc.hasNext()) {

System.***out***.println("读取到的内容是:" + sc.next());

}

}

}

**package** com.guigu.transfer;

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** java.io.PrintStream;

**import** java.util.Scanner;

**public** **class** RedirectOut {

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

PrintStream pw = **new** PrintStream("D:/guigu.txt");

System.*setOut*(pw);

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

**while**(scanner.hasNext()) {

//重定向到文件输入内容

System.***out***.println(scanner.next());

}

}

}

### 切割流

**package** com.guigu.splittest;

**import** java.io.BufferedInputStream;

**import** java.io.BufferedOutputStream;

**import** java.io.File;

**import** java.io.FileInputStream;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.util.Properties;

**public** **class** SplitFileTest {

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

*splitFile*(**new** File("C:/JDK\_1.6\_API.chm"));

}

**public** **static** **void** splitFile(File srcFile) **throws** IOException {

BufferedInputStream bis = **new** BufferedInputStream(**new** FileInputStream(srcFile));

BufferedOutputStream bos = **null**;

**byte**[] buffer = **new** **byte**[1024 \* 1024]; // 1M的缓冲区域

**int** hasRead = 0;

**int** count = 1;

**while** ((hasRead = bis.read(buffer)) != -1) {

bos = **new** BufferedOutputStream(**new** FileOutputStream(**new** File("C:/test", (count++) + ".part")));

bos.write(buffer);

bos.close();

}

// 保存源文件的信息

Properties prop = **new** Properties();

prop.setProperty("partCount", count + "");

prop.setProperty("srcFilename", srcFile.getName());

bos = **new** BufferedOutputStream(**new** FileOutputStream(**new** File("C:/test", "info.properties")));

prop.store(bos, "save partfiles info");

bos.close();

bis.close();

}

}

SequenceInputStream 表示其他输入流的逻辑串联。它从输入流的有序集合开始，并从第一个输入流开始读取，直到到达文件末尾，接着从第二个输入流读取，依次类推，直到到达包含的最后一个输入流的文件末尾为止。

**package** com.guigu.splittest;

**import** java.io.BufferedOutputStream;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.SequenceInputStream;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.Enumeration;

**public** **class** SequenceInputStreamTest {

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

ArrayList<FileInputStream> al = **new** ArrayList<>();

**for** (**int** x = 1; x <= 36; x++) {

al.add(**new** FileInputStream("C:/test/" + x + ".part"));

}

// 把集合转换为枚举类型

Enumeration<FileInputStream> en = Collections.*enumeration*(al);

// 把多个源 合并为一个

SequenceInputStream sis = **new** SequenceInputStream(en);

BufferedOutputStream bos = **new** BufferedOutputStream(**new** FileOutputStream("D:/jdk.chm"));

**byte**[] buffer = **new** **byte**[1024];

**int** hasRead = 0;

**while** ((hasRead = sis.read(buffer)) != -1) {

bos.write(buffer);

}

bos.close();

sis.close();

}

}

### RandomAccessFile

RandomAccessFile是Java输入流和输出流体系中功能最丰富的文件访问类，它提供了众多的方法来访问文件的内容，它既可以读取文件内容，也可以向文件中输出数据、与普通的输入输出流不同的是RandomAccessFile支持随机访问的方式，程序可以跳过文件的任意位置直接读取数据。

此类的实例支持对随机访问文件的读取和写入。随机访问文件的行为类似存储在文件系统中的一个大型 byte 数组。存在指向该隐含数组的光标或索引，称为文件指针；输入操作从文件指针开始读取字节，并随着对字节的读取而前移此文件指针。如果随机访问文件以读取/写入模式创建，则输出操作也可用；输出操作从文件指针开始写入字节，并随着对字节的写入而前移此文件指针。写入隐含数组的当前末尾之后的输出操作导致该数组扩展。该文件指针可以通过 getFilePointer 方法读取，并通过 seek 方法设置。

值

Mode的含意

"r" 以只读方式打开。调用结果对象的任何 write 方法都将导致抛出 IOException。

"rw" 打开以便读取和写入。如果该文件尚不存在，则尝试创建该文件。

"rws" 打开以便读取和写入，对于 "rw"，还要求对文件的内容或元数据的每个更新都同步写入到底层存储设备。

"rwd" 打开以便读取和写入，对于 "rw"，还要求对文件内容的每个更新都同步写入到底层存储设备。

**package** com.guigu.randomaccessfiletest;

**import** java.io.IOException;

**import** java.io.RandomAccessFile;

**public** **class** RandomAccessFileTest {

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

/\*\*

\* RandomAccessFile

\* 1、既可以读取文件也可也写入

\* 2、只能对文件操作

\* 3、内部维护了一个大型的byte数组，将字节输入流和字节输出流进行了封装

\* 4、通过索引的方式对数组中的元素进行操作，获取和设置索引的方法是getFilePoniter seek两种方式

\* 5、随机访问的原理是：就是通过操作索引的方法对指针进行自定义，指定位置来进行读写

\*

\*/

// writeFile();

*readFile*();

}

**public** **static** **void** writeFile() **throws** IOException{

//1.创建随机读取文件的对象

RandomAccessFile raf =**new** RandomAccessFile("D:/guigu.txt", "rw");

//2.将数据写入到文件中

raf.write("张三".getBytes());

raf.writeInt(97);

raf.write("李四".getBytes());

raf.writeInt(99);

//获取当前指针的位置

System.***out***.println(raf.getFilePointer());

raf.write("赵六".getBytes());

raf.writeDouble(3.14);

System.***out***.println(raf.getFilePointer());

}

**public** **static** **void** readFile() **throws** IOException{

RandomAccessFile raf =**new** RandomAccessFile("D:/guigu.txt", "r");

//跳过指定的字节

raf.seek(8\*1);

**byte** [] buffer =**new** **byte**[4];

raf.read(buffer);

String name =**new** String(buffer);

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

**int** age =raf.readInt();

System.***out***.println("age:"+age);

//写入的顺序必须和读取的顺序是一致的是

raf.read(buffer);

String name2 =**new** String(buffer);

System.***out***.println("name2:"+name2);

**double** d =raf.readDouble();

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

}

}

## 对象序列化

对象序列化的目标是将对象保存在磁盘中或允许在网络中直接传输对象。序列化机制是允许把内存中的Java对象转换为与平台无关的二进制流。从而允许把二进制流保存到硬盘上。通过网络将自动二进制流传输到另外一个网络节点。那么其他程序，一旦获得这种二进制流都可以通过二进制流恢复成原来的对象。

### 对象序列化的含义和意义

序列化机制容许将实现序列化的Java对象转换成字节序列。这些字节序列可以保存到磁盘上，或者通过网络传输。对以后重新恢复成原来的对象，序列化机制可以使对象脱离程序而独立存在。

对象序列化是指将一个Java对象写入到IO流中，以此对应的是对象的反序列化，直指从IO流中恢复某个Java对象。

**package** com.guigu.serializable;

**import** java.io.FileInputStream;

**import** java.io.IOException;

**import** java.io.ObjectInputStream;

**public** **class** ReadObject {

**public** **static** **void** main(String[] args) **throws** IOException, ClassNotFoundException {

ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream("object.txt"));

Person p = (Person) ois.readObject();

System.***out***.println("女神的名字是:"+p.getName()+",存在的岁月是:"+p.getAge());

}

}

注意： 反序列化 读取的仅仅是Java对象，不是Java类 所以对应的反序列的java的class文件一定要存在 否则抛出ClassNotfoundException.

当前Person对象只有一个有参的构造函数，没有无参的构造函数，而在反序列化的时候并没有传递参数进行读取Java对象，表明反序列化操作无需通过构造函数进行初始化java对象

## NIO

BufferedReader读取输入流中的数据，如果没有读到有效的数据，程序将在此处阻塞该线程的继续执行。如果使用的InputStream的read方法读取数据，如果数据源没有数据，他也会阻塞该线程的执行。阻塞式的IO流的读取。

从jdk1.4开始java提供了一系列改进的输入和输出处理的新功能，这些被称为NIO，新增了许多处理输入输出的类这些类放在java.nio这个包下。

### Java新IO的概述

新IO和传统的IO有相同的目的，都是用来进行输入和输出操作，但是新IO使用了不同的方式处理输入输出，新IO采用内存映射的方式处理输出和输出，新IO将文件或者文件的一段区域映射到内存中，这样可以像访问内存一样访问文件信息，这种方式模拟了操作系统中虚拟内存的概念。这样的方式输入和输出比传统方式快很多。

Java中新IO相关的包

java.nio 主要是包含各种Buffer相关的类

java.nio.channels 主要包含Channel和Selector相关的类

java.nio.channels.spi 主要是包含Channel相同的服务提供者编程接口

java.nio.charset 主要是与字符编码集相关的

java.nio.charset.spi 包含与字符编码集相关的服务提供者编程接口

Channel(通道)和Buffer(缓冲)都是新IO的两个核心概念，Channel是对传统输入和输出体系的模拟，在新IO系统中所以的数据都需要通过Channle(通道)进行传输，Channel与传统的InputStream和OutputStream最大的区别在于Channel提供了一个map方法。通过map方法可以直接将一块数据映射到内存中。

Buffer可以理解为一个容器，它的本质是一个数组，发送到Channel中所有的对象必须首先放到Buffer中，从而Channel中读取数据也必须先放到Buffer。

除了Channel和Buffer以外，新IO还提供了用于字符编码的映射，也支持非阻塞式的输入和输出。

### 使用Buffer

从内部结构看Buffer就像一个数组，他可以保存多个类型相同的数据，Buffer是一个抽象类，依赖子类实现功能，最常用的子类是ByteBuffer,它可以在底层字节数组上进行get和set操作。除了Bytebuffer以外还有CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, ShortBuffer。

**在Buffer中有三个重要的概念：容量(Capacity)、界限(limit)、位置(position)**

容量（Capacity）：缓冲区的容器，表示该Buffer的最大数据容量，即可以最多存储多少数据，缓冲区的容器不能为负数，创建以后不能改变。

界限(limit):第一个不应该被读写的缓冲区的位置。(不能读写的位置)

位置(Position)用于指明下一个可用来读写的位置的索引。

**package** com.guigu.nio.buffer;

**import** java.nio.CharBuffer;

**public** **class** BufferTest {

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

//创建Buffer

CharBuffer buffer =CharBuffer.*allocate*(8);

System.***out***.println("capacity:"+buffer.capacity());

System.***out***.println("limit:"+buffer.limit());

System.***out***.println("position:"+buffer.position());

//存入数据

buffer.put('a');

buffer.put('b');

buffer.put('c');

System.***out***.println("存储三个元素后position:"+buffer.position());

//调用flip()操作 反转此缓冲区。

buffer.flip();

System.***out***.println("执行flip操作后limit:"+buffer.limit());

System.***out***.println("执行flip操作后position:"+buffer.position());

//调用clear方法 清除此缓冲区。

buffer.clear();

System.***out***.println("执行clear操作后limit:"+buffer.limit());

System.***out***.println("执行clear操作后position:"+buffer.position());

System.***out***.println("清除缓存区以后数据仍然存在:"+buffer.get(2));

}

}

### 使用Channel

Channel类与传统的流对象的区别主要有两个:

* Channle可以直接将指定文件的部分或者全部映射成Buffer
* 程序不能直接访问Channel中的数据，，包括读取和写入都不行，Channel只能与buffer进行交互。如果想要从Channel中取数据，必须先用Buffer从Channel中取出一些数据，然后让程序从Buffer中读取数据即可。

**package** com.guigu.nio.buffer;

**import** java.io.File;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.nio.CharBuffer;

**import** java.nio.MappedByteBuffer;

**import** java.nio.channels.FileChannel;

**import** java.nio.charset.Charset;

**import** java.nio.charset.CharsetDecoder;

**public** **class** FileChannelTest {

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

File file =**new** File("D:/guigu.txt");

