技 术 文 件

技术文件名称：**Java中的基本技术问题**

技术文件编号：

版 本：V1.1

文件质量等级：A级

共 页

(包括封面)

拟 制 张 腾

审 核

会 签

标准化

批 准

深圳市中兴通讯股份有限公司

**目 录**

[1 背景 6](#_Toc71536430)

[2 系统总体介绍 6](#_Toc71536431)

[2.1 应用范围 6](#_Toc71536432)

[2.2 技术参考（文献及资料） 6](#_Toc71536433)

[3 基本概念问题 6](#_Toc71536434)

[3.1 Java开发概念 6](#_Toc71536435)

[3.1.1 开发工具 6](#_Toc71536436)

[3.1.2 JDK的基本使用 (java development kit) 6](#_Toc71536437)

[3.1.3 Java文件类型 8](#_Toc71536438)

[3.2 Java工程结构 8](#_Toc71536439)

[3.2.1 应用程序（application） 8](#_Toc71536440)

[3.2.2 不具备main函数的小应用程序（applet） 10](#_Toc71536441)

[3.2.3 有main函数的applet 11](#_Toc71536442)

[3.3 Java语言基础知识 11](#_Toc71536443)

[3.3.1 类 11](#_Toc71536444)

[3.3.2 对象 39](#_Toc71536445)

[3.3.3 数据类型 43](#_Toc71536446)

[3.3.4 流程控制、运算符和Java赋值操作 47](#_Toc71536447)

[3.3.5 断言 55](#_Toc71536448)

[3.3.6 接口 Interface 55](#_Toc71536449)

[3.3.7 包 59](#_Toc71536450)

[3.4 Java JDK package 60](#_Toc71536451)

[3.4.1 java.lang和java.util 60](#_Toc71536452)

[3.4.2 java.io 65](#_Toc71536453)

[3.4.3 java.awt：图形和组件 67](#_Toc71536454)

[3.4.4 java.awt：布局管理器 71](#_Toc71536455)

[3.4.5 java.awt：事件处理 73](#_Toc71536456)

[3.5 Exception异常处理（错误处理） 75](#_Toc71536457)

[3.5.1 创建并抛出自己的异常 77](#_Toc71536458)

[3.5.2 什么时候使用异常？ 77](#_Toc71536459)

[3.5.3 什么时候不要使用异常？ 78](#_Toc71536460)

[3.5.4 构造函数中抛出异常的处理方法 78](#_Toc71536461)

[3.5.5 常见的Java异常 79](#_Toc71536462)

[3.5.6 子类中覆盖方法的异常 80](#_Toc71536463)

[3.5.7 消失的异常 80](#_Toc71536464)

[3.5.8 异常的关键概念 82](#_Toc71536465)

[3.5.9 从堆栈中获取抛出的异常，重定位到字符串 82](#_Toc71536466)

[3.6 GC垃圾收集 83](#_Toc71536467)

[3.6.1 对象何时适合于垃圾收集 83](#_Toc71536468)

[3.6.2 结束方法（最终化） 83](#_Toc71536469)

[3.6.3 GC部分问题回答 84](#_Toc71536470)

[3.7 Java DOC文档 86](#_Toc71536471)

[3.7.1 Javadoc帮助 86](#_Toc71536472)

[3.7.2 Javadoc标签 86](#_Toc71536473)

[4 基本技术问题 87](#_Toc71536474)

[4.1 图形编程 87](#_Toc71536475)

[4.1.2 Swing特性 88](#_Toc71536476)

[4.2 数据库编程 88](#_Toc71536477)

[4.2.1 JDBC介绍(Java数据库互连) 88](#_Toc71536478)

[4.2.2 JDBC的四种类型 89](#_Toc71536479)

[4.2.3 JDBC类 89](#_Toc71536480)

[4.2.4 SQL数据库操作 89](#_Toc71536481)

[4.2.5 JDBC实例 89](#_Toc71536482)

[4.2.6 JDBC注意事项 90](#_Toc71536483)

[4.2.7 Java DataStore数据库 90](#_Toc71536484)

[5 专题技术问题 91](#_Toc71536485)

[5.1 RMI（Remote Method Invocation） 91](#_Toc71536486)

[5.1.1 远程接口（interface） 91](#_Toc71536487)

[5.1.2 远程接口实现类 91](#_Toc71536488)

[5.1.3 RMI服务器类 92](#_Toc71536489)

[5.1.4 RMI客户端类 93](#_Toc71536490)

[5.1.5 设置RMI安全管理器 94](#_Toc71536491)

[5.1.6 RMI程序的编译 95](#_Toc71536492)

[5.1.7 RMI程序的运行 95](#_Toc71536493)

[5.2 网络编程和通信 96](#_Toc71536494)

[5.2.1 网络编程基础 96](#_Toc71536495)

[5.2.2 网络包 96](#_Toc71536496)

[5.2.3 网络编程实例 97](#_Toc71536497)

[5.3 对象序列化（串行化） 97](#_Toc71536498)

[5.3.1 字节流、数据流 97](#_Toc71536499)

[5.3.2 字符流 98](#_Toc71536500)

[5.3.3 对象流 98](#_Toc71536501)

[5.3.4 注意事项 100](#_Toc71536502)

[5.4 Java多线程 100](#_Toc71536503)

[5.4.1 如何编写Java多线程程序 100](#_Toc71536504)

[5.4.2 线程间的同步 112](#_Toc71536505)

[5.4.3 线程的其它注意事项 116](#_Toc71536506)

[5.4.4 Java多线程的缺点 116](#_Toc71536507)

[5.4.5 线程Block或Stop的问题 116](#_Toc71536508)

[5.4.6 Java单线程实例 118](#_Toc71536509)

[5.4.7 Java多线程实例 118](#_Toc71536510)

[5.5 JNI与系统交互 120](#_Toc71536511)

[5.6 创建自己的package 120](#_Toc71536512)

[5.7 JavaBean 120](#_Toc71536513)

[6 高效编程指南 120](#_Toc71536514)

[6.1 创建及销毁对象 120](#_Toc71536515)

[6.1.1 考虑用静态工厂方法替代构造函数 121](#_Toc71536516)

[6.1.2 使用私有构造函数强化singleton属性 123](#_Toc71536517)

[6.1.3 用私有构造函数强化不可实例化的能力 126](#_Toc71536518)

[6.1.4 避免创建重复对象 127](#_Toc71536519)

[6.1.5 消除对过期对象的引用 129](#_Toc71536520)

[6.1.6 避免使用终结程序 131](#_Toc71536521)

[6.2 对象的通用方法 133](#_Toc71536522)

[6.2.1 重写equals时要遵守通用约定 133](#_Toc71536523)

[6.2.2 重写equals时永远要重写hashCode 139](#_Toc71536524)

[6.2.3 永远要重写toString 142](#_Toc71536525)

[6.2.4 谨慎地重写clone 143](#_Toc71536526)

[6.2.5 实现Comparable的问题 148](#_Toc71536527)

[6.3 类和接口 153](#_Toc71536528)

[6.3.1 最小化类和成员的可访问能力 153](#_Toc71536529)

[6.3.2 倾向于非可变性 155](#_Toc71536530)

[6.3.3 组合优于继承 159](#_Toc71536531)

[6.3.4 设计和文档化继承 160](#_Toc71536532)

[6.3.5 接口优于抽象类 160](#_Toc71536533)

[6.3.6 接口只能用来定义类型 160](#_Toc71536534)

[6.3.7 静态成员优于非静态的 160](#_Toc71536535)

[7 思考问题 160](#_Toc71536536)

[7.1 Java与C++有什么区别？ 160](#_Toc71536537)

[7.2 你在编程中出过什么样的错误？ 160](#_Toc71536538)

[7.3 你知道对象是如何在内存中组织的吗？Java真的不用管理内存分配吗？ 161](#_Toc71536539)

[7.4 有哪些方法可以优化Java程序的性能？如何定位性能瓶颈？ 161](#_Toc71536540)

[7.5 什么是内存泄漏？JVM一般在什么情况下开始垃圾回收？ 161](#_Toc71536541)

[7.6 Swing主要采用什么设计模式？存在哪些线程安全问题？ 161](#_Toc71536542)

[8 编码规则 162](#_Toc71536543)

[8.1 Java规则 162](#_Toc71536544)

[8.2 Java编程30条建议 162](#_Toc71536545)

[9 部分小知识 164](#_Toc71536546)

Java基本技术问题

# 背景

|  |  |  |
| --- | --- | --- |
| 版本号 | 完成时间 | 原因 |
| 1.0 | 2003/01/04 | 建立文档 |
| 1.1 | 2003/03/17 | 加入Java高效编程指南 |
| 1.1.02 | 2003/03/17 | 加入SCJP Roseanne Zhang |

**"If you give me a fish, You feed me for the day. If you teach me how to fish, you feed me for life."**

"**Don't believe everything you read.**"

**Some thing legal does not mean it is recommended.**

**Who say the life is not beautiful?**

**Follow the rule of KISS (Keep It Simple and Stupid/Straight forward), please!!**

# 系统总体介绍

## 应用范围

中兴CDMA事业部。

## 技术参考（文献及资料）

束闻、王国良、林勇民（译）；Java2编程21天自学，清华大学出版社，北京：2002.4

田丽辒、龚志翔、龚超（译）；Java2认证考试指南，机械工业出版社，北京：2002.3

闻山等（译），Joshua Bolch（著）；Java高效编程指南，机械工业出版社，北京：2002.1

CDMA事业部多种Java培训教材

# 基本概念问题

## Java开发概念

### 开发工具

源文件编辑：任何文本编辑器（UltraEdit）

编译器：JDK1.3、1.4或Jbuilder等

### JDK的基本使用 (java development kit)

JDK主要由两部分组成：运行环境（基本类库：rt.jar或classes.zip、dt.jar）和JVM (Java virtual machine虚拟机)：Java程序运行的基础。JRE（Java runtine environment）

Java JDK 环境配置（右键我的电脑🡪属性🡪高级🡪环境环境变量）：

Path = Jdk安装目录\bin //bin运行路径

Classpath = Jdk安装目录\jre\lib\rt.jar ; Jdk安装目录\lib\dt.jar //指定所有类库的路径

Java DOS窗口命令：

* 源文件的编译命令 javac

编译器javac将源文件\*.java转换为可执行文件\*.class。一个文件可能会产生多个.class文件，但每个.class文件都是有序号的，编译的过程中从main入口，看到那个类被实例化就产生相应的.class文件。

javac 类名.java （例如：javac Welcome.java）

javac \*.java //编译文件夹中所有.java的文件

jikes.exe 完全符合JLS标准，而javac并不完全符合JLS标准。

The following code is not compilable by javac, but perfectly compilable by jikes and runnable by java:

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

class SmpException extends Exception {}

public class InstInitTest {

int k, l;

{

k = 10;

System.out.println("It's ok to JLS, but it isn't ok to javac compiler");

if (k > 0)

throw new SmpException();

l = k;

}

public InstInitTest() throws SmpException{}

public static void main(String[] args) {

try {

new InstInitTest();

} catch (SmpException e){}

}

}

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

* 应用程序的运行命令Java

解释器java执行编译成功后生成的JVM字节码文件\*.class，在字节码下载和执行的过程中，解释器负责维护它的完整性、正确性和安全性。

java Welcome para1 para2 //不能用java Welcome.class

* Java小应用程序的运行命令

appletviewerHello.html

* package的编缉器

jar //将自己的文件打成包

* Java类反编译命令Javap

Javap –c Test$1

### Java文件类型

* 源代码文件（.java）：包括Java源代码。一个文件对应一个公共类。
* 类文件（.class）：包括已编译的Java源文件；每个标准Java规范中描述为已系列的字节代码。一个类文件对应于一个Java类。
* Java序列化文件（.ser）：包括Java对象状态和类描述的序列化表示。
* JavaARchive文件（.jar）：包括一个压缩的（可选）文件集，对应于ZIP文件格式。JAR文件通常包括Java类文件、序列化文件、manifest文件，以及多媒体文件（声音或视频）。Java 2 平台可以直接执行JAR文件。
* JavaManifest文件（MANIFEST.MF）：用于描述归案文件内容。

## Java工程结构

应用程序与小应用程序(applet)

### 应用程序（application）

Java通过将一个类名传给Java解释器来运行一个程序。当调用一个独立程序时，Java在运行时会寻找该类的main()方法。应用程序是含有方法main()的程序，由主文件和一些相关类的文件组成。主文件的一般构架为：

public class Manager { //Manager.java

//变量与方法声明

public static void main(String[] args){

Manager myManager = new Manager();

System.out.println( aReferenceToNull ); // is fine

}

}

类文件的特点：

1. 完全面向对象，所有的变量、函数均只能定义在类中，类以外不能有全局变量、全局函数，也没有传统的动态连接库（DLL）；
2. 类的声明与实现是合一的，不能像C++那样可分离（分为\*.h与\*.CPP）；
3. 每个文件只能有一个public 类，和多个私有类，但文件名必须是public 类的名字；
4. 能被其它文件引用的类，必须是public类。

方法main()中各部分概要：

1. public意味着这个main()方法对其它的类和对象也是可用的。方法main()必须被声明成public的；
2. static意味着main()方法是一个静态（类）方法；
3. void意味着main()方法并不返回任何值；
4. main()方法必须有一个程序参数，它是一个字符串数组。args[0]指的是类名后的第一个命令行参数，而不是该程序的名字。args是一个字符类数组的引用。

由于main方法是静态的，因此，当你用定义main方法的类名来调用Java解释器时，解释器会启动main方法，而不创建该类的实例。因此，访问该类内或者类外的方法被限制，并且mian方法只能访问该类中的静态实例字段。为了消除这种限制，大多数Java程序员使用该方法来创建自己的类的对象，然后就可以调用该对象的实例方法。

1. system.exit(int value)，也就是说只要是int都能使虚拟机退出，因此System.exit('a');同样的是合法的.

例子：

class test{

public static void main(){

System.out.println(“Hello World!”);

}

}

问题：该程序能否编译？若能编译，能否运行？

答案：程序编译通过，但运行时抛出NoSuchMethodError: main。因为Java解释器专门寻找声明为public和static的main()方法，该方法不返回任何数值，并且需要但有String对象的数组作为参数。即为public static void main(String[] args)。

针对类文件特点的第3点：考虑一个没有public类的Java文件。

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*文件名 D.java \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

class A {}

class B {}

class C {}

class E {

public static void main(String[] args) {

System.out.println("Strange thing in E.");

}

}

class F {

public static void main(String[] args) {

System.out.println("Strange thing in F.");

}

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*END \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

编译后，可以采用两种方式运行该文件（请仔细考虑为什么？）：

java E //output: Strange thing in E.

java F //output: Strange thing in F.

带有main()方法的抽象类同样可以运行，例如：

abstract class cannot be instantiated. However, if the main method only calls static method of the class, you can run it just like you run any C program.

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

public abstract class A {

static public int sum(int a, int b) { return a + b; }

abstract public void aMethod();

// abstract static combination not allowed

// abstract public static void aStaticMethod(); //not compilable

public static void main(String args[]) {

System.out.println("Sum of 3 and 5 is : " + sum(3, 5));

// The following code will not be compilable

// A a = new A();

// a.aMethod();

}

}

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

针对类文件特点的第4点，考虑一个问题。

**Q. For the following program, when I run it with " java test \* ", why it has a number of parameters as many as the number of files under the directory?**

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

class test{

public static void main(String args[]){

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

System.out.println("Para"+i+":"+args[i]);

}

if (args.length<=0) {

System.out.println("No para");

}

}

}

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

**A:**

This is a wild card in dos, "\*" stands for everything fit the context under the current directory. Have you ever tried to compile all Java files in the directory by typing "javac \*.java" or delete all files and subdirectories under the current directory by typing "del \*"?

DOS is doing the same thing in all above.

考虑下面的程序存在什么问题:

public class TestArrays{

public static void main(String[] args){

int[] array1={2,3,5,7,11,13,17,19};

p(array1);

public void p(int[] array){

int i;

for(i=0;i<array.length;i++)

System.out.print(array[i]+" ");

}

}

}

回答：其实很简单，在main函数内部定义方法public void p(int[] array){……}是错误的。

### 不具备main函数的小应用程序（applet）

小程序依赖于外部程序与用户进行交互，它是java.applet.Applet类的扩展，这意味着小程序是子类的完整范例。

含有几个特殊的函数，例如：init()：创建用户界面、start()：恢复线程、paint(Graphics g)：编写所有的绘图代码、stop()：中止线程、destroy()：终止线程，必须继承java.applet.Applet，且必须在html中运行，可用浏览器或命令appletviewer\*.htm启动。

类文件：类名.class

启动applet 的html语句：

<html>

<body>

<applet code = “类名.class” width= 300 height =100>

<param name = paramName value = paramValue>

</applet>

</body>

</html>

你可以使用Applet类中定义的方法getParameter()来获得参数或<applet>标签的值；如果参数根本没有定义，它返回null或者是包含参数值的子符串。其中paramValue是否包含引号是没有任何影响的。

### 有main函数的applet

即可用 java运行，也可用浏览器运行。

## Java语言基础知识

### 类

#### 类的声明

Java中源文件中需要保护三个基本内容：1**.** 包（package）定义；2**.** 任意数目的导入（import）语句；3**.** 任意的公有何私有类及接口。在Sun的JDK中，你只可以定义一个公开（public）类，如果你确实只定义了一个公开类，那么源文件一定以此类的名字命名。

Java中所有的类都是从类Object中继承下来，Object是Java类层次结构中所有类的超类。即Java中所有的类都是基于名为Object的基类创建的。因此，Object类中所有方法和变量都适合于所有其它的类。

Java的类名可以为任何字符串，但必须以字符、下划线（\_）或者美元符（$）开头。根据命名规范，类名的首字母应该是大写的字母。

大多数类定义有部分或者全部这些组件：

[Keywords] class MyClass [extends Superclass] [implements Interface]{

//calss variables

//static initializers

//instance initializers

//instance variables

//constructors

//class methods

//instance methods

}

#### 类的属性和行为

Java变量名（标识符）必须以字符、下划线（\_）或者美元符（$）开头，不能以数字开头。Java变量名是区分大小写的。例如：\_5\_、hello$是一个合法的变量名；而a-class、hello#是一个非法的变量。类中使用变量来保存数据。

##### 实例变量、字段、域

实例变量、字段（instance variable）定义了某个特定对象的属性。对象的类定义了该属性的类型，每一个实例都保存着它自己的属性值。即实例变量只存在于某对象的特定实例，这意味着给定类的每个不同实例都有一个名称相同的变量，但Java在内存中的不同地方分别为不同的实例存储值。

实例变量也叫做对象变量（object variable）。如果变量被声明在方法定义之外，而且不被关键字static所修饰，那么该变量就是实例变量。

通常，实例变量被声明在类声明之后，任何其它方法声明之前。但它们只需要在所有方法之外就算正确。

##### 类变量

类变量（class variable）定义了整个类的属性，是在类本身定义和存储的变量。该变量适用于类本身以及它的所有实例，因此不管该类创建了多少对象，该类变量所保存的值只有一个，即static变量只存在于内存的一个地方。类变量适合于在同一个类的不同对象之间进行通信，或者在一组对象中记录类范围上的信息。如果变量被声明在方法定义之外，且被关键字static所修饰，那么该变量就是类变量（类变量不能被子类更改）。

实例变量和类变量的定义被给出了一个依赖于它们存储信息类型的初值（默认值）：

数字变量 0 （byte：0 short：0 int：0 long：0l float：0.0f double：0.0d）

字符 '\0'或\u0000 布尔型 false 对象 null

##### 局部变量、方法变量

局部变量、方法变量(local varivale)用在方法内部定义，甚至方法内更小的语句块里。局部变量可以在方法的任何地方被声明；局部变量在程序里使用之前必须被给定值（初始化），否则程序将无法正确编译。只有当Java解释器执行到这些方法和语句块时，它们才能被用到，离开这个方法和块之后就不复存在。

class A{

static int myarg=1;

public static void main(String[] args){

//Int myarg ; 这样定义不行，局部变量在使用前必须初始化

Int myarg = 3; //这样定义可以

System.out.println(myarg);

}

}

程序可以正常编译，输出结果为：3

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block.

在Java中，不可在两个嵌套的块内声明两个完全同名的局部变量。而在C++中，允许在一个嵌套块内对一个变量进行重新定义；定义后，内部定义就会掩盖外部定义。

例如：{ outer block { inner block } }

Java没有能够用于程序的任何部分的全局变量（global variable）。实例变量和类变量被用来从一个对象向另一个对象传递信息，这样就取代了全局变量的需求。

方法定义了一个对象的行为，即在对象创建时发生的任何事情，以及在它的生存周期内能够执行的各种任务。

##### 实例方法

实例方法(instance method)是对象的方法，它通过管理对象的实例变量来对该对象进行操作；实例方法指出的是一个特定对象的行为。在方法名前面缺少关键字static使得该分发成为实例方法。实例方法只对某个特定的对象起作用，而不是整个对象的类。实例方法简称为方法。

##### 类方法、静态方法

类方法、静态方法(class method)是作用在类本身而不是某个特定的类实例上的方法；类方法指出的是属于该类的所有对象的行为。类方法对类本身的任何实例都是可用的，而且也能够被其它类使用；与实例方法不同，类在调用类方法是不需要有一个该类的实例。使用关键字static声明类方法。

静态(static)方法与类有关，并且不操作类的任何实例。静态方法只能访问静态变量（字段）。更进一步说，由于静态方法与特定的对象无关，你不能像对待非静态方法一样，使用this或者根据名称来引用实例成员。如果某方法需要访问某对象的非静态实例字段，则它不能是静态方法（或者该方法必须首先获取某实例变量）。

class Avg {

public static void main(String[] args) {

double a = 5.1; double b = 20.32; double c = 32.921;

System.out.println(findAvg(a, b, c));

}

double findAvg(double a, double b, double c) {

return (a + b + c) / 3.0;

}

}

该程序编译错误在于：静态main方法尝试调用名为findAvg()的非静态方法。由于没有调用findAvg()的Avg实例，因而导致编译错误。修改这个问题，或者声明findAvg()为静态方法，或者创建Avg实例，并且当调用findAvg()时使用。

实例方法和类方法:

class Aa{

public static void test(){}

public void test(){}

}

以上定义错误，报出test方法已经定义的编译错误

如果一个方法没有使用任何实例类成员，包括方法和字段，那么应该把方法声明为静态的，以提高代码的封装性。

**【错误示例】**

public class ASI {

public int add (int i, int j) { //违例

return i + j;

}

}

**【改正示例】**

public class ASIFixed {

public static int method (int i, int j) { //正确

return i + j;

}

}

将参数传递给方法：

当调用一个带有对象参数的方法时，你所传递给方法体内的对象时用引用来传递的。不管你在方法内对该对象进行了什么操作，它都会影响到最初的对象。

记住这些对象包含数组和包含在数组中的所有对象。当你将一个数组传递到一个方法中并修改了它的内容时，最初的数组也被修改了。

而另一方面，原始数据类型是按值传递到方法里的，这个值在方法内部发生的任何事情都不会影响在调用代码中的原始值。

#### 修饰语、关键字、限定符（modifier）

类修饰语：public、abstract、final和strictfp；

变量修饰语：public、static、final、transient和volatile；

访问控制关键字：public、private、friendly（默认的访问修饰符）和protected；

其它关键字：this。

##### public

修饰语public说明如果你导入一个包，就只能访问包中那些声明为public的类。

##### private

private修饰变量为私有变量，只能被该类自身访问和修改，不能被任何其它类，包括该类的子类直接获取或引用。

如果两个对象是同一个类的实例，那么其中一个对象可以访问另一个对象中的private变量。One object can access a private variable of another object of the same class. The private field can be accessed from static method of the class through an instance of the same class too.

例如：

public class Test1 {

private int i;

Test1(int ii) {

i = ii;

System.out.println("Self: " + i);

// to avoid infinite recursion

if (i != 3) {

Test1 other = new Test1(3);

other.i++;

System.out.println("Other: " + other.i);

}

}

public static void main(String[] args) {

Test1 t = new Test1(5);

// The private field can be accessed from static method of the same class

// through an instance of the same class

System.out.println(t.i);

}

}

编译通过，输出为：

Self: 5

Self: 3

Other: 4

5

如果超类定义了一个private方法，那么其它类不能够访问和使用它。如果在它的子类中定义了一个与超类中private方法相同名字和参数的方法，那么该方法与超类中的private方法无关。这只是一种不好的编程方式而已。

If super class defined a private method, it hides itself, nobody else can see it or use it. That is exactly what **private** means. It does not need a subclass to define a same name & signature method to be hided!!!

If the subclass use the same name and signature to define another method, it does not hide anything, it is just itself and totally irrelevant with the super class same name method. In addition, it should be only considered a bad coding style, even it is legal.

例如：

public class Base {

public static void main(String[] args) {

Sub sub = new Sub();

System.out.println(sub.g()); // 20, not –100

System.out.println(sub.f()); // -100, not 20

}

private int f() { return 20; }

// statc bound with Base.f() compiler does not know any derived class has f()

// or not and does not care either

protected int g() { return f(); }

}

class Sub extends Base {

//f() here has nothing to do with Base.f()| In real world programming, never do this!

public int f() { return -100; }

}

##### strictfp

修饰语strictfp可以对类和方法使用strictfp，以确保浮点运算能被严格地处理。如果应用到方法，方法中使用浮点值自动遵循严格的浮点规则；如果应用到类，类中所以方法都遵循严格的浮点规则。

##### static

修饰语static用来创建类方法和类变量，类方法和类变量可以使用类名后跟点号和变量或方法名得形式来访问。也可以使用该类的一个对象的名字来访问，但对类方法和类变量而言，使用类名会更好些。实例变量和实例方法不能通过类名来引用。

修饰语static的深入理解：static变量不能在方法内声明（不管是一般方法还是静态方法）（a static variable can not be delared within a method）；static方法的内部可以声明一个内部类（a static method can contain an inner class）；static方法内部不能声明一个静态的内部类（a static method can not contain an static innner class）。

例子：

class Test {

void method() {

//static int i = 3; // not compilable

}

static void smethod() {

//static int i = 5; // not compilable

class local {

}

/\* not compilable

static class slocal {

}

\*/

}

}

* 类的静态初始化段

例1：静态初始化代码段、静态变量初始化和构造函数

类定义中的一块代码也可以声明为静态的，这样的程序块叫做静态初始化段。

class A{

static int i=5;

static { System.out.println("A1"); }

{ System.out.println("A2"); }

public A(){System.out.println("A3"); } //构造函数

}

class test{

public static void main(String args[]){

//System.out.println("static i="+ A.i);

//new A();

}

}

若运行System.out.println("static i="+ A.i); 输出结果为： A1 static i=5

若运行new A(); 输出结果： A1 A2 A3

若运行System.out.println("static i="+ A.i); new A(); 输出结果为： A1 static i=5 A2 A3

加载一个类时，它的静态初始化代码被执行。静态初始化代码段与静态变量初始化同时进行，它们按在程序中出现的顺序执行。在静态初始化期间若引用到未加载的类，该类先加载并初始化。在初始化序列中，若引用到在它之前的一个未初始化的类时，就会发生循环。这将导致一个NoClassDefFoundException错误。

例2：Static类、变量、函数的初始化

首次使用的地方也就是static代码段初始化发生的地方。装载的时候，所有static对象和static代码段都会按照本来的顺序初始化（亦即它们在类定义代码里写入的顺序）。当然，static代码段只会初始化一次。

由于Java中的一切东西都是对象，每个对象的代码都存在于独立的文件中，除非真的需要代码，否则那个文件是不会载入的。通常，我们可认为除非那个类的一个对象构造完毕，否则代码不会真的载入。由于static方法存在一些细微的歧义，所以也能认为”类代码在首次使用的时候载入”。

// Beetle.java: The full process of initialization.

class Test{

static int x = prt("test");

static int prt(String s) {

System.out.println(s);

return 1;

}

}

class Insect {

int i = 9;

int j;

Insect() {

prt("i = " + i + ", j = " + j);

j = 39;

}

static int x1 = prt("static Insect.x1 initialized");

static int prt(String s) {

System.out.println(s);

return 47;

}

}

public class Beetle extends Insect {

int k = prt("Beetle.k initialized");

Beetle() {

prt("k = " + k);

prt("j = " + j);

}

static int x2 = prt("static Beetle.x2 initialized");

static int x3 = prt("hello");

static int prt(String s) {

System.out.println(s);

return 63;

}

public static void main(String[] args) {

//prt("Beetle constructor");

//Beetle b = new Beetle();

//Test.prt("^\_^");

}

}

第一次：该程序执行的输出结果为：

static Insect.x1 initializedstatic Beetle.x2 initializedhello第二次：如果尝试把对Test.prt("^\_^")；这一句的注释去掉，输出结果为：

static Insect.x1 initializedstatic Beetle.x2 initializedhellotest^\_^第三次：如果再尝试把对prt("Beetle constructor")；这一句的注释去掉，输出结果为：

static Insect.x1 initialized

static Beetle.x2 initialized

hello

Beetle constructor

test

^\_^

第四次：如果再把对Beetle b = new Beetle()；这一句的注释去掉，则输出结果为：

static Insect.x1 initialized

static Beetle.x2 initialized

hello

Beetle constructor

i = 9, j = 0

Beetle.k initialized

k = 63

j = 39

test

^\_^

对Beetle运行Java时，发生的第一件事情是装载程序到外面找到那个类。在装载过程中，装载程序注意它有一个基础类（即extends关键字要表达的意思），所以随之将其载入。无论是否准备生成那个基础类的一个对象，这个过程都会发生。若基础类含有另一个基础类，则另一个基础类随即也会载入，以此类推。接下来，会在根基础类（此时是Insect）执行static初始化，再在下一个衍生类执行，以此类推。保证这个顺序是非常关键的，因为衍生类的初始化可能要依赖于对基础类成员的正确初始化。

例如：在第一次的执行结果中，虽然我们似乎没有通过任何语句来显式调用Insect和Beetle类的任何方法或成员，但是这两个类还是先后初始化了其static成员，这是因为我们从public类的main函数开始运行java程序的时候，我们就需要载入Beetle类及其基础类Insect。从第一、二次的执行结果可以看出：仅当我们需要使用Test类的时候，它才被装入，它的static成员也就才被初始化，这正好验证了我们在前面提到的一句话：”首次使用的地方也是static初始化发生的地方”。

此时，必要的类已全部装载完毕，所以能够创建对象。首先，这个对象中的所有基本数据类型都会设成它们的默认值，而将对象句柄设为null。随后会调用基础类构造函数。在这种情况下，调用是自动进行的。但也完全可以用super来自行指定构建器调用（就象在Beetle()构建器中的第一个操作一样）。基础类的构建采用与衍生类构建器完全相同的处理过程。基础顺构建器完成以后，实例变量会按本来的顺序得以初始化。最后，执行构建器剩余的主体部分。

**Q. Why cannot we declare a static variable in a member function of a class in Java like in c++?**

**A:**

**Simple answer:** That is Java creator's decision

**A little Guess work by me:** Java want that static means per-class, like classmethod, class variable, are static. Per-method or per-file like c++ is not consistent with this kind of thinking, and not OO. Don't forget C++ is a hybrid language (OO/not OO).

The static can not be a modifier before the keyword class.

static class TestClass{} //Error

##### final

修饰语final与类、方法和变量一起使用来指明它们不会被修改。

* 一个final变量不能改变值（final variables cannot be changed）；

情形1：final数据

Java语言通过final属性告诉编译器某个数据是常数。常数主要应用于下述两个方面：1．编译期常数，它永远不会改变；2．在运行期初始化的一个值，我们不希望它发生变化。

对于编译期的常数，编译器（程序）可将常数值“封装”到需要的计算过程里。也就是说，计算可在编译期间提前执行，从而节省运行时的一些开销。在Java中，这些形式的常数必须属于基本数据类型（Primitives），而且要用final关键字进行表达。在对这样的一个常数进行定义的时候，必须给出一个值。

一个类的属性（可为static也可以不是static的，类型可以为基本数据类型或对象句柄）如果被声明为final，那么在声明的时候必须为之赋值。

对于基本数据类型，final会将值变成一个常数；但对于对象句柄，final会将句柄变成一个常数；进行对象句柄声明时，必须将句柄初始化到一个具体的对象，而且永远不能将句柄变成指向另一个对象。但是对象本身是可以修改的。

下面是演示final字段用法的一个例子：

// FinalData.java: The effect of final on fields

class Value { int i = 1; }

public class FinalData {

// Can be compile-time constants

final int i1 = 9;

static final int I2 = 99;

// Typical public constant:

public static final int I3 = 39;

// Cannot be compile-time constants:

final int i4 = (int)(Math.random()\*20);

static final int i5 = (int)(Math.random()\*20);

Value v1 = new Value();

final Value v2 = new Value();

static final Value v3 = new Value();

// final Value v4; // Java Error: no initializer

// Arrays:

final int[] a = { 1, 2, 3, 4, 5, 6 };

public void print(String id) {

System.out.println( id + ": " + "i4 = " + i4 + ", i5 = " + i5);

}

public static void main(String[] args) {

FinalData fd1 = new FinalData();

// fd1.i1++; // Error: can't change value

fd1.v2.i++; // Object isn't constant!

fd1.v1 = new Value(); // OK -- not final

// fd1.v2 = new Value(); // Error: Can't change handle

// fd1.v3 = new Value(); // Error: Can't change handle

for(int i = 0; i < fd1.a.length; i++)

fd1.a[i]++; // Object isn't constant!

// fd1.a = new int[3]; // Error: Can't change handle

fd1.print("fd1");

System.out.println("Creating new FinalData");

FinalData fd2 = new FinalData();

fd1.print("fd1");

fd2.print("fd2");

}

}

由于i1和I2都是具有final属性的基本数据类型，并含有编译期的值，所以它们除了能作为编译期的常数使用外，在任何导入方式中也不会出现任何不同。I3是我们体验此类常数定义时更典型的一种方式：public表示它们可在包外使用；Static强调它们只有一个；而final表明它是一个常数。注意：对于含有固定初始化值（即编译期常数）的fianl static基本数据类型，它们的名字根据规则要全部采用大写。也要注意i5在编译期间是未知的，所以它没有大写。

不能由于某样东西的属性是final，就认定它的值能在编译时期知道。i4和i5向大家证明了这一点。它们在运行期间使用随机生成的数字。例子的这一部分也向大家揭示出将final值设为static和非static之间的差异。只有当值在运行期间初始化的前提下，这种差异才会揭示出来。因为编译期间的值被编译器认为是相同的。这种差异可从输出结果中看出：

fd1: i4 = 11, i5 = 6

Creating new FinalData

fd1: i4 = 11, i5 = 6

fd2: i4 = 19, i5 = 6

注意对于fd1和fd2来说，i4的值不是唯一的，但i5的值不会由于创建了另一个FinalData对象而发生改变。那是因为它的属性是static，而且在载入时初始化，而非每创建一个对象时初始化。

从v1到v4的变量向我们揭示出final句柄的含义。正如大家在main()中看到的那样，并不能认为由于v2属于final，所以就不能再改变它的值。然而，我们确实不能再将v2绑定到一个新对象，因为它的属性是final。这便是final对于一个句柄的确切含义。我们会发现同样的含义亦适用于数组，后者只不过是另一种类型的句柄而已。将句柄变成final看起来似乎不如将基本数据类型变成final那么有用。

情形2：空白final

Java 允许创建"空白final"，它们属于一些特殊的字段。尽管被声明成final，但却未得到一个初始值。无论在哪种情况下，空白final都必须在实际使用前得到正确的初始化。而且编译器会主动保证这一规定得以贯彻。然而，对于final关键字的各种应用，空白final具有最大的灵活性。

举个例子来说，位于类内部的一个final字段现在对每个对象都可以有所不同，同时依然保持其"不变"的本质。下面列出一个例子：

//: BlankFinal.java: "Blank" final data members

class Poppet { }

class BlankFinal {

final int i = 0; // Initialized final

final int j; // Blank final

final Poppet p; // Blank final handle

// Blank finals MUST be initialized in the constructor:

BlankFinal() {

j = 1; // Initialize blank final

p = new Poppet();

}

BlankFinal(int x) {

j = x; // Initialize blank final

p = new Poppet();

}

public static void main(String[] args) {

BlankFinal bf = new BlankFinal();

}

}

现在强行要求我们对final进行赋值处理：要么在定义字段时使用一个表达式，要么在每个构造函数中。这样就可以确保final字段在使用前获得正确的初始化。

其它例子：

//final变量初未初始化

public class Test{

static int sn;

int n;

final static int fsn; //编译错误

final int fn6; //编译错误

}

//final变量直接初始化

public class Test{

static int sn;

int n;

final static int fsn = 3;

final int fn = 6;

}

//final变量采用静态初始化段初始化

public class Test{

static int sn;

int n;

final static int fsn;

final int fn;

static { fsn=6; }

{ fn =8; }

}

//采用构造函数进行初始化

public class Test{

static int sn;

int n;

final static int fsn;

final int fn;

static { fsn=6; }

Test(){ fn =8; }

Test(int pn){ fn =pn; }

}

情形3：final方法参数

Java 允许将方法的参数设成final属性，方法是在自变量列表中对它们进行适当的声明。这意味着在一个方法的内部，我们不能改变自变量句柄指向的东西。如下所示：

class Gizmo {

public void spin() { }

}

public class FinalArguments {

void with(final Gizmo g) {

//g = new Gizmo(); // Illegal -- g is final

g.spin();

}

void without(Gizmo g) {

g = new Gizmo(); // OK -- g not final

g.spin();

}

// void f(final int i) { i++; } // Can't change: You can only read from a final primitive:

int g(final int i) {

return i+1;

}

public static void main(String[] args) {

FinalArguments bf = new FinalArguments();

bf.without(null);

// bf.with(null); // Error: cause java.lang.NullPointerException

bf.with(new Gizmo());

System.out.println(bf.g(5)); //输出6

System.out.println(bf.g(8)); //输出9

}

}

注意此时仍然可以为final自变量分配一个null（空）句柄，同时编译器不会捕获它，编译通过。但在运行时会报出NullPointerException异常。

方法f()和g()向我们展示出基本类型的方法参数为final时会发生什么情况：我们只能读取自变量，不可改变它。

* 一个final方法不能被任何子类重写，但可以被子类继承（final method cannot be override）；

之所以要使用final方法，可能是出于对两方面理由的考虑。第一个是为方法“上锁”，防止任何继承类改变它的本来含义。设计程序时，若希望一个方法的行为在继承期间保持不变，而且不可被覆盖或重写，就可以采取这种做法。

采用final方法的第二个理由是程序执行的效率。将一个方法设成final后，编译器就可以把对那个方法的所有调用都置入“嵌入”调用里。只要编译器发现一个final方法调用，就会（根据它自己的判断）忽略为执行方法调用机制而采取的常规代码插入方法（将自变量压入堆栈；跳至方法代码并执行它；跳回来；清除堆栈自变量；最后对返回值进行处理）。相反，它会用方法主体内实际代码的一个副本来替换方法调用。这样做可避免方法调用时的系统开销。当然，若方法体积太大，那么程序也会变得雍肿，可能得不到嵌入代码所带来的任何性能提升。因为任何提升都被花在方法内部的时间抵消了。Java编译器能自动侦测这些情况，并颇为“明智”地决定是否嵌入一个final方法。然而，最好还是不要完全相信编译器能正确地作出所有判断。- 这一点类似于C++的inline函数。

同样可以这样理解：将方法声明为final的第二个理由是使该类运行得更加有效。通常，当Java运行某个方法时，它会首先在当前类中查找该方法，接下来检查它的超类，并一直沿着类层次向上查找直到该方法被找到。出于灵活性和容易开发的目的，此过程牺牲了一些速度。如果某个方法时final的，Java编译器能将该方法的二进制代码直接放到调用它的任何程序里，从而优化了程序，实现了代码内联技术（inlining the code）。

通常，只有在方法的代码量非常少，或者想明确禁止方法被覆盖的时候，才应考虑将一个方法设为final。

类内所有的私有（private）方法无需声明也是final的，因为它们在任何层次结构下都不能在子类中被重写，即将方法声明为static和private都隐含了final。可为一个private方法添加final关键字，但却不能为该方法提供任何额外的含义。

不能定义一个abstract类型的final方法。

* 一个final类不能被任何类继承，在类前面用final，使类不能被子类化（final classes cannot be inherited）。

如果说类是final的（在它的定义前冠以final关键字），就表明自己不希望从这个类继承，或者不允许其他任何人采取这种操作。换言之，出于这样或那样的原因，该类肯定不需要进行任何改变；或者出于安全方面的理由，我们不希望进行子类化（子类处理）。

在一个final类中所有的方法本身自动都是final的，不必在它们的声明中使用修饰语final，虽然可以为final类内的方法添加final关键字，但这样做没有任何意义。

在一个final类中数据成员既可以是final，也可以不是，取决于我们具体选择。

因为final类是不能再扩展的。如果想阻止程序员对某个特定的类建立子类，可以将这个类声明为final类。

例如：String类是final的，不能被继承。

##### abstract

修饰语abstract声明抽象类，抽象类是不能被实例化的类（不能被实例化生成自己的对象），只能做为超类，必须被其它类继承和实现。抽象类可以包含一般的方法声明和也可以包含抽象方法的声明（It is not a requirement for an abstract class to have any abstract methods），抽象方法是一个没有方法主体实现的方法声明，这些方法在抽象类的子类中被实现。不能在非抽象类中声明抽象方法。

抽象类的特点有：

1. 如果子类派生自一个带有abstract方法的超类，而且在子类中没有为那个abstract方法提供定义，那么该方法在子类中仍为abstract。结果，子类也是一个abstract类而且必须显式的声明为一个abstract类，且不能实例化。即如果你把某方法声明为抽象的，你也必须把其所在的类定位抽象的。
2. 即使一个类是abstract的，它仍有面向子类的正常的继承规则的实例数据和非abstract方法。

你必须将一个类声明为abstract类型，如果：

1. 它的方法都是abstract类型；
2. 它继承了定义abstract方法的类的信息，而且不会实现这些方法；
3. 它扩展了一个接口，但是不实现它的方法。

即使自己定义的类不包括任何abstract类型的方法，你也可以随意地决定不需要别的程序员初始化你的类，并将你的类声明为abstract类型。

如果一个类是abstract类型，那么abstract关键字应该放在类定义之前；如果一个方法是abstract类型，那么abstract关键字应该放在方法定义之前。

没有声明任何abstract方法的abstract类的子类，在默认情况下不是abstract类型的。

抽象方法（abstract method）为超类或界面提供了一种手段，它实际上是必须由子类实现的一组协议。标记为abstract的方法必须在其子类中定义。抽象方法没有方法体，而声明由分号结束。

使用abstract方法的规则如下：构造函数不能为abstract；abstract方法不能为static类型（你不能覆盖一个static方法：由于static方法是类方法，没有this或super，因为没有当前对象对static方法起作用，所以无法调用超类的行为）、private 方法不能为abstract；abstract方法必须在声明它的类的子类中定义，不能覆盖超类中的abstract方法；包含abstract方法的类和继承abstract方法的类是抽象abstract类（只要有一个方法声明为abstract的，那么整个类就是abstract的）；初始化抽象类或直接调用abstract方法，将导致编译时刻错误；不能将一个abstract方法定义为native类型。

例子：

//class abstract Animal{ 这样声明抽象类是错误的

abstract class Animal{

abstract void gorw();

// abstract void gorw() { }; 错误声明，抽象方法应没有方法体。

// abstract gorw(); 错误声明，抽象方法没有说明返回类型。

}

##### synchronized

修饰符synchronized：synchronized方法任何时间只能被一个线程调用。Synchronized方法可以属于一个类或者对象。如果某线程进入某synchronized实例方法，则任何其它线程不能再调用该对象的任何其它synchronized实例方法。如果某线程进入某synchronized类方法，则任何其它线程不能调用该类的任何其它synchronized类方法。

When you synchronize a method of a class, as in:

public synchronized int getAge(){ ... }

that method will acquire a monitor lock before execution. If the synchronized method is static, e.g. a class method, the lock is obtained over the class. Otherwise, the lock is obtained over an instance object of the class.

##### native

修饰符native：声明某方法为native，这意味着它是一种运行平台的“本地化”语言进行实现的。换句话说，它是用一种诸如C的语言编写的，并且为该特定平台进行了编译。native方法没有方法体，甚至没有空花括号。尽管native方法可以抛出异常，但它不能是抽象的。

A native method can not have a method body. And the keyword native can't be put in front of the return value of the method.

声明一个native方法：

public native void test(); //正确

public native void test(){} //错误

##### transient

修饰符transient：transient变量不能用作对象的永久状态存储。这表明这个变量不会被串行化，而且能够使敏感的数据不被写入流中。transient类型的变量不能声明为final或static。

##### volatile

修饰符volatile：volatile表明在一个多处理器的环境中，变量可以异步修改。

##### this

关键字this指向了当前对象，this是对类的当前实例的引用，所以应该只在实例方法的方法体内使用它。类方法不能使用this。特别地，当一个局部变量具有与实例变量相同的名字时，由于变量作用域的关系，实例变量在方法中被局部变量隐藏，但可以在实例变量名前使用关键字this来达到在局部变量作用域内引用实例变量的目的。

#### 对象的类型转换

可以把一种对象类型转换为另一种类型。（但必须满足类型一致性原理）

如果在类层次关系上进行向下类型转换（Down cast form superclass to subclass），你必须强制转换类的类型。

如果在类层次关系上进行向上类型转换（Upper cast from subclass to superclass），你可以不做类型转换就可以把某对象分配给一个超类引用。

只能在层次关系中进行向上或向下的类型转换，不能在两个兄弟（sibling）间或不相关的类间进行类型转换。

例如：

class Parent {

public static void main(String[] args) {

Derived1 d1 = new Derived1();

Derived2 d2 = new Derived2();

Parent p = new Parent();

}

}

class Derived1 extends Parent{}

class Derived2 extends Parent{}

p = d1; //编译，运行通过

d1 = p; //编译出错：incompatible types 类似于将一个float值赋给一个int变量。这实在试图转换一个具有更多内容的值给一个内容较少的类型。

d1 = (Derived1)p; //进行强制转换，编译通过，这只是因为编译器可能允许这种类型转换，并不表示事实就是这样；是否合法依赖于运行时对象的类型是什么。对于这段代码来说，对象引用p引用了类Parent的一个实例，而Parent不是从Derived1继承而来的，所以运行出错，抛出java.lang.ClassCastException异常

但可以这么来做：

Parent p = new Derived1 ();

d1 = (Derived1)p; //编译，运行通过

d1 = d2; //编译出错：incompatible types

d1 = (Derived1)d2; //编译出错：inconvertible types

#### 内类（nested class）和匿名类（anonymous class）

内类（an inner class）是定义在其它类内部的类。

如果在与封装类（EnclosingClass）的实例变量同一层次中定义一个内类，那么不管这些实例变量是何种访问控制（甚至是private），内类也可以对它们进行访问，这就像定义在类中的方法可以访问类的变量一样。

如果内类是在方法内定义的，则此内类可以访问封装类的实例变量，并且可以访问该方法的局部变量和参数。

实例：

class A{

public int x;

private int y;

class B{

protected void method1(){ }

class C(){

protected void method2(){ }

}

}

}

class D extends A{

public float z;

}

在没有其它实例的引用时，method2()可以访问定义在A中的变量x、y和在B中定义的方法method1()。但不能访问定义在D中的变量z（当一个变量属于封闭类的子类时，如果没有对其进行引用，内类不能访问该变量）。

如果要应用来自于内类的局部变量和参数，那么这些变量及参数应该声明为final类型。如果想引用封装类的当前实例，可以这样写：EnclosingClassName.this；如果需要使用一个完全限定的名字引用内类，可以这样写：EnclosingClassName.InnerClassName。

内部类的特点：一个内部类的对象可访问创建它的那个对象的属性情况；同一个包内，内部类可被其它类隐藏起来；打算动态定义回调函数时，采用匿名内部类比较容易；在编写事件驱动程序时，内部类非常方便。

如果内类没有被定义为静态（static）的，那你只能创建来自于非静态方法的内类的实例。静态内类（a static nested class）不会接收一个隐含的EnclosingClassName.this指针，这就是要在封闭类（封装类）的静态方法中初始化静态内类的原因。

Member class declarations describe nested classes that are members of the surrounding class. Member classes may be static, in which case they have no access to the instance variables of the surrounding class; **or** they may be inner classes.

实例1：

class First {

public static void main(String[] args) {

new Second().sayGoodnightGracy(); //静态方法创建内类实例 （错误）

}

class Second extends First { //内类 //编译错误，内类为非静态的，不能用静态方法创建内类实例；若修改为 static class Second extends First 则正确。即静态内类可以在封闭类的静态方法中进行实例化。

void sayGoodnightGracy() {

System.out.println("Goodnight, Gracy");

}

}

}

实例2：

class First {

public static void main(String[] args) {

new First().test();

}

void test() {

new Second().sayGoodnightGracy(); //非静态方法创建内类实例

}

class Second extends First { //内类

void sayGoodnightGracy() {

System.out.println("Goodnight, Gracy");

}

}

}

内类的修饰符不能为transient，见下例：

public class Test {

public static void main(String argv[]){

}

// all ok but one

public class MyInner1 { }

protected class MyInner2 { }

private class MyInner3 { }

class MyInner4 { }

final class MyInner5 { }

static class MyInner6 { }

abstract class MyInner7 { }

//transient class MyInner8 { } // not compilable

}

匿名类（Anonymous Classes）：匿名类的末尾要用分号(;)来结束，因为匿名类的定义就像定义语句一样，需要在结尾加分号。匿名类不具有构造函数，Java会隐含地调用其超类的构造函数。

有关匿名类的访问方法：（Is it possible that an anonymous class instance is referred outside of its scope of definition? Yes! Anonymous class can be referred by the interface it implements or class it extends.）

import java.io.\*;

public class InnerLocalAnonymous {

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

OuterA o = new OuterA();

Keeper k1 = o.getKeeper1(1); System.out.println(k1.getInt());

Keeper k2 = o.getKeeper2(2); System.out.println(k2.getInt());

Keeper k3 = o.getKeeper3(3); System.out.println(k3.getInt());

Keeper k4 = o.getKeeper4(4); System.out.println(k4.getInt());

Keeper k5 = o.getKeeper5(5); System.out.println(k5.getInt());

// m() does not exist to outside world of getKeeper5(). not compilable

// System.out.println(k5.m());

System.in.read();

}

}

class OuterA {

Keeper getKeeper1(final int i) {

//Local class

class MyKeeper1 implements Keeper {

public int getInt() { return i; }

}

// return local class instance

return new MyKeeper1();

}

Keeper getKeeper2(int i) {

// return inner class instance

return new MyKeeper2(i);

}

// inner class

class MyKeeper2 implements Keeper {

final int j;

MyKeeper2(int i) { j=i; }

public int getInt() { return j; }

}

Keeper getKeeper3(final int k) {

// return anonymous class instance

return new MyKeeper2(k){

public int getInt() { return j + 4; }

};

}

Keeper getKeeper4(final int i) {

// return anonymous class instance but different than getKeeper3()

return new Keeper(){

public int getInt() { return i; }

};

}

Keeper getKeeper5(final int i) {

// return anonymous class instance but different than getKeeper3()

// and getKeeper3() added a method noone else knows.

return new Keeper(){

public int getInt() { return m(); }

// m() does not exist outside getKeeper5()

public int m() { return i; }

};

}

}

// The common interface

interface Keeper { int getInt(); }

Anonymous class always extends the class (1) or implement the interface (2) after the keyword new. In the second case, it extends Object. I assume that **extends or implements** can only be implicit, in case of anonymous classes. See the following example:

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

import java.awt.\*;

import java.awt.event.\*;

class Outer extends Frame {

public Outer(String s){

super(s);

addWindowListener(new WindowAdapter(){

// anonymous class extends WindowAdapter

public void windowClosing(WindowEvent e){ System.exit(0); }

} );

Button btn = new Button("Exit");

btn.addActionListener( new ActionListener(){

// anonymous class implements ActionListener, and extends Object

public void actionPerformed(ActionEvent e){ System.exit(0); }

} );

add(btn);

}

public static void main(String[] args) {

Outer f = new Outer("Test");

f.setSize(400, 300);

f.setVisible(true);

}

}

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

**Q. What is the differences between static or non-static inner classes?**

**A:**

1) In the new Sun (JLS2) nomenclature there is no such thing as a static inner class. There are toplevel classes and nested classes, and nested classes are by definition divided into static and inner.

2) If it is static, it is not inner class, you use it just as top level class, but put a qualifier as EncloseingClass.EnclosedClass.

3) Inner must be attached to an instance of the Enclosing class, something like new EnclosingClass().new EnclosedClass() or used in a instance method of the EnclosingClass.

4) Attention: JLS2 has got rid of "Nested top level class" confusing terminology now. The correct name is static nested class which in contrary to inner class (non - static nested class) in JLS2 now.

**Q. An anonymous class cannot have any constructors. true or false?**

**A:**

You cannot explicitly define constructors for anonymous class, but compiler generates an constructor for it with the same signature as its parent constructor called. If the parent has more than one contructors, the anonymous will have one and only one constructor.

Here is an example. After you compile it, then do **javap Test$1**, you will see the compiler generated **one and only one** constructor with the same signature as A constructor used, even class A has two constructors.

// Test.java

class A {

int a;

// 2 contructors for A

A() { a = 100; }

A(int aa) { a = aa; }

public int m(int n) { return a + n; }

}

public class Test {

public static void main(String[] args) {

// Anonymous class extends A, override method m()

A oa = new A(12) { public int m(int n) { return a - n;} };

System.out.println(oa.m(5)); // 7

}

}

// javap Test$1 output

/\*

C:\CODE>javap Test$1

Compiled from Test.java

final class Test$1 extends A {

Test$1(int);

public int m(int);

}

**Q. An anonymous class can only access static fields of the enclosing**

**class. true or false?**

**A: false.**

An anonymous class object can access static and instance fields when it has an associated instance of the enclosing class (i.e. defined in a instanse method). An anonymous class can only access static fields when it is in a static context (i.e. defined in a static method). "

What it can't access is local variables (unless they are declared final). Attention: parameters passed to the method are treated the same as local variables, since it is passed by value and a local copy are really being used.

**Q. Why does local class defined in a method only can access final local**

**variables or final parameters?**

**A:**

Local class (anonymous or with a name) defined in a method can be returned by the method, and it can live much longer than the method itself.

However, the local variables are usually on the method calling-stack, and will be out of scope when the method returns. And the parameter passs into the methods are local copies of them (Read the topic pass-by-value). They will be out-of-scope too when the calling stack returns.

However, if it is final, the local class can treat them as constant, and don't care what happens to the original variable any more.

**Don't use inner class unless it is a very simple case. You are better-off without using inner classes in complicated projects.**

**Nested classes (Carls Notes)**

->get from the Java Language Specification and from Sun's Java Tutorial

\*\*\* A nested class is a class that is defined inside another class.

\* There are two distinct types of nested classes:

- static nested classes (or top-level nested classes)

- inner classes (which are always associated with an instance of

an enclosing class).

\* (Nested) inner class types:

- member classes,

- local classes (method or code block),

- anonymous classes.

\* Top-level classes (all classes are either top-level or inner)

- static nested classes,

- package member classes.

\* Access to enclosing class:

- all outer-class members (inc. private) are accessible to an inner class [Usually without scope modifiers, but if they are hidden by an inner class name, you can use Outer.this.outerVar]

- static nested classes cannot acccess instance variables from enclosing classes (there is no instance), but can access their static variables (and classes i would think).

\* Instantiation:

- For accessible inner classes: Outer.Inner i = new Outer().new Inner(); Even if you are inOuter's scope, you need to have an Outer instance.

- For accessible static nested classes: Outer.Nested nested = new Outer.Nested();

\* Local inner classes cannot access non-final local variables or method arguments.

\* Nested classes generally have the same options with regard to modifiers as do variables declared in the same place.