//创建FileInputStream以该文件输入流创建FileChannel

FileChannel inChannel =**new** FileInputStream(file).getChannel();

//以FileOutputStream创建FIleBuffer 用来控制输出

FileChannel outChannel=**new** FileOutputStream("a.txt").getChannel();

//将FileChannle中全部的数据映射为byteBuffer然后进行操作

//把file文件中所有的内容映射到内存中

MappedByteBuffer buffer =inChannel.map(FileChannel.MapMode.***READ\_ONLY***, 0, file.length());

//把映射到内存中的数据写出到指定的文件中

outChannel.write(buffer);

buffer.clear();

//创建解码器

//设置编码集

Charset charSet=Charset.*forName*("GBK");

CharsetDecoder decoder=charSet.newDecoder();

//使用解码解把内存的内容输出到控制台

CharBuffer charBUffer=decoder.decode(buffer);

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

}

}

### 使用CharSet

**package** com.guigu.nio.buffer;

**import** java.nio.charset.Charset;

**import** java.util.SortedMap;

**public** **class** CharSetTest {

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

SortedMap<String, Charset> map = Charset.*availableCharsets*();

map.forEach((key, value) -> System.***out***.println(key + "--" + value));

}

}

**package** com.guigu.nio.buffer;

**import** java.io.IOException;

**import** java.nio.ByteBuffer;

**import** java.nio.CharBuffer;

**import** java.nio.charset.Charset;

**import** java.nio.charset.CharsetDecoder;

**import** java.nio.charset.CharsetEncoder;

**public** **class** CharsetTransfor {

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

Charset cn = Charset.*forName*("GBK");

// 获得相关的编码器和解码器

CharsetEncoder cnEncoder = cn.newEncoder();

CharsetDecoder cnDecoder = cn.newDecoder();

// 创建一个CharBuffer对象

CharBuffer cbuffer = CharBuffer.*allocate*(8);

cbuffer.put("武");

cbuffer.put("则");

cbuffer.put("天");

cbuffer.flip();

//将CharBuffer中的字符序列 转换为字节序列

ByteBuffer bbuffer=cnEncoder.encode(cbuffer);

//测试是否转换成功

**for** (**int** i=0;i<bbuffer.capacity();i++) {

System.***out***.print(bbuffer.get(i)+" ");

}

//将ByteBuffer解析为字符序列

System.***out***.println(cnDecoder.decode(bbuffer));

}

}

## Java7新增的NIO2

### Path、Paths、和Files核心的API

**package** com.guigu.nio2;

**import** java.nio.file.Path;

**import** java.nio.file.Paths;

**public** **class** PathTest {

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

//以当前路径创建Path对象

Path path=Paths.*get*(".");

System.***out***.println("Path里包含的路径数量:"+path.getNameCount());

System.***out***.println("Path的根路径信息:"+path.getRoot());

System.***out***.println("Path的绝路径:"+path.toAbsolutePath());

System.***out***.println("AbsolutePath路径包含的路径信息:"+path.toAbsolutePath().getNameCount());

System.***out***.println(path.toAbsolutePath().getName(0));

Path path2=Paths.*get*("D:", "guigu.txt");

System.***out***.println(path2);

System.***out***.println(path2.getNameCount());

System.***out***.println(path2.getRoot());

}

}

**package** com.guigu.nio2;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.nio.charset.Charset;

**import** java.nio.file.Files;

**import** java.nio.file.Paths;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** FilesTest {

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

//复制文件

Files.*copy*(Paths.*get*("D:", "guigu.txt"), **new** FileOutputStream("hello.txt"));

//判断文件是否是隐藏

System.***out***.println(Files.*isHidden*(Paths.*get*("D:", "guigu.txt")));

//判断指定文件的大小

System.***out***.println("得到指定文件的大小:"+Files.*size*(Paths.*get*("D:","abc","guigu.txt")));

//一次性读取指定文件的所有行

List<String> lines=Files.*readAllLines*(Paths.*get*("D:","abc","guigu.txt"));

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

//一次性把多个字符写入到指定文件中

List<String> strs=**new** ArrayList<>();

strs.add("清风徐来水波不惊");

strs.add("床前明月光");

Files.*write*(Paths.*get*("hello.txt"), strs, Charset.*forName*("GBK"));

//使用Java8新增的Stram的API累出当前目录下所有的文件和子目录

Files.*list*(Paths.*get*("C:","WorkSpace","J2001")).forEach(path->System.***out***.println(path));

}

}

### 使用FileVisitor遍历文件和目录

**package** com.guigu.nio2;

**import** java.io.IOException;

**import** java.nio.file.FileVisitResult;

**import** java.nio.file.Files;

**import** java.nio.file.Path;

**import** java.nio.file.Paths;

**import** java.nio.file.SimpleFileVisitor;

**import** java.nio.file.attribute.BasicFileAttributes;

**public** **class** FileVisitorTest {

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

Files.*walkFileTree*(Paths.*get*("C:", "WorkSpace", "J2001"), **new** SimpleFileVisitor<Path>() {

// 访问文件的时候触发

@Override

**public** FileVisitResult visitFile(Path file, BasicFileAttributes attrs) **throws** IOException {

System.***out***.println("正在访问:" + file + "文件");

// return super.visitFile(file, attrs);

**return** FileVisitResult.***CONTINUE***;// 继续往下迭代

}

// 访问目录的时候触发

@Override

**public** FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) **throws** IOException {

System.***out***.println("正在访问:" + dir + "路径文件夹");

// return super.preVisitDirectory(dir, attrs);

**return** FileVisitResult.***CONTINUE***;

}

});

}

}

### 使用WatchService监控文件的变化

**package** com.guigu.nio2;

**import** java.io.IOException;

**import** java.nio.file.FileSystem;

**import** java.nio.file.FileSystems;

**import** java.nio.file.Paths;

**import** java.nio.file.StandardWatchEventKinds;

**import** java.nio.file.WatchEvent;

**import** java.nio.file.WatchKey;

**import** java.nio.file.WatchService;

**public** **class** WatchServiceTest {

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

// 获得系统文件的WatchService对象

WatchService watchService = FileSystems.*getDefault*().newWatchService();

// 为c盘注册监听

Paths.*get*("C:/").register(watchService, StandardWatchEventKinds.***ENTRY\_CREATE***,

StandardWatchEventKinds.***ENTRY\_DELETE***, StandardWatchEventKinds.***ENTRY\_MODIFY***);

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

// 获取下一个改变的事件

WatchKey key = watchService.take();

**for** (WatchEvent<?> event : key.pollEvents()) {

System.***out***.println(event.context() + "---文件发生了：" + event.kind() + "事件");

}

// 重置watchKey

**boolean** valid = key.reset();

**if** (!valid) {// 如果设置失败则退出

**break**;

}

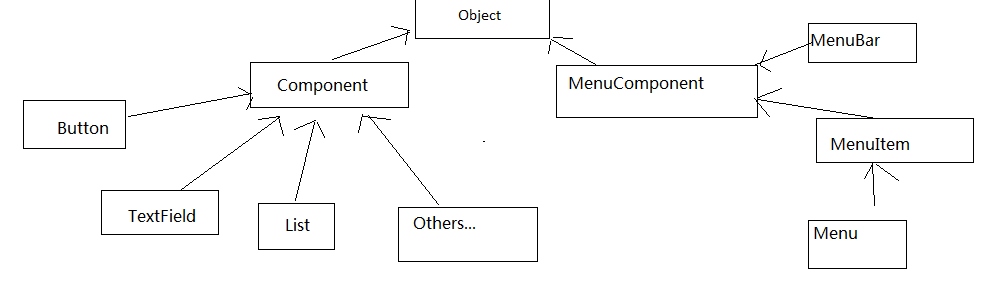
}

}

}

# GUI图形化界面

Java使用AWT和Swing完成图形化界面，其中AWT被称为抽象工具集。



## AWT和Swing容器

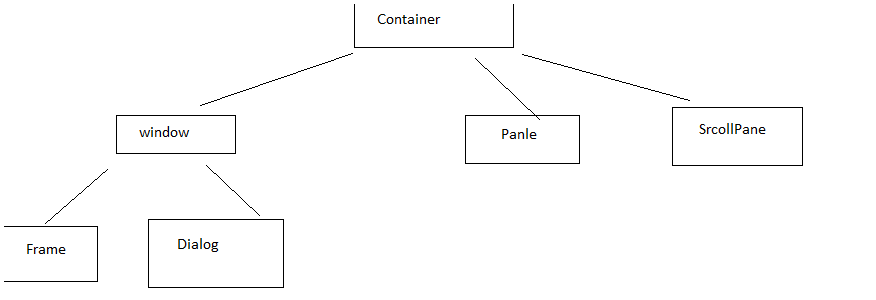
任何窗口都可以被分解为一个空的容器，容器里存储大量的基本组件，通过设置这些组件的大小和位置就可以完成一个图像化界面。

java.lang.Object

java.awt.Component

java.awt.Container

Container是Component的一个子类，其中可以存储其他的组件信息。



### JFrame框架

**package** com.guigu.awt;

**import** javax.swing.JFrame;

**public** **class** MyFrame {

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

// 创建窗体

JFrame jf = **new** JFrame("第一个窗口程序");

// 设置窗体的大小和位置

jf.setBounds(50, 50, 500, 400);

// 窗体默认 是不可见的 可以通过setVisible设置可见

jf.setVisible(**true**);

// 设置窗体的关闭行为

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

}

### Panle面板

Panle是组件中的另外一容器，它是不能独立存在的，必须存放在其他的容器里。

**package** com.guigu.awt;

**import** java.awt.Button;

**import** java.awt.TextField;

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**public** **class** MyPanel {

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

JFrame jf =**new** JFrame("测试JPanle");

//设置大小

jf.setBounds(10, 10, 600, 400);

//创建Panel容器

JPanel jp=**new** JPanel();

//在Jpanle上存放一些组件

jp.add(**new** Button("点我"));

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

//将JPanle存放于 JFrame中

jf.add(jp);

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

}

### JScrollPane

**package** com.guigu.awt;

**import** java.awt.TextField;

**import** javax.swing.JFrame;

**import** javax.swing.JScrollPane;

**public** **class** MyScrollPanel {

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

JFrame jf = **new** JFrame("测试滚动窗口");

jf.setBounds(50, 50, 600, 400);

JScrollPane jsc = **new** JScrollPane(JScrollPane.***VERTICAL\_SCROLLBAR\_ALWAYS***,JScrollPane.***HORIZONTAL\_SCROLLBAR\_ALWAYS***);

// 向JScrollPane中添加数据

jsc.add(**new** TextField(60));

// jsc.add(new Button("点击"));

jf.add(jsc);

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

}

### 工具类Toolkit

**package** com.guigu.awt;

**import** java.awt.Dimension;

**import** java.awt.Toolkit;

**import** javax.swing.JFrame;

**public** **class** JFrameCenter {

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

JFrame jf = **new** JFrame();

// jf.setSize(600, 400);

//得到屏幕的大小

Toolkit tk =Toolkit.*getDefaultToolkit*();

Dimension d=tk.getScreenSize();

jf.setBounds((**int**)(d.getWidth()/2-300), (**int**)(d.getHeight()/2-200), 600, 400);

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

}

## AWT的布局管理器

为了使生成的图像化界面拥有更好的外观，可以通过布局管理器管理组件摆放的位置。

### BorderLayout

**package** com.guigu.layout;

**import** java.awt.BorderLayout;

**import** java.awt.Button;

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**public** **class** BorderLayoutDemo {

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

JFrame jf = **new** JFrame("测试布局管理器");

// 设置JFrame的布局管理器

jf.setLayout(**new** BorderLayout(15, 15));

JPanel jp = **new** JPanel();

jp.add(**new** Button("南面1"));

jp.add(**new** Button("南面2"));

// 分别向五个区域添加组件

jf.add(**new** Button("北面"), BorderLayout.***NORTH***);

jf.add(jp, BorderLayout.***SOUTH***);

jf.add(**new** Button("西面"), BorderLayout.***WEST***);

jf.add(**new** Button("东面"), BorderLayout.***EAST***);

jf.add(**new** Button("中间"), BorderLayout.***CENTER***);

jf.setBounds(59, 60, 600, 500);

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

}

**package** com.guigu.layout;

**import** java.awt.BorderLayout;

**import** java.awt.Button;

**import** java.awt.TextField;

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**public** **class** BorderLayoutDemo2 {

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

JFrame jf = **new** JFrame("测试");

jf.setLayout(**new** BorderLayout(10, 10));

JPanel jp = **new** JPanel();

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

jf.add(jp, BorderLayout.***NORTH***);

// 在创建一个面板

JPanel jp2 = **new** JPanel();

jp2.setSize(5, 20);

String[] names = { "1", "2", "3", "4", "5", "6", "7", "8", "9", "0", "+", "-", "\*", "/", "=" };

**for** (String butName : names) {

jp2.add(**new** Button(" " + butName + " "));

}

jf.add(jp2, BorderLayout.***CENTER***);

// 根据组件自动填充大小

jf.pack();

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

}

### FlowLayout

**package** com.guigu.layout;

**import** java.awt.Button;

**import** java.awt.FlowLayout;

**import** javax.swing.JFrame;

**public** **class** FlowLayoutDemo {

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

JFrame jf = **new** JFrame("测试");

jf.setLayout(**new** FlowLayout(FlowLayout.***CENTER***, 20, 10));

**for** (**int** i = 0; i < 10; i++) {

jf.add(**new** Button("按钮" + (i + 1) + "号"));

}

jf.pack();

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

}

### GridLayout

**package** com.guigu.layout;

**import** java.awt.BorderLayout;

**import** java.awt.Button;

**import** java.awt.GridLayout;

**import** java.awt.TextField;

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**public** **class** GridLayoutTest {

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

JFrame jf = **new** JFrame("简易计算器");

JPanel jpText = **new** JPanel();

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

jf.add(jpText, BorderLayout.***NORTH***);

JPanel butPanel = **new** JPanel(**new** GridLayout(3, 5));

String[] names = { "1", "2", "3", "4", "5", "6", "7", "8", "9", "0", "+", "-", "\*", "/", "=" };

**for** (String ss : names) {

butPanel.add(**new** Button(" " + ss + " "));

}

jf.add(butPanel);

jf.pack();

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

}

### 绝对定位

**package** com.guigu.layout;

**import** javax.swing.JButton;

**import** javax.swing.JFrame;

**public** **class** AbsoluteLayoutDemo {

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

JFrame jf = **new** JFrame("测试");

jf.setLayout(**null**);

JButton jb = **new** JButton("按钮1");

// 单独设置JButton的大小和位置

jb.setBounds(40, 40, 60, 40);

JButton jb2 = **new** JButton("按钮2");

jb2.setBounds(100, 200, 50, 30);

jf.add(jb);

jf.add(jb2);

jf.setBounds(0, 0, 600, 600);

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

}

## 组件

AWT’提供了以下组件

Button按钮

CheckBox 复选按钮

CheckBoxGroup

Lable 标签

List 列表框

TextArea 文本域

TextField文本框

**package** com.guigu.layout;

**import** java.awt.BorderLayout;

**import** java.awt.Checkbox;

**import** java.awt.CheckboxGroup;

**import** java.awt.Choice;

**import** java.awt.List;

**import** java.awt.TextArea;

**import** javax.print.attribute.standard.MediaSize.NA;

**import** javax.swing.Box;

**import** javax.swing.JButton;

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**import** javax.swing.JTextField;

**public** **class** ComponentTest {

// 定义框架

JFrame jf = **new** JFrame();

// 定义下方的区域 的两个组件

JButton ok = **new** JButton("ok");

JTextField jtf = **new** JTextField(50);

// 定义上方右侧的区域

List colorList = **new** List();

// 定义上方区域的 左侧区域的 上半部分

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

//// 定义上方区域的 左侧区域的 下半部分

Choice edu = **new** Choice();

// 单选按钮

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

// 定义单选

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

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

// 定义复选按钮

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

/\*\*

\* 初始化数据

\*/

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

// 列表框填充数据

colorList.add("赤色");

colorList.add("橙色");

colorList.add("黄色");

colorList.add("绿色");

colorList.add("青色");

colorList.add("蓝色");

colorList.add("紫色");

// 下拉列表定义数据

edu.add("硕士");

edu.add("本科");

edu.add("专科");

edu.add("高中");

// 完成布局 组装最下方的区域

JPanel bottom = **new** JPanel();

bottom.add(jtf);

bottom.add(ok);

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

// 创建一个Jpanel存储上方 左侧下方的数据

JPanel checkPanle = **new** JPanel();

checkPanle.add(edu);

checkPanle.add(male);

checkPanle.add(female);

checkPanle.add(mattied);

// 创建一个垂直的Box 分为上下两个部分

Box topLeft = Box.*createVerticalBox*();

topLeft.add(ta);

topLeft.add(checkPanle);

// 在定义一个Box用于存储整个上方的区域

Box top = Box.*createHorizontalBox*();

top.add(topLeft);

top.add(colorList);

jf.add(top);

jf.pack();

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

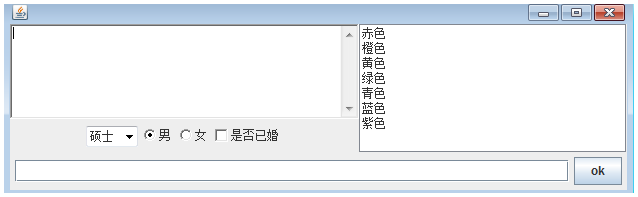
}

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

**new** ComponentTest().init();

}

}



## 事件处理

### Java事件的模型的流程

为了使图形化界面能接收用户的请求，必须给各个组件添加事件处理机制，在事件处理机制的过程中主要涉及到三个类对象。

* Event Source(事件源) 事件发生的场所，通常就是各个组件，比如按钮，窗口，菜单等组件
* Event(事件) 事件封装了GUI组件上发生的特定的事件，即用户的一次操作，如果需要获取GUI组件上发生的事件相关的信息 必须通过Event对象获取
* Event Listener(事件监听) 负责监听事件源发生的事件，并且对各个事件做出处理和响应。

**package** com.guigu.event;

**import** java.awt.BorderLayout;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** javax.swing.JButton;

**import** javax.swing.JFrame;

**import** javax.swing.JTextField;

**public** **class** ActionListenerEventTest {

JFrame jf = **new** JFrame("测试窗体");

JTextField jtf = **new** JTextField(30);

JButton jb = **new** JButton("点我");

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

jf.setBounds(100, 100, 400, 400);

// 对这个JButton按钮添加监听事件

jb.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

System.***out***.println("用户点击了按钮....");

jtf.setText(jtf.getText()+"hello guigu hello cat");

}

});

jf.add(jtf, BorderLayout.***NORTH***);

jf.add(jb, BorderLayout.***SOUTH***);

jf.pack();

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

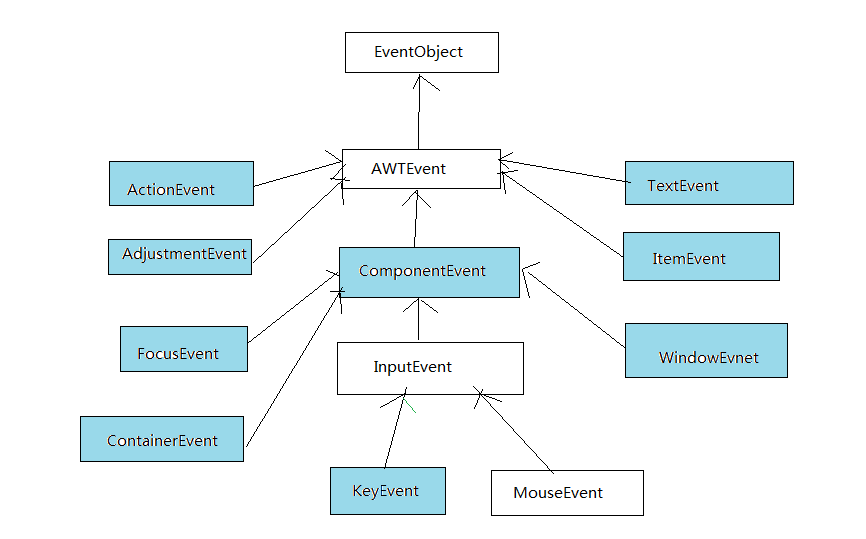
**public** **static** **void** main(String[] args) {

**new** ActionListenerEventTest().init();

}

}

### 事件和事件源



不同的事件需要使用不同的监听器监听，不同的监听器需要实现不同的监听器接口，当指定事件发生后，事件监听器就会调用所包含的事件处理器来处理。

事件、监听器接口和处理器之间的对应关系

|  |  |  |
| --- | --- | --- |
| 事件 | 监听器接口 | 处理器及触发时机 |
| ActionEvent | ActionListener | actionPerformed：按钮、文本框、菜单项被单击时触发 |
| AdjustmentEvent | AdjustmentListener | adjustmentValueChanged：滑块位置发生改变时触发 |
| ContainerEvent | ContainerListener | componentAdded：向容器中添加组件时触发 |
| componentRemoved：从容器中删除组件时触发 |
| FocusEvent | FocusListener | focusGainer：组件得到焦点时触发 |
| focusLost：组件失去焦点时触发 |
| ComponentEvent | ComponentListener | componentHidden：组件被隐藏时触发 |
| componentMoved：组件位置发生改变时触发 |
| componentResized：组件大小发生改变时触发 |
| componentShown：组件被显示时触发 |
| KeyEvent | KeyListener | keyPressed：按下某个按键时触发 |
| keyReleased：松开某个按键时触发 |
| keyTyped：单击某个按键时触发 |
| MouseEvent | MouseListener | mouseClicked：在某个组件上单击鼠标键时触发 |
| mouseEntered：鼠标进入某个组件时触发 |
| mouseExited：鼠标离开某个组件时触发 |
| mousePressed：在某个组件上按下鼠标键时触发 |
| mouseReleased：在某个组件上松开鼠标键时触发 |
| MouseMotionListener | mouseDragged：在某个组件上移动鼠标，且按下鼠标键时触发 |
| mouseMoved：在某个组件上移动鼠标，且没有按下鼠标键时触发 |
| TextEvent | TextListener | textValueChanged：文本组件里的文本发生改变时触发 |
| ItemEvent | ItemListener | itemStateChanged：某项被选中或取消选中时触发 |
| WindowEvent | WindowListener | windowActivated：窗口被激活时触发。 |
| windowClosed：窗口调用dispose()即关闭时触发。 |
| windowClosing：用户单击窗口右上角的“X”按钮时触发 |
| windowDeactivated：窗口失去激活时触发 |
| windowDeiconified：窗口被恢复时触发 |
| windowIconified：窗口最小化时触发 |
| windowOpened：窗口首次被打开时触发。 |

**package** com.guigu.event;

**import** java.awt.event.MouseEvent;

**import** java.awt.event.MouseListener;

**import** java.awt.event.MouseMotionListener;

**import** java.awt.event.MouseWheelEvent;

**import** java.awt.event.MouseWheelListener;

**import** javax.swing.JButton;

**import** javax.swing.JFrame;

**public** **class** MouseEventDemo {

JFrame jf = **new** JFrame();

JButton jb1 = **new** JButton("guigu");

**public** MouseEventDemo() {

jb1.addMouseListener(**new** MouseListener() {

@Override

**public** **void** mouseReleased(MouseEvent e) {

System.***out***.println("鼠标的松开...1");