\* Unlike class-level nested classes, local classes are executed in the method's sequence of execution so you can't create an instance of the local class before it is declared.

\* The static keyword marks a top-level construct (class, method or field) and can never be subject to an enclosing instance.

- no inner class can have a static member.

- no method can have a static member.

\* Interfaces automatically attach 'public static final' to field and class members (thus making them top-level nested rather than member inner classes).

\* A nested class cannot have the same simple name as any of its enclosing classes (note: there is no similar restriction on method or variable names).

内部类笔记：

static inner class :只能访问static的变量和方法。

member inner class :

EnclosingClass.MemberClass mc = new EnclosingClass().new MemberClass();

member inner class 不能有static变量。

local inner class :

在method或者代码块里定义的inner class。

Local inner classes, like local variables,

cannot be declared public, protected, private, or static.

Like member inner classes, local inner classes cannot have static members

and they cannot have the same name as the enclosing class in which they are defined.

匿名内部类 是一种local inner class

The rule is that a local or an anonymous inner class

can access only those local variables and method arguments that are declared final。

This means that you cannot change the value of local variables

from within the code of the inner class. They are read-only.

#### 存取器方法

类中有一个实例变量，它对所包含的值有严格的限制。为了防止某个外部的类不正确地设置该实例变量，将其声明为private。如果其它类要访问该实例变量则需要使用存取器方法（ 比较好的解决方法是为该变量提供get方法和set方法）。

存取器方法因为能够对某些内容地存取访问权限而得名，否则，那些内容会具有不能被访问得限制。通过使用某个方法来提供对一个私有变量得访问，你可以控制如何使用该变量。

通常，用于读取和写入变量得存取器可以是分离的。读取方法有一个以get开始的名字，而一个写入方法有一个以set开始的名字。

使用方法来访问实例变量是面向对象编程中一种常用的技术。这种实现使得类具有更多的可复用性，因为它防止了对变量的不正确使用。

public class ZipCode {

private int zipCode = -1; //实例变量

public ZipCode(int inZipCode) {

setZipCode(inZipCode);

}

public void setZipCode(int inZipCode) {

if ((inZipCode > 9999) & (inZipCode < 100000)) //使用有限制

zipCode = inZipCode;

}

public int getZipCode() {

return zipCode;

}

public static void main(String[] arguments) {

ZipCode zip1 = new ZipCode(75044);

if (zip1.getZipCode() != -1)

System.out.println("Zip1: " + zip1.getZipCode());

}

}

如果关心存取器方法的速度，你总可以将存取器方法声明为final，在大多数情况下它们在速度上可以与直接的实例变量相仿。

通过给类定义访问方法来获得并且设置变量的值是一个良好的编程习惯。

① 只允许其它类使用访问方法，这样可以隐藏该类的实现，并且可以使其它类大大地减少对该类数据布局的依赖。

② 可以确保对象数据的完整性。set方法可以确保类为变量分配的值有意义，而不是任意分配一个值。如果出现异常情况，该方法可以纠正问题，并且可能抛出异常，通知调用程序该操作没有发生，或许还会给出原因。

③ 在多线程环境下，你可以更多地控制变量数据。例如，可以使访问方法同步化，以便于当某线程在别处操作数据时，其它线程不会获得或设置该数据。

④ 只需要通过定义get或set方法，就可以根据需要将变量设置为只读或只写。

#### 变量与方法的能见度

能见度：是指一个类的变量或方法是否可被另一个类访问或继承。

蓝色表示在不同包中的类；**黑色**表示在同一包中的类。

|  |  |  |  |
| --- | --- | --- | --- |
| 条件  类型 | 继承 | 访问 | 备注 |
| Public | Yes/yes | Yes/yes |  |
| private | No/no | No/no |  |
| protected | Yes/yes | Yes/no |  |
| 缺省：表示友元 | Yes/no | Yes/no | 在同一个包(package)可以相互访问 |

如果不指明访问关键字，如public、protected等，就表示是友好的（friendly），允许同一个包内的其它类访问，包外的类不能访问。

四种访问控制级别：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Visibility** | Public | Protected | Default | Private |
| 在同一个类里 | Yes | Yes | Yes | Yes |
| 从同一包中的任何类里 | Yes | Yes | Yes | No |
| 从包外的任何类 | Yes | No | No | No |
| 从同一包中的子类 | Yes | Yes | Yes | No |
| 从包外的子类 | Yes | Yes | No | No |

Assume we have the following code in the file /abc/def/Q.java:

//File: /abc/def/Q.java:

package def;

public class Q {

private int privateVar;

int packageVar;

protected int protectedVar;

public int publicVar;

}

and this code is in /abc/Tester.java:

//File: /abc/Tester.java:

import def.Q;

public class Tester extends Q {

Q sup = new Q();

Sub sub = new Sub();

public void someMethod() {

// First, try to refer to sup's memebers.

sup.privateVar = 1; // Line 1

sup.packageVar = 2; // Line 2

sup.protectedVar = 3; // Line 3

sup.publicVar = 4; // Line 4

// Next, try to refer to this object's members

// supplied by class Q.

privateVar = 5; // Line 5

packageVar = 6; // Line 6

protectedVar = 7; // Line 7

publicVar = 8; // Line 8

// Next, let's try to access the members of

// another instance of Tester.

Tester t = new Tester();

t.privateVar = 9; // Line 9

t.packageVar = 10; // Line 10

t.protectedVar = 11; // Line 11

t.publicVar = 12; // Line 12

// Finally, try to refer to the members in a

// subclass of Tester.

sub.privateVar = 13; // Line 13

sub.packageVar = 14; // Line 14

sub.protectedVar = 15; // Line 15

sub.publicVar = 16; // Line 16

}

}

and this code is in /abc/Sub.java:

//File: /abc/Sub.java:

public class Sub extends Tester {

}

Just as with Question 1, assume the directory, /abc, is in the compiler's CLASSPATH.

When you try to compile Tester.java there will be several compiler errors. For each of the labeled lines above, decide whether or not a compiler error will be generated.

Answer 2

General Visibility

When one class extends another class, instances of the subclass will contain their own copies of all members declared in the superclass (including any private variables). In the example above all instances of Tester will contain all four of the variables declared in Q. However, not all of these variables will be accessible (i.e., visible) by instances of Tester.

In the descriptions below, we say a variable is inherited by a subclass if it is accessible to instances of the subclass.

sup.privateVar = 1; // Line 1

private members are only accessible from within the class in which they are declared. Because privateVar was declared in class Q, it is not accessible from class Tester. The following compiler error results:

Tester.java:10: privateVar has private access in def.Q

sup.privateVar = 1; // Line 1

^

sup.packageVar = 2; // Line 2

package members are only accessible to classes within the same package as the class that declares them. In the case, the member packageVar, is declared in class Q, in the def package. package is accessible only from within classes in the def package. Since the Tester class is in the "default package" it does not have access to packageVar. The following compiler error results:

Tester.java:11: packageVar is not public in def.Q; cannot be accessed from outside package

sup.packageVar = 2; // Line 2

^

sup.protectedVar = 3; // Line 3

Even though class Tester extends class Q, instances of Tester are not allowed access to protected members in instances of Q, proper. Because sup is an instance of Q, proper, a Tester instance will not have acces to its protected members. An instance of class Tester can only access protected members in instances of classes that are of type Tester. This includes classes that inherit from Tester (such as the Sub class). The following compiler error results:

Tester.java:12: protectedVar has protected access in def.Q

sup.protectedVar = 3; // Line 3

^

sup.publicVar = 4; // Line 4

public members are available "everywhere". Consequently, Tester's reference to sup.publicVar compiles without error.

privateVar = 5; // Line 5

private members are only accessible from within the class in which they are declared. Because privateVar was declared in class Q, it is not accessible from class Tester. Note that on Line 1 we were trying to refer to a private member in another object (the member sup). On line 5 we are trying to refer the copy of privateVar that Tester contains by virtue of being a subclass of Q. In terms of the visibility of privateVar, however, it does not matter. privateVar is a private member declared in the Q class and can only be accessed from within that class. The following compiler error results:

Tester.java:17: privateVar has private access in def.Q

privateVar = 5; // Line 5

^

packageVar = 6; // Line 6

As was true for line 2, package members are only accessible to classes within the same package as the class that declares them. Here we are attempting to access the packageVar member contained in an instance of the Tester class. In other words, an instance of Tester is trying to access the packageVar it contains by virture of its inheritance from Q. The protectedVar member was declared by a class in the def package and Tester is not in that package. Therefore, Tester cannot access the packageVar member. The following compiler error results:

Tester.java:18: packageVar is not public in def.Q; cannot be accessed from outside package

packageVar = 6; // Line 6

^

protectedVar = 7; // Line 7

There is no compiler error for line 7. Remember that a protected member is accessible within:

an instance of a class in the same package as the class that declared it (yes, this is exactly the same restriction that applies to package members)

an instance, s1, of a class, S, that extends a class, B, where B declares the protected member and where s1 is referring to one of the following:

its own copy of the protected member inherited from B

a copy of the protected member owned by another instance, s2, of S (and therefore, inherited from B)

a copy of the protected member owned by an instance of a subclass of S

The language here is getting a bit abstract. In our coded example above, class B is represented by class Q and class S is represented by class Tester. Instance s1 is simply an instance of Tester.

What's being said is that an instance of Tester can refer to its own copy of protectedVar, the copy of protectedVar contained in some other instance of Tester, or the copy of protectedVar contained in an instance of a subclass of Tester. An instance of Tester cannot, however, refer to a copy of protectedVar in an object that is not an instance of Tester.

publicVar = 8; // Line 8

As was the case with line 12, public members are available "everywhere". Consequently, Tester's reference to publicVar compiles without error.

t.privateVar = 9; // Line 9

As was the case with lines 1 and 5, the member, privateVar, is declared in class Q and can therefore only be accessed by instances of class Q, proper (i.e., non-subclasses of class Q). The following compiler error results:

Tester.java:25: privateVar has private access in def.Q

t.privateVar = 9; // Line 9

^

t.packageVar = 10; // Line 10

As was the case with lines 2 and 6, the member, packageVar, is declared in class Q which is in the def package. Only instances of classes in the same package as class Q can access the package variable, packageVar. Remember that because the Tester class does not have a package statement it is placed in Java's "default package". The following compiler error results:

Tester.java:26: packageVar is not public in def.Q; cannot be accessed from outside package

t.packageVar = 10; // Line 10

^

t.protectedVar = 11; // Line 11

You may want to refer back to the description of line 7. If we have a class (call it B) that contains a protected member (call it p) and another class (call it S) extends B, all instances of S can access protected members in any instance of class S, including subsclasses of S.

Once again, in this coded example, class B is class Q, class S is class Tester, and p is the protected member protectedVar. Line 11 is an example of an instance of Tester attempting to reference a protected member in another instance of Tester (represented by the member, t). Since an instance of Tester is attempting to reference a protected member in another instance of Tester, the line compiles cleanly.

t.publicVar = 12; // Line 12

As was the case with lines 4 and 8, public members are available "everywhere". Consequently, Tester's reference to t.publicVar compiles without error.

sub.privateVar = 13; // Line 13

As was the case with lines 1, 5, and 9, the member, sub.privateVar, is declared in class Q and can therefore only be accessed by instances of class Q, proper (i.e., non-subclasses of class Q). The following compiler error results:

Tester.java:32: privateVar has private access in def.Q

sub.privateVar = 13; // Line 13

^

sub.packageVar = 14; // Line 14

As was the case with lines 2, 6, and 10, the member, sub.packageVar, is declared in class Q which is in the def package. To access this member the class attempting the access (Tester, in this case) must be in the def package. Remember that since Tester does not have a package statement it is placed in Java's "default package". The following compiler error results:

Tester.java:33: packageVar is not public in def.Q; cannot be accessed from outside package

sub.packageVar = 14; // Line 14

^

sub.protectedVar = 15; // Line 15

You may want to refer back to the descriptions of lines 7 and 11. Actually, this line compiles cleanly for exactly the same reason line 11 compiled cleanly. Remember that class Sub extends class Tester. Consequently, any instances of Sub are also considered instances of Tester.

sub.publicVar = 16; // Line 16

As was the case with lines 4, 8 and 12, public members are available "everywhere". Consequently, Tester's reference to sub.publicVar compiles without error.

最重要的是说清楚了protect的范围。看起来,挺累,但真的讲得很好。

#### 类的设计

类的一致性原则：

不变式： 一个类的不变式是指一种状态，即在任何时候该类的每个对象都满足条件（当这个对象处于平衡状态时）。

协变性：每个子类操作的前置条件不应强于其超类的前置条件。如：子类方法的参数范围，必须大于其超类类的相应函数的参数的取值范围。

抗变性：每个子类操作的后置条件不应弱于其超类的后置条件。如：子类方法的返回值范围，必须小于其超类的相应函数的返回值范围。

使用“是什么”和“有什么”的面向对象关系：

“是什么”定义了一个超类和一个子类之间的一种直接关系：子类是超类的一种。

“有什么”描述了一个对象的一部分和另一个对象之间的关系，它们通常是不同的类型：一个对象的部分是另一个对象。

### 对象

从类建立对象的过程叫做实例化（instantiation），创建出来的对象叫做实例（instances）。类实例是对象的特定实现。所有的对象都必须创建

；对象是代码与数据的组合，不同的对象共享代码区，但有不同的数据区。

对象是信息的容器，对象所包含的信息就是其数据（实例变量）

。对象的行为是由该对象接收的信息和它所执行的动作所实现的；对象的状态是由其实例变量的当前值决定的。

#### 预定义的对象实例

* **null**

null in Java cannot be a primitive value, it is for reference only. 0 in Java can only be a number, and it cannot be an object reference.

null：对象被声明为null意味着变量不被分配任何值。一般来说，使用null来代表尚未创建的对象。但null对象可以访问静态成员变量或静态方法。

例子：

class ST {

static int n = 3; //静态变量

static int getInt() { return n + 35; } //静态方法

public static void main(String[] args) {

ST st = null; // st has the type of ST

// ST st; //not compilable

// the following two are equalient

System.out.println(st.n); //OK 3

System.out.println(ST.n); //perfect 3

System.out.println(st.getInt()); // 38

// the above lines are equivalent to the followings

System.out.println(ST.n); // 3

System.out.println(ST.getInt()); // 38

}

}

It is the class type, it doesn't need a "reference variable ". You can use a object reference to call a static method, of course, provided it has the right type.

However, you don't have to. The above example clearly said that. t is not referring to anything but null, but has a MyClass type.

* **this**

this：使用this可以显示地引用某对象的当前实例。

* **super**

super：super是对超类的引用。它通常被用做当前类的超类中某成员的引用方式。

* **.class**

public class MyClass{

public static void main(String[] s) {

MyClass m = new MyClass();

System.out.println(MyClass.class); // class MyClass

System.out.println(m.getClass()); // class MyClass

}

}

#### 对象的引用

在程序里有obj1和obj2两个对象，当使用obj2 = obj1时，obj1和obj2被认为是同一个变量。运算符=并不将一个对象复制到另一个对象中，而是把两个变量指向同一个对象。

import java.awt.Point;

class ReferencesTest {

public static void main(String[] arguments) {

Point pt1, pt2;

pt1 = new Point(100, 100);

pt2 = pt1; //创建一个从pt2到pt1的引用，而不是把pt2创建成一个从pt1复制过来的新对象；pt2是一个与pt1对象相同的引用，两者中的任何一个都可以用来引用该对象或者改变它的变量。

pt1.x = 200;

pt1.y = 200;

System.out.println("Point1: " + pt1.x + ", " + pt1.y);

System.out.println("Point2: " + pt2.x + ", " + pt2.y);

}

}

输出： Point1: 200, 200 Point2: 200, 200

引用与调用的区别：被引用的类在引用它的类被实例化的时候也被实例化；被调用的类只有在它被使用时才被实例化。

#### Pass-by-value还是Pass-by-reference

DO NOT MIX pass-by-reference and pass-by-value.In Java, every thing is passing by value.

**Pass-by-value** : You make a copy of the parameter (argument), and use it inside of the methods. The value of the parameter (argument) will not be affected outside of the method.

**Pass-by-references** : you do NOT make a copy of the parameter, you use the parameter itself!

some authoritative quotation to support the saying "In Java, everything is pass -by-value"

**A:** Some people will say incorrectly that objects are passed "by reference." In programming language design, the term pass by reference properly means that when an argument is passed to a function, the invoked function gets a reference to the original value, not a copy of its value. If the function modifies its parameter, the value in the calling code will be changed because the argument and parameter use the same slot in memory. The Java programming language does not pass objects by reference; it passes object references by value. Because two copies of the same reference refer to the same actual object, changes made through one reference variable are visible through the other. There is exactly one parameter passing mode -- pass by value -- and that helps keep things simple.

**B:** In Java methods, arguments are passed by value. When invoked, the method receives the value of the variable passed in.

When the argument is of primitive type, pass -by-value means that the method cannot change its value. When the argument is of reference type, pass-by-value means that the method cannot change the object reference, but can invoke the object's methods and modify the accessible variables within the object.

This is often the source of confusion--a rogrammer writes a method that attempts to modify the value of one its arguments and the method doesn't work as expected.

Let's look at such method and then investigate how to change it so that it does what the programmer originally intended.

// PassByValueTest.java

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

public class PassByValueTest {

public static void main(String [] args) {

String arr[]=new String[2];

arr[0]="hello";

arr[1]="hi";

// nothing will change here

swap(arr[0], arr[1]);

System.out.println(arr[0] + ", " + arr[1]); //hello, hi

// two Strings are actually swapped

swap(arr, 0, 1);

System.out.println(arr[0] + ", " + arr[1]); //hi, hello

}

// useless swap method, since you swap the copies of two strings' reference

// inside the method Strings outside are not affected.

public static void swap(String s1,String s2){

String tmp = null;

tmp = s1; s1 = s2; s2 = tmp;

}

// real practical swap here: swap array elements with two indices. very

// useful for sorting algorithm, etc. reference of arr will never change (pass-by-value)

// but the contents of arr will change permanently

// since the copy of arr reference still refer to the same object.

public static void swap(String arr[], int ix1, int ix2) {

String tmp = null;

tmp = arr[ix1]; arr[ix1] = arr[ix2]; arr[ix2] = tmp;

}

}

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

class ValHold{ public int i = 10; }

public class ObParm{

public static void main(String argv[]){

ObParm o = new ObParm();

o.amethod();

}

public void amethod(){

ValHold v = new ValHold();

v.i=30;

another(v);

System.out.println(v.i);

}

public void another(ValHold v){

v.i = 20;

ValHold vh = new ValHold();

v = vh;

System.out.println(v.i);

}

}

系统输出为：10 20

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

Java passes arguments only by value. When an object instance is passed as an argument to a method, the value of the argument is a reference to the object. The contents of the object can be changed in the called method, but the object reference is never changed.

public class Test{

public static void main(String args[]){

String str=new String("World");

char ch[]={'H','e','l','l','o'};

change(str,ch);

System.out.println(str + "and" + ch);

}

public static void change(String str, char ch[]){

str="Changed"; ch[0]='C';

}

}

What is the result after execution?

输出结果：World and Cello

### 数据类型

#### 八种基本数据类型

Java不进行所冲突数据类型的自动化转换，Java编译器会检测所有的表达式和参数以确保所有的类型都是兼容的。

变量的声明与C++完全类似，但是没有联合、结构和指针。Java没有与C++中typedef对应的东西，要在Java里声明新的类型，可以声明一个新的类，然后变量就可以用该类作为它们的类型。

Java中有8种基本数据类型（不表示对象）：

byte:：8位，取值范围是 -128 ~ 127，不是0~255

short：16位，取值范围是 -32,768 ~ 32,767

int：32位，在C/C++中，int意味着整数类型大小依赖于目标机器，可能是16或32位；Java数值类型的大小是与平台无关的。取值范围为：

-2,147,483,648 ~ 2,147,483,647

long：64位，长整数需使用后缀大写L或者小写l（建议使用大写L），比如：88888888L；

取值范围为：-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807

整型数的位数是从 -2^(位数-1) 到2^(位数-1)-1。针对整型数，编译器检查数的取值范围。

float：单精度，32位浮点数。取值范围为：-3.40292347E+38 ~ 3.40292347E+38

double：双精度，64位浮点数。Java中浮点数默认情况下是double类型，如：3.02。如表示为float加后缀F，如：3.544F。

针对浮点数，编译器检查数的精度（d还是f）和范围。note that floor(double), ceil(double) and rint(double) return doubles not ints

Java中的八进制数是以0开头的，如：0123，十六进制是以0x开头的，如：0xAB。

负整数是用补码表示的。补码：取反加1。

Don't forget in Java, except char, every number is signed. ~3 = -4

~3 is ~(0000 0000 0000 0000 0000 0000 0000 0011)

or 1111 1111 1111 1111 1111 1111 1111 1100 is -4

思考下面代码的输出：

public class Abs {

public static void main (String args[]) {

System.out.println(Math.abs(Integer.MIN\_VALUE)); //-2147483648

System.out.println(Math.abs(Integer.MIN\_VALUE+1)); //2147483647

System.out.println((byte)Math.abs(-128)); //-128

System.out.println((byte)Math.abs(-127)); //127

}

}

例子：

数字的字面值默认是double或者是int的。

float f = 18; //编译正确

float f = 3.2; //编译报错，应改为float f = 3.2F;

float f = 1/3; //编译正确，f = 0

int i= f; //编译报错，应改为int i = (short)f

long temp = (int)3.9; //temp的值为3

long test = 012; //编译正确，十进制数为10（一个八进制数）

//将long变为float不用使用强制类型转换的方法 Converting from long to float is not a narrowing conversion. Even it may lose some significant digits.

long longNumber = 123456789;

float floatNumber = longNumber; //编译正确

//42e1 是double型的；0x0123 是int型的

class My{

static char c; static byte b; static long l; static float f;

public static void main(String[] ss) {

b=c; //not compilable （存在正负数的问题）

c=b; //not compilable either

l=f; //not compilable

f=l; //OK

}

}

若一个浮点数被零除，java不会报告出错，比如：System.out.println(1/(1.0/0))能够正常的输出0.0，而不报溢出；但整数除零在运行时会报错Exception。

Java中没有提供任何unsigned（“无符号”）类型，Java所有的整数都是有符号的。因此，byte字节类型的变量取值范围是-128~127,不是0~256。

当把一个超过变量所能容纳的整数值赋给该变量时，它会发生溢出（overflow），在这种情况下，数字从一个范围末端绕回到另一个的开始。

例如：一个int变量从2147483647（可接收值）变为2147483648（不可接收值）时，它会绕回到最低的可接收值-2147483648。int + int is supposed to be int.

例如：一个byte变量从127变为128时，会报编译错误。127 + 1 results 128, which is an int operation too. The result is out of range of byte. Compile time error!!!

char：16位，用于单个字符，是用两个字节表示的无符号整数（Unicode），取值范围为：0 ~ 65535（或者为：’\u0000’ ~ ’\uFFFF’ ）。char类型不存在负的字符值。从int转换成char需要强制转换；但char转换成int类型，无须强制转换。char类型的

变量不是整数，但可以把它像整数一样使用。可以把两个字符累加，并且递增某字符的值。

任何被声明为char类型的类变量或实例变量的缺省值是'\u0000'。不要把'\u0000'和空白字符相混，空白字符的值是'\u0020'。

boolean：只有两个值：true和false。不能与任意一个数据类型相互转化，不能进行算数运算。Boolean的缺省值为false。

boolean在内存中占用一个byte来存放，虽然它只使用了1 bit，另外7 bits没有使用。

CASTING and conversion

1．基本类型的conversion

1.1 boolean 型的不能被converte成别的类型。

1.2 向外扩展可以converte。窄化不能converte，得强制转化。

2．字面值的conversion

数字的字面值默认是double或者是int的。窄化赋值错。

float f = 1.234 // 错！

float f = 1.234f //对

int的字面值可以窄化给byte,char,short。

int i=10 //对

byte bb = 10 //对

byte b = i; //错

#### 数组

数组：我们所定义的数组都是Object类的一个子类的实例。数组本身是一个对象，用new来创建数组（You must explicitly instantiate each array, which is not as efficient as allocating one big chunk of memory.），并为它分配合适的大小。Java中的数组是在运行时分配的对象，所以可以用变量来设置它们的长度，而这在C++中时不可能的。

The number of elements in an array is stored in the length attribute in the array object.

初始化数组时用的花括号只有在数组被声明时才用的着。这就是说，花括号只能作为初始化程序使用。例如：

double[][] identityMatrix = {{1.0, 0.0, 0.0}, {0.0, 1.0, 0.0}, {0.0, 0.0, 1.0}};

double[][] identityMatrix = {{1.0, 0.0, 0.0}, {0.0, 1.0, 0.0}, {0.0, 0.0, 1.0},};

注意：可以在多维数组初始化的最后一行放一个逗号。

Java的一维数组实际相当于C++的指向一个堆数组的指针。

换言之： int [] numbers = new int[50];

for (int index=0;index <50; index++)

System.out.println(numbers[index]); //将输出50个0，数组被初始化

System.out.println(numbers.length); //将输出50

并不等价于： int numbers[50];

而是等价于： int \* numbers = new int[50];

int iA[3]; //非法

int ii[] = new int[] { 1,2,3 }; //合法

int ii[] = new int[3]; //合法

int[] ii = { 1,2,3 }; //合法

Arrays are objects in Java. Array elements are just member variables of the class, and they are initialized to default value as any other class of objects.

MyObject[] myobjarray;

There is no array here, only an array reference type variable called myobjarray. If it is a field in another class, myobjarray will be initialized to null. If it is in a method, it will not be initialised. No exceptions here!!!

MyObject[] myobjarray = new MyObject[5];

You are initializing myobjarray by constructing a MyObject[5] array, inside the new object instance of the MyObject[5] array, all fields will be initialized to default values, which are myobjarray[0] = null, myobjarray[1] = null, ... and the length field would be initialized to 5. No exceptions here either!!!

Method for creating an array of five empty Strings:

String a [ ] = new String [5]; for (int i = 0; i < 5; a[i++] = "");

String b [ ] = {"", "", "", "", ""};

To create a 3x3x3 array of integers, you would do the following:

public final class Matrix {

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

int i, j;

int[][] matrix[];

matrix = new int[3][][];

for(i = 0; i < matrix.length; i++) {

matrix[i] = new int[3][];

for(j = 0; j < matrix[i].length; j++)

matrix[i][j] = new int[3];

}

}

}

Which one of the following is correct to create an array?

A. float f[][] = new float[6][6];

B. float []f[] = new float[6][6];

C. float f[][] = new float[][6];

D. float [][]f = new float[6][6];

E. float [][]f = new float[6][];

Explanation:

In Java the declaration format allows the square brackets to be at the left or right of the variable name. But the new float[][6] is illegal.

Correct Answer: A,B,D,E 15 of 60

Q. How to create an array with actually 3 HashMaps in it?

A: See the following code snippets

HashMap[] maps = new HashMap[3];

for (int i=0; i<3; i++) {

maps[i] = new HashMap();

}

//or

HashMap[] maps = new HashMap[] {new HashMap(), new HashMap(), new HashMap()};

### 流程控制、运算符和Java赋值操作

#### 流程控制

* **switch**

switch (x){ }中的x只对原始类型byte、char、short和int起作用，但不能是long，float，double和boolean或者对象引用，每一个case语句中的值都应该是常量。A switch argument can be any type that can implicit-cast to an int (byte/char/short/int but not boolean or long). The argument and cases can also be compile-time constant expression.

要比较多个字符串（字符串在Java里是对象），必须使用嵌套的if语句。

例如1：

char mychar = 'c';

switch(mychar){

default :

case 'a' : System.out.println("a"); break;

case 'b' : System.out.println("b"); break;

}

该程序段的输出是：a

首先default子句是完全有效的。因为在default子句后的结尾没有break语句，随意如果没有于a或b匹配的选项，则a总在运行。

例如2：continue在switch-case中的例子：

public class A {

public static void main(String args[]) {

int x = 0;

while (x < 10) {

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

switch (x) {

case 5: x += 2; continue;

case 2: break;

case 7: break;

default: x++;

};

x++;

}

}

}

正确的输出为：0 2 3 5 7 8

* **if和while**

if和while语句的表达式一定要是一个boolean值。C程序员习惯于使用0或者null作为false。

\* if (...)

statement1;

statement2; //...always executed

boolean m = true;

if ( m = false )

System.out.println("False");

else

System.out.println("True");

输出为True.