}

@Override

**public** **void** mousePressed(MouseEvent e) {

System.***out***.println("鼠标的按下...1");

}

@Override

**public** **void** mouseExited(MouseEvent e) {

System.***out***.println("鼠标的退出...1");

}

@Override

**public** **void** mouseEntered(MouseEvent e) {

System.***out***.println("鼠标的进入...1");

}

@Override

**public** **void** mouseClicked(MouseEvent e) {

// System.out.println(e.getButton());

System.***out***.println("------------------------------------"+e.getClickCount());

**int** i = e.getButton();

**if** (MouseEvent.***BUTTON1*** == i) {

System.***out***.println("点击了左键");

} **else** **if** (MouseEvent.***BUTTON2*** == i) {

System.***out***.println("点击了滑轮");

} **else** **if** (MouseEvent.***BUTTON3*** == i) {

System.***out***.println("点击了右键");

}

System.***out***.println("鼠标的点击...1");

}

});

jb1.addMouseMotionListener(**new** MouseMotionListener() {

@Override

**public** **void** mouseMoved(MouseEvent e) {

//得到当前鼠标的位置

System.***out***.println("x的坐标是:"+e.getX());

System.***out***.println("y的坐标是:"+e.getY());

System.***out***.println("鼠标的移动......");

}

@Override

**public** **void** mouseDragged(MouseEvent e) {

System.***out***.println("鼠标的拖动.......!!!!");

}

});

jb1.addMouseWheelListener(**new** MouseWheelListener() {

@Override

**public** **void** mouseWheelMoved(MouseWheelEvent e) {

System.***out***.println("鼠标滑轮的滚动....");

}

});

jf.setBounds(100, 100, 400, 300);

jf.add(jb1);

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

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

**new** MouseEventDemo();

}

}

#### 使用适配器模式简化监听

**package** com.guigu.event;

**import** java.awt.event.MouseEvent;

**import** java.awt.event.MouseListener;

**public** **class** MyTest **implements** MouseListener {

@Override

**public** **void** mouseClicked(MouseEvent e) {

// **TODO** Auto-generated method stub

}

@Override

**public** **void** mousePressed(MouseEvent e) {

// **TODO** Auto-generated method stub

}

@Override

**public** **void** mouseReleased(MouseEvent e) {

// **TODO** Auto-generated method stub

}

@Override

**public** **void** mouseEntered(MouseEvent e) {

// **TODO** Auto-generated method stub

}

@Override

**public** **void** mouseExited(MouseEvent e) {

// **TODO** Auto-generated method stub

}

}

**package** com.guigu.event;

**import** java.awt.event.MouseEvent;

**import** javax.swing.JButton;

**import** javax.swing.JFrame;

**public** **class** MouseAdpaterDemo {

**public** MouseAdpaterDemo() {

JFrame jf = **new** JFrame("测试");

jf.setSize(600, 400);

JButton jb1 = **new** JButton("按钮1");

// 只需要对鼠标的点击行为监听

jb1.addMouseListener(**new** MyTest() {

@Override

**public** **void** mouseClicked(MouseEvent e) {

System.***out***.println("监听到了鼠标的点击行为...");

}

});

jf.add(jb1);

jf.pack();

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

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

**new** MouseAdpaterDemo();

}

}

**package** com.guigu.event;

**import** java.awt.event.MouseAdapter;

**import** java.awt.event.MouseEvent;

**import** java.awt.event.MouseWheelEvent;

**import** javax.swing.JButton;

**import** javax.swing.JFrame;

**public** **class** MouseAdpaterDemo {

**public** MouseAdpaterDemo() {

JFrame jf = **new** JFrame("测试");

jf.setSize(600, 400);

JButton jb1 = **new** JButton("按钮1");

// 只需要对鼠标的点击行为监听

// jb1.addMouseListener(new MyTest() {

// @Override

// public void mouseClicked(MouseEvent e) {

// System.out.println("监听到了鼠标的点击行为...");

// }

//

// });

jb1.addMouseListener(**new** MouseAdapter() {

@Override

**public** **void** mouseClicked(MouseEvent e) {

System.***out***.println("鼠标的监听...点击");

}

// @Override

// public void mouseWheelMoved(MouseWheelEvent e) {

// System.out.println("滑轮滚动....");

// }

});

jb1.addMouseWheelListener(**new** MouseAdapter() {

@Override

**public** **void** mouseWheelMoved(MouseWheelEvent e) {

System.***out***.println("滑轮滚动....");

}

});

jf.add(jb1);

jf.pack();

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

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

**new** MouseAdpaterDemo();

}

}

**package** com.guigu.event;

**import** java.awt.FlowLayout;

**import** java.awt.Toolkit;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** java.awt.event.WindowAdapter;

**import** javax.swing.JButton;

**import** javax.swing.JFrame;

**public** **class** EventDemoTest **extends** WindowAdapter **implements** ActionListener {

JButton jb1 = **null**;

JButton jb2 = **null**;

**public** EventDemoTest() {

JFrame jf = **new** JFrame("测试");

jb1 = **new** JButton("点我有声音");

jb2 = **new** JButton("点我打开新窗口");

jb1.addActionListener(**this**);

jb2.addActionListener(**this**);

jf.setLayout(**new** FlowLayout());

jf.add(jb1);

jf.add(jb2);

jf.pack();

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

@Override

**public** **void** actionPerformed(ActionEvent e) {

**if** (e.getSource() == jb1) {

System.***out***.println("哈哈...");

Toolkit.*getDefaultToolkit*().beep();

}

**if** (e.getSource() == jb2) {

JFrame newFrame = **new** JFrame("新窗口");

newFrame.setSize(600, 400);

newFrame.setVisible(**true**);

}

}

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

**new** EventDemoTest();

}

}

### 简易计算器

**package** com.guigu.calc;

**import** java.awt.BorderLayout;

**import** java.awt.Color;

**import** java.awt.FlowLayout;

**import** java.awt.GridLayout;

**import** java.awt.event.MouseAdapter;

**import** java.awt.event.MouseEvent;

**import** javax.swing.JButton;

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**import** javax.swing.JTextField;

**import** javax.swing.SwingConstants;

**public** **class** Calc {

**private** JFrame jf = **new** JFrame("简易计算器");

// 面板1 装第一排的按钮

**private** JPanel jp1 = **new** JPanel();

// 面板2 装 面板3和面板4

**private** JPanel jp2 = **new** JPanel();

// 装下方左侧的数字 和右侧的符号

**private** JPanel jp3 = **new** JPanel();

**private** JPanel jp4 = **new** JPanel();

// 定义第一排的按钮

**private** JButton clear = **new** JButton("clear");

**private** JButton back = **new** JButton("BackSpace");

**private** JButton CE = **new** JButton("CE");

**private** JButton About = **new** JButton("about");

// 定义下方左排的数字

**private** JButton[] numbers = **new** JButton[10];

**private** JButton point = **new** JButton(".");

**private** JButton pm = **new** JButton("+/-");

// 右派的按钮

**private** JButton add = **new** JButton("+");

**private** JButton sub = **new** JButton("-");

**private** JButton mul = **new** JButton("\*");

**private** JButton div = **new** JButton("/");

**private** JButton ds = **new** JButton("1/x");

**private** JButton prec = **new** JButton("%");

**private** JButton sqrt = **new** JButton("sqrt");

**private** JButton equ = **new** JButton("=");

// 文本框

**private** JTextField text = **new** JTextField();

/\*\*

\* 完成组件之间的布局

\*/

**public** **void** layout() {

// 左排的按钮

jp3.setLayout(**new** GridLayout(4, 3, 2, 2));

**for** (**int** i = 9; i >= 0; i--) {

numbers[i] = **new** JButton(String.*valueOf*(i));

jp3.add(numbers[i]);

}

jp3.add(pm);

jp3.add(point);

jp4.setLayout(**new** GridLayout(4, 2, 2, 2));

div.setForeground(Color.***RED***);

jp4.add(div);

jp4.add(sqrt);

mul.setForeground(Color.***RED***);

jp4.add(mul);

jp4.add(prec);

sub.setForeground(Color.***RED***);

jp4.add(sub);

jp4.add(ds);

add.setForeground(Color.***RED***);

jp4.add(add);

jp4.add(equ);

jp2.setLayout(**new** GridLayout(1, 2, 4, 4));

jp2.add(jp3);

jp2.add(jp4);

// 第一排的按钮

jp1.setLayout(**new** FlowLayout(FlowLayout.***RIGHT***));

back.setForeground(Color.***blue***);

jp1.add(back);

CE.setForeground(Color.***blue***);

jp1.add(CE);

clear.setForeground(Color.***blue***);

jp1.add(clear);

About.setForeground(Color.***blue***);

jp1.add(About);

text.setHorizontalAlignment(SwingConstants.***RIGHT***);//设置对齐方式

text.setEditable(**false**);//设置不可编辑

jf.add(text, BorderLayout.***NORTH***);

jf.add(jp1);

jf.add(jp2, BorderLayout.***SOUTH***);

jf.pack();

jf.setVisible(**true**);

// 设置禁止最大化

jf.setResizable(**false**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

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

layout();

//设置初始化文本的内容是0

text.setText("0");

//添加监听

numbers[9].addMouseListener(**new** MouseAdapter() {

@Override

**public** **void** mouseClicked(MouseEvent e) {

//得到文本的值

String currentText=text.getText();

**if**(currentText.length()==1 && currentText.startsWith("0")) {

text.setText("9");

}**else** {

text.setText(currentText.concat("9"));

}

}

});

}

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

**new** Calc().init();

}

}

## 菜单

### 顶部菜单

MenuBar菜单条

**package** com.guigu.menu;

**import** java.awt.BorderLayout;

**import** java.awt.EventQueue;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**import** javax.swing.border.EmptyBorder;

**import** javax.swing.JMenuBar;

**import** javax.swing.JMenu;

**import** javax.swing.JMenuItem;

**import** javax.swing.JTextArea;

**public** **class** SimpleMenu **extends** JFrame {

**private** JPanel contentPane;

**private** JTextArea textArea\_1;

/\*\*

\* Launch the application.

\*/

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

EventQueue.*invokeLater*(**new** Runnable() {

**public** **void** run() {

**try** {

SimpleMenu frame = **new** SimpleMenu();

frame.setVisible(**true**);

} **catch** (Exception e) {

e.printStackTrace();

}

}

});

}

/\*\*

\* Create the frame.

\*/

**public** SimpleMenu() {

setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

setBounds(100, 100, 550, 436);

JMenuBar menuBar = **new** JMenuBar();

setJMenuBar(menuBar);

JMenu menu = **new** JMenu("\u6587\u4EF6");

menuBar.add(menu);

JMenuItem menuItem = **new** JMenuItem("\u65B0\u5EFA");

menu.add(menuItem);

JMenuItem menuItem\_1 = **new** JMenuItem("\u6253\u5F00");

menu.add(menuItem\_1);

JMenuItem menuItem\_2 = **new** JMenuItem("\u4FDD\u5B58");

menu.add(menuItem\_2);

JMenuItem menuItem\_3 = **new** JMenuItem("\u9000\u51FA");

menu.add(menuItem\_3);

JMenu menu\_1 = **new** JMenu("\u7F16\u8F91");

menuBar.add(menu\_1);

JMenuItem menuItem\_4 = **new** JMenuItem("\u526A\u5207");

menu\_1.add(menuItem\_4);

JMenuItem menuItem\_5 = **new** JMenuItem("\u590D\u5236");

menu\_1.add(menuItem\_5);

JMenuItem menuItem\_6 = **new** JMenuItem("\u7C98\u8D34");

menu\_1.add(menuItem\_6);

JMenu menu\_2 = **new** JMenu("\u6CE8\u91CA");

menu\_1.add(menu\_2);

JMenuItem menuItem\_7 = **new** JMenuItem("\u6DFB\u52A0\u6CE8\u91CA");

menu\_2.add(menuItem\_7);

JMenuItem menuItem\_8 = **new** JMenuItem("\u53D6\u6D88\u6CE8\u91CA");

menu\_2.add(menuItem\_8);

JMenu menu\_3 = **new** JMenu("\u54C8\u54C8");

menu\_1.add(menu\_3);

contentPane = **new** JPanel();

contentPane.setBorder(**new** EmptyBorder(5, 5, 5, 5));

setContentPane(contentPane);

contentPane.setLayout(**null**);

textArea\_1 = **new** JTextArea();

textArea\_1.setRows(20);

textArea\_1.setBounds(41, 42, 337, 197);

contentPane.add(textArea\_1);

menuItem.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

textArea\_1.append("用户点击了新建的操作");

// 可以执行相关的代码

}

});

menuItem\_3.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

System.*exit*(0);

}

});

}

}

### 右键菜单

**package** com.guigu.menu;

**import** java.awt.BorderLayout;

**import** java.awt.EventQueue;

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**import** javax.swing.border.EmptyBorder;

**import** javax.swing.JTextArea;

**import** javax.swing.JPopupMenu;

**import** java.awt.Component;

**import** java.awt.event.MouseAdapter;

**import** java.awt.event.MouseEvent;

**import** javax.swing.JMenu;

**import** javax.swing.JMenuItem;

**public** **class** PopMenu **extends** JFrame {

**private** JPanel contentPane;

/\*\*

\* Launch the application.

\*/

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

EventQueue.*invokeLater*(**new** Runnable() {

**public** **void** run() {

**try** {

PopMenu frame = **new** PopMenu();

frame.setVisible(**true**);

} **catch** (Exception e) {

e.printStackTrace();

}

}

});

}

/\*\*

\* Create the frame.

\*/

**public** PopMenu() {

setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

setBounds(100, 100, 487, 365);

contentPane = **new** JPanel();

contentPane.setBorder(**new** EmptyBorder(5, 5, 5, 5));

setContentPane(contentPane);

JPopupMenu popupMenu = **new** JPopupMenu();

*addPopup*(contentPane, popupMenu);

JMenu menu = **new** JMenu("\u6D4B\u8BD5");

popupMenu.add(menu);

JMenuItem menuItem = **new** JMenuItem("\u6D4B\u8BD51");

menu.add(menuItem);

JMenuItem menuItem\_1 = **new** JMenuItem("\u6D4B\u8BD52");

menu.add(menuItem\_1);

JMenu menu\_1 = **new** JMenu("\u526A\u5207");

popupMenu.add(menu\_1);

JMenuItem menuItem\_2 = **new** JMenuItem("\u4ECE\u6587\u4EF6\u526A\u5207");

menu\_1.add(menuItem\_2);

JMenuItem menuItem\_3 = **new** JMenuItem("\u4ECE\u7F51\u7EDC\u526A\u5207");

menu\_1.add(menuItem\_3);

JMenuItem menuItem\_4 = **new** JMenuItem("\u590D\u5236");

popupMenu.add(menuItem\_4);

JMenuItem menuItem\_5 = **new** JMenuItem("\u62F7\u8D1D");

popupMenu.add(menuItem\_5);

contentPane.setLayout(**null**);

JTextArea textArea = **new** JTextArea();

textArea.setColumns(30);

textArea.setRows(50);

textArea.setBounds(10, 24, 451, 183);

contentPane.add(textArea);

}

**private** **static** **void** addPopup(Component component, **final** JPopupMenu popup) {

component.addMouseListener(**new** MouseAdapter() {

**public** **void** mousePressed(MouseEvent e) {

**if** (e.isPopupTrigger()) {

showMenu(e);

}

}

**public** **void** mouseReleased(MouseEvent e) {

**if** (e.isPopupTrigger()) {

showMenu(e);

}

}

**private** **void** showMenu(MouseEvent e) {

popup.show(e.getComponent(), e.getX(), e.getY());

}

});

}

}

## 位图和绘图

绘图是开发游戏和一些复杂页面的基础。

既然是绘图需要准备哪些？

1. 用什么样的笔去画图，这只笔从哪里来？
2. 在什么位置画图， 画图板是用什么构成的。
3. 谁来调用画图的动作。

在Component中有三个与绘图相关的方法，

paint(Graphics g)

update(Graphics g)

repaint()

其实三个绘图方法是 repaint会调用update方法，而update方法调用paint方法。

1、画笔是谁？Graphics 有了这个类就可以画图

这个笔是从paint方法 或者update方法获取，

另外还有BufferedImage可以获取画笔。

2、继承了Jpanel或者Canvas这两个类都可以作为画图板

**package** com.guigu.gui;

**import** java.awt.Color;

**import** java.awt.Graphics;

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**public** **class** GraphTest {

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

**new** GraphTest();

}

**public** GraphTest() {

JFrame jf =**new** JFrame();

jf.setSize(600, 400);

MyDrawArea mda =**new** MyDrawArea();

jf.add(mda);

mda.repaint();

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

//1.得到画板

**class** MyDrawArea **extends** JPanel{

@Override

**public** **void** paint(Graphics g) {

g.setColor(Color.***red***);

//绘制一个矩形

g.drawRect(200, 300, 100, 100);

g.fillOval(20, 30, 200, 300);

g.drawLine(300, 300, 500, 500);

g.drawLine(500, 500, 300, 400);

}

}

}

在随机的位置绘制矩形 随机绘制椭圆形

**package** com.guigu.gui;

**import** java.awt.BorderLayout;

**import** java.awt.Canvas;

**import** java.awt.Color;

**import** java.awt.Dimension;

**import** java.awt.Graphics;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** java.util.Random;

**import** javax.swing.JButton;

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**public** **class** GrahpTest2 {

**public** **final** String RECT\_SHAPE="rect";

**public** **final** String OVAL\_SHAPE="oval";

JFrame jf =**new** JFrame("简单绘图");

JButton rect =**new** JButton("绘制矩形");

JButton oval =**new** JButton("绘制椭圆形");

//定义变量 保存需要绘制什么样的图形字符串

**public** String shape="";

**public** MyCanvas mc =**new** MyCanvas();

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

**new** GrahpTest2();

}

**public** GrahpTest2() {

JPanel jp =**new** JPanel();

//分别为两个按钮添加监听事件

rect.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

//设置需要绘制的图形

shape=RECT\_SHAPE;

mc.repaint();

}

});

oval.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

shape=OVAL\_SHAPE;

mc.repaint();

}

});

jp.add(rect);

jp.add(oval);

mc.setPreferredSize(**new** Dimension(250, 200));

jf.add(mc);

jf.add(jp,BorderLayout.***SOUTH***);

jf.pack();

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

//定制绘图板

**class** MyCanvas **extends** Canvas{

@Override

**public** **void** paint(Graphics g) {

Random random =**new** Random();

**if**(shape.equals(RECT\_SHAPE)) {

//绘制矩形

g.setColor(Color.***GREEN***);

//随机的位置进行绘制

g.drawRect(random.nextInt(200), random.nextInt(180), 40, 60);

}

**if**(shape.equals(OVAL\_SHAPE)) {

//绘制矩形

g.setColor(Color.***RED***);

//随机的位置进行绘制

g.drawOval(random.nextInt(200), random.nextInt(180), 50, 40);

}

}

}

}

### 位图Image抽象类和ImageIO

**package** com.guigu.gui;

**import** java.awt.Graphics;

**import** java.awt.Image;

**import** java.io.File;

**import** java.io.IOException;

**import** javax.imageio.ImageIO;

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**public** **class** GrahpImage {

Image image = **null**;

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

**new** GrahpImage();

}

**public** GrahpImage() {

**try** {

image = ImageIO.*read*(**new** File("image/ww.jpg"));

} **catch** (IOException e) {

e.printStackTrace();

}

JFrame jf = **new** JFrame();

jf.setSize(800, 600);

**final** MyDrawArea mda = **new** MyDrawArea();

jf.add(mda);

mda.repaint();

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

**class** MyDrawArea **extends** JPanel {

@Override

**public** **void** paint(Graphics g) {

g.drawImage(image, 30, 40, **null**);

}

}

}

## Swing中的高级组件

### JtoolBar工具条

**package** com.guigu.swingad;

**import** java.awt.BorderLayout;

**import** java.awt.Insets;

**import** java.awt.Toolkit;

**import** java.awt.datatransfer.Clipboard;

**import** java.awt.datatransfer.DataFlavor;

**import** java.awt.datatransfer.StringSelection;

**import** java.awt.datatransfer.UnsupportedFlavorException;

**import** java.awt.event.ActionEvent;

**import** java.io.IOException;

**import** javax.swing.AbstractAction;

**import** javax.swing.Action;

**import** javax.swing.ImageIcon;

**import** javax.swing.JButton;

**import** javax.swing.JFrame;

**import** javax.swing.JMenu;

**import** javax.swing.JMenuBar;

**import** javax.swing.JPanel;

**import** javax.swing.JScrollPane;

**import** javax.swing.JTextArea;

**import** javax.swing.JToolBar;

**public** **class** JToolBarTest {

JFrame jf =**new** JFrame("测试工具条");

JTextArea jta =**new** JTextArea(6,35);

JToolBar jtb =**new** JToolBar();

JMenuBar jmb=**new** JMenuBar();

JMenu edit =**new** JMenu("编辑");

//得到剪切板的内容

Clipboard clipboard =Toolkit.*getDefaultToolkit*().getSystemClipboard();

//创建两个按钮 复制 和粘贴 并且为这两个按钮实现相关的功能

Action pastAction=**new** AbstractAction("粘贴",**new** ImageIcon("ico/past.jpg")) {

@Override

**public** **void** actionPerformed(ActionEvent e) {

//判断剪切板中是否有内容

**if**(clipboard.isDataFlavorAvailable(DataFlavor.***stringFlavor***)) {

//取出剪切板中的内容

**try** {

String content =(String) clipboard.getData(DataFlavor.***stringFlavor***);

jta.replaceRange(content, jta.getSelectionStart(), jta.getSelectionEnd());

//讲选中的内容替换为剪切板中的内容

}**catch** (Exception e1) {

e1.printStackTrace();

}

}

}

};

//实现复制功能

Action copyAction=**new** AbstractAction("复制",**new** ImageIcon("ico/cut.jpg")) {

@Override

**public** **void** actionPerformed(ActionEvent e) {

StringSelection contents =**new** StringSelection(jta.getSelectedText());

//讲StringSelection对象放入到剪切板即可

clipboard.setContents(contents, **null**);

//如果剪切板中存在内容 将剪切板激活

**if**(clipboard.isDataFlavorAvailable(DataFlavor.***stringFlavor***)) {

pastAction.setEnabled(**true**);

}

}

};

**public** JToolBarTest() {

//将粘贴功能设置不可用

pastAction.setEnabled(**false**);

jf.add(**new** JScrollPane(jta));

//创建两个按钮 并且添加功能

JButton copyBt =**new** JButton(copyAction);

JButton pastBt =**new** JButton(pastAction);

JPanel jp =**new** JPanel();

jp.add(copyBt);

jp.add(pastBt);

jf.add(jp,BorderLayout.***SOUTH***);

//工具条中也添加这个两个功能

jtb.add(copyAction);

jtb.add(pastAction);

//向菜单条中添加功能

edit.add(copyAction);

edit.add(pastAction);

jmb.add(edit);

jf.setJMenuBar(jmb);

//设置工具条之间按钮的间距

jtb.setMargin(**new** Insets(20, 10, 5, 30));

jf.add(jtb,BorderLayout.***NORTH***);

jf.setVisible(**true**);

jf.pack();