The expressions in the block of do/while loop will be executed at least once. If the condition of this loop is not met the loop will stop after once execution, otherwise, it will continue to loop until the condition is no met.

* **标号labels**

在循环中使用标号。

Labelled break and continue statements:

- The label referred to by a labelled break statement is attached to a statement block, which could be a loop or switch but doesnt have o be.

- The label referred to by a labelled continue statement must be ttached to a loop.(A "continue" statement must be enclosed in a "while", "do" or "for" statement.)

考虑：

void looper(){

int x=1;

while (x <=10){

two:

System.out.println(++x);

if (x!=5) break two;

}

}

该段程序不能编译，因为标签two: 到break two:之间不存在循环（while、for等）。

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*注释说明了label的所有用法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

public class Test {

// lbla: //compilable Error: identifier expected

public static void main(String[] args) {

// lblb: //compilable Error: A declaration cannot be labeled

Object o = new Object();

int n = 1;

lblc:

if (o instanceof Object) {

while (true) {

n++;

if (n > 5) break lblc;

System.out.println(n);

}

}

System.out.println("Finished n");

lblx:

for (int i = 3; i < 7; i++){

if (i==5) continue lblx; // skip number 5

System.out.println(i);

}

lbly: // legal before another lable

lblz:

{

System.out.println("Watch:");

break lbly;

// compilable Error: statement not reached

//System.out.println("This will never be printed out.");

}

}

}

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

#### 运算符

for循环语句中可以使用逗号(,)，例如：

public class CommaOperator {

public static void main (String[] args) {

for (int i = 1, j = i + 10; i < 5; i++, j = i \* 2)

System.out.println("i= " + i + " j= " + j);

}

}

则程序的输出依次为： i= 1 j= 11

i= 2 j= 4

i= 3 j= 6

i= 4 j= 8 //好好理解for语句是如何循环的

//for循环的执行顺序

for (语句a;条件b;表达式c){

语句d

}

语句a只在第一次执行。b为真则继续，执行语句d。c在语句d后执行，回到条件b，为真则再继续。

&&（逻辑与）和||（逻辑或）运算符被称为短路运算符，是因为它们在返回结果前并不总是计算第二个操作数。例如：

class Exp {

public static void main(String[] args) {

int i = 10;

int j = 12;

if (true & true) //该表达式是正确的

if ((i < j) || (i = 3)) //应为if ((i < j) || (i = =3))

System.out.println("hello");

}

}

问题出在i = 3，它的结果是一个整数i，i的值被设为了3。因为对于运算符||来说，整数不是合法的参数，所以这段代码编译不通过。

移位运算符面向的运算对象是二进制的位。左移位运算符（<<）能将运算符左边的运算对象向左移动运算符右侧指定的位数（在低位补0）。有符号右移位运算符（>>）则将运算符左边的运算对象向右移动运算符右侧指定的位数。有符号右移位运算符（>>）使用了符号扩展：若值为正，则在高位插入0；若值为负，则在高位插入1。无符号右移位运算符（>>>），它使用了零扩展：符号位保持并向右移动，无论正负，都在高位插入0。该运算符C或C++没有的。若对char，byte或者short进行移位处理，那么在移位进行之前，它们会自动转换成一个int。

There are two right shift operators in Java. They are >> and >>>. >> is the arithmetic(signed) right shift operator and >>> is the logical(unsigned) right shift operator.

<<.和>>>运算，补0。

>>的时候，正数补0，负数补1。

instanceof运算符可以检验某对象是否是某类或该类的子类的实例；还可以检验某对象的类或子类是否实现了某接口。对于instanceof操作符，如果第一个操作数是null，它就会返回false，而不管随后出现的操作数类型。如果通过一些有效类（其中，某特定对象不可能从instanceof中指定的某类继承的话）可以决定的话，则编译器将不会允许这些代码通过编译。

instanceof 操作符比较一个类是否出去另一个类（或者超类）。

There is a very important rule when using instanceof.

It is not legal to attempt to compare an instance to a class

if the instance is not in the same object hierarchy as the class.

If we attempt to compare an instance with a class that is not part of the hierarchy,

the compiler will produce an error.

it is legal to test whether a null object (or null itself)

is an instance of a class. This will always result in false,

例如：

class Tree { }

class Pine {

public static void main(String[] args) {

Pine[] p = new Pine[1];

if (p instanceof Tree[])

System.out.println("p inherits from Tree[]");

}

}

编译器将解释为Pine[]不能从Tree[]处继承。

“= =”和“equals”的区别，现有如下选项：

A. String a1 = “ABC”; Object a2 = “ ABC”;

B. Boolean a1 = new Boolean(true); Boolean a2 = new Boolean(true);

C. Object a1 = new Object(); Object a2 = new Object();

D. Boolean a1 = new Boolean(true); Object a2 = new Boolean(true);

使if(a1= = a2)为true的是(A)；使if(a2.equals(a1))为true的是(ABD)

涉及到两个知识点：String在内存中的组织；“= =”和“equals”的区别：equals()方法用于检测一个对象的值，= =运算符用于检查对象引用本身。

String s1 = “ab”;

Object o = s1+”c”;

String s = “abc”;

boolean b = o.equals(s);

Object的equals()方法是检查对象的引用，String的equals()方法是检查对象值。在下面的代码中：boolean b = o.equals(s);

变量o指向一个类型为String的对象。因此，o.equals(s)调用的是String类的equals()方法，而不是Object的equals()方法。

++,--后缀的优先级最高，其次是++,--前缀。

x= ++x + x++ 是先做x++,再做++x.

#### Java evaluating from left to right

**Java evaluating from left to right.** The rule "Evaluating from left to right" is for resolving the addresses and evaluation of expression.

It works this way:

int x = 0;

x = x++;

1) Get value of x (0 for now), put it somewhere

2) increase the value of x (x value is 1 now)

3) assign the value stored somewhere back to x (x value is 0 again)

A better example to illustrate the concepts is:

int y = 5;

int y = y++ + y++; //(the y value will be 11)

1) Get value of 1st y (5 for now), put it somewhere

2) increase value of y (y value is 6 now)

3) Get value of the 2nd y (6 for now), put it somewhereelse

4) increase value of y (y value is 7 now)

Attention: from 1 to 4 is strictly left to right. The evaluation of the right hand side expression is finished now. However, there is no such rule in C language.

5) adding the value in somewhere and some where else together. assign it to y

(y = 5 + 6 = 11)

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

class ArrTest{

public static void main(String args[]) {

int i = 0;

int[] a = {3,6};

a[i] = i = 9;

System.out.println(i + " " + a[0] + " " + a[1]); // 9 9 6

}

}

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

This is another good example the great Java evaluation rule applies. Java resolves the addresses from left to right. a[i] which is the address of a[0], then i which is the address of i, then assign 9 to i, then assign 9 to a[0].

再给出一个例子：

The output is: Determine the result of attempting to compile and run the following code:

public class Tester {

public static void main(String[] args) {

int i = 10, j = 3, k = 7;

int p = 30;

p += k -= j <<= i %= 4;

System.out.println("i=" + i);

System.out.println("j=" + j);

System.out.println("k=" + k);

System.out.println("p=" + p);

}

}

Answer 6

Operator Associativity

The output is:

i=2

j=12

k=-5

p=25

Remember that the various assignment operators in Java (=, +=, -=, \*=, /=, %=, <<=, >>=, >>>=, &=, ^=, |=) associate right to left. In other words, the assignment statement in the program above is executed as if it had been written:

p += (k -= (j <<= (i %= 4)));

这个不难，从左到右就好了。

public class Tester {

static int x;

public static void main(String[] args) {

int i = 1;

int j = (i + (i = 4)) \* (i = 3);

System.out.println(j);

int y = z() \* (z() + z());

System.out.println(y);

}

private static int z() {

return ++x;

}

}Answer 9

Order of Evaluation I

The output is:

15

5

This question illustrates the fact that Java evaluates the operands of each operator in a left-to-right manner. In other words, before an operator can be applied to it's operands the left operand must be evaluated first. After the left operand has been evaluated the right operand is evaluated (unless the operator is one of the conditional operators &&, ||, and ? :, in which case the right operand may not be evaluated). Keep in mind that either the left or right operand may itself be an expression containing several operators.

Consider the expression (i + (i = 4)) \* (i = 3) from the question. Before the multiplication operator can be applied its left operand must be evaluated. This operand is i + (i = 4). This operand is an expression consisting of an addition operator whose left operand is i and whose right operand is (i = 4). Since Java evaluates left operands before right operands, the addition operator "sees" its left and right operands as 1 and 4, respectively. The result, 5, then serves as the left operand for the multiplication operator.

The multiplication operator's right operand is another expression, i = 3, containing an assignment operator (=). Although the evaluation of this expression is trivial, you should note that Java will still evaluate the left operand of the assignment operator (the variable i) before evaluating the right side (the integer literal 3). After both sides have been evaluated the assignment operator is applied. The result is, of course, 3. Once both operands have been evaluated the multiplication operator can be applied. In this case the result is 15.

The evaluation of variable y in the question follows the same pattern of evaluating the left operand of the multiplication operator before the right operand. The assignment of y is then evaluated as: 1 \* (2 + 3).

这个很有用吧？我以前不清楚的。

**Q. Why the following code print out "BAC", instead of "ABC"?**

// C.java

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

class C {

public static void main(String arg[]) {

System.out.println("A"+new C());

}

public String toString() {

System.out.print("B");

return "C";

}

}

// output:

// BAC

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

**A:** You need understand concepts here: Java left-to-right evaluation rule and side effect.

Java evaluates the expression

**"A"+new C()**

following the same rule. It gets "A" first, which is a String literal, put it somewhere. Then it evaluate

**new C()**

it construct a C Object first, then invoke toString() method of C Object, and gets the value of C object, which is "C", then concatenates "A" and "C" together, and println "AC".

Where does "B" come from? Inside the toString() method of C Object, there is a

**System.out.print("B");**

which is invoked when Java evaluate the above expression. It is printed out before the evaluation completed. That is why "B" is printed first.

### 断言

断言

1．assert不具有继承功能

2．断言的使用：public的方法不应该使用断言才测验precondition，private方法可以。

对于postcondition，建议使用assert。

不要在assert的语句中修改变量值。可能导致错误。

### 接口 Interface

#### 接口的定义与使用

接口的目的：依靠实现某接口来定义你的类，会把该类的对象标识为该改接口的实例。这意味着可以把接口用做所期望行为的某种暗示器。对于接口来说，在其中定义包含相关的行为是接口的基本思想。例如在Java中，FocusListener接口中定义了两个方法：focusGained()和focusLost()。

接口的用途：管理类之间的相似性，而不必建立类的关系；定义多个类希望实现的方法；开放对象的编程接口，但不开放对象本身。如果某个公有接口被声明在一个源文件中，则接口必须与源文件具有相同的名字。

接口是一个纯粹的设计，只进行变量和抽象方法的定义，没有任何实现细节，不能被实例化；主要是为了对外提供统一的接口，隐藏实现的细节；必须用类来实现它。注意当你包含了某个接口之后，你必须实现该接口中所有的方法，你不能挑选你所需的方法去实现。

接口中声明的所有（任何）方法隐含为public和abstract，不能在一个接口内将方法声明为private或protected，也不能将方法声明为static或final，也不能为strictfp、native或者synchronized。接口中声明的变量隐含为public、static和final，不能为private、protected、volatile、transient和synchronized；变量必须有初始值（即不能在接口中设置变量）

。接口中声明的所有类隐含为static。

因此，在类中实现接口时必须使用public修饰语。接口中所有变量在实现接口的派生类中均当成常量来使用。

实现多个接口可能导致复杂性。若两个不同的接口定义了相同的方法，可以用下面的方法来解决这个问题：

1．如果每个接口中的方法都有相同的格式，在类里实现一个方法，该定义在这两个接口中都适应；

2．如果该方法具有不同的参数列表，那么它是一个简单的方法重载：你要实现两种方法格式，每一个定义只适用于它自己的接口定义；（相当于多态性）

3．如果方法具有相同的参数列表，但返回值不同，则你不能创建一个适合两个接口的方法。存在接口设计上的问题，需重新设计。

接口必须有公共的或包保护。要注意不带修饰语public的接口不会自动将它们的方法转换成public和abstract，而且也不会将它们的常量转换成public。非public的接口有的也是非public的方法和常量，它们只能被同一个包中的类和其它接口所使用。

接口内方法的参数：通过将你的方法参数定义成接口类型，你可以创建通用的参数，它们适用于可能实现这个接口的任何类。然后在某个类实现这个方法时，你可以采用通用的接口参数，并强制转换成相应的对象。

public interface Fruitlike{

public abstract germinate(Fruitlike self);

}

class Fruit implements Fruitlike{

//……

}

public class Orange extends Fruit{

public germinate(Fruitlike self){

Orange theOrange = (Orange)self;

//……

}

}

在大多数情况下，在任何可以使用类的地方，也可以使用一个接口来代替。与类不同的是接口层次可以多重继承。

**Interface与Static的比较：**

1) Inner interfaces are implicitly static, no matter you put static modifier or not. Attention: inner interfaces is very rarely used.

2) Outer interface are **NOT** static, just like outer class.

3) All members (attributes & methods) of interface are implicitly public.

4) All attributes defined in interface are implicitly static and final.

5) All methods defined in interface are **NOT** static.

6) All classes defined in an interface are implicitly static

Interface can be extended by another interface, but not by a class. For anonymous class, it can be confusing. e.g.

interfacename ref = new interfacename() {

//implementation code here

}

The new anonymous class defined here is still implementing the interface named interfacename, not extending the interface.

**Q. Can we pass an interface as parammeter to a method? If yes, how does it work?**

**A: Yes, definitely.**

**It is not only correct, but also is recommended. Since the implementation class can change, but the method will stay the same. There are tons of excellent examples in the Java collection framework. It is also very useful to implement some design patterns such as FactoryMethod Pattern.**

**How to use it? When you make the method call, you must pass a concrete implementation of the interface to it.  
See the following example.**

**// TestFace.java**

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

**interface IFace{**

**void drawFace();**

**}**

**class RealFace implements IFace {**

**public void drawFace(){**

**System.out.println("I'm drawing a beautiful face");**

**}**

**}**

**public class TestFace {**

**public static void callIFace(IFace face) {**

**face.drawFace();**

**}**

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

**IFace face = new RealFace();**

**callIFace(face);**

**}**

**}**

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

#### 充分理解接口的含义用

Understanding Interfaces

An interface describes the public methods that a class should implement along with the calling conventions of those methods. An Interface says nothing about implementation of its methods. In essence, an interface is a contract of guaranteed operations, or behaviors. A class that implements an interface must provide code to execute when the methods of the interface are invoked.

An object of class that implements an interface can be cast into the type of that interface, then be passed to other objects under the interface’s type. This allows for finer abstraction with class design.

For example, the interface java.awt.event.ActionListener has a method public void actionPerformed(ActionEvent e). A class that implements ActionListener as in:

public class AClass implements java.awt.event.ActionListener{ ... ... }

should provide a method body for public void actionPerformed(ActionEvent e). This can be an empty body - a do-nothing implementation. An instance object of AClass can then be cast into java.awt.event.ActionListener and passed to other objects. The receiving object will only have to know the contract method of java.awt.event.ActionListener, namely, public void actionPerformed(ActionEvent e), which they can call when an action is happened. This way, all the specific behaviors of the instance object of AClass can be kept away from objects that receive AClass as an ActionListener and not as AClass.

Implementors of Java have also used empty interfaces to tag objects. These interfaces are referred to as marker interfaces. For example, the interface java.rmi.Remote has no method signatures. It serves only to identify remote objects. The interface java.lang.Cloneable which serves as an indication to the clone method in class Object that it is legal for that method to make a field-for-field copy of instances of a class that has implemented java.lang.Cloneable.

#### 克隆接口（Cloneable）

使用Cloneable接口：为了创建某对象的副本，可以调用clone()方法（定义在类Object中的protected方法）。并不是所有的类都允许克隆其对象，只有那些实现了Cloneable接口的对象才可以被克隆，否则将抛出CloneNotSupportedException异常。Cloneable接口并不定义任何必须实现的方法。当克隆对象时，Java虚拟机不会调用对象的构造函数。克隆仅仅是原对象的一个准确副本（make a field-for-field copy of instances of that class）。

克隆对象可能产生一些不可预知的副作用。如果创建了引用其它对象的一个对象，则该对象的克隆也会引用相同的对象。对象克隆也许会对被引用的对象作更改，这种变化接下来可能会被原对象反映出来。

Cloneable接口不声明任何方法；也不让实现者实现任何东西。任意被确定为Cloneable实例的类让Object类的clone()方法知道该类能够被克隆。该Cloneable 接口只是识别开发任意的意图，并不使开发人员实现任何行为。The clone() method always returns an object of type Object, you must cast it to the actual type of the object you are cloning.

一个对象只能请求其它对象的克隆，其中后者所在类与被克隆对象属于同一类型，或者是被克隆对象的子类。意思是只允许方法请求某种对象的克隆，该对象满足在类层次中该对象的类与本身处于相同分支上的条件。

例子：

class Tree {

protected int age;

}

class Pine extends Tree implements Cloneable {

Pine() {

age = 2;

}

}

class Maple extends Tree {

public static void main(String[] args) {

Pine p1 = new Pine();

Pine p2 = new (Pine)p1.clone();

System.out.println("Pine is " + p2.age);

}

}

编译会出错，因为Maple与Pine在类层次中不属于同一分支，clone（）被保护，而且定义clone()的Object类来自另一个不同的包。

对Clone的理解：

The key concept you need to understand is the default implementation of clone() is a shallow clone(). The primitive type is cloned, for example, the hashCode() (type long) of the original and the clone will have the same value. If you change the clone's primitive type field value, the original will stay the same as before.

But the reference type just clone the same object's reference. Therefore when you make a change in the data member object, the original and the clone will be both changed.

I learned this the hard way. Then I had to write my deepClone() method to do my job.

### 包

按照两种重要的方式创建包有助于使自己的API干净整洁：能够更好地限制对方法和变量的访问；能够定义子系统。

包是类和接口的容器，被用于把名字空间组织成松散的层次。包有助于把类和特定功能联系起来。包声明package必须位于Java源文件中第一个非空行（注释除外），然后才是import语句。类只能属于一个包，因而类文件只能有一个包声明。

包命名规范：包名的首字符是小写的字母（Sun's convention: package name starts with a lowercase letter.），这样做可以在查找到类的显示引用时能够更加容易地区分包名和类名。这就是为什么类名按照规范应该是以大写字母开头的原因。

每个包名必须是唯一的，这样才能最大程度上使用包。如果存在相同的包名，则会出现导致运行时错误的命名冲突问题。对于大型的项目，绝对应该在项目的初期就应该指定命名规范，以免出现命名混乱的现象。

package告诉Java编译器将该文件中的类分配给该命名包。包的名字决定了要编译的类的一个目录结构。例如在文件开始出有： package awt； 说明这个文件里你所有的类都应该包含在自己定义的awt包中。

import只是指明Java编译器在哪里能够找到某个类，它并不进行任何操作来扩大某个类的大小。导入语句的主要思想是帮助Java找到合适的方法，并且避免名字空间的冲突。当使用一个带有星号的import语句时，导入的只是该包内的（public）公共类，对程序的运行性能没有任何影响，只是增加了编译的时间。只有public类才能被该类所在的包外的成员访问。

在Java中，核心函数都位于java.lang包，并且该包是自动导入的。

包使得别人能够充分利用你的类。只有public类才能被该类所在的包外的成员访问。可以按照四种方法为一个类的成员指定访问控制：public、protected、private和默认值。

在默认情况下，你的所有类均可以引用同一包中的其它类，并且所有方法均可以调用同一包中的其它方法，以及访问同一包中的变量。public和protected不影响同一包中类和成员之间的访问；private限制对定义了私有成员的类的访问；如果有可能其它某个包将会为你所定义的类定义一个子类，并且你还希望该类的子类能够访问该类中具体的成员，那么可以将这些成员设置为protected。

应该牢记：不写访问控制关键字不应源自于懒惰，而是由你的设计意图决定的。

对于包，你可以完全限制类名以便Java知道在哪里能找到这个类。这使得你能够使用与其它包中定义的类相同的类名。

## Java JDK package

### java.lang和java.util

包装类Boolean、Byte、Character、Double、Float、Integer、Long和Short。除了Character之外的所有包装类都有一个叫做valueOf()的静态（static）方法，它会解析字符串，构造一个相同的类类型的包装类，然后返回包装类。可以用equals()方法检查两个包装类是否相同。

ceil()会找到下一个最大整数；floor返回紧邻的最小整数；round()返回与某浮点数值最接近的整数值，random()返回一个在0.0到1.0之间的双精度数。

#### String和StringBuffer类

String类支持静态（不变的）字符串；StringBuffer类则支持可修改的字符串。String和StringBuffer类都支持16位Unicode字符。

String类（String类使用关键字final定义，不能被继承）是相对稳定的类，一旦初始化后就不可以改变。具有不可变性（immutable），它的任何修改（substring、replace、trim、+、toUpperCase等）操作都会产生新的对象。只有+和+=两个运算符被String类重载。

A JVM has a string pool where it keeps at most one object of any String. String literals always refer to an object in the string pool. String objects created with the new operator do not refer to objects in the string pool but can be made to using String's intern() method. Two String references to 'equal' strings in the string pool will be '=='.

有下列语句：

String s1 = “Abc”; String s2 = s1;

s1 +=”xyz”; s2.toUpperCase();

System.out.println(s2);

程序输出结果是\_\_\_\_

A. ABCXYZ; B. Abc; C. ABC; D. ABCxyz. 答案：（B）

String类的不变性，用String类创建的字符串是只读的。当你将一些字符用运算符”+”加到某String对象上时，那么结果是一个新的String对象，而不是原来String对象的一个修改值。

在Java中每个String字面值，像String str1 = ”Hello, World!”; 都是由一个String实例来表示。Java类可以拥有字符处缓冲池。当一个包含字符串文字的程序进行编译时，编译器通常是将这个String文字加到这个类的一个缓冲池中。然而，如果这个类中已经存在了这个字符串文字，那么编译器不会创建一个副本，而是从这个缓冲池中使用已有的字符串。这样设计是为了节省内存，而且不引起程序错误。

必须注意以下两点：

* + String字符串被放在一个特定类的缓冲池中

例如：String str1 = ”Monday”; String str2 = ”Monday”;

则有：str1 = = str2（true）并且str1.equals(str2) （true）

* + 当你使用一个新的String字符串构造函数String()时，一个新的String对象实例被创建在程序运行的内存中（非缓冲池中）。

例如：String str1 = ”Monday”; String str2 = new String("Monday");

则有：str1 = = str2（false）并且str1.equals(str2) （true）

例如：String str1 = new String("Monday"); String str2 = new String("Monday");

则有：str1 = = str2（false）并且str1.equals(str2) （true）

因为equals()方法必须比较字符串中的实际字符，所以这个方法执行起来比相等运算符（= =）要慢。如果一个程序中执行了很多使用equals()方法的字符串比较代码，你可能会注意到响应时间会变得更长。要解决这个问题，你可以使用String类的intern()方法将字符串加入到字符串的缓冲池中。一旦你这么做了，就可以用相等（= =）运算来做比较运算了。

String数组： String [] s = { “a”, “b”, “c” };

String类的方法：

charAt()：此方法返回子符串内位置在索引处的子符。第一个字符的位置索引是0，最后一个字符的位置索引是n-1。

substring(int beginIndex, int endIndex)：是取从beginIndex开始，到endIndex之前的子符，不包括endIndex那个子符。

将int 转换为String类型：

String S = String.valueOf(ii); // ii is an int

String S = Integer.toString(ii);

String S = (new Integer(ii)).toString();

将String 转换为int类型：

int ii = Integer.parseInt(S); // S is a string

StringBuffer类用来处理大部分字符串运算，它是可读写的。StringBuffer doesn't override equals. StringBuffer类的构造函数：

StringBuffer()：构造一个空的字符串缓冲区；

StringBuffer(int length)：构造一个具有指定初始长度的字符串缓冲区；

StringBuffer(String initialString)：根据String（字符串）对象构造字符串缓冲区。

StringBuffer类的append()方法和insert()方法被重载，可以支持任何基本的Java数据类型。StringBuffer类不是从String类中继承而来的。如果你想把用StringBuffer对象表示的字符串作为参数传递给一个方法，就必须从StringBuffer对象处获取一个String对象（通过调用toString()方法来实现）。

例如： StringBuffer strBuf = new StringBuffer("12345");

strBuf.append("67890");

结果：在执行了第二行代码后，strBuf还是引用相同的对象实例。因为StringBuffer类只能读写，strBuf.append("67890")语句只是修改了已经存在的strBuf对象，因此引用不变。

例如： StringBuffer strBuf = new StringBuffer("12345");

String str = strBuf.toString();

结果：str = = strBuf是否正确？（由于str和strBuf的对象类型不匹配，编译时出错）；判断str.equals(strBuf)？（false）尽管String和StringBuffer对象有相同的内容，但它们是不同的对象。你可以比较一个字符串和另一个字符串，也可以比较一个字符串缓冲区和另一个字符串缓冲区，但是你不能比较一个字符串和另一个字符串缓冲区。

//又一个例子Test.java

public class Test {

public static void main(String[] args) {

String s1 = new String("hello");

String s2 = new String("hello");

StringBuffer sb1 = new StringBuffer("hello");

StringBuffer sb2 = new StringBuffer("hello");

System.out.println("String: s1.equals(s2) = " + s1.equals(s2));

System.out.println("StringBuffer: sb1.equals(sb2) = " + sb1.equals(sb2));

}

}

/\* output

String: s1.equals(s2) = true

StringBuffer: sb1.equals(sb2) = false

\*/

以上输出的结果是由于：String类重写（override）Object类的equals方法，而StringBuffer类没有重写。

StringBuffer中的方法是append，String中的方法是concat，并且，试图在其他方法中通过调用该方法改变String是徒劳的。

#### Collection接口（数据结构）

因为在JAVA中没有结构（struct），所以只能用类和对象的自我应用来实现传统的数据结构，比如：链表、树、队列等等。

Set是不允许有重复值的集合。有些集合运行null值，但如果它们有null值时，这个值只能出现一次。然而，在HashSet类中，null值是不允许的。AbstractSet和HashSet类创建了没有排序的集合。TreeSet类则创建通过二进制树来排序的集合。

List是允许一个对象多次出现的有序集合。顺序可以是自然排列或按对象加入到集合的顺序排列。因为列表是有序的，所以它的对象可以被索引。ListIterator接口提供了迭代列表中元素的方法。抽象的列表可以被随机地、通过数组、通过链表或者通过双向链表进行访问。

Map是使用一系列值来查找或索引所存储数据的集合。借助于映射，你可以根据一个关键字（key）域进行搜索。关键字的键值必须是唯一的。Map可以有重复的值，但不能有重复的键值。在JDK1.2中，映像替代了字典，并作为关键字与值（associate key with values）相联系的首选技术。AbstractMap、HashMap和WeakHashMap类创建的映射是无序的。TreeMap类创建的是通过二进制树来排序的映射。

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

Collection简单的Collection可以是无序的，并不必有任何限制。你可以使用任意一种类型的对象，而且一个对象可以出现多次。List和Collection具有重复的元素。

设定合适的初始大小，比如：ArrayList是根据需要进行内存的扩充分配，会极大的影响性能。

Java.util包中的数据结构：

接口Iterator：提供了以某种定义号的顺序在一个元素列表中进行重复的一种标准办法，这通常是许多数据结构里的共同任务。接口Iterator定义的方法：hasNext()、next()和remove()。

BitSet类：实现了一组标记（一个标记是一个布尔变量值），可以被单独地设置或清除。例如：

class SomeBits{

public static final int READABLE = 0;

public static final int WRITEABLE = 1;

}

BitSet bits = new BitSet(2);

bits.set(SomeBits.WRITEABLE);

bits.clear(SomeBits. READABLE);

Vector类：实现了一个可扩展的对象数组。类Vector是被设计来处理类Object的，因此需要对使用方法get()后的数据进行类型转换。

Vector具有两个与大小相关的不同属性：大小size（向量中对象数目）和容量capacity（分配来存放所用对象的内存数量）；trimToSize()方法可以使容量分配与大小相匹配。Vector是从0下标开始编号的。

add()方法将在当前位置插入一个对象，并且使Vector中在其后的对象的下标发生变化；

contaions()方法搜索特定的对象；

indexOf()方法找出基于这个对象本身的下标；

Vector vectorA = new Vector(25); //分配足够的内存来支持25个对象

Vector vectorA = new Vector(25,5); //初始25个对象，以后每次增加5个对象

Beware of adding a primitive to a Vector. More generally, you can't use a primitive where an Object is required (eg. the equals method).