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

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

**new** JToolBarTest();

}

}

### JoptionPane

参数：

这些方法的参数遵守一致的模式：

parentComponent

定义作为此对话框的父对话框的 Component。通过两种方式使用此参数：包含它的 Frame 可以用作对话框的父 Frame，在对话框的位置使用其屏幕坐标。一般情况下，将对话框紧靠组件置于其之下。此参数可以为 null，在这种情况下，默认的 Frame 用作父级，并且对话框将居中位于屏幕上（取决于 L&F）。

message

要置于对话框中的描述消息。在最常见的应用中，message 就是一个 String 或 String 常量。不过，此参数的类型实际上是 Object。其解释依赖于其类型：

Object[]

对象数组被解释为在纵向堆栈中排列的一系列 message（每个对象一个）。解释是递归式的，即根据其类型解释数组中的每个对象。

Component

该 Component 在对话框中显示。

Icon

该 Icon 被包装在 JLabel 中并在对话框中显示。

其他

该对象通过调用其 toString 方法被转换为 String。结果被包装在 JLabel 中显示。

messageType

定义 message 的样式。外观管理器根据此值对对话框进行不同地布置，并且通常提供默认图标。可能的值为：

ERROR\_MESSAGE

INFORMATION\_MESSAGE

WARNING\_MESSAGE

QUESTION\_MESSAGE

PLAIN\_MESSAGE

optionType

定义在对话框的底部显示的选项按钮的集合：

DEFAULT\_OPTION

YES\_NO\_OPTION

YES\_NO\_CANCEL\_OPTION

OK\_CANCEL\_OPTION

用户并非仅限于使用此选项按钮集合。使用 options 参数可以提供想使用的任何按钮。

options

将在对话框底部显示的选项按钮集合的更详细描述。options 参数的常规值是 String 数组，但是参数类型是 Object 数组。根据对象的以下类型为每个对象创建一个按钮：

Component

该组件被直接添加到按钮行中。

Icon

创建的 JButton 以此图标作为其标签。

其他

该 Object 通过使用其 toString 方法转换为字符串，并使用该结果作为 JButton 的标签。

icon

要置于对话框中的装饰性图标。图标的默认值由 messageType 参数确定。

title

对话框的标题。

initialValue

默认选择（输入值）。

当选择更改时，调用生成 PropertyChangeEvent 的 setValue 方法。

如果已为所有输入 setWantsInput 配置了 JOptionPane，则还可以侦听绑定属性 JOptionPane.INPUT\_VALUE\_PROPERTY，以确定何时用户输入或选择了值。

当其中一个 showXxxDialog 方法返回整数时，可能的值为：

YES\_OPTION

NO\_OPTION

CANCEL\_OPTION

OK\_OPTION

CLOSED\_OPTION

**package** com.guigu.swingad;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** javax.swing.JButton;

**import** javax.swing.JFrame;

**import** javax.swing.JOptionPane;

**public** **class** JoptionPaneTest {

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

JFrame jf = **new** JFrame();

JButton jb = **new** JButton("ok");

jb.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

// JOptionPane.showMessageDialog(null, "我是提示的内容", "我是标题", JOptionPane.INFORMATION\_MESSAGE);

// int result=JOptionPane.showConfirmDialog(null, "你确认吃饭了吗", "吃了吗?", JOptionPane.YES\_NO\_CANCEL\_OPTION, JOptionPane.WARNING\_MESSAGE);

// System.out.println(result);

// Object [] options= {"哈哈","呵呵","嘿嘿","嘎嘎"};

// Object re=JOptionPane.showOptionDialog(null, "选择你的笑容", "笑一个", JOptionPane.DEFAULT\_OPTION, JOptionPane.INFORMATION\_MESSAGE, null, options, options[2]);

// System.out.println(re);

//

// String inputValue=JOptionPane.showInputDialog(null, "请输入你的姓名", "注册", JOptionPane.INFORMATION\_MESSAGE);

// System.out.println(inputValue);

Object [] possibleValues= {"博士","硕士","本科","专科","高中","初中","小学","幼儿园"};

Object selectValue=JOptionPane.*showInputDialog*(**null**, "选择现在的阶段", "测试", JOptionPane.***INFORMATION\_MESSAGE***, **null**, possibleValues, possibleValues[0]);

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

}

});

jf.add(jb);

jf.pack();

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

}

### JSplitPane

**package** com.guigu.swingad;

**import** java.awt.Dimension;

**import** javax.swing.JButton;

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**import** javax.swing.JScrollPane;

**import** javax.swing.JSplitPane;

**public** **class** JSplitPaneTest {

JPanel jp1 =**new** JPanel();

JPanel jp2 =**new** JPanel();

JPanel jp3 =**new** JPanel();

JFrame jf =**new** JFrame();

**public** JSplitPaneTest() {

//为三个JPanel设置合适的大小

jp1.setPreferredSize(**new** Dimension(150, 300));

jp2.setPreferredSize(**new** Dimension(300,150));

jp3.setPreferredSize(**new** Dimension(300,150));

//定义三个按钮

JButton jb1 =**new** JButton("1");

JButton jb2 =**new** JButton("2");

JButton jb3 =**new** JButton("3");

jp1.add(jb1);

jp2.add(jb2);

jp3.add(jb3);

//创建一个垂直分割的面板

JSplitPane left =**new** JSplitPane(JSplitPane.***VERTICAL\_SPLIT***,**true**,jp1,**new** JScrollPane(jp2));

left.setOneTouchExpandable(**true**);

left.resetToPreferredSizes();

JSplitPane context =**new** JSplitPane(JSplitPane.***HORIZONTAL\_SPLIT***, left,jp3);

jf.add(context);

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

jf.pack();

}

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

**new** JSplitPaneTest();

}

}

### JProgressbar进度条

**package** com.guigu.swingad;

**import** java.awt.FlowLayout;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** javax.swing.Box;

**import** javax.swing.BoxLayout;

**import** javax.swing.JCheckBox;

**import** javax.swing.JFrame;

**import** javax.swing.JProgressBar;

**public** **class** JProgressBarTest **extends** JFrame {

//创建一个进度条

JProgressBar bar =**new** JProgressBar(JProgressBar.***HORIZONTAL***);

JCheckBox interminate =**new** JCheckBox("不确定的进度条");

JCheckBox noBorder =**new** JCheckBox("不绘制边框");

**public** JProgressBarTest() {

Box box =**new** Box(BoxLayout.***X\_AXIS***);

box.add(interminate);

box.add(noBorder);

//设置布局管理器

**this**.setLayout(**new** FlowLayout());

**this**.add(box);

**this**.add(bar);

//设置进度条的初始值和最大值

bar.setMinimum(0);

bar.setMaximum(100);

bar.setStringPainted(**true**);

noBorder.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent arg0) {

bar.setBorderPainted(!noBorder.isSelected());

}

});

interminate.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

bar.setIndeterminate(interminate.isSelected());

bar.setStringPainted(!interminate.isSelected());

}

});

**this**.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

**this**.pack();

**this**.setLocation(100, 100);

**this**.setVisible(**true**);

**for**(**int** i=0;i<100;i++) {

bar.setValue(i+1);

//下面的代码将暂停100毫秒 模拟操作其他内容

**try** {

Thread.*sleep*(100);

} **catch** (InterruptedException e1) {

e1.printStackTrace();

}

}

}

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

**new** JProgressBarTest();

}

}

### Jtree

#### JTree的基础

**package** com.guigu.swingad;

**import** javax.swing.JFrame;

**import** javax.swing.JScrollPane;

**import** javax.swing.JTree;

**import** javax.swing.tree.DefaultMutableTreeNode;

**public** **class** JTreeTest {

JFrame jf =**new** JFrame();

JTree tree ;

//定义几个初始化的节点

DefaultMutableTreeNode root =**new** DefaultMutableTreeNode("中国");

DefaultMutableTreeNode zhejiang =**new** DefaultMutableTreeNode("浙江");

DefaultMutableTreeNode shandong =**new** DefaultMutableTreeNode("山东");

DefaultMutableTreeNode hangzhou =**new** DefaultMutableTreeNode("杭州");

DefaultMutableTreeNode wenzhou =**new** DefaultMutableTreeNode("温州");

DefaultMutableTreeNode ningbo =**new** DefaultMutableTreeNode("宁波");

DefaultMutableTreeNode jining =**new** DefaultMutableTreeNode("济宁");

DefaultMutableTreeNode jinan =**new** DefaultMutableTreeNode("济南");

DefaultMutableTreeNode qingdao =**new** DefaultMutableTreeNode("青岛");

**public** JTreeTest() {

zhejiang.add(hangzhou);

zhejiang.add(wenzhou);

zhejiang.add(ningbo);

shandong.add(jinan);

shandong.add(qingdao);

shandong.add(jining);

root.add(zhejiang);

root.add(shandong);

tree=**new** JTree(root);

//设置取消节点之间的连线

tree.putClientProperty("JTree.lineStyle","None");

//取消显示根节点

tree.setRootVisible(**false**);

jf.add(**new** JScrollPane(tree));

jf.pack();

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

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

**new** JTreeTest();

}

}

#### Jtree的监听

**package** com.guigu.swingad;

**import** javax.swing.Box;

**import** javax.swing.BoxLayout;

**import** javax.swing.JFrame;

**import** javax.swing.JScrollPane;

**import** javax.swing.JTextArea;

**import** javax.swing.JTree;

**import** javax.swing.event.TreeSelectionEvent;

**import** javax.swing.event.TreeSelectionListener;

**import** javax.swing.tree.DefaultMutableTreeNode;

**import** javax.swing.tree.TreeSelectionModel;

**public** **class** SelectTree {

JFrame jf = **new** JFrame("监听树的节点");

JTree tree;

JTextArea contentText = **new** JTextArea(6, 30);

// 定义几个初始化的节点

DefaultMutableTreeNode root = **new** DefaultMutableTreeNode("中国");

DefaultMutableTreeNode zhejiang = **new** DefaultMutableTreeNode("浙江");

DefaultMutableTreeNode shandong = **new** DefaultMutableTreeNode("山东");

DefaultMutableTreeNode hangzhou = **new** DefaultMutableTreeNode("杭州");

DefaultMutableTreeNode wenzhou = **new** DefaultMutableTreeNode("温州");

DefaultMutableTreeNode ningbo = **new** DefaultMutableTreeNode("宁波");

DefaultMutableTreeNode jining = **new** DefaultMutableTreeNode("济宁");

DefaultMutableTreeNode jinan = **new** DefaultMutableTreeNode("济南");

DefaultMutableTreeNode qingdao = **new** DefaultMutableTreeNode("青岛");

**public** SelectTree() {

zhejiang.add(hangzhou);

zhejiang.add(wenzhou);

zhejiang.add(ningbo);

shandong.add(jinan);

shandong.add(qingdao);

shandong.add(jining);

root.add(zhejiang);

root.add(shandong);

tree = **new** JTree(root);

// 设置取消节点之间的连线

// 设置一次只能选中一个TreePath

tree.getSelectionModel().setSelectionMode(TreeSelectionModel.***SINGLE\_TREE\_SELECTION***);

// 对tree添加

tree.addTreeSelectionListener(**new** TreeSelectionListener() {

@Override

**public** **void** valueChanged(TreeSelectionEvent e) {

**if** (e.getOldLeadSelectionPath() != **null**) {

contentText.append("原来选中的节点路径是:" + e.getOldLeadSelectionPath().toString() + "\r\n");

contentText.append("新选中的路径信息是:" + e.getNewLeadSelectionPath().toString() + "\r\n");

}

}

});

tree.setRootVisible(**true**);

Box box = **new** Box(BoxLayout.***X\_AXIS***);

box.add(**new** JScrollPane(tree));

box.add(**new** JScrollPane(contentText));

jf.add(box);

jf.pack();

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

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

**new** SelectTree();

}

}

#### 改变树的外观

**package** com.guigu.swingad;

**import** java.awt.Color;

**import** javax.swing.ImageIcon;

**import** javax.swing.JFrame;

**import** javax.swing.JScrollPane;

**import** javax.swing.JTree;

**import** javax.swing.tree.DefaultMutableTreeNode;

**import** javax.swing.tree.DefaultTreeCellRenderer;

**public** **class** ChangeAllCellRender {

JFrame jf = **new** JFrame();