Stack类：实现了一个后进先出（LOFI）的元素堆栈。

接口Map：定义了一个框架来实现某个基本的键映射数据结构。定义了一种存储和获取基于某个键值的信息的方法。接口Map中的键值可以是任何内容。

Dictionary类：一个抽象类，定义了用于将键值（一个键值是一个在数据结构中用来引用或查找某个值的标志符）与数值进行匹配的数据结构。

Hashtable类：类Hashtable来自类Dictionary，实现了接口Map，并且提供了对键映像数据结构的完整实现。在接口Map中定义的所有抽象方法都在Hashtable中被实现了。Hashtable的实际用处在于表示那些按照值来进行查找或索引太耗时的数据。

contaions()方法检查某个对象是否存放在该表中；

contaionsKey()方法是基于某个键值来查找；

//负载系数：越靠近1.0导致更高的内存使用率；越靠近0.0导致更加有效的查找。

Hashtable hashTableA = new Hashtable(20); //缺省负载系数为0.75

Hashtable hashTableA = new Hashtable(20,0.25F); //具有初始容量和负载系数

Collections

\* The Collection interface is extended by Set and List (and SortedSet).

- a set contains no duplicates.

- a list is a sequence and can have duplicates.

\* The Map interface does not extend Collection and is extended by SortedMap.

\* There are abstract classes for each of the main interfaces (Collection, Set, List and Map) and implementations extend these.

\* Set implementations

- HashSet

- TreeSet (implements SortedSet)

\* List implementations

- ArrayList

- LinkedList

\* Map implementations

- HashMap

- TreeMap (implements SortedMap)

\* Vector and Hashtable (extends Dictionary) are older collection classes.

- Vector has been made to extend AbstractList and is like a sync'd ArrayList (maha:set)z.

- Hashtable still extends Dictionary but it implements Map.

- In contrast to other collection types, Vector and Hashtable are synchronized.

\* There are Collections and Arrays classes to provide static methods for general algorithms on collections and arrays.

\* Stack extends Vector

- it has methods push(Object), pop, peek and search(Object).

- Push is identical to addElement but also returns the added Object.

- The top of a stack is the last element in the vector (and has index 1 as far as the search method is concerned).

集合框架

1.容器分两类 collection 和map

collection：存放独立元素。list 里的元素可以重复，set不能有重复的。

map：存放一对key-value对应的元素

ArrayList,LinkedList extends list接口。

ArrayList:允许快速随机访问，但是当元素的安插和删除的位置在List中央的时候，

效率会比较低。最好使用ListIterator来遍历，不要添加删除。

LinkedList:提供最佳循序访问，以及成本低的在”list中安插和删除”。

可以用来做Stack(LIFO)；也可以用来制作Queue(FIFO)；

也可以用来做deque(两边开口的queue)。

HashSet,TreeSet extends Set 接口

HashSet:查询时间比较少，所有元素必须定义hashCode()；

TreeSet:有序的Set(是Sorted Set的唯一一份实现)。

### java.io

#### java.io包

* InputStream和OutputStream
* FilterInputStream和FilterOutputStream

Filter流可以从另一个流读取字节，或者把字节写入其它流。Filter流的类是InputStream和OutputStream的子类。虽然InputStream和OutputStream的一些子类能被自身使用，但FilterInputStream和FilterOutputStream以及它们的子类要同其它的流对象一起使用。

当创建Filter流时，你必须制定它将要连接的流。通过把InputStream和OutputStream的一个实例作为属性（取决于创建FilterInputStream还是FilterOutputStream）来创建构造函数。Filter不定义不含参数的构造函数，这意味着它必须通过某种方式连接起来；一个Filter流必须要么引用另一个Filter流，要么引用InputStream或者OutputStream流。

Filter流用某些方式来处理流。通过连接多个Filter流，可把多个过滤器添加到字节流中。可以把你所想要的所有Filter流连接在一起，链中的每个Filter流能处理从某资源读取的字节或者要写入的字节。链中的第一个Filter流必须与最先创建的文件或其它资源相链接。

* DataInput和DataOutput
* Reader和Writer

假定使用UTF-16定义的字符集创建了一个文件，在默认情况下，本地编码集被设置为ISO 8859-8。怎样编写写入该文件的代码？

其中：myFile为想写入的文件名；stream是与该文件相关的InputStream对象；s是String对象。

FileOutputStream stream = new FileOutputStream(myFile);

OutputStream out = new OutputStream(stream, “UTF-16”);

out.write(s);

* InputStream.read()的工作原理

// TestRead.java

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

import java.io.\*;

public class TestRead {

public static void main(String argv[]) {

try {

byte[] bary = new byte[]{-1, 0, 12, 23, 56, 98, 23, 127, -128};

ByteArrayInputStream bais = new ByteArrayInputStream(bary);

System.out.print("{ ");

while (test(bais)){ System.out.print(", "); }

System.out.println(" }");

// output: { 255, 0, 12, 23, 56, 98, 23, 127, 128, -1 }

// Notice 2 negative byte value becomes positive

// -1 is added to the end to signal EOF.

}catch (IOException e) {

System.out.println(e);

}

public static boolean test(InputStream is) throws IOException {

// Read one byte at a time and put it at the lower order byte of an int

// That is why the int value will be always positive unless EOF, which will be -1

int value = is.read();

System.out.print(value);

// return true as long as value is not -1

return value == (value & 0xff);

}

}

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

#### 使用文件 File

要创建一个文件，可以使用File、RandomAccessFile或者FileOutputStream类的实例；要创建一个新目录，可以使用File类的实例方法mkdir()或者mkdirs()。要删除一个文件，应该使用File类的实例。

创建新的RandomAccessFile对象，只有”r””和”rw是合法的模式。

Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change. However, you can use one File object to find the directory you want, and then create another File object for the directory you want to go to.

* 递归删除一个非空目录（Recursively Delete Directories in Java）：

In Java, a readable, writable directory should be empty before an attempt is made to remove it. The following code recursively empties a directory, and its sub-directories, and then removes them.

private void deleteFile(File p\_file) {

String FILE\_SEPARATOR = System.getProperty("file.separator");

// If it is a directory, empty it first

System.out.println("Deleting: "+p\_file.getPath());

if(p\_file.isDirectory()) {

String[] dirList = p\_file.list();

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

File aFile = new File(p\_file.getPath()+FILE\_SEPARATOR+dirList[i]);

if(aFile.isDirectory()) {

deleteFile(aFile);

}

aFile.delete();

}

}

p\_file.delete();

}

Note that a File object should be writable by the current user of the code in order to be removed.

* 复制一个文件

String fileToRead = "someFileToRead";

String fileToWrite = "someFileToWrite";

FileInputStream fis = null;

FileOutputStream fos = null;

try {

fis = new FileInputStream(fileToRead);

fos = new FileOutputStream(fileToWrite);

byte[] readBuffer = new byte[2152];

int bytesIn = 0;

while((bytesIn = fis.read(readBuffer)) != -1) {

fos.write(readBuffer, 0, bytesIn);

}

}

catch (Exception e) { //handle exception }

finally { //close the streams, the close operation may throw exceptions

try { fos.close(); }

catch (Exception e){ }

finally {

try { fis.close }

catch(Exception e){ }

}

}

### java.awt：图形和组件

#### AWT基础

继承层次：

Component→Container→Panel→Applet

→Window↘Frame

MenuContainerInterface↗

Component类是AWT类层次中的顶层类。Component是屏幕和界面上所有界面元素的抽象超类。Component类定义许多处理方法处理事件、定位窗口、调整窗口和更新显示器的方法。Component对象也负责设置和获取前景颜色、背景颜色和字体。

Container类是用于存放其它组件的Component子类。添加到容器的组件被记录在列表中，并且列表的顺序定义了容器内组件的跟踪顺序。当把一个组件添加到容器时没有指定索引，则此组件将被添加到列表尾。容器负责安排它们所包含的任何组件。

Panel类是Container类的具体子类。该类被用于实现容器，并且从Container类继承几乎所有的功能。Panel提供了程序可以连接任何其它组件的空间，包括其它Panel。Panel不包含标题栏或者边界。Panel类是Applet类的超类。当界面输出被导向Applet小程序，它会在Panel对象的表明绘画。

Window类会创建不包含在任何其它对象的顶层窗口。Windows直接位于桌面。在大多数情况下不会直接创建Window对象，而是使用Window的子类Frame。

Frame类会创建具有标题栏和边界的窗口。通常，Frame窗口被应用于应用程序，而不能与小程序同时使用。实际上，如果你想从小程序内创建Frame窗口，则会出现一个警告，告诉用户该窗口是由小程序产生的，而不是由应用程序产生的。这种警告有助于维护安全性。

#### Applet和AWT

Applet重新绘图的顺序为：rapaint()、updata()、paint()。

其中updata()方法依次做三件事：用背景颜色来填充对象背景，以便清除对象原有背景；设定当前的绘图色为对象的前景色；调用Paint()方法，并向它传递所接受的图形对象。

#### 在应用程序中创建框架窗口范例

import java.awt.\*;

import java.awt.event.\*;

class SimpleFrame extends Frame {

SimpleFrame(String title) {

super(title);

MyWindowAdapter adapter = new MyWindowAdapter(this);

addWindowListener(adapter);

}

public static void main(String args[]) {

Frame f;

f = new SimpleFrame("My Frame Window");

f.setSize(300, 200);

f.setVisible(true);

}

public void paint(Graphics g) {

g.drawString("Application Frame: Testing...", 15, 100);

}

}

class MyWindowAdapter extends WindowAdapter {

SimpleFrame simpleFrame;

public MyWindowAdapter(SimpleFrame simpleFrame) {

this.simpleFrame = simpleFrame;

}

//关闭框架窗口，覆盖WindowListener的方法windowClosing

public void windowClosing(WindowEvent we) {

simpleFrame.setVisible(false);

}

}

#### 在小程序中创建框架窗口范例

import java.awt.\*;

import java.awt.event.\*;

import java.applet.\*;

class MySimpleAppletFrame extends Frame {

MySimpleAppletFrame(String title) {

super(title);

MyWindowAdapter2 adapter = new MyWindowAdapter2(this);

addWindowListener(adapter);

}

public void paint(Graphics g) {

g.drawString("Applet Frame: Testing...", 15, 100);

}

}

class MyWindowAdapter2 extends WindowAdapter {

MySimpleAppletFrame mySimpleAppletFrame;

public MyWindowAdapter2(MySimpleAppletFrame mySimpleAppletFrame) {

this.mySimpleAppletFrame = mySimpleAppletFrame;

}

public void windowClosing(WindowEvent we) {

mySimpleAppletFrame.setVisible(false);

}

}

public class MySimpleApplet extends Applet {

Frame f;

public void init() {

f = new MySimpleAppletFrame("My Frame Window");

f.setSize(300,200);

f.setVisible(true);

}

public void start() {

f.setVisible(true);

}

public void stop() {

f.setVisible(false);

}

public void paint(Graphics g) {

g.drawString("Applet Window", 15, 100);

}

}

#### 字体和颜色

当改变组件的外观时，下面这些重要的类和方法要牢记：

* Font类定义字体。为了定义一个新的Font对象，需要提供字体的名字、字体类型（由Font类定义的一组常数）和字体的尺寸。然后通过调用setFont()并且传递新的Font对象，来设置被组件使用的字体。
* Courier是均衡字体，这意味着每个字母的宽度都相同。当你想将文本与其它行的文本排列在一起时，就会发现使用均衡字体是很方便的。
* 你可以使用setForeground()来设置组件的文本颜色，也可以使用setBackground()来设置组件的背景颜色。将某Color对象传递给这两种方法。一般总是使用从Color类（例如：Color.blue、Color.red和Color.gray）中引用的预先已经定义好的Color对象来命名3中颜色。许多普通的颜色由Color类中的静态变量来表示。

解决java中文问题，针对applet和awt：

1：

Font f = new Font(UIResource.getString("Default\_font"),Font.PLAIN,12);

UIManager.put("Label.font",f);

UIManager.put("Label.foreground",Color.black);

UIManager.put("Button.font",f);

UIManager.put("Menu.font",f);

UIManager.put("MenuItem.font",f);

UIManager.put("List.font",f);

UIManager.put("CheckBox.font",f);

UIManager.put("RadioButton.font",f);

UIManager.put("ComboBox.font",f);

UIManager.put("TextArea.font",f);

2：

Font f = new Font("隶书",Font.PLAIN,15);

UIManager.put("Button.font",font);

UIManager.put("ToggleButton.font",font);

UIManager.put("RadioButton.font",font);

UIManager.put("CheckBox.font",font);

UIManager.put("ColorChooser.font",font);

UIManager.put("ToggleButton.font",font);

UIManager.put("ComboBox.font",font);

UIManager.put("ComboBoxItem.font",font);

UIManager.put("InternalFrame.titleFont",font);

UIManager.put("Label.font",font);

UIManager.put("List.font",font);

UIManager.put("MenuBar.font",font);

UIManager.put("Menu.font",font);

UIManager.put("MenuItem.font",font);

UIManager.put("RadioButtonMenuItem.font",font);

UIManager.put("CheckBoxMenuItem.font",font);

UIManager.put("PopupMenu.font",font);

UIManager.put("OptionPane.font",font);

UIManager.put("Panel.font",font);

UIManager.put("ProgressBar.font",font);

UIManager.put("ScrollPane.font",font);

UIManager.put("Viewport",font);

UIManager.put("TabbedPane.font",font);

UIManager.put("TableHeader.font",font);

UIManager.put("TextField.font",font);

UIManager.put("PasswordFiled.font",font);

UIManager.put("TextArea.font",font);

UIManager.put("TextPane.font",font);

UIManager.put("EditorPane.font",font);

UIManager.put("TitledBorder.font",font);

UIManager.put("ToolBar.font",font);

UIManager.put("ToolTip.font",font);

UIManager.put("Tree.font",font);

### java.awt：布局管理器

布局管理器的模式不建议使用指定构件的大小和位置，而是主张选择合适的布局管理器，再由布局管理器按照一个布局策略，确定每个构件具体的大小和位置。

#### FlowLayout

FlowLayout是默认的布局管理器，将组件从左到右，从上到下依次排列，并且每一行的组件对水平方向居中。如果不需要每一行都居中，可使用设置对齐方式的常数FlowLayout.LEFT、FlowLayout.CENTER和FlowLayout.RIGHT。例如：

setLayout(new FlowLayout(FlowLayout.LEFT));

当有新组件以FlowLayout加入到容器中时，FlowLayout将根据容器宽度将新的组件放置于前一组件同一行上，直到不能再容纳新组件的插入。FlowLayout使这一排居中，然后再新增一排来放置下面的组件。这样FlowLayout使每个组件获得最佳尺寸，并且不随意改动它们（这一点与其它布局管理器不同）。

如果使用FlowLayout的容器尺寸被改变，那么所有其中的组件都必须重新安排。The size of the components is not constrained by the FlowLayout manager. When the area is resized the size of these components is not changed.

#### BorderLayout

BorderLayout根据北（North）、南（South）、东（East）、西（West）方向来安排组件，余下的即是中心（Center），它是其它区域布满后余下来的（如果东西区域没有放置组件，中心区域即指从容器左端到容器右端的全部区域）。每个区域可以包含一个组件。North区域的组件获得容器顶部的位置，使South区域的组件位于容器底部。布局管理器强迫North和South组件与容器等宽。

BorderLayout不受添加组件顺序的影响，必须指定添加组件的区域。方法add( Component, String )或者add( String, Component )中，第一个参数是被添加的组件，第二个参数是String对象：

BorderLayout.NORTH、BorderLayout.SOUTH、BorderLayout.EAST、BorderLayout.WEST或BorderLayout.CENTER

在BorderLayout方式下，如果容器小于组件的最佳尺寸，那么组件就被压缩；反之组件就被拉伸。

BorderLayout方式有一个局限性：在一个特定区域不能显示多个组件。如果向一个区域增加了多个组件，那么只有最后加上去的组件才能显示。

BorderLayout方式的好处在于：与FlowLayout相比，它能够确保组件保持相互间的关系不变（即是程序界面尺寸发生变化）.

若想用BorderLayout方式在同一区域显示多个组件，那就可以在容器内再加一个容器。如果不想让组件间紧密层叠，BorderLayout( int, int )可以指定区域间水平与垂直的间距。

可拉伸的组件：按钮、标签和文本框

不可拉伸的组件：复选框

The components are laid out according to their preferred sizes and the constraints of the container's size. The NORTH and SOUTH components may be stretched horizontally; the EAST and WEST components may be stretched vertically; the CENTER component may stretch both horizontally and vertically to fill any space left over.

#### GridLayout

GridLayout方式可以指定放置组件的矩形网格，网格中每个单元高度与宽度相同（采用这种方式，可以确保设置的Button的高度和宽度相同）。组件未适应网格可以在水平与垂直方向上拉伸。网格的数目是由容器的尺寸，以及需要创建多少网格决定的。

GridLayout初次创建时必须指定网格拥有的行数和列数。GridLayout安排组件的顺序时从左到右、从上到下。当组件的数目大于列数时，GridLayout保持行数不变而自动增加列数。

//1行5列，行之间的间隔为10，列之间的间隔为0

setLayout(new GridLayout(1, 5, 10, 0));

#### CardLayout

CardLayout用来堆叠组件，一个组件在另一个组件的上面，每次只能看到一个组件。

#### GridBagLayout

GridBagLayout是把组件严格限定在网格之中，而且网格尺寸都是一样的，每个组件只能放置于一个网格内。GridBagLayout不要求每行每列有相同的尺寸，而且组件可以占据多于一个的网格。

#### XYLayout

根据组件的X和Y坐标来放置组件，这是最灵活、最容易使用的一种布局方式。这种布局管理器允许使用者拖动组件到任意指定的位置和大小，无需编码。

#### PaneLayout

PanelLayout方式通过为容器中已选定的组件划分显示区域来安排组件。在PanelLayout中的组件可以调整自身的大小来填满显示区域。PanelLayout使用PaneConstraints对象来控制组件的位置和大小。

PaneConstraints（String name1, String name2, int pos, float per）：四个参数分别代表组件本身、被划分区域的组件、该控件处于被划分组件的位置（ROOT、TOP、RIGHT、LEFT和BOTTOM）、给该组件占被划分组件的百分比。

#### VerticalFlowLayout

VerticalFlowLayout是将组件依次按照顺序从上到下，每一行的组件都竖直居中于当前的容器。

#### BoxLayout 2

将FlowLayout和VerticalFlowLayout的功能结合起来封装的布局管理器。

#### OverlayLayout 2

类似于CardLayout，但设置组件在布局管理器中的属性后，一次可以看到多个组件。

#### 默认的布局管理器

由于每一个容器类都有与之管理关联的布局管理器，所以容器都有一个默认的布局管理器。容器Farme、Dialog和Window： 默认布局管理器BorderLayout

容器Panel： 默认布局管理器FlowLayout

#### 综合应用布局管理器

import java.awt.\*;

import java.applet.\*;

public class FlowFiveC extends Applet {

public void init() {

Panel p1 = new Panel();

p1.setLayout(new GridLayout(1, 5));

p1.add(new Button("First"));

p1.add(new Button("Second"));

p1.add(new Button("Third"));

p1.add(new Button("Fourth"));

p1.add(new Button("Fifth"));

Panel p2 = new Panel();

p2.setLayout(new FlowLayout(FlowLayout.LEFT));

p2.add(p1);

setLayout(new BorderLayout());

add(p2, BorderLayout.SOUTH);

}

}

这段代码中，首先将按钮放置在Panel（面板）对象p1上，该对象用GridLayout方式排列这些按钮（确保设置的Button的高度和宽度相同）；然后，把p1对象添加到使用FlowLayout方式的Panel对象p2；最后将对象p2 加入到使用BorderLayout方式的小程序中。

### java.awt：事件处理

#### 事件类

Java包定义了许多不同的事件类，它们分别代表特定的事件类型。位于事件层次顶端的EventObject位于java.util中，不是在java.util中。这使得事件具有更好的通用性，并且不依赖于AWT。然而所有由AWT的组件调用的事件，其使用的事件类是在java.util中的AWTEvent的子类。

#### 事件监听

必须显示地指出哪些对象应该处理发生在特定组件中的具体事件。通过告诉组件哪个对象会监听其所感兴趣的事件，你可以创建事件句柄。

例如，在用户界面上放置一个按钮，并且想让应用程序知道什么时候用户会点击这个按钮，这样在编写代码时就要遵循以下的步骤：

* 定义一个实现监听器接口的类。有各种各样的监听器接口，其中每一种都代表了不同类型的事件。例如，MouseListener类声明处理鼠标移动的接口方法（mouseClicked、mouseReleased等）。
* 为这个类创建一个实例。
* 通过调用一个叫做addABCListener的方法，并将这一方法传递给监听器类的实例，就可以告诉你感兴趣的组件是谁产生的事件，是哪些类型的事件，哪个对象处理这些事件。还有一些不同的addABCListener方法，这些方法名字里的ABC三个字母被添加到组件列表中的监听器类型所替代。

这是一个观察者模式的设计。Subject是组件本身，Observer是ABCListener。当组件状态发生变化时，ABCListener就可以收到通知，进行更新。例如：

jTextField1.addActionListener(new java.awt.event.ActionListener() {

public void actionPerformed(ActionEvent e) {

jTextField1\_actionPerformed(e);

}

});

One component can registers one or more listeners. One listener can be registered by multiple components. After being registered the listener can receive and process the events from these components.

All the methods in the event listener interfaces return void.

Four listeners don't have their corresponding event adapters. They are ActionListener, ItemListener, AdjustmentListener and TextListener.

#### 实现事件监听接口（例子）

import java.awt.\*;

import java.awt.event.\*;

import java.applet.Applet;

public class ClickApplet extends Applet {

public void init() {

Button b = new Button("Click me!");

b.addMouseListener(new OurClickHandler()); //创建实例，并加入

add(b);

}

}

class OurClickHandler implements MouseListener { //定义类实现接口

public void mouseClicked(MouseEvent e) {

System.out.println("button clicked");

}

// Left-over interface methods.

public void mousePressed(MouseEvent e) { }

public void mouseReleased(MouseEvent e) { }

public void mouseEntered(MouseEvent e) { }

public void mouseExited(MouseEvent e) { }

}

## Exception异常处理（错误处理）

Java对于异常要求在方法中声明抛出异常的方法，它们要捕捉被调用方法所抛出的异常。Java中的异常是实际的对象，都是从类Throwable继承而来的类的实例。可以这样使用：

try{ } catch(Throwable t) { t.printStackTrace(); }

也可以在代码中的任何位置使用：

new Throwable().printStackTrace();

Throwable有两个子类：Error和Exception。Error的实例是在Java运行时环境中（Java虚拟机）的内部错误（Error and subclasses of Error are thrown by the JVM and should not be managed by the developers.）。Exception是在Java里创建的任何异常的超类。Exception的子类分为两组：运行时异常RuntimeException类及其子类（Runtime exceptions also should not be caught nor extended by user programs. They are meant to be propagated to the top level of the call stack and should be managed by the runtime system.）和其它异常。

Throwable、Erroe和Exception的继承关系如下：

class Throwable extends Object

// Never subclass or catch these !!

class Error extends Throwable

class Exception extends Throwable

// Never subclass or catch these

class RuntimeException extends Exception

// Creating user exceptions

class MyException extends Exception

可以为单个try使用多个catch块，在一个多catch块中，第一个匹配的catch块会被执行，而余下的则被忽略；它们的顺序为：从你想捕获的特定异常到你想捕获的异常的超类。试图捕获超类的异常首先导致编译时的错误。例如，若把带有Exception类型的catch块放在带有IOException类型的catch块前会导致编译错误。可以使用带有finally而不带catch语句的try语句。如果多层中某层程序发生异常，则会一层层向上抛，直到有捕捉为止。你不能在try块自身里面再用try。

If two exceptions thrown in the same try block, when the first exeption is thrown, the second exception is ignored. Because: An exception is thrown, which means the code cannot continue its normal path. The later code will not be excuted no mater there is another throw exception statement or not. This bad situation will continue until the thrown exception is caught or the program will die at the top level.

为了指明某个方法可能抛出一个异常，可以在方法的后面添加关键词throws，并带有该方法所抛出异常的名字（若需抛出多种类型的异常，可以将其放在一个throws语句中用逗号隔开）。但隐式异常（即非检测异常：Error类或RuntimeException类或者它们的任何子类）通常由Java运行时自己抛出，不必声明在方法中抛出。Error和RuntimeException是未检测异常不必捕获。

比捕获异常并将它们忽略更好的方法是将它们传递给外层的调用方法。与其在方法体内使用try和catch语句，还不如用一个throws语句来声明方法，以便它能够抛出适当的异常，即方法后加异常：

public void connectMe() throws Exception{

**...**

}

public void testExp{

try {

connectMe(); //在testExp里必须要做try catch 操作，不然编译错。

return; //即使try中的代码有return语句，finally块仍然运行

} catch (Exception e){

System.out.println(“Exception catched”);

return; //即使catch中的代码有return语句，finally块仍然运行

} catch (RuntimeException e){

System.out.println(“RuntimeException catched”);

return; //该catch语句在程序中不会运行

} finally{

System.out.println("finally Execute");

}

System.out.println("testExp Execute"); //finally运行后，该语句可以照常运行

}

try块后可以跟catch块或finally块，或两者皆可。finally块和catch块均不能单独使用，必须与try块一起使用。

为什么使用多个catch？

For instance, if you are designing a user interface for an application, your program could throw exceptions under several conditions. However, you may not want to propagate all the exceptions up to the user interface module and especially not to the end user. In such a case, you may want to catch the exception, partially handle it (i.e. log it in a file), and then propagate up a higher level exception that makes sense to the user interface module.

例如：

1. public void propagateException throws UIException {

2. try {

3. // Method code here

4. }

5. catch (FileNotFoundException fnfe) {

6. // log FileNotFoundException to file

7. throw new UIException();

8. }

9. catch (IOException ioe) {

10. // log IOException to file

11. throw new UIException();

12. }

13. }

In this example, the calling application expects the user-defined exception, UIException. On Lines 5-8, FileNotFoundExceptions are caught and converted to a UIException. On Lines 9-12, IOExceptions are caught and converted to a UIException. The calling method will only be aware of UIExceptions.

异常在创建时含有描述性的字符串，解释异常为什么发生，可以用getMessage()访问该字符串，它在Exception的超类Throwable中具体实现。也可以调用printStackTrace查看异常信息。

### 创建并抛出自己的异常

所有用户创建的异常应该是Exception而非Error层次的一部分。

public class ServerTimedOutException extends Exception{

public SevertimedOutException (){} //构造函数1：不带任何参数

public SevertimedOutException (String msg){ //构造函数2：带字符串参数

super(msg);

}

public String getReason(){..}

public String getPort() {..}

}

使用自己的异常：

try{

}catch{

throw new SeverTimedOutException(..);

}

### 什么时候使用异常？

如果你的方法调用了另一个有throws子句的方法，你可以进行如下的一种方法：

1. 通过使用try和catch语句来处理异常
2. 通过对你的方法定义添加throws子句来将异常沿着调用链路传递
3. 执行前两种方法，即通过使用try和catch语句来捕获异常，然后通过使用throw来显示地将它重新抛出

在某个方法抛出多个异常的情况下，可以区别地处理每一个异常。例如：可以捕获其中的一些异常；同时让另一些异常沿着调用链向上传递。

如果方法抛出了自己的异常，应该使用throws子句来声明它抛出了哪些方法。如果方法重写了带有一个throws子句的超类方法，则可以抛出相同类型的异常或者那些异常的子类，但不能抛出任何不同类型的异常。

如果方法使用了一个throws子句来声明，不要忘记在方法体内使用throws语句来实际抛出该异常。避免直接抛出异常Exception，应该使用抛出其子类的异常，避免捕获不到某些异常

**【错误示例】**

public class NTX {

void lostDetailsTest () throws Exception { //违例，编译通过

throw new java.io.IOException ("IO exception");

}

}

**【改正示例】**

public class NTXFixed {

void lostDetailsTest () throws java.io.IOException { //正确

throw new java.io.IOException ("IO exception");

}

}

### 什么时候不要使用异常？

如果异常是可预料的或者可以用一个简单的语句来避免的某个事情，就不该使用异常。异常应该只用在那些真正超出你控制的异常情况中。

### 构造函数中抛出异常的处理方法

父类的定义：

public class A {

int iValue = 10;

public A(int initValue) throws Exception {

if(initValue<10) { throw new Exception("initial value smaller than 10"); }

}

}

子类的定义：

public class B extends A {

private String iString = null;

public B(int initValue, String initString) {

super(initValue);

iString = initString;

}

}

编译时会抛出异常："Exception java.lang.Exception must be caught, or it must be declared in the throws clause of this constructor"

将子类改写为：

public class B extends A {

private String iString = null;

public B(int initValue, String initString) {

try {

super(initValue); //调用父类构造函数必须在第一句中

} catch(Exception e) {

System.out.println(e.getMessage);

}

iString = initString;

}

}

编译时同样会抛出异常："No constructor matching A() found in A and Constructor invocation must be the first thing in a method".

分析：If A's constructor throws an exception, it implies that the A instance could not be fully constructed. Therefore, the instance of B within which that instance of A resides cannot be created either. Thus B's constructor should throw at least the A's Exception.

将子类重新改写为：

public class B extends A {

private String iString = null;

public B(int initValue, String initString) throws Exception {

super(initValue);

iString = initString;

}

}

编译通过。**:**)

### 常见的Java异常

* + **检测（捕获）异常**

ClassNotFoundException（在java.lang中定义）：说明某类被调用，但是没有找到相应的类文件。该类名或者不正确，或者该类对程序无效。

IOException（在java.io中定义）：在读写文件时发生错误的信号。在你使用流的方法时常常不得不处理这种异常。

FileNotFoundException（在java.io中定义）：当访问一个文件时，你必须准备处理这种异常。记住，这是因为你创建了某File对象并不意味着该文件存在。

IllegalAccessException（在java.lang中定义）：当你在某类中尝试加载，但是当前执行方法没有访问指定的类时，则会发生这种异常。

InterruptedException（在java.lang中定义）：当你休眠线程或挂起线程时，必须准备处理这种异常；休眠线程或挂起线程可能在休眠时间结束之前或在调用resume()之前被中断，以防在这种情况下抛出异常。

MalformedURLException（在java.net中定义）：当创建URL时，必须准备处理这种异常，以防提供的URL字符串不合要求。

* + **非检测（捕获）异常 RuntimeException**

ArithmeticException（在java.lang中定义）：在出现不合法的数学运算发生时，比如；整数被零除，JVM将抛出这种异常。

ArrayIndexOutOfBoundsException（在java.lang中定义）：如果你使用非法的索引值来访问数组时，该异常就会抛出。该异常说明该索引要么是负值，要么大于或者等于该数组的大小。

ArrayStoreException（在java.lang中定义）：如果你尝试在对象数组中保存错误类型的对象时，则会发生这种异常。

ClassCastException

例如：

Object x = new Integer(0); System.out.println((String)x); 编译可以通过，但运行时产生ClassCastExceptio.

EmptyStackException（在java.lang中定义）：该异常可以由Stack类中的方法抛出。如果你尝试从空的堆栈中检索项，则会发生该异常。

IndexOutOfBoundsException（在java.lang中定义）：如果你尝试通过过大或过小的index去取数组中的值，则该异常发生。

IllegalParameterException（在java.lang中定义）：如果你尝试向方法传递非法或者不合适的值，则该异常发生。

NullPointerException（在java.lang中定义）：如果调用使用null对象引用的方法，就会看到这种异常。如果在编程时不注意，则很容易在自己的代码中蔓延。例如：你可以建立两个方法调用，并希望methodOne()返回将调用methodTwo()的对象。如果methodOne()返回null，则调用methodTwo()将抛出NullPointerException。

NumberFormatException（在java.lang中定义）：如果将字符串转换成数字，你得准备处理这种异常，以防止该字符串实际上并不代表数字。

StringIndexOutOfBoundsException（在java.lang中定义）：说明尝试使用在字符串边界外的索引。

### 子类中覆盖方法的异常

当覆盖某方法时，你必须列出覆盖代码可能抛出的异常。你只可以列出在所继承的方法定义中定义的异常或者异常的子类。（方法也可以抛出任何非检测异常，即使其在throw子句中没有声明）

在覆盖子类中某方法时，你不能把新的异常类型添加到你所继承的方法中（即覆盖的方法不能抛出它的超类未声明的异常）。你可以抛出在方法的超类中所列的异常的子集，但是远在这一层下面的子类不能再列出在它之上的异常。

例子：

class First {

void test() throws Exception1, Exception2 { …… }

}

class Second extends First {

void test() { …… }

}

class Third extends Second {

void test() { …… }

}

问题：Third类中的test方法会抛出什么异常？

A. Exception1 B. Exception2 C. no checked exceptions.

分析：子类中的方法不能添加可能抛出的新异常类型。由于子类的超类（Second）不定义其test()方法中的任何异常，Third类中的test方法也不能添加新的异常类型。

答案：C

### 消失的异常

A thrown exception is guaranteed to be caught. An exception handling block (a catch or finally block) is allowed to forget the reason it was entered. If a handling block uses the return statement, the exception disappears (see method returnInHandler() in the example code). You should never put a return inside a catch or finally block unless you really intend to dismiss the exception.

More insidiously, if an exception is thrown inside a handling block and is allowed to propagate out of that block, the original exception disappears. The method doubleThrow() demonstrates this behavior, in which the ImportantException disappears and the NotImportantException propagates. You should be especially careful to catch NullPointerException and IOException, which are commonly thrown when blocks finally attempt to close resources that weren't properly allocated in the first place.

EXAMPLE:

class ImportantException extends Exception{}

class NotImportantException extends Exception{}

public class DisappearingExceptions{

static void doubleThrow() throws Exception{

try {

throw new ImportantException();

} finally{

throw new NotImportantException();

}

}

static void returnInHandler() throws Exception{

try {

throw new ImportantException();

}finally{

return;

}

}

public static void main(String[] args){

try{

doubleThrow();

}catch(Exception e){

System.err.println(e);

}

try{

returnInHandler();

System.out.println("No exception");

}catch(Exception e){

System.err.println(e);

}

}

}

The output is:

NotImportantException

No exception

### 异常的关键概念

Exceptions

\* Non-runtime exceptions are called checked exceptions.

\* Even if a method explicitly throws a runtime exception, there is no obligation for the caller to acknowledge the exception. One consequence of this is that the restriction on exceptions thrown by an overriding method only applies to checked exceptions.

\* A try block's finally clause is called unless the JVM is exited (i think).

- a return in try or catch does not prevent finally from being executed.

\* A try block's finally statement will be executed (unless the thread dies) before control leaves the try/catch/finally scope. It will be executed before unhandled exceptions (from try or catch) are passed back up the calling stack.

\* If you return from a try and finally does not return ... 1) the return value is calculated, 2) finally executes and 3) the method returns with the value calculated prior to executing finally.

\* If you have a return in both try and finally, the finally's value is always returned.

\* If try/catch excludes continuation following finally, it is a compile error to have any statements there.

\* Primitive floating point operations do not throw exceptions. They use NaN and infinity instead.

\* A constructor can throw any exception.

### 从堆栈中获取抛出的异常，重定位到字符串

import java.io.\*;

public class ExceptionToString {

public ExceptionToString() {

//====== Method one ======

try {

int zero= 0;

System.out.println("Causing exception with 1/0 "+1/zero);

} catch (Exception e) {

StringWriter swriter = new StringWriter();

e.printStackTrace(new PrintWriter(swriter));

String text = swriter.toString();

System.out.println("Caught individual exception " + text);

}

//====== Method two ======

ByteArrayOutputStream baos = new ByteArrayOutputStream();

System.setErr( new PrintStream( baos ) );

try {

int zero= 0;

System.out.println("Causing exception with 1/0" + 1/zero);

} catch (Exception e) {

e.printStackTrace();

}

byte []output = baos.toByteArray();

String f = new String( output );

System.out.println("Caught exception text = "+f);

}

public static void main(String []args) {

new ExceptionToString();

}

}

## GC垃圾收集

大多数情况下，你永远不会知道垃圾收集什么时候发生。Java将垃圾收集进程作为一个低优先级线程运行，当空闲内存较少时，垃圾收集程序就会运行。

How GC implemented is vender and platform dependent. Which means: It is vender and platform dependent that when and in what order the eligible for GC objects will be GCed.

However, which objects are eligible for GC is independent of implementations. Different vendor may use different algorithms. What algorithm used to find the objects which are eligible for GC does not matter. What matters is they use the same principle.

The object is no longer referenced or referenced only by objects which are eligible for GC. Unless they have a bug...

Ojects referenced by block-level variables are probably not available for garbage collection until their enclosing method's scope is exited.

Beware of local vars from a try block used in its catch block (out of scope).

### 对象何时适合于垃圾收集

当你的程序不再引用某个对象的时候，则该对象就会被考虑收集到垃圾收集堆里。例如：

假如你分配了一个对象，并且只为它指派了一个方法变量。当这个方法返回时，你就无法在引用这个方法了。因此，这个对象就永远消失了。那意味着当方法返回时，这个对象就可以考虑收集到垃圾收集堆了。（即当一个方法返回时，它的方法变量将不在有效，这样垃圾收集程序会知道所有该方法变量引用的对象都可以考虑收集到垃圾收集堆了）

你可能创建一个对象，给它指派一个对象引用，然后将这个对象引用设置为null值或者设置为其它的对象。一旦你失去了对这个对象的引用，这个对象就可以考虑收集到垃圾收集堆了。

仅因为你的程序可能失去对一个对象的引用，并不意味着JVM会立刻收回这个对象的内存，甚至根本不回收。JVM仅会在需要更多的内存以继续执行程序时，才会进行垃圾收集。

### 结束方法（最终化）

一个结束方法是在对象被作为垃圾回收和释放内存之前被调用的。结束方法是finalize()，Java会为程序中每一个对象只调用一次finalize()方法。类Object定义了一个默认的不做任何事情的结束方法。

要为自己的类创建一个结束方法，需要重写方法finalize()：（记住finalize()声明为protected，不返回任何值，并且抛出一个Throwable类型异常）

protecte void finalize() throws Throwable{

super.finalize();

}

调用fianlize() 并不会使得一个对象被收集到垃圾中，只有删除了对该对象的所有引用后，才导致它被加上删除标记。

结束方法最适合用在对象删除的优化上：例如，用它来去掉对其它对象的引用。在大多数情况下，你根本不必使用finalize()。

使用Runtime对象来强迫垃圾收集的例子：

class Test {

protected void finalize() throws Throwable {

super.finalize();

System.out.println("We're in finalize");

}

}

class GC {

public static void main(String[] args) {

GC gc = new GC();

gc.test();

}

void test() {

Test t = new Test();

t = null;

fullGC();

}

void fullGC() {

Runtime rt = Runtime.getRuntime();

long isFree = rt.freeMemory(); //返回空闲内存总数

long wasFree;

do {

wasFree = isFree;

rt.gc(); //在这个方法返回之前，JVM已经执行了垃圾收集

isFree = rt.freeMemory();

} while (isFree > wasFree); //强迫空闲内存发生变化

rt.runFinalization();//在这个方法返回之前，JVM已经为所有还没有运行finalize()方法的等候垃圾收集的对象执行了finalize()方法。

}

}

### GC部分问题回答

**Q. Any class that includes a finalize method should invoke its superclass' finalize method, why?**

**A:**

The point is that finalize methods are not automatically "chained" - if you subclass a class that has an important finalize, your finalize should call it. Obviously if you don't know what the superclass finalize does, you should call it to be safe.

By contrast, When you create an object with the new keyword, the superclass constructor(s) are called first. This is automatically "chained".

**Q. When an object becomes eligible for garbage collection?**

**A:**

An object becomes eligible for garbage collection when there is no way for any active thread to reach that object through a reference or chain of references.

Set all references to an object to null is **sufficient** for an object eligible to GC, but **not necessary**.

Here isare simple example with proper comments:

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

public class Test1{

public static void main(String[] args) {

Object a = new Object();

Object b = a;

a = null;

// the Object is still referred by b. // which is not eligible for garbage collection.

Object[] ary = new Object[5];

for (int i=0; i<5; i++) {

ary[i] = new Object();

}

// do something here

ary = null;

//Now all objects in the array are eligible for GC

// even none of their references are null.

}

}

**Q. How many objects are eligible for GC in the following code after d = null?**

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

public class Test{

public static void main(String[] args) {

Object a = new Object(); // the object original referenced by object reference a

Object b = new Object();

Object c = new Object();

Object d = new Object();

d=c=b=a;

d=null;

}

}

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

**A:**

Just remember that **operator = is right associate**, then you should be able to figure out the answer by yourself. The equivalent statement is d=(c=(b=a));

Answer: Three.

1) After b=a, the object original referenced by b is eligible for GC.

2) After c=(b=a), the object original referenced by c is eligible for GC.

3) After d=(c=(b=a)), the object original referenced by d is eligible for GC.

4) After d=null, nothing new is eligible for GC.

5)The object original referenced by a is not eligible for GC, since it is still referred

by references a, b, c.

6)Make sure you understand the differences between physical object and object reference (pointer to object).

## Java DOC文档

Javadoc产生四种类型的文件：

* packages.html：已经生成了文档的包及包中所含类的列表
* AllNames.html：所有方法及变量名字的字母索引
* tree.html：已经生成了文档的类的列表
* Classname.html：类的文档

### Javadoc帮助

除了下面两种常见的注释写法：

// everthing to the end of the line is a comment 和

/\* everything in here is a commner \*/

Java还定义了第三种特殊的写法：

/\*\* everything in here is a javadoc commner \*/

在默认情况下，这种特殊的javadoc注释只在公有类。或者公有或受保护的变量和方法前有意义。当运行javadoc时，如果使用参数 –private 选项，你会看到为私有变量和方法生成的javadoc风格的注释。

### Javadoc标签

Javadoc标签有：@see、@return、@param、@exception和@throws（这四个标签可以带记录方法的特征）。

调用javadoc的方法：javadoc JavaDocTest.java

例子：

* Test类的文档

/\*\*

\* Test machine for objects.

\* This class tests objects for the performance and accuracy

\* by executing standard tests.

\* @see CanBeTested

\* @version 1.0

\* @author ZTE

\*/

public class Tester{ …… }

* runTest方法的文档

/\*\*

\* Performs the standard test.

\* Invokes test() for the object to be tested and informs

\* this method how much testing should be performed.

\* @return A String that explains the results of the test

\* @param targer The debug level

\* @param start The debug level

\* @exception exceptionname anytext

\* @throws exceptionname anytext

\* @see CanBeTested

\*/

public String runTest (int debugLevel, int debugStart) { …… }

* debugLevel变量的文档

/\*\*

\* Debug output level.

\* @see CanBeTested

\*/

public int debugLevel;

注意javadoc注释中的第一行非常特殊，这一行祈祷总结的作用，并且javadoc在索引中要用到这一行。

# 基本技术问题

## 图形编程

#### 事件模型

Button

menu

事件

处理

对象

必须要实现接口：

public interface ActionListener{

public void actionperformed(Action evt);

}

public class myPanel extends Jpanel implements ActionListener{

public void actionperformed(ActionEvent evt){

}

}

JButton myButton = new JButton(“clear”);

myButton.addActionListener(new ActionListener()

{

public void actionPerformed(ActionEvent e){

jMenuHelpAbout\_actionPerformed(e);

}

});

或者：

myButton.addActionListener(new myPanel());

### Swing特性

#### 设置外观

外观的管理是由在包javax.swing中的一个用户界面管理器类UIManager来实现的。

try{

//使用操作系统默认界面风格

UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());

//设置为Metal方式

UIManager.getCrossPlatformLookAndFeelClassName();

//模拟motif界面风格

UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");

}catch(Exception e){

e.printStackTrace();

}

#### 标准对话框

类JOptionPane提供四种标准的对话框：

Confirm对话：showConfirmDialog

Input对话框：showInputDialog

Message对话框：showMessageDialog

Option对话框：showOptionDialog

String[] choices = { "Personal", "Commercial", "Unknown" };

int Result = JOptionPane.showOptionDialog(null, "What type of site is it?",

"Site Type", 0, JOptionPane.QUESTION\_MESSAGE,

null, choices, choices[0]);

## 数据库编程

### JDBC介绍(Java数据库互连)

Java database connectivity是Java的数据库连接，屏蔽掉数据库的物理实现细节。JDBC是SQL级别的API；在JDBC接口中，允许SQL语句作为参数嵌入到方法中；为了在独立于特定数据库的模式下完成这项工作，JDBC需要数据库厂商（Oracel、SQL Server、MySQL等）提供其接口的运行时实现（Driver）。程序设计者不用关心SQL语句的执行过程，不管当前是什么类型的数据库，都可以运行同样的代码。实现一次编程、一次编译、随处运行。

ODBC是开发Windows环境下数据库访问的一种标准尽管业界已经把ODBC看作Windows下数据库通信的基本方法，但它对Java的转换做得并不好。首先，ODBC是一种C API，不支持Java的平台无关性。其次，ODBC的设计太复杂，造成了对Windows环境进行外部转换的失败。为了了解ODBC的一点点内容，就必须懂得许多知识。

JDBC通过一系列的Java的接口来达到它的目的。每一个接口分别由不同的厂商来实现。为特定的数据库引擎实现的JDBC接口的类的集合被称为JDBC驱动程序（Driver）。JDBC的整个要点在于隐藏每个数据库的细节，应用开发人员只需考虑应用程序的设计，根本不必关心它们低层类的实现。

### JDBC的四种类型



Type1：JDBC-ODBC桥，提供到ODBC API的通路，通常需要在客户端系统上安装软件。

Type2：DB-Library、OCI等，标准的API驱动程序（C/C++），需要在客户端安装软件。

Type3：网络API，可以在服务器级别转化为特定数据库的访问，客户端上的JDBC驱动程序使用套接字来调用服务器上的中介应用程序，它将客户端的请求转化为所需驱动程序上的具体API。优点：灵活，不需要客户端软件，单个驱动程序就可以完成对多个数据库的访问。

Type4：使用网络协议构建的数据库引擎，利用Java套接字来直接与数据库通信。这是最直接、最纯粹的Java方法。

### JDBC类

DiverManager类：用于对数据库驱动程序的管理、注册、注销以及连接等，主要功能是可以自动为数据库访问加载合适的驱动程序，进行数据库的连接。

Connection接口：能够处理锁定表、提交和恢复数据、断开连接等功能，还能处理一些特殊类型的调用。

Statement接口：主要用来处理和执行几种不同的SQL语句，包括查询（Select）和更新（updata、insert和delete）。

ResultSet接口：用来对Statement对象返回的结果进行控制。通过ResultSet接口可以得到全部的查询结果，并可以对该结果进行定位、更新等操作。

异常类：共有4种类型的异常 SQLException→SQLWarning→DataTruncation

→BatchUpdateException

### SQL数据库操作

Select Name, Age From Student Where Name = ‘小李‘

Update Student set Age =20 where Name = ‘小李‘

Insert into 表名 values(‘zhang’,24)

Delete from 表名 where Name = ‘xiaowang’

### JDBC实例

利用JDBC访问数据库是基于以下4个步骤：

1. 实例化用于当前数据源的驱动程序
2. 利用这个驱动程序建立数据库的连接
3. 发送SQL语句进行数据库访问
4. 数据库将结果返回给本地变量

import java.sql.\*; //JDBC package

String data = "jdbc:odbc:World Energy";

try {

//基于JDBC-ODBC桥

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Connection conn = DriverManager.getConnection(data, "", "");

Statement st = conn.createStatement();

ResultSet rec = st.executeQuery("SELECT \* " +"FROM Coal " + "WHERE " +

"(Country='" + arguments[0] + "') " +"ORDER BY Year");

System.out.println("FIPS\tCOUNTRY\tYEAR\t" +"ANTHRACITE PRODUCTION");

//for(int j=1;j<=rec.getMetaData().getColumnCount();j++)

// System.out.print(rec.getMetaData().getColumnName(j)+"\t");

while(rec.next()) {

System.out.println(rec.getString("FIPS") + "\t" + rec.getString(2) + "\t"

+ rec.getInt("YEAR") + "\t" + rec. getFloat (4));

}

st.close();

conn.close();

} catch (SQLException s) {

System.out.println("SQL Error: " + s.toString() + " "

+ s.getErrorCode() + " " + s.getSQLState());

} catch (Exception e) {

System.out.println("Error: " + e.toString() + e.getMessage());

}

//基于JDBC驱动程序

//String url = "jdbc:oracle:thin:@10.42.19.95:1521:omcSource"; //Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();

//Connection con = DriverManager.getConnection(url,"omc","omc");

其中：10.42.19.95可以为远程服务器

1521是Oracle服务器运行的默认端口号

omcSource为数据源

### JDBC注意事项

SQL的列索引是从1开始而不是从0开始的，并且SQL的列名不区分大小写，这样当出现多个列名相同时，只使用第一个匹配的列。

### Java DataStore数据库

#### 建立Java DataStore数据库

#### 建立Java DataStore数据库中的表

# 专题技术问题

## RMI（Remote Method Invocation）

RMI的基本概念是：某应用程序的一个对象（或许是客户端：本地对象）调用完全不同的机器（或许是服务器：远程对象）上运行的其它应用程序中的方法。客户可以传递参数，甚至能够得到返回值。RMI使用对象串行化来传递参数，并且返回两个应用程序之间的值。

要使用RMI，必须构造四个主要的类：远程对象的本地接口、RMI客户、远程对象实现和RMI服务器。RMI服务器生成远程对象实现的一个实例，并用一个特殊的URL注册它，RMI客户在远程服务器上查找对象，若找到就把它转换成本地接口类型，然后就像一个本地对象一样使用它。

### 远程接口（interface）

远程接口类仅仅是一个接口，而不是实现，RMI客户端可以直接使用它，RMI服务器必须通过一个远程对象来实现它，并用某个URL注册它的一个实例。需要为远程对象定义一个接口。步骤如下：

① 定义一个声明方法的公共接口（public interface），本地对象可以使用RMI来调用这些方法；

② 从本地对象可以用RMI机制来调用的公共接口中的每个方法中都必须能够抛出远程异常java.rmi.RemoteException。

③ 该公共接口必须继承自另一个名叫java.rmi.Remote的接口。

本地对象使用这个接口来决定对远程对象进行什么样的操作。

public interface SimpleRMIInterface extends java.rmi.Remote

{

public java.util.Date getDate() throws java.rmi.RemoteException;

}

### 远程接口实现类

远程接口实现类必须扩展RemoteObject类，并且实现远程接口。在远程接口实现类中，你可以定义在远程接口中没有声明的方法，但是这些方法不能被客户端的对象调用，客户端只能调用在远程接口中声明的方法。要定义一个远程接口实现类，实现步骤如下：

① 必须扩展java.rmi.RemoteObject类或者它的一个子类，例如UnicastRemoteObject，来定义远程类。当你要实现一个类，并且该类的对象不能通过多个服务器来复制时，此时扩展UnicastRemoteObject是个通用的方法；

② 实现远程接口中定义的所有方法；

③ 关于构造函数：

a. 调用super();

b. 使用Naming.rebind(name,this);告诉RMI registry该对象可以通过”String name”来调用。

c. 构造函数必须抛出RemoteException。远程类必须定义自己的构造函数，默认的无参数构造函数不起作用，因为该对象的构造函数必须声明抛出一个RemoteException异常；

import java.rmi.\*;

import java.rmi.server.UnicastRemoteObject;

public class SimpleRMIImpl extends UnicastRemoteObject

implements SimpleRMIInterface

{

public SimpleRMIImpl(String name) throws RemoteException {

super();

try {

//将远程接口的对象实例绑定到URL上

Naming.rebind(name, this);

}

catch (Exception e) {

if (e instanceof RemoteException)

throw (RemoteException)e;

else

throw new RemoteException(e.getMessage());

}

}

public java.util.Date getDate() {

System.out.println("SimpleRMIImpl.getDate()");

return new java.util.Date();

}

}

### RMI服务器类

RMI服务器类创建远程接口实现类的对象的一个实例，然后用一个特定的URL来注册它，所谓注册就是通过Naming.bind或Naming.rebind来将远程接口实现类的实例绑定到URL上。具体实现步骤为：

① 创建并安装一个新的服务器端安全管理器（注意RMI与applets使用不同的安全管理器）；

② 创建远程接口实现类的实例对象，该实例对象可以使RMI客户端与RMI服务器端建立联系。

import java.rmi.\*;

import java.rmi.server.UnicastRemoteObject;

public class SimpleRMIServer

{

public static void main(String[] argv) {

System.setSecurityManager(new RMISecurityManager());

try {

//The IP is the server IP，通过构造函数来实现绑定

SimpleRMIImpl implementation =

new SimpleRMIImpl("//10.42.19.215/SimpleServer");

System.out.println("SimpleRMIImpl ready");

}

catch (Exception e) {

System.out.println("Exception occurred: " + e);

}

}

}

### RMI客户端类

RMI客户端类使用Naming.lookup在指定的RMI服务器上查找对象，若找到就把它转换成本地接口类型，然后像一个本地对象一样使用它。

与CORBA不同之处在于RMI客户必须知道提供远程服务主机的URL，这个URL可以通过rmi://host/path或rmi://host:port/path来指定，如果省略端口号，就使用1099。

Naming.lookup可能产生三个异常：MalformedURLException、RemoteException、NotBoundException，三个异常都需要捕获。RemoteException、Naming和NotBoundException在java.rmi.\*中定义，MalformedURLExcepti on在java.net.\*中定义。另外，客户机将向远程对象传递串行化对象Serializable，所以还应在程序中输入java.io.\*。

具体实现步骤如下：

① 创建并安装一个新的客户端安全管理器；

② 使用静态方法Naming.lookup()得到远程接口实现类的一个引用，注意客户端使用远程接口（interface）来保存该引用并调用远程接口中的方法；

import java.rmi.\*;

import java.rmi.registry.\*;

import java.rmi.server.\*;

import java.util.Date;

public class SimpleRMIClient

{

public static void main(String[] argv) {

String serverName = "";

System.setSecurityManager(new RMISecurityManager());

if (argv.length != 1) {

try {

serverName = java.net.InetAddress.getLocalHost().getHostName();

System.out.println("ServerName is: "+ serverName);

}

catch(Exception e) {

e.printStackTrace();

}

}

else {

serverName = argv[0];

}

if (serverName == "") {

System.out.println("usage: java SimpleRMIClient

<IP address of host running RMI server>");

System.exit(0);

}

try {

//bind server object to object in client

//The IP is the server IP

SimpleRMIInterface myServerObject =

(SimpleRMIInterface) Naming.lookup("//10.42.19.215/SimpleServer");

//invoke method on server object

Date d = myServerObject.getDate();

System.out.println("Date on server is " + d);

}

catch(Exception e) {

System.out.println("Exception occured: " + e);

System.exit(0);

}

System.out.println("RMI connection successful");

}

}

### 设置RMI安全管理器

安全管理器必须是运行的，应为RMI要通过网络来装载类。针对客户端和服务器端分别建立安全管理器：SimpleClientRMI.policy和SimpleServerRMI.policy。内容可以设为一样（未进行任何限制）：

grant {

// allows anyone to listen on un-privileged ports

//permission java.net.SocketPermission "\*:1024-65535", "listen,accept,connect";

permission java.security.AllPermission;