JTree tree;

// 定义几个初始化的节点

DefaultMutableTreeNode root = **new** DefaultMutableTreeNode("中国");

DefaultMutableTreeNode zhejiang = **new** DefaultMutableTreeNode("浙江");

DefaultMutableTreeNode shandong = **new** DefaultMutableTreeNode("山东");

DefaultMutableTreeNode hangzhou = **new** DefaultMutableTreeNode("杭州");

DefaultMutableTreeNode wenzhou = **new** DefaultMutableTreeNode("温州");

DefaultMutableTreeNode ningbo = **new** DefaultMutableTreeNode("宁波");

DefaultMutableTreeNode jining = **new** DefaultMutableTreeNode("济宁");

DefaultMutableTreeNode jinan = **new** DefaultMutableTreeNode("济南");

DefaultMutableTreeNode qingdao = **new** DefaultMutableTreeNode("青岛");

**public** ChangeAllCellRender() {

zhejiang.add(hangzhou);

zhejiang.add(wenzhou);

zhejiang.add(ningbo);

shandong.add(jinan);

shandong.add(qingdao);

shandong.add(jining);

root.add(zhejiang);

root.add(shandong);

tree = **new** JTree(root);

// 改变树的外观是通过DefaultTreeCellRender的类

DefaultTreeCellRenderer cellRender = **new** DefaultTreeCellRenderer();

// 设置节点的背景色

cellRender.setBackgroundSelectionColor(**new** Color(150, 150, 150));

cellRender.setBackgroundNonSelectionColor(**new** Color(220, 220, 220));

//设置图标 处于折叠状态下的图标 非节点

cellRender.setClosedIcon(**new** ImageIcon("ico/close.png"));

//设置展开的图标

cellRender.setOpenIcon(**new** ImageIcon("ico/open.png"));

//设置节点的图标

cellRender.setLeafIcon(**new** ImageIcon("ico/leaf.png"));

// 让树节点应用我们改变的样式

tree.setCellRenderer(cellRender);

tree.setRootVisible(**true**);

jf.add(**new** JScrollPane(tree));

jf.pack();

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

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

**new** ChangeAllCellRender();

}

}

### JTable

#### 简单的表格

**package** com.guigu.swingad;

**import** javax.swing.JFrame;

**import** javax.swing.JScrollPane;

**import** javax.swing.JTable;

**public** **class** JTableTest {

JFrame jf = **new** JFrame("简单表格");

// 定义二维数组作为表格数据

Object[][] tableData = { **new** Object[] { "张三", 30, "女" }, **new** Object[] { "李四", 40, "男" },

**new** Object[] { "王五", 28, "男" } };

// 定义一维数组 作为标题

Object[] coumTitle = { "姓名", "年龄", "性别" };

JTable table = **new** JTable();

**public** JTableTest() {

table = **new** JTable(tableData, coumTitle);

jf.add(**new** JScrollPane(table));

jf.pack();

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

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

**new** JTableTest();

}

}

#### TableModel TableColumModel 监听器

**package** com.guigu.swingad;

**import** javax.swing.JFrame;

**import** javax.swing.JScrollPane;

**import** javax.swing.JTable;

**import** javax.swing.event.TableModelEvent;

**import** javax.swing.event.TableModelListener;

**import** javax.swing.table.AbstractTableModel;

**public** **class** SimpleJTable {

JFrame jf = **new** JFrame("简单表格");

JTable table;

MyTableModel model;

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

**new** SimpleJTable();

}

**public** SimpleJTable() {

model = **new** MyTableModel();

// model.addTableModelListener(new TableModelListener() {

//

// @Override

// public void tableChanged(TableModelEvent e) {

// int row = e.getFirstRow();

// int col = e.getColumn();

// System.out.println(row);

// System.out.println(col);

// System.out.println(model.getValueAt(row, col));

// }

// });

table = **new** JTable(model);

// table.setEnabled(true);

jf.add(**new** JScrollPane(table));

jf.pack();

jf.setVisible(**true**);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

}

**class** MyTableModel **extends** AbstractTableModel {

// 定义二维数组作为表格数据

Object[][] tableData = { **new** Object[] { "张三", 30, "女" }, **new** Object[] { "李四", 40, "男" },

**new** Object[] { "王五", 28, "男" } };

// 定义一维数组 作为标题

Object[] coumTitle = { "姓名", "年龄", "性别" };

@Override

**public** **int** getRowCount() {

**return** tableData.length;

}

@Override

**public** **int** getColumnCount() {

**return** coumTitle.length;

}

@Override

**public** Object getValueAt(**int** rowIndex, **int** columnIndex) {

**return** tableData[rowIndex][columnIndex];

}

@Override

**public** String getColumnName(**int** column) {

**return** (String) coumTitle[column];

}

@Override

**public** **boolean** isCellEditable(**int** rowIndex, **int** columnIndex) {

**return** **true**;

}

@Override

**public** **void** setValueAt(Object aValue, **int** rowIndex, **int** columnIndex) {

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

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

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

tableData[rowIndex][columnIndex] = aValue;

// 更新表格

fireTableCellUpdated(rowIndex, columnIndex);

}

}

}

# 多线程

## 线程的概述

Java语音提供了非常优秀的多线程的支持，程序可以通过非常简单的方式启动多线程，多线程包括创建，启动线程，控制线程，以及多线程的同步。

### 线程的和进程

几乎所有的操作系统都支持进程 的概念，所有运行的任务通常对应一个进程。当一个程序进入到内存中 即变为一个进程。进程是处于运行中的程序，并且具有一定的独立性。进程是包含线程的，也就是一个进程可以包含多个线程。

进程具有如下的特性:

* 独立性： 进程是系统中独立存在的实体，它可以拥有自己独立的资源，每一个进程都有自己的私有空间。没有经过进程本身的允许其他进程是不可以访问当前进程的地址。
* 动态性：进程与线程最大的区别是程序是一个动态的指令集合，而进程是一个正在系统中活动的指令的集合。在进程中加入了时间的概念。
* 并发性：多个进程可以在单个处理器上并发执行。多个进程之间不会相关影响

### 多线程的优势

线程是程序中独立的，并发的执行流。 线程是进程的划分的最小单位，线程中是可以共享内存，文件语句，每个进程都有对应的状态。

## 线程的创建

**package** com.guigu.thread;

**public** **class** ThreadTest1 **extends** Thread {

@Override

**public** **void** run() {

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

System.***out***.println("我是线程一......");

}

}

}

**package** com.guigu.thread;

**public** **class** ThreadTest2 **implements** Runnable{

@Override

**public** **void** run() {

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

System.***out***.println("我是线程2....");

}

}

}

**package** com.guigu.thread;

**public** **class** Go {

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

ThreadTest1 t1 = **new** ThreadTest1();

ThreadTest2 t2 = **new** ThreadTest2();

// 让线程运行

t1.start();

// 如果使用实现接口的方式 实现线程 需要再次包装

Thread t3 = **new** Thread(**new** ThreadTest2());

t3.start();

// 在当前程序中在创建一个线程

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

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

System.***out***.println("我是主线程3........");

}

}

}).start();

}

}

### 两种方式创建线程的对比

通过继承Thread类 以及实现Runnable接口两种方式实现多线程。 这两种方式有如下区别：

采用实现Runnable接口的方式:

* 线程只是实现了Runnable接口 还可以继承其他的类
* 在这种方式下可以多个线程共享一个target对象，适合多个相同的线程处理同一份资源，从而将cpu，代码和数据进行分开，形成一个清晰的模型关系，较好的体现了面向对象的思想。

劣势： 编程稍微复杂一些，如果需要访问当前线程必须使用Thrad.currentThread（）这个方法

采用继承Thread类：

劣势：因为继承了Thread类 无法再继承其他的类

优势： 编写程序相对于简单， 如果需要访问当前线程，可以直接使用this获取当前线程。

## 线程的生命周期

当线程被创建并且启动以后，它经历了五种状态，新建，就绪，运行，阻塞，死亡、当线程在运行的时候不能一直占有cpu的执行权，cpu会在对个线程之间进行调度，线程也会多次进行阻塞和运行状态的切换。

* 当线程对象被创建出来就进入到了新建状态，和其他的java对象一样，仅仅是由java虚拟机为其分配内存。
* 当线程调用start方法以后，线程进入到就绪状态，其实内部还是依赖jvm的调度，当jvm执行这个线程那么才进入到运行状态，否则就是就绪状态。
* 如果线程就绪状态获得了cpu的执行权，那么线程就进入到运行状态，这个线程不会一直霸占cpu的执行权，在执行的过程中会被jvm从cpu调度下来，以便其他的线程执行，从cpu上调度下来就进入到阻塞状态。
* 线程进入阻塞状态的情况：
  + 线程调用一个阻塞的方法，该方法返回之前该线程一直阻塞。
  + 线程调用sleep方法进入阻塞
  + 线程尝试获取同步监视器(锁)，但是该同步监视器被其他线程所持有
  + 线程调用了suspend方法挂起线程。
* 线程由阻塞状态进入到运行状态的的情况：
* 调用的阻塞方法返回
* 调用的sleep方法到期
* 线程成功获取同步监视器
* 被suspend的方法 调用resume方法恢复
* 死亡

线程在以下三种方式 线程就会进入到死亡状态

* Run方法执行完毕，线程正常结束
* 线程抛出异常
* 直接调用一个stop方法结束线程

## 线程的控制

### Join线程

**package** com.guigu.thread2;

**public** **class** JoinTest {

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

**new** JoinTest().init();

}

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

MainThread mt = **new** MainThread();

mt.setName("主线程");

mt.start();

}

**class** MainThread **extends** Thread {

@Override

**public** **void** run() {

**for** (**int** i = 0; i < 50; i++) {

**if** (i == 30) {// 执行另外一个线程 打印另外一个线程的1-50

JoinThread jt = **new** JoinThread();

jt.setName("加塞进程...");

jt.start();

**try** {

jt.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

System.***out***.println(**this**.getName() + "," + i);

}

}

}

**class** JoinThread **extends** Thread {

@Override

**public** **void** run() {

**for** (**int** i = 0; i < 50; i++) {

System.***out***.println(**this**.getName() + "," + i);

}

}

}

}

### 线程睡眠

可以使用Thread类的sleep方法 可以让线程休眠一下。

**package** com.guigu.thread2;

**public** **class** ThreadSleep {

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

**for**(**int** i=0;i<100;i++) {

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

//让当前程序每隔1秒执行一次

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

### 线程的让步

**package** com.guigu.thread2;

**public** **class** ThreadYield **extends** Thread {

**public** ThreadYield() {

}

**public** ThreadYield(String msg) {

**super**(msg);

}

@Override

**public** **void** run() {

**for** (**int** i = 0; i < 50; i++) {

System.***out***.println(**this**.getName() + "," + i);

**if** (i == 20) {

Thread.*yield*();// 线程的让步 从cpu调度下来 两者共同再次争抢cpu的执行权

}

}

}

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

// 启动两条线程

ThreadYield ty1 = **new** ThreadYield("高级线程");

//可以设置优先级 但是这个优先级 在Windows下有时是不起作用的

// ty1.setPriority(Thread.MAX\_PRIORITY);

ty1.setPriority(5);

ty1.start();

ThreadYield ty2 = **new** ThreadYield("低级线程");

// ty2.setPriority(Thread.MIN\_PRIORITY);

ty2.setPriority(4);

ty2.start();

}

}

## 线程的安全

### 同步代码块

线程安全：<<Java并发编程实践>>作者定义了如下内容： 多个线程访问一个类的对象，如果不考虑这些线程在运行环境下的调度和交替执行， 并且不需要额外同步的以及调用方法 代码也不需要额外的协调，那么这个类的行为仍然是正确的，那么这类就是线程的安全类。