};

安全策略文件

Java提供的应用程序环境的安全策略，使得不同的代码对系统资源拥有不同的访问许可。Java应用程序安全策略由Policy对象来表达，通过定义安全策略文件来实现。Java安全策略文件分为三级：系统安全策略文件、用户安全策略文件和缺省安全策略文件。当Java应用程序启动后，按照先后顺序加载安全策略内容。

下面结合典型的安全策略文件内容对安全策略文件的定义进行说明：

grant{

//对系统和用户目录设置“读”的权限

permission java.util.PropertyPermission “user.dir”,”read”;

permission java.util.PropertyPermission “user.home”,“read”;

permission java.util.PropertyPermission “java.home”,“read”;

permission java.util.PropertyPermission “java.class.path”,“read”;

permission java.util.PropertyPermission “user.name”,“read”;

//其他安全策略内容

};

上述安全策略文件定义了应用程序对用户目录、用户登录目录、JDK安装目录、用户名称等系统内容具有读的权限。

### RMI程序的编译

* 编译RMI客户端和服务器端

Javac SimpleRMIInterface.java

Javac SimpleRMIImpl.java

Javac SimpleRMIServer.java

Javac SimpleRMIClient.java

或者直接运行 Javac \*.java

* 生成客户端承接模块和服务器框架

rmic SimpleRMIImpl

这将生成存根SimpleRMIImpl\_Stub.class和大纲SimpleRMIImpl\_Skel.class。存根代表客户端的远程对象，大纲代表服务器端的远程对象。

Java借助于存根和大纲在本地对象和远程对象之间通信。

### RMI程序的运行

* 客户端和服务器端文件的分配

服务器端文件：（目录假定为：E:\temp\forzt）

SimpleServerRMI.policy、SimpleRMIServer.class、SimpleRMIInterface.class

SimpleRMIImpl.class、SimpleRMIImpl\_Stub.class和SimpleRMIImpl\_Skel.class

客户端文件：（在另一台机器上，目录假定为：D:\java\simple）

SimpleClientRMI.policy、SimpleRMIClient.class、SimpleRMIInterface.class

和SimpleRMIImpl\_Stub.class

* 启动RMI注册

在目录E:\temp\forzt下运行rmiregistry。不管有多少个远程对象，本操作只需做一次。

* 运行服务器程序

在目录E:\temp\forzt下运行：

java -Djava.security.policy=file:E:\temp\forzt \SimpleServerRMI.policy SimpleRMIServer

或者运行（指明Server所在的路径）：

java -Djava.rmi.server.codebase=file:E:\temp\forzt\classes\

-Djava.security.policy=file:E:\temp\forzt \SimpleServerRMI.policy SimpleRMIServer

* 运行客户端程序

在目录D:\java\simple下运行

java -Djava.security.policy=file:D:\java\simple\SimpleClientRMI.policy SimpleRMIClient

程序将在客户端程序显示：

Date on server is Mon Mar 10 21:57:40 CST 2003

RMI connection successful

## 网络编程和通信

### 网络编程基础

TCP/IP具有很强的适应性，可以将长消息分成消息片发送，消息片可以选择发送路径来到达正确的目的地，但是它们到达的顺序可能与发送的顺序不同。不但如此，TCP还能将消息片重新组装成正确的顺序，构成发送端的长消息。

服务程序通过监听不同的端口来接收不同的消息。共有个端口，每个端口都分配了一个号码。这些号码分为两组：1024以下的端口号是为系统应用预留的（Web协议，ftp等）；从1024到之间的端口都可以任意支配。

IP地址和端口号的组合唯一地标识机器上的某一个服务。将IP地址和端口组合起来的机制被认为是套接字（socket）。套接字标识双向通信的一端，当客户端请求与某一特定端口处的服务器相连时，该服务器识别并跟踪用于与客户端交流的套接字。

这有助于解释服务器是如何跟踪多个客户端，并能与多个客户端进行交流的。虽然服务器通过同一端口与多个客户端通信，但是它通过套接字（端口＋IP地址）来判定信息的目的地和发源地。服务器会跟踪每一个客户端的套接字，它知道从某套接字发来的数据与从另一套接字发来的数据不同，它们分别来自不同的客户端。服务器还知道根据套接字分别给不同的客户端的请求做出响应。

设计服务器和客户端：服务器在某一特定的端口上监听连接；当收到连接时，服务器获得一个用于通信的套接字，并开始与位于该套接字的客户端对话。客户端初始化这一连接，并借助于代表服务器的套接字于服务器通信。所有从客户端接收或向客户端发送的数据都是从套接字中读取或向套接字中写入的。位于套接字层下面的TCP协议保证从某特定客户端发过来的数据传送到正确的服务器线程的正确套接字。

标准的服务器设计是让服务器始终运行，始终监听于客户端的连接。一旦连接上，就开始于客户端对话。

### 网络包

* InetAddress

InetAddress类对IP地址进行编码。InetAddress的对象不是通过构造函数来创建的，而是通过调用InetAddress定义的3种静态方法之一得到的。

InetAddress ipAddress = InetAddress.getByName(“java.sun.com”);

InetAddress ipAddress = InetAddress.getLocalHost();

* Socket和ServerSocket

Socke类定义了一个套接字，标识双向通信的一端。你需要一个IP地址和一个端口来创建Socket对象。

服务器端：

//服务器创建ServerSocket对象来监听特定的端口(5001)，最大连接客户端数量限制为25个（缺省为50个）

ServerSocket serversock = new ServerSocket(5001,25);

//服务器通过执行下面的代码，来等待客户端的连接. accept()一直等到客户端与服务器进行连接

Socket sock = serversock.accept();

客户端：

//获得服务器端的IP地址

InetAddress ipAddress = InetAddress.getByName(“java.sun.com”);

//如果服务器监听端口5001等待与客户端建立连接，则创建连接于该服务器的套接字

Socket s = new Socket(ipAdderss, 5001);

作为一个程序员，只需要知道的就是服务器监听的原始端口号；服务器和客户端为了建立专门的连接所协商的端口不需要由你来决定，Java类会为你做这些事情。事实上，如果你不在套接字中调用所创建的分别用于识别本地和远程端口的方法：getLocalPort()和getPort()，就将永远认识不到ServerSocket的accept()方法已经为你分配了服务器的端口。

Socket的其它构造函数。例如：你可以将本地端口指定为与远程端口不同的号码。或许你想让套接字的本地端口号为3333，但是你仍旧想连接到监听5001端口的服务器，这可以通过下面的代码来创建套接字并且连接服务器：

Socket sock = new Socket(ipServer, 5001, ipLocal, 3333);

* Internet流

假设sock变量是一个Socket对象。

读入数据：

DataInputStream remoteIn = new DataInputStream(sock.getInputStream());

然后可以用remoteIn从套接字中读取数据，并从相连的机器上获得数据。例如要读一个整数，可以这样写：

int myInt = remoteIn.readInt();

输出数据：

DataOutputStream remoteOut = new DataOutputStream(sock.getOutputStream());

然后可以用remoteOut向套接字中写入数据，并发送到相连的机器上。例如要输出一个整数5，可以这样写：

remoteOut.writeInt(5); remoteOut.writeUTF(“Write a String”);

### 网络编程实例

## 对象序列化（串行化）

一个流（Stream）是程序中数据所途径的一条路径；是一个将数据从一个地方运送到另一个地方的一个对象。一个输入流（input straem）将数据从某个数据源发送到程序中，而一个输出流（output stream）将数据从程序中往外发送到目的端。

字节流和字符流把数据从处理它们的Java类中分离出来。为了以后的使用，必须通过某个流来读取它并将其转换成该类能够使用的某种格式。对象流则使得数据可以表示成对象的一部分，而不是对象外的某个内容。

### 字节流、数据流

字节流（byte stream）能够传送值的范围从0到255的整数。

FileInputStream和FileOutputStream针对存储在磁盘、光盘或其它存储设备上的文件里的字节流。

DataInputStream和DataOutputStream针对被过滤过的字节流，从中可以读取诸如整数和浮点数之类的数据。

FileOutputStream file = new FileOutputStream("400primes.dat");

BufferedOutputStream buff = new BufferedOutputStream (file); //相当于过滤流

DataOutputStream data = new DataOutputStream (buff);

data.writeInt(8888);

创建一个从被压缩ZIP文件读取数字的数据输入流：

DataInputStream dataIn = new DataInputStream (new ZipInputStream (

new FileInputStream ("record.zip")));

### 字符流

字符流（character stream）是一个特殊类型的字节流，它处理由ASCII字符集或Unicode所表示的任何文本化的数据。

FileReader file = new FileReader("ReadSource.java");

BufferedReader buff = new BufferedReader(file);

boolean eof = false;

while (!eof) {

String line = buff.readLine();

if (line == null)

eof = true;

else

System.out.println(line);

}

buff.close();

### 对象流

对于一个要保存到某个目的端的对象而言，它必须被转换成串行化的形式。一个对象通过实现Serializable接口来指明它被用于流中。Serializable接口并不包含必须出现在实现它的类里的任何方法。Serializable的唯一目的是指明可以用串行方式存储和获取得类的对象。串行化使得对象具有持续性persistence（即一个对象在创建它的程序外存在和起作用的能力），若需将对象的某些变量排除在串行化之外，需要使用修饰语transient。

#### 对象输出流

import java.io.\*;

import java.util.\*;

public class ObjectToDisk {

public static void main(String[] arguments) {

Message mess = new Message(); //创建对象

String author = "Author";

String recipient = "Recipient";

String[] letter = { "ABCDEFGHIJKLMNOPQRSTUVWXYZ."};

Date now = new Date();

mess.writeMessage(author, recipient, now, letter); //对象属性的赋值

try {

FileOutputStream fo = new FileOutputStream("Message.obj");

ObjectOutputStream oo = new ObjectOutputStream(fo);

oo.writeObject(mess); //将对象mess写入到oo对象输出流中

oo.close();

} catch (IOException e) {

System.out.println("Error -- " + e.toString());

}

}

}

class Message implements Serializable {

int lineCount; String from, to;

Date when; String[] text;

public transient int limit = 55; //此属性不能被串行化

void writeMessage(String inFrom, String inTo, Date inWhen, String[] inText) {

text = new String[inText.length];

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

text[i] = inText[i];

lineCount = inText.length;

to = inTo;

from = inFrom;

when = inWhen;

}

}

#### 对象输入流

import java.io.\*;

import java.util.\*;

public class ObjectFromDisk {

public static void main(String[] arguments) {

try {

FileInputStream fi = new FileInputStream("message.obj");

ObjectInputStream oi = new ObjectInputStream(fi);

Message mess = (Message) oi.readObject();

System.out.println("Message:\n");

System.out.println("From: " + mess.from);

System.out.println("To: " + mess.to);

System.out.println("Date: " + mess.when + "\n");

for (int i = 0; i < mess.lineCount; i++)

System.out.println(mess.text[i]);

oi.close();

} catch (Exception e) {

System.out.println("Error -- " + e.toString());

}

}

}

### 注意事项

所有的数据都可以被表示成单个字节和一连串的字节。在Java里，一个整数用四个字节来表示，它们按照叫做大数结尾的顺序来摆放；可以通过从左向右将字节组合起来判断出这个这个整数值。在C程序里很可能是以一个小数结尾的顺序来表示整数，这意味着字节应该从右到左摆放来决定最后的结果。

在处理流时，char和int两种原始数据类型可以交互使用。因为一个char在Java内部表示成一个整数值。

## Java多线程

### 如何编写Java多线程程序

#### Java方法的类型

\* Know where the basic thread methods are:

- wait, notify and notifyAll are Object instance methods.

- start, stop, suspend, resume and interrupt are Thread instance methods.

- sleep and yield are Thread static methods

\* Deprecated methods

- stop is unsafe because it releases all locks and may leave objects in an inconsistent state.

- suspend is deprecated because its failure to release locks makes it prone to deadlock. Calling wait in a sync block is safer.

#### 创建Java线程

创建线程有两种方法：

继承java.lang.Thread类及其子类，并重写run方法（本身是空的）：

class PrimeThread extends Thread {}

实现java.lang.Runnable接口及其子接口，实现接口的同时实现了其中的run方法：

class PrimeRun implements Runnable {}

提示：如果你的类必须继承其它一些类（例如Applet小程序），那就应该用Runnable接口来实现要创建的线程。

当创建了一个新的线程，它仅仅是一个空的Thread对象，系统资源并没有分配给它。当线程处于这个状态时，你只能启动该线程。如果调用除了start之外的方法是没有任何意义的，并将导致IllegalThreadStateException。start方法创建该线程运行所需的系统资源，计划线程的启动（根据线程优先级）和调用该线程的run方法。

另外，当线程已经启动，如果再次调用start()方法或调用setDaemon()方法，会导致异常IllegalThreadStateException的抛出。调用deprecated 的countStackFrames()方法，同样也会导致IllegalThreadStateException的抛出。

#### 编写线程执行内容

java.lang.Runnable接口定义了一个方法：public void run()。

Java.lang.Thread类实现了java.lang.Runnable 接口。

将要在线程中执行的内容放在run()方法中即可。

#### 启动Java线程

继承自java.lang.Thread类的情况：

PrimeThread p = new PrimeThread();

p.start();

实现java.lang.Runnable接口的情况：

PrimeRun p = new PrimeRun();

new Thread(p).start(); //等价于Thread runP = new Thread(p); runP p.start();

线程被启动后，预先run()方法就开始被执行。当run()方法执行完成后，线程的生命期也就结束了。

**Q**: After a thread started, if the reference of the thread is set to null, what will happen to the thread? Does it stop? Is it eligible for GC?

**A:** No. Live thread will continue running even its reference is set to null.

When you call Thread start() method, JVM will create a new Thread, then call run() method of the new Thread. The new Thread will run concurrently with the oringinal calling Thread.

If you directly call the run() method, the Thread will act as a normal Java object. No new Thread will be created. The code will run sequentially.

Try the following code to see the difference, and get a feeling of concurrency.

// MyRun.java

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

public class MyRun implements Runnable {

public static void main(String argv[]) {

MyRun r = new MyRun();

Thread t = new Thread(r);

// t is still running after Main finished

t.start();

// No new thread created, Main finishes after run() returns

// t.run();

try {

// See concurrency if you call t.start()

// See sequential if you call t.run()

for (int i = 0; i < 5; i++) {

Thread.sleep(10);

System.out.println("in Main: " + i);

}

}catch (InterruptedException e) {

System.out.println(e);

}

System.out.println("Main thread finished...");

}

public void run(){

try {

for (int i = 0; i < 5; i++) {

Thread.sleep(20);

System.out.println("in Run: " + i);

}

}catch (InterruptedException e) {

System.out.println(e);

}

System.out.println("Run method finished...");

}

}

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

系统的输出为：

in Main: 0

in Main: 1

in Run: 0

in Main: 2

in Main: 3

in Run: 1

in Main: 4

Main thread finished...

in Run: 2

in Run: 3

in Run: 4

Run method finished...

#### Java线程的状态迁移

线程共有下列四个状态（Bruce Eckel lists），而且总是处于其中的一个状态，并可在不同状态间迁移。

New （新建）：当用一个new 操作符创建一个线程时；

Runnable （可运行）：线程对象的start()方法被调用后（并非真的一定在运行）；

Blocked （阻塞）：当线程调用sleep()或wait()方法、等待加锁成功、等待I/O结束、suspend()方法被调用（不推荐）时；

Dead （终止）：线程正常退出或遇到非预期的异常突然死亡，导致线程进入终止状态。

以下是线程的状态图，需要好好理解：



#### 阻塞和激活Java线程

线程在运行过程中，下列操作会使线程进入阻塞状态：调用sleep方法；调用wait方法；调用join方法；等待同步加锁；等待I/O操作完成。

线程也许活着但却不能运行，详细的说有以下原因：1.该线程不是具有最高优先级的线程，因此得不到CPU时间。2.线程已经被sleep()方法置成休眠。3.有多个线程具有相同的最高优先级，JVM正在这些线程间切换；此时正被讨论（in question）的线程正在等待CPU时间。4.线程堵塞，正在等待I/O调用，或者没有成功获得监视器的而而正在等待监视器的锁。5.因为这个线程调用了wait()，所以线程正在等待条件。线程正在等待notify()或notifyAll()。6.线程通过调用yield()方法显示地放弃控制。

线程进入阻塞状态后，不进行任何操作，释放对所有资源的锁定。

下面分别讨论几种阻塞情况。

■调用sleep方法：

Thread类提供了两个静态sleep方法：

static void sleep(long millis)和static void sleep(long milis, int nanos)

示例

try {

Thread.sleep(1000);

} catch(InterruptedException e) {

e.printStackTrace();

}

当线程调用sleep方法时，线程进入阻塞状态；进入阻塞状态后，线程锁定的资源将被解锁；当sleep期限到达时，线程被唤醒，进入运行状态。当sleep期限未满而线程对象被调用interrupt方法时，线程将立即被唤醒，并抛出InterruptedException异常。

当在一个synchronized method或synchronized block中调用Thread.sleep()时，Thread并不释放线程锁。

public class SleepTest implements Runnable{

public void run() {

while (true){ sleepInsteadOfWaitNotify(); }

}

synchronized void sleepInsteadOfWaitNotify() {

while (true) {

System.out.println(Thread.currentThread().getName() + "Go to sleep");

try{ Thread.sleep(400); }

catch (Exception e){ System.out.println(e); }

System.out.println(Thread.currentThread().getName() + "wake up");

}

}

public static void main (String[] args){

SleepTest t = new SleepTest();

Thread t1 = new Thread(t);

Thread t2 = new Thread(t);

Thread t3 = new Thread(t);

t1.setName("t1"); t2.setName("t2"); t3.setName("t3");

t1.setDaemon(true); t2.setDaemon(true); t3.setDaemon(true);

t1.start(); t2.start(); t3.start();

try {

// let daemon threads run

Thread.sleep(10000);

}

catch (InterruptedException e) { }

System.out.println("If user thread is dead, all daemon threads die. ");

}

}

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

■调用wait方法：

Object类提供了3个wait方法：

void wait()：当前线程进入无限期阻塞状态，直到被唤醒。

void wait(long timeout)：当前线程进入阻塞状态，直到被唤醒或超时期限到达。

void wait(long timeout, int nanos)：当前线程进入阻塞状态，直到被唤醒或超时期限到达。

使用wait方法，线程可以阻塞在任何一个对象上。示例：

LinkedList queue = new LinkedList();

queue.wait();

线程被阻塞后，所有本线程加锁的资源被释放。

Object 类提供了下列方法用于唤醒调用wait阻塞的线程：

notify()：唤醒一个线程。被唤醒的通常是等得最长的线程，但不能保证被唤醒是等待最长的线程。如果有多个线程阻塞，则任选一个唤醒。

public final void notify()

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened.

The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object. The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object;

notifyAll()：唤醒所有的线程。JVM可能会选择一个优先级最高的线程运行

public final void notifyAll()

Wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.

The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object. The awakened threads will compete in the usual manner with any other threads that might be actively competing to synchronize on this object;

Sun的官方网站。清楚地说明了两个问题：

1、notify只唤醒这个对象上多个线程之一个。notifyAll是所有的。

2、不管是notify还是notifyAll，被唤醒的进程还要竞争这个对象上的锁。（注意，当执行wait的时候，要暂时释放这个对象的锁，所以醒来的时候还需要重新竞争得到锁）

Q. How to call wait(), notify(), notifyAll() in syncronized static methods?

A:

Direct calling those methods will not compile since those are instance methods inherited from Object. When you call wait(), you are actually calling this.wait(), this does not exist in static methods.

How can we solve the problem? There are two ways to do it.

1. Call wait(), notify(), notifyAll() methods of your Class object, by using MyClass.class.wait(). Syncronized static methods locks the Class, you release lock of the same object.

2. Instantiate a static Object dummy in your class, you can use syncronized block, which locks dummy, in your static method. Then you can call dummy.wait(), dummy.notify(), ... in those blocks.

This is better when you have more than one set of static methods, which needs locked independently, you can use more than one dummy objects for that purposes.

If your problem is something like that a Queue problem where add and remove are static menthods working on static array. Lock the array object!!! See sample code snippet below.

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

static Object[] ary = new Object[50];

static void method1() {

System.out.println("do something not need to lock the ary...");

syncronized(ary){

System.out.println("do something on ary Object..");

try{

ary.wait();

}

catch(InterruptedException e) {

}

System.out.println("do something else on ary Object..");

}

System.out.println("do something else not need to lock the ary...");

}

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

一个带有陷阱的程序：

下面这个例子怎么产生结果：

1 Waiting for calculation...

2 Waiting for calculation...

3 Waiting for calculation...

4 Waiting for calculation...

5 Waiting for calculation...

Starting calculating ...

1 cal.total: 4950

2 cal.total: 4950

4 cal.total: 4950

3 cal.total: 4950

5 cal.total: 4950

notify只是唤醒等待此对象的多个线程之一，怎么全部唤醒了呢？

public class Reader extends Thread {

Calculator c;

public Reader(Calculator calc, String name) {

c = calc;

setName(name);

}

public void run() {

synchronized(c) {

try {

System.out.println(getName()+" Waiting for calculation...");

c.wait();

} catch (InterruptedException e) {}

}

System.out.println( getName()+" cal.total: " + c.total);

}

public static void main(String [] args) {

Calculator calculator = new Calculator();

calculator.start();

new Reader(calculator, "1").start();

new Reader(calculator, "2").start();

new Reader(calculator, "3").start();

new Reader(calculator, "4").start();

new Reader(calculator, "5").start();

}

}

class Calculator extends Thread {

int total;

public void run() {

try {

Thread.sleep(1000); //为了保证 Reader 所有的线程进入 wait

} catch (InterruptedException e ) {}

System.out.println("Starting calculating ...");

synchronized(this) {

for(int i=0;i<100;i++) {

total += i;

}

// notify();

}

}

}

处于阻塞状态的线程对象调用interrupt方法，线程被立即唤醒，抛出interruptedException异常。线程被唤醒后，进入运行状态。

■线程进行I/O操作：

正在进行I/O操作的线程，当I/O操作指令已发出但操作尚未完成时，进入阻塞状态。由I/O操作引起的线程阻塞，对线程对象调用interrupt()方法，只能将线程的interrupt状态置位，无法将其唤醒。

只能等待I/O操作完成，线程才能被唤醒。

■调用join方法：

Thread类提供了3个join方法：

void join()：阻塞当前线程，直到指定的线程对象终止。

void join(long millis)：阻塞当前线程，直到指定线程终止或阻塞期限到达。

void join(long millis, int nanos)：阻塞当前线程，直到指定线程终止或阻塞期限到达。

线程阻塞后，所有本线程加锁的资源释放。当指定线程终止，或等待期限到达，当前线程被唤醒。阻塞状态下线程调用interrupt()方法，线程将立即被唤醒(结束sleep或者wait状态)，并抛出InterruptedException异常。

Q. What does the Thread.join() method do? Can you give an example?

A: The current running thread currT call another thread jobthread.join(), then wait until the jobthread to die.

For example, The current running thread which spawns the jobthread to do something (do a long calculation, load several images, call somebody through satilite, etc. etc.). The jobthread has been coded to do its job and then terminate or die, in other words, the jobthread.run() method ends. The current thread will rely on jobthread's completion to continue its own work. The effect of calling jobthread.join() statement is that currT waits for jobthread to complete

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

public class JoinDemo {

public static void main(String[] args) {

Job job = new Job();

Thread jobthread = new Thread(job);

jobthread.start();

try {

// the current thread will wait until the job done

jobthread.join();

}

catch(Exception e){

}

System.out.println("Job done");

// do something else

}

}

class Job implements Runnable {

public void run(){

int i;

for (i=1; i<=200; i++ ){

if (i % 10 != 0) {

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

}

else {

System.out.println(i);

}

}

}

}

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

#### 结束Java线程运行

当run()方法结束，或线程执行过程中发生了未预期的异常时，线程就会结束运行。该线程给认为是死的（dead），一个死线程不能重新启动(a dead Thread cannot be restarted.)。如果你要再次运行线程的任务，就必须构造并启动一个新的线程实例。

记住，尽管死的线程作为一个对象（Thread或Runnable）仍然存在，但是它不能像一个独立的线程那样运行。当线程结束运行时，线程所占用的所有资源都被释放，等待回收。结束线程运行的最佳方法是退出线程对象的run()方法。

虽然Thread类提供了一个stop()方法，但属于禁用方法，因为stop()方法的调用者无法确定何时中止线程是安全的。除非迫不得已，否则不要使用stop()方法。

Thread类还提了一个destroy()方法，但该方法未提供实现。

**Q. How can we stop a thread without call stop()?**

**A:**

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

**public class T extends Thread {**

**boolean runflag = true;**

**public T(String name){**

**super(name);**

**}**

**public void run() {**

**int i = 0;**

**while (runflag) {**

**System.out.println(getName() + ": " + i++);**

**try**

**{**

**sleep((int)(400\*Math.random()));**

**}**

**catch (InterruptedException e) {**

**}**

**}**

**System.out.println(getName() + " is stopping.");**

**}**

**public void setRunFlagFalse() {**

**runflag = false;**

**}**

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

**T t1=new T("t1");**

**T t2=new T("t2");**

**T t3=new T("t3");**

**t1.start();**

**t2.start();**

**t3.start();**

**try**

**{**

**// let three threads run**

**Thread.sleep(1000);**

**}**

**catch (InterruptedException e) {**

**}**

**// stop them**

**t1.setRunFlagFalse();**

**t2.setRunFlagFalse();**

**t3.setRunFlagFalse();**

**}**

**}**

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

#### Java线程对象的其它特性

Java线程具有以下属性：

线程的优先级：在Java语言中，每个线程都有一个优先级。缺省情况下，一个线程继承其父进程的优先级。由于CPU的调度是抢占式的，JVM的线程调度程序总是选取处于runnable状态的且有最高优先级的线程投入运行。优先级最高的线程是Java正在运行的线程，其它线程必须等待。如果多个线程具有同样的最高优先级，Java会在它们之间切换，通常会快的使你意识不到线程正在“执行-休眠”的循环里交替。Java能在任何数量的有相同优先级的线程间切换。

Java调度器的任务是使高优先级的线程能始终先运行：支持时间片的平台，同等优先级的线程以轮流的方式顺序使用时间片；不支持时间片的操作系统中，同等优先级的线程抢占处理器直至任务结束，其他的线程则等待。

可以用Thread类的setPriority(int newPriority)方法来改变线程的优先级。其中newPriority参数的取值范围为1-10间的正整数，取值越大则优先级越高。由于JVM需要将设定的优先级映像到操作系统的优先级，因此在不同的OS的运行表现不尽相同。

设置优先级应在创建线程后的任何时间均可，没有任何的构造函数能够设置线程的优先级。

JVM基于优先级等级决定线程何时运行，但是这并不意味着一个低优先级的线程一直不运行。这一点很重要，因为不应该依靠优先级精确预测什么时候你的程序会发生什么事。

你不必依靠JVM在具有相同优先级的线程上切换。通过调用yield()，当前执行的线程会放弃控制，使其它线程获得执行的机会。如果这样，Java挑选一个新的线程来运行，但是如果刚才放弃的是具有最高优先级的线程，那么它也有可能又立刻运行。

线程名：每个线程都有String类型的线程名。多个线程可以拥有同样的线程名。当不作定义时，JVM为线程分配缺省的线程名。可以通过Thread类的getName()方法和setName()方法获取或修改线程名。

Every thread has a name for identification purposes. More than one thread may have the same name. If a name is not specified when a thread is created, a new name is generated for it.

Nothing changes by assigning your Thread a name. It only gives you the convenience of debug, since it has a name you know.

所属线程组；

线程状态（运行/挂起）；

ready：当你创建了一个线程，线程并不马上执行。你必须调用线程的start()方法，然后线程才进入ready状态，等待调度线程把它置成running状态。一个在先前执行过的线程，如果进入了waiting状态，之后又准备好继续执行，这个线程也能进入ready状态。如果调度程序运行线程生成，调用yield()方法也能将线程置成ready状态。在使用监控器和同步的时候，线程也可能因为调用notify()方法或notifyAll()方法而进入等待状态。

running：当线程调度程序给线程分配了CPU时间的时候，线程处于running状态。这意味着线程正在执行，线程能够被中断或者离开这个状态。

waiting：线程会由于许多不同的原因进入waiting状态。线程可能在等待输入或者正在休眠。线程可能被中断或者它的执行被阻塞。在使用监控器和同步的时候，线程可能因为调用wait()方法而进入等待状态。

中断（是/否）：

Java线程对象的特色操作：

Interrupt()：中断线程；当线程的run()方法返回时，线程就中止了。既然不赞成使用stop()方法，那么run()方法就应该有一个循环检查逻辑来判断自己是否应当中止。当一个线程处于阻塞状态时，无法主动检测自己是否应当中止。此时可以调用线程对象的interrupt()方法。当interrupt()方法作用于一个处于阻塞状态的线程时，run()方法将会抛出InterruptedException异常，可以将线程从阻塞中唤醒的作用。当interrupt()方法作用于一个处于运行状态的线程时，不产生任何异常。

线程的属性之一是“interrupt状态”。当线程对象被调用interrupt()方法时，interrupt状态就会被置位。当InterruptedException异常被抛出后，interrupt状态就被清除了。有两个方法可以检测线程被中断：

interrupted()：静态方法，用于检测当前线程是否已被中断，并且重置当前线程的interrupt状态。

isInterrupted()：实例方法，用于检测线程实例是否被中断。本方法不重置interrupt状态。

yield()：暂停当前线程；多线程的工作环境下，必须注意避免利己线程的出现。所谓利己线程是指从不主动放弃对处理机占用的线程。在某些OS中，利己线程会引发系统性能恶化。Thread类提供了一个静态方法static void yield()：暂停当前运行的线程，线程状态不改变；在线程的工作过程中，必须确保有不间断的yield或sleep方法调用。同时，要记住，使用yield()时，优先级最高地线程未必首先运行。

currentThread()：获取当前线程引用；Thread类提供了currentThread静态方法，用于获取调用者所在线程的引用。static Thread currentThread()

dumpStack()：打印当前线程堆栈信息；Thread类提供了dumpStack静态方法，用于获取当前线程的堆栈信息。static void dumpStack()。Debug场合有很大用途。比如：想知道某个方法的调用者方法是谁；想知道在执行流中每个方法占用的时间等。

#### Java线程组

一组功能相关的线程可以组成一个线程组。构造一个线程组：

ThreadGroup g = new ThreadGroup(name);

向线程组中加入成员：

Thread t = new Thread(g, threadName);

线程组可以拥有子线程组。缺省情况下，新创建线程组是当前线程组的子线程组，也可特别指定。

可以对同组内的线程进行集中管理。

中断线程组中的所有线程：g.Interrrupt()；

获取线程组中处于运行状态的线程数目：g.activeCount();

获取最高优先级：g.getMaxPriority();。

#### 守护线程

守护线程是一种特殊线程：当JVM要确定是否退出时，守护线程是被忽略的。也就是说，当有10个守护线程，而你开发的最后一个线程退出时，JVM将结束整个进程运行。守护线程创建的线程也是守护线程。当JVM启动时，通常只有一个非守护线程在运行main方法。

Java程序会一直运行到只剩下守护线程。Java运行时由运行程序的守护线程构成。线程在创建时可以被设置成守护线程或者用户线程。

可以通过Thread类的getDaemon方法判断一个线程是否是守护线程，通过setDaemon方法将对线程的daemon属性进行置位和复位，但变更只能发生在start()方法被调用以前。

### 线程间的同步

The key concept of synchronized keyword is locking the code to prevent other thread to access the critical code when one thread is processing it. You need acquire the monitor/lock from an object to lock it, and one object only has one lock/monitor. That is why when one thread acquired the lock/monitor, other threads have to wait.

Why you need to lock the code? Mostly you need to keep the integrity of your data, and which is very application specific. The locking mechanism actually knows nothing about it. It is NOT responsible for your data integrity at all. It is you, the application programmer, who is responsible for it.

Java has built-in support to prevent collisions over one kind of resource: the memory in an object. Since you typically make the data elements of a class private and access that memory only through methods, you can prevent collisions by making a particular method synchronized. Only one thread at a time can call a synchronized method for a particular object (although that thread can call more than one of the object’s synchronized methods). Here are simple synchronized methods:

synchronized void f() { /\* ... \*/ }

synchronized void g(){ /\* ... \*/ }

Each object contains a single lock (also called a monitor) that is automatically part of the object (you don’t have to write any special code). When you call any synchronized method, that object is locked and no other synchronized method of that object can be called until the first one finishes and releases the lock. In the example above, if f(　) is called for an object, g(　) cannot be called for the same object until f(　) is completed and releases the lock. Thus, there’s a single lock that’s shared by all the synchronized methods of a particular object, and this lock prevents common memory from being written by more than one method at a time (i.e., more than one thread at a time).

#### 基于同步锁的理解

多线程环境下系统必须保证能够可靠地同步各线程的动作。实现同步的关键在于提供原子操作能力。Java通过提供synchronized关键字，提供了原子操作能力，即保证了一次只有一个线程执行对象或类的代码。在使用任何一个Java对象时，都必须关注它是否具有thread-safe特性。

Java的同步机制是基于对象锁来实现的。每个对象都有一个仅被一个线程控制的锁计数器，用来标识锁拥有者调用了多少被同步的方法。当锁拥有者调用一个新的同步方法时，锁计数器加1。一个同步方法终止时，锁计数器减1。

Every Java Object has one lock, and only one lock. If you want enter synchronized code, you have to aquire the lock first.

When you enter the static synchronized code, you need to acquire the lock on the Class object. When you enter the instance synchronized code, you need to acquire the lock on the instance object. However, they are different objects with different locks, and independent of each other.

What is critical section? Only one thread can enter that section of code.

Only code section in static synchronized method can be considered as critical section, since the Class object is locked. Two threads can never enter that code section at the same time.

Code block in instance synchronized method is not critical section. Since one instance of object is locked, but other thread still can enter the same section of code through different instance of object

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

class HasCriticalSection {

static Object o1 = new Object();

Object o2 = new Object();

// attention, the method is not even synchronized

void aMethod() {

// Make a code block as critical section

synchronized(o1) {

// lots of code here in the critical section

// only one thread can enter here.

}

}

// attention, the method is not even synchronized

void bMethod() {

// This code block is not critical section

// since o2 is not static member of the class

synchronized(o2) {

// only one thread can enter here,

// per instance of HasCriticalSection Object

}

}

// When one thread enters this cMmethod, the instance object is locked

// No other thread can enter cMetuod() or dMethod()

// which will NOT prevent other thread enter aMethod() or bMethod()

// Of course, you need aware of it, and take advantage of it.

synchronized void cMethod() {

// lots of code here

}

synchronized void dMethod() {

// lots of code here

}

}

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

**Q. How many locks on a Java object? If the answer is one, is it possible to make 2 different methods locked individually? Which means only one thread can enter an individual method, but different threads can enter different methods at the same time?**

**A: Only one lock per object in Java.**

It is possible to make 2 different methods locked individually. Then how?

Using synchronized blocks instead of synchronized methods. Each proposed blocks can lock the same or different member objects. This ought to depend on the relevancy of those blocks. If you want only one thread can enter this group of synchronized blocks, then synchronized on the same object. Otherwise, different objects.

In this way, we are not limited on synchronized methods, which allows only one Thread entering all its synchronized methods. A lot of flexibility can be achieved.

同步消除了一个线程在改变另一个线程的稳定对象状态时发生的并发错误。

1．可以在方法前增加synchronized关键字来实现对对象的同步访问。示例：

public synchronized int getNextMediaID();

调用方法getNextMediaId时，导致对象this被锁定。当方法返回或异常终止时，对象被自动解锁。

用synchroinzed关键字来修饰一个实例方法，当线程调用该方法时，会导致方法所在的对象被加锁（锁计数器加1）

用synchronized关键字来修饰一个类方法（静态方法），会导致类本身被加锁。

同步方法中涉及到的非本对象之外的其它对象，应当具备线程安全属性。

2．另一种同步访问的方法是在语句块前使用synchroinzed关键字（代码块）。示例：

synchronized (pct) {

for(int i=0; i<PCT\_SIZE; i++) {

if (pct[i] == Thread.currentThread()){

return pct[i].getPID();

}

return null;

}

}

使用这种方法进行同步，锁定的对象可以任意指定。可以细化同步对象，提高运行效率。语句块中涉及的其它对象，应具备线程安全特性。

#### 基于监视器的理解

Java在语言的内部提供了在多个线程间协调和同步的方式。每个对象和类都有一个监视器（monitor）。线程可以暂时用于监视器的所有权，过些时候再释放（以便另一个线程可以对该监视器享有暂时的所有权）。一次只有一个线程拥有一个特定的监视器。通过拥有监视器，一个线程阻止其它线程对监视器所属对象或类所定义的同步方法进行操作。

有三种类型的代码能够被同步化：

类方法或实例方法：要声明一个方法为同步化的代码，在声明方法时必须要使用关键字synchronized。

一个方法中的任何代码块：要声明一个代码块为同步化代码，在这个块的前面要使用关键字synchronized，然后在圆括号里指明这些代码要获得哪个对象或那个类的监视器。

如果代码是类方法，则可以在代码块中使用关键字synchronized从类中获得监视器；如果代码是实例方法，则可以在代码块中使用关键字synchronized从对象中获得监视器。关键字synchronized保证了一次只有一个线程执行对象或类的代码。如果你定义了一个实例方法为synchronized，根据优先级，任何覆盖这个方法的子类都可以是synchronized。

如果你进入同步方法，而且唤起了一个异步方法，同步方法将一直有效。因此如果一个子类覆盖了一个异步方法，并且把它声明为同步方法，即使该方法用关键字super调用异步的超类方法，执行同步方法的线程也会继续控制对象的监视器。线程只有在同步方法返回后才放弃监视器。

当等待的线程暂停时，它会释放对象的监视器，并且等待重新获得监视器的通知。这通过调用wait()来进行。通常wait()放在while子句的里面。这么做的想法是源于一个线程被唤醒后，它的等待条件还没有改变。因此，如果有必要应该重新检查条件，线程应该再等待一次。

关于调用wait()的两件事情：第一，wait放于try-catch块里。wait()方法也许会抛出一个InterruptedException异常情况，因此代码必须准备处理异常情况。第二，wait()被放在同步块里，wait()和notify()方法只能从同步代码里调用。块或方法需要同步的监视器必须是属于线程正在等待的对象。此外，如果你没有对象的监视器而企图调用wait()，Java运行时会抛出异常IllegalMonitorStateException。

在同步代码里放置对wait()和notify()的调用时需要做两件事情。第一，要保证当前运行的代码拥有监视器，这样代码能在调用wait()和notify()的时候释放监视器。第二，要尽量确保被用来决定条件的对象内容是稳定的。例如，一个while循环包含wati()调用，在代码调用notify()后离开while循环前，条件不能变为false，这一点很重要。

notify()方法唤醒等待重新获取对象监视器的线程。

Java用监控器(monitor)实现同步。所有使用synchronized方法的对象都是一个监控器，监控器每次只允许一个线程运行对象的一个synchronized方法，如果有多个synchronized方法，那么一次只能有一个对象的一个synchronized方法活动，而其他试图调用synchronized方法的线程则必须等待，当synchronized方法运行结束后，监控器才会打开该对象的锁并让高优先级的处于就绪状态的线程调用synchronized方法。

关于对象锁定和synchronized()指令遵循以下：

1. 每个对象有一把锁.这个锁通常与对象本身相关联

2. 为了执行synchronized(obj)语句，线程应能取得对象的锁,取锁后对象就不再持有这把锁.

3. 如果线程开始执行synchronized(obj)指令时锁不在obj中,则线程停止运行，直到锁返回对象.

4. 线程离开取锁进入块时，将锁交还给对象.

5. 可以在程序任何地方用

synchronized(obj)

{

执行代码;

}

这样的语句保护数据项目。

### 线程的其它注意事项

关于线程调度问题：

JVM规范中对线程调度的要求如下：

所有Java线程都有一个优先级。原则上，JVM会将最高优先级可运行线程投入运行。（由于各种OS线程调度策略的差异，以及优先级定义的映像精确度有限，并不保证总能做到这一点。）一个系统可对线程使用时间片，但并不是必须的。如果存在时间片，将时间片用于全部线程或仅用于优先级内，由实现者自行决定。

由此可见，调度规范相当宽松，不同OS的JVM实现差异很大。因此，对调用行为永远不要做出最佳设想，而总是作最坏的假定。

必须设定线程能在任何点随时交换。一定不能要求线程在某时间交换。如果需保证不同线程的过程，必须显式地编程以保证这一过程发生。

Java规范中并未说明对线程数目的限制，与具体的平台实现有关。

关于Thread类的isAlive()方法：

isAlive()方法告诉调用者在调用时线程是否处于运行状态，但应当注意当获得这个信息时，线程状态可能已经发生变化。

IsAlive方法返回true意味着线程已经启动并且没有停止，你可以知道线程是Runnable或者不能Runnable的。

IsAlive方法返回false意味着线程是一个新建的线程或者是一个已经死掉的线程。

通过isAlive方法你不能区分一个新建的线程或者一个已经死掉的线程，同样也不能区分一个Runnable或者不能Runnable的线程。

线程对象能重复使用吗？

不能。一但线程结束，它就不再存在，堆栈被释放，内部线程结构被清除，所剩下的只有线程对象的空壳，线程不能再被启动。虽然Thread对象不能复用，但runnable对象可以复用。

### Java多线程的缺点

线程调度效率不高；在不同的硬件平台和操作系统上表现不尽相同；线程优先级不可靠；

无有效的死锁检测和异常中止手段

### 线程Block或Stop的问题

1. 下面哪些可以引起一个线程停止运行？

2. 下面哪些不可以引起一个线程停止运行？

A. Exiting from a synchronized block.

B. Calling the wait method on an object.

C. Calling the notify method on an object.

D. Calling the notifyAll method on an object.

E. Calling the setPriority method on a thread object.

F. Calling the yield method

G. Calling the start method on another Thread object

B. Calling the wait method on an object

F. Calling the yield method

这两个肯定可以引起一个线程停止运行！调用wait()可以使此线程由running状态转为同步对象的waiting pool里；调用yield()使此线程由running状态转为runnable状态。

G. Calling the start method on another Thread object

这个不能选，因为调用start()使此新建的线程进入rnnable状态！

E. Calling the setPriority method on a thread object

调用此方法可以设置一个线程对象的优先级！java thread schedualer采用preemptive机制，即优先级高的线程可以抢先被执行，同优先级的线程采用轮流机制。但是这种情况是对所有的runnable线程来说的。此项也不能选！

A，C，D涉及到线程的同步问题！执行这三种情况的线程的状态肯定running，这三种情况可以使此线程把同步对象的lock让出来，让处于此同步对象的waiting pool里的线程进出此同步对象的lock pool!!并不是让他们马上运行！所以我觉得这三种情况也不是答案！！

F不是肯定，只是有可能！！因为如果没有其他的runnable线程，此线程继续运行！

我在amay's notes上看到的！

there are 7 possible for causing thread to stop executing:

1．exiting from synchronized block

2．calling the wait method on the object

3．calling read method on the InputStream object

4．priorer thread enters ready state

5．calling system.exit(0)

6．The thread executes a sleep() call

7．A call to the thread's stop method

\* Blocking

- There are 5 ways a thread can be blocked - sleep, wait, suspend, synchronization, io blocking.

- sleep and suspend do not release locks held by the thread.

1.public class SyncTest {

2.private int x;

3.private int y;

4.private synchronized void setX (int i) {x=1;}

5.private synchronized void setY (int i) {y=1;}

6.public void setXY(int i){setX(i); setY(i);}

7.public synchronized Boolean check() {return x !=y;}

8.}

Under which conditions will check () return true when called from a different class?

A.Check() can never return true

B.Check() can return true when setXY is called by multiple threads

C.Check() can return true when multiple threads call setX and setY separately.

D.Check() can only return true if SyncTest is changed to allow x and y to be set separately.

选B

1.public class SyncTest {

2.private int x;

3.private int y;

4.public synchronized void setX (int i) {x=1;}

5.public synchronized void setY (int i) {y=1;}

6.public synchronized void setXY(int i){set X(i); setY(i);}

7.public synchronized Boolean check() {return x !=y;}

8.)

Under which conditions will check () return true when called from a different class?

A.check() can never return true

B.check() can return true when setXY is called by multiple threads

C.check() can return true when multiple threads call setX and setY separately.

D.Check() can only return true if SyncTest is changed to allow x and y to be set separately.

E.Not of All

选C （注意B、C选项的修饰符public和private）

### Java单线程实例

import java.\*;

class A {

public static void main(String[] args) {

new A ().go();

}

void go() {

Thread a = new Thread() {

//Runnable a = new Runnable(){ 这种方式也可以

public void run(){

System.out.println("Anonymous Inner Class");

}

}; //采用内类来实现

Thread t = new Thread(a);

t.start();

}

}

### Java多线程实例

#### 变量同步的方法

变量不能使用关键字synchronized。这意味着在下面的代码中，尽管一个线程可能处于updateBalance()代码执行过程中，另一个线程也许还会来读取balance。

class Account {

double balance;

synchronized void updateBalance(double amount) {

balance += amount;

}

}

因此需要利用存取器方法提供变量balance的访问方法并实现它的同步访问。

class Account {

private double balance;

synchronized double getBalance() {

return balance;

}

synchronized void setBalance(double newBalance) {

balance = newBalance;

}

synchronized void updateBalance(double amount) {

setBalance(getBalance() + amount);

}

}

#### 线程同步的例子

import java.awt.\*;

import java.applet.\*;

import java.awt.event.\*;

public class ClickApplet extends Applet implements MouseListener {

boolean clicked;

int counter;

public void init() {

add(new ClickCanvas(this));

add(new ClickCanvas(this));

addMouseListener(this);

}

public void mousePressed(MouseEvent e)

{

synchronized (this) {

clicked = true;

notifyAll(); //notify();

}

counter++;

}

// stubbed methods from MouseListener interface

public void mouseClicked(MouseEvent e) {}

public void mouseReleased(MouseEvent e) {}

public void mouseEntered(MouseEvent e) {}

public void mouseExited(MouseEvent e) {}

}

class ClickCanvas extends Canvas implements Runnable{

ClickApplet applet;

ClickCanvas(ClickApplet applet) {

this.applet = applet;

setBackground(Color.blue);

setSize(30, 30);

new Thread(this).start();

}

public void run() {

while (true) {

synchronized (applet) { //同步的代码与对象的实例有关，其中一个实例处于wait状态；另一个实例在代码synchronized (applet){}之外，处于阻塞状态。这样理解有误！！！

while (!applet.clicked) {

try { applet.wait(); }

catch (InterruptedException x) { }

}

applet.clicked = false;

}

repaint(250);

}

}

public void paint(Graphics g) {

g.drawString(new Integer(applet.counter).toString(), 10, 20);

}

}

## JNI与系统交互

## 创建自己的package

## JavaBean

# 高效编程指南

## 创建及销毁对象

关注创建和销毁对象：什么时候、如何创建对象，什么时候、如何避免创建对象；如何保证对象被适时地销毁；对象销毁之前如何处理清除工作。

### 考虑用静态工厂方法替代构造函数

公共的静态工厂方法（static factory method）是一种返回类实例的静态方法。类可以给客户提供静态工厂方法来取代构造函数，或者在提供构造函数的同时，提供静态工厂方法。用静态工厂方法替代构造函数，既有优势又有不足。

**与构造函数相比，静态工厂方法的一个优势是，它们具有自己的名字**。构造函数的参数自身无法描述被返回的对象，而选用名字合适的静态工厂方法可以使类的使用更加容易，产生的客户代码更容易理解。

类对于一个给定的签名只能有一个构造函数。尽管可以采用提供两个构造函数的方法来避开这一限制（这两个构造函数的参数列表相同，但参数类型的顺序不一样）。但这不是好主意，使用这种API的用户将无法分清两个构造函数，如果不参考相关类的文档，将无法知道代码要做什么。

因为静态工厂方法有名字，它们不会具有构造函数那样的限制（每个签名只能有一个方法）。当类在相同的签名下需要多个构造函数时，应该考虑使用静态工厂方法替代构造函数，通过精心地选择方法名可以很好地区分它们。

**与构造函数相比，静态工厂方法的第二个优势是，它们不需要在每次调用时都去创建一个新的对象**。这使得非可变类可以使用预先构造地实例，或者在构造阶段先缓存这些实例，然后反复地使用它们，从而避免创建不必要地重复对象。例如静态工厂方法：

public static Boolean valueOf(boolean b){

return(b?Boolean.TRUE: Boolean.FALSE);

}

方法Boolean.valueOf(boolean)阐明了这一特性：它从不需要创建对象。当频繁地请求相同的对象时，尤其是创建对象所需代价很大时，使用这一技术可以大大地提高性能。

静态工厂方法为重复调用而返回同一对象的能力，也可以用来控制某一时刻实例的存在情况。有两个理由使静态工厂方法可以做到这一点。首先，它能够使类保证实例是singleton；其次，它能够使非可变类保证没有两个相等的实例同时存在，即只有当a==b时，a.equals(b)才成立。如果类能够保证这一点，那么它的客户就能够使用”==”操作来代替equals(OBject)方法，这样做可以使性能得到实质性的提高。String.intern方法用一种有限的形式实现了这种优化。

**与构造函数相比，静态工厂方法的第三个优势是，它们可以返回到返回类型的任何子类型**（subtype）**的对象**。这使用户在选择返回对象的类时具有很大的灵活性。

这一灵活性的体现是API可以返回一种对象同时又不需要使该对象的类成为公共的。这种方式下类实现的隐藏可以带来紧凑的API。这种特性使静态工厂方法是基于接口结构的，接口为它提供了中性的返回类型。客户用接口访问被返回的对象，而不是通过对象的类，这种做法是一种好的行为。

公共的静态工厂方法返回对象所对应的类不仅可以是非公共的，而且依据静态工厂的参数的值，类在每次调用时可以变化，使得返回声明类型的任何子类型的类都是允许的。而且为了增强软件的可维护性，返回的对象对应的类可以随版本而变化。

静态工厂方法返回对象所属的类，甚至不必在包含该静态工厂的类的设计时刻就存在。这种灵活性，使静态工厂方法实现了如服务提供者框架Java Cryptography Extension（JCE）的基础。服务提供者框架是一种系统，在这种系统中，提供者提供了可用API的多种实现给框架用户。这种框架需要一种机制以注册这些实现，使它们对用户可用。框架的用户在使用API时不用考虑它们在使用哪一种实现。

在JCE中，系统管理员通过编辑一个叫做Properties文件来注册要实现的类。使用静态工厂方法的客户以关键字做为参数，静态工厂方法在映射列表中查找Class对象，该映射列表从Properties文件（在该文件中增加条目，将字符串关键字映射成相应的类名字，Properity文件注册实现类）初始化而来。找到需要的类后，使用Class.newInstance方法初始化该类。下面的程序框架说明了这种技术（

import java.util.\*;

// Provider framework sketch

public abstract class Foo {

// Maps String key to corresponding Class object

private static Map implementations = null;

// Initializes implementations map the first time it's called

private static synchronized void initMapIfNecessary() {

if (implementations == null) {

implementations = new HashMap();

// Load implementation class names and keys from Properties file,

// translate names into Class objects using Class.forName and store

// mappings....

}

}

public static Foo getInstance(String key) { //静态工厂方法

initMapIfNecessary();

Class c = (Class) implementations.get(key);

if (c == null)

return new DefaultFoo();

try {

return (Foo) c.newInstance();

} catch (Exception e) {

return new DefaultFoo();

}

}

public static void main(String[] args) {

System.out.println(getInstance("NonexistentFoo"));

}

}

class DefaultFoo extends Foo {

}

**静态工厂方法的主要缺陷是类在没有公共或受保护的构造函数时不能被子类化**。对于由公共静态工厂方法返回的非公共类也是如此。例如，不能子类化Collections Framework中的任何集合类。然而这也许会有一些好处：因为Java语言鼓励使用复合而不是继承。

**静态工厂方法的另一个缺陷是名字之间不容易区分**。在API文档里，它们不会象构造函数那样是唯一的，而且静态工厂方法代表与规范的一种背离。因而，要对提供了静态工厂方法而不是构造函数的类进行初始化，从类文档中找到解决办法是困难的。这种缺陷可以通过遵循标准的命名约定来消除。例如下面两个名字已经很流行了：

* valueOf：不严格的说，该方法返回一个与参数同值的实例，它实际是一种类型转换操作符。
* getInstance：返回一个由参数描述的实例，但实例不一定由相同的值。对于singleton类，它返回唯一的实例。这个名字在提供者框架下是普遍的。

总的来说，在权衡过了静态工厂方法和构造函数后，如果没有其它因素影响你的选择，那么最好还是简单地使用构造函数，因为这符合规范。

### 使用私有构造函数强化singleton属性

#### singleton模式的实现

singleton类是一种只能被实例化一次的简单类。这种类典型地被用来表示那些本性上具有唯一特性的系统组件，如视频显示或文件。实现singleton模式有两种途径，这两种途径都以保持构造函数私有及提供公共的静态成员允许客户访问它的唯一实例为基础。

* 一种实现方法是：公共的静态成员是一个final域。

**// Singleton with final field**

public class Elvis {

**public static final Elvis INSTANCE = new Elvis();**

private Elvis() {

// ...

}

// ... // Remainder omitted

public static void main(String[] args) {

System.out.println(Elvis.INSTANCE);

}

}

私有构造函数仅被调用一次，以初始化公共的静态final域Elvis.Instance。一旦Elvis类被初始化，仅有一个Elvis实例存在。客户的任何行为都不能改变这一点。

* 第二种实现方法是：提供了公共的静态工厂方法，取代了公共的静态成final域。

**// Singleton with static factory**

public class Elvis {

**private static final Elvis INSTANCE = new Elvis();**

private Elvis() {

// ...

}

**public static Elvis getInstance() {**

return INSTANCE;

}

// ... // Remainder omitted

public static void main(String[] args) {

System.out.println(Elvis.INSTANCE);

}

}

对所有的静态方法Elvis.getInstance的调用，都返回同一个对象引用，没有其它的Elvis实例会被创建。

第一种方法的突出优点是类成员的声明使类清楚地表明它是singleton：公共的静态域是final，所以这个域将总是包含着相同的对象引用。这种方式也许还会带来一些性能上的优势，但一个好的JVM实现应该通过在第二个方法中将调用内嵌到静态工厂的方法来消除这一点。

第二种方法的突出优点是在于它给使用者提供类灵活性，当你要把这个类改为非singleton的时候，无需修改API。singleton类的静态工厂方法返回了类的唯一实例，但这个方法可以很容易被修改，比如，使它变成每一个调用该方法的线程返回一个唯一的实例。

总而言之，如果你确信该类将永远是singleton的，那么应该使用第一种方式。否则，使用第二种方式更好。

为了使singleton类可串行化，仅仅在它的声明中增加一个implements Serializable是不够的。为了保证类是singleton，还必须要提供一个readResolve方法（保证它具有singleton的属性）。否则对每一个串行化实例的解串行化操作将导致一个新实例的创建。

**// readResolve method to preserve singleton property**

private Object readResolve() throws ObjectStreamException {

/\*

\* Return the one true Elvis and let the garbage collector

\* take care of the Elvis impersonator.

\*/

return INSTANCE;

}

#### singleton模式实现的种种问题

Singleton模式是一个非常有用的模式，Singleton 模式通过控制对象地创建，限制对象的数量只有一个。Singleton模式是一个很常见的模式，比如：COM中的AddRef()、Release()机制；COM、CORBA的单线程、多线程服务机制，其中的单线程机制的实现等。

但是在某些情况下会产生多个对象被创建的情况，这是我们极力向避免的。下面是一些JAVA应用中的产生多个Singleton的尴尬情形。

JAVA中的Singleton的实现有两种情况：eager Initialization和lazy Initalization

Listing