**package** com.guigu.safethread;

**public** **class** SafeTheadTest {

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

**new** SafeTheadTest();

}

PrintString ps = **new** PrintString();

**public** SafeTheadTest() {

// 线程一

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

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

ps.showString("AAAAAAAAAAAAAAA");

}

}

}).start();

// 线程二

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

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

ps.showString("BBBBBBBBBBBBBBBBB");

}

}

}).start();

}

}

**class** PrintString {

**public** **void** showString(String str) {

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

System.***out***.print(str.charAt(i));

}

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

}

}

**synchronized**

同步代码块 就是定义了一组原子性的操作 所谓的原子性的操作，就是说只允许一个线程进入。直到这个线程运行完毕，下一个线程才可以进入，同步代码块的格式如下

**Synchronized(this) { }**

**package** com.guigu.safethread;

**public** **class** SafeTheadTest {

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

**new** SafeTheadTest();

}

PrintString ps = **new** PrintString();

**public** SafeTheadTest() {

// 线程一

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

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

ps.showString("AAAAAAAAAAAAAAA");

}

}

}).start();

// 线程二

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

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

ps.showString("BBBBBBBBBBBBBBBBB");

}

}

}).start();

}

}

**class** PrintString {

// synchronized是一把锁 这个锁 锁住的方法 一次只允许一个线程进入到这个方法体内

// public synchronized void showString(String str) {

// for (int i = 0; i < str.length(); i++) {

// System.out.print(str.charAt(i));

// }

// System.out.println();

// }

//方式二 锁住 要执行的代码

**public** **void** showString(String str) {

**synchronized** (**this**) {

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

System.***out***.print(str.charAt(i));

}

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

}

}

}

**public** **void** showString(String str) {

**synchronized** (PrintString.**class**) { //锁住类的字节码 或者锁住 this代指当前 对象 或者str

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

System.***out***.print(str.charAt(i));

}

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

}

}

**同时开启两个线程 ，每个线程自加到1000 两个线程求和 应该是2000**

**package** com.guigu.safethread;

**public** **class** SafeThreadTest2 {

**public** **int** i = 0;

AddTest at = **new** AddTest();

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

**new** SafeThreadTest2();

}

**public** SafeThreadTest2() {

// 线程1 自加10000

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**for** (**int** i = 0; i < 10000; i++) {

at.add();

}

}

}).start();

// 线程2 自加10000

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**for** (**int** i = 0; i < 10000; i++) {

at.add();

}

}

}).start();

//等待上方线程运行的结束

// System.out.println(Thread.activeCount()); //得到当前线程的活跃的数量

//在上方两个线程没有执行结束 一直阻塞

**while**(Thread.*activeCount*()!=1) {

}

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

}

**class** AddTest {

**public** **synchronized** **void** add() {

i++;

}

}

}

### 线程安全的应用

**package** com.guigu.safethread;

**public** **class** Account {

**private** String cardNo;

**private** **double** money;

**public** Account(String cardNo, **double** money) {

**super**();

**this**.cardNo = cardNo;

**this**.money = money;

}

**public** String getCardNo() {

**return** cardNo;

}

**public** **void** setCardNo(String cardNo) {

**this**.cardNo = cardNo;

}

**public** **double** getMoney() {

**return** money;

}

**public** **void** setMoney(**double** money) {

**this**.money = money;

}

}

**package** com.guigu.safethread;

**public** **class** ATM **extends** Thread {

// 模拟账户

**private** Account account;

// 模拟取钱

**private** **double** caMoney;

**public** ATM(String name, Account account, **double** caMoney) {

**super**(name);

**this**.account = account;

**this**.caMoney = caMoney;

}

@Override

**public** **void** run() {

**synchronized** (account) {

// 如果账户余额大于要取钱的数目

**if** (account.getMoney() > caMoney) {

System.***out***.println(**this**.getName() + "取钱成功,共取钱" + caMoney);

//

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

// 设置账户的余额

account.setMoney(account.getMoney() - caMoney);

// 检测账户的余额

System.***out***.println("余额是:" + account.getMoney());

} **else** {

System.***out***.println(**this**.getName() + "账户余额不足...");

}

}

}

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

Account a = **new** Account("88888888888", 6000);

**new** ATM("张三", a, 5000).start();

**new** ATM("李四", a, 5000).start();

}

}

## Wait notify

当线程A运行的过程中遇到不满足的条件需要等待，等待线程B去更改系统的状态，当线程B更改了系统的状态，再次唤醒线程A，线程A查看是否满足条件，如果满足条件则继续执行，如果不满足条件则进入等待状态。

面试题 ：wait 和sleep有什么区别？

1. Sleep必须指定时间，wait可以指定 也可以不指定
2. Sleep和wait都是让线程处于休眠状态，释放执行权
3. 持有锁的线程执行sleep是不释放锁， 持久锁的线程执行wait是释放锁
4. Sleep时间到了自动苏醒，wait 只能被notify进行唤醒

**package** com.guigu.waitnotify;

**public** **class** ThreadWait {

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

**new** ThreadWait();

}

PrintString ps = **new** PrintString();

**public** ThreadWait() {

// 线程1 调用f1方法

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

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

ps.f1();

}

}

}).start();

// 线程2 调用f2方法

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

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

ps.f2();

}

}

}).start();

}

**class** PrintString {

**boolean** flag = **true**;

**public** **synchronized** **void** f1() {

**while** (flag) {

**try** {

wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

System.***out***.println("我是线程AAAAAAAAAAAAAAAAAAAAAAA");

flag = **true**;

notifyAll();// 唤醒所有的线程

}

**public** **synchronized** **void** f2() {

**while** (!flag) {

**try** {

wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

System.***out***.println("我是线程BBBBBBBBBBBBBBBBBBBBBBB");

flag = **false**;

notifyAll();

}

}

}

课堂练习 三个线程同时执行 ，要求顺序打印ABC

**package** com.guigu.waitnotify;

**public** **class** ThreadWait2 {

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

**new** ThreadWait2();

}

PrintString ps = **new** PrintString();

**public** ThreadWait2() {

// 线程1 调用f1方法

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

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

ps.f1();

}

}

}).start();

// 线程2 调用f2方法

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

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

ps.f2();

}

}

}).start();

// 线程2 调用f2方法

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

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

ps.f3();

}

}

}).start();

}

**class** PrintString {

**public** **int** i = 1;

**public** **synchronized** **void** f1() {

**while** (i != 1) {

**try** {

wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

System.***out***.println("我是线程AAAAAAAAAAAAAAAAAAAAAAA");

i = 2;

notifyAll();// 唤醒所有的线程

}

**public** **synchronized** **void** f2() {

**while** (i != 2) {

**try** {

wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

System.***out***.println("我是线程BBBBBBBBBBBBBBBBBBBBBBB");

i = 3;

notifyAll();

}

**public** **synchronized** **void** f3() {

**while** (i != 3) {

**try** {

wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

System.***out***.println("我是线程CCCCCCCCCCCCCCCCCCCCCCCCCC");

i = 1;

notifyAll();

}

}

}

### 生产者和消费者

**package** com.guigu.waitnotify2;

**public** **class** Product {

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

**private** String name;

**public** Product(**int** id, String name) {

**super**();

**this**.id = id;

**this**.name = name;

}

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

**return** id;

}

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

**this**.id = id;

}

**public** String getName() {

**return** name;

}

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

**this**.name = name;

}

}

**package** com.guigu.waitnotify2;

**import** java.util.LinkedList;

**public** **class** Table {

// 定义容器存储数据

LinkedList<Product> list = **new** LinkedList<>();

// 定义桌子的容量

**public** **int** max = 10;

/\*\*

\* 存储

\*

\* **@param** p

\*/

**public** **synchronized** **void** put(Product p) {

// 在存储之前先判断是否达到最大容量

**while** (size() == max) {

**try** {

wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

list.add(p);

notifyAll();

}

/\*\*

\* 取产品

\* **@return**

\*/

**public** **synchronized** Product take() {

**while** (!(size() > 0)) {

**try** {

wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

Product p = list.removeLast();

notifyAll();

**return** p;

}

/\*

\* 用于检测桌上的容量

\*/

**public** **int** size() {

**return** list.size();

}

**public** **void** checkTable() {

System.***err***.println("当前桌上的容量是:"+size());

}

}

**package** com.guigu.waitnotify2;

/\*\*

\*

\*生产者

\*/

**public** **class** ProducerThread **extends** Thread {

**private** Table table;

**public** ProducerThread(Table table) {

**super**();

**this**.table = table;

}

@Override

**public** **void** run() {

**int** i = 0;

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

**if**( i==Integer.***MAX\_VALUE***) {

i=0;

}

++i;

Product p =**new** Product(i, "产品:"+i);

table.put(p);

System.***out***.println("生产者创建了一个产品,放到桌子上");

**try** {

Thread.*sleep*(200);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

**package** com.guigu.waitnotify2;

/\*\*

\* 消费者

\*

\*/

**public** **class** CustomerThread **extends** Thread{

**private** Table table;

**public** CustomerThread(Table table) {

**super**();

**this**.table = table;

}

@Override

**public** **void** run() {

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

Product p =table.take();

System.***out***.println("消费者消费了一个产品,产品的ID是:"+p.getId()+"产品的名字是:"+p.getName());

**try** {

Thread.*sleep*(200);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

**package** com.guigu.waitnotify2;

**public** **class** Go {

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

// 必须保证是同一个桌子

Table table = **new** Table();

// 创建五个生产者

**for** (**int** i = 0; i < 5; i++) {

ProducerThread pt = **new** ProducerThread(table);

pt.start();

}

// 创建三个消费者

**for** (**int** i = 0; i < 3; i++) {

CustomerThread ct = **new** CustomerThread(table);

ct.start();

}

// 在创建一个线程用于检测桌子上的容量

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

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

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

table.checkTable();

}

}

}).start();

}

}

## 死锁

所谓死锁：就是指两个或者两个以上的线程在执行的过程中，因为争抢资源，而造成相关等待的现象，如果没有外力的作用，他们将无法继续执行， 此时系统成为死锁状态或者系统。

为什么会产生死锁：

* 因为系统资源不足
* 进程运行推进的顺序不正确
* 资源分配不当

产生死锁的条件有四个：

* 互斥条件 ：所谓互斥就是进程在某一段时间内独占资源。
* 请求与保持条件：一个进程因为请求资源而阻塞时，对已获得的资源保持不释放。
* 不剥夺条件：进程已经获得的资源，在未使用完成之前，不能强行剥夺。
* 循环等待条件： 若干进程 形成一种头尾项链的循环等待资源 的关系

避免死锁：

上述的四个条件 只要破坏其中一个条件即可避免死锁的产生。

**package** com.guigu.deaththread;

**public** **class** DeathThread **implements** Runnable {

**private** **boolean** flag;

**public** DeathThread(**boolean** flag) {

**this**.flag = flag;

}

@Override

**public** **void** run() {

**if** (flag) {

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

**synchronized** (MyLock.***LOCKA***) {

System.***out***.println("lock a haha ");

**synchronized** (MyLock.***LOCKB***) {

System.***out***.println("lock b hehe");

}

}

}

} **else** {

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

**synchronized** (MyLock.***LOCKB***) {

System.***out***.println("lock b haha ");

**synchronized** (MyLock.***LOCKA***) {

System.***out***.println("lock a hehe");

}

}

}

}

}

}

**class** MyLock {

**public** **static** **final** Object ***LOCKA*** = **new** Object();

**public** **static** **final** Object ***LOCKB*** = **new** Object();

}

**package** com.guigu.deaththread;

**public** **class** Go {

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

DeathThread d1 =**new** DeathThread(**true**);

DeathThread d2 =**new** DeathThread(**false**);

Thread t1 =**new** Thread(d1);

Thread t2 =**new** Thread(d2);

t1.start();

t2.start();

}

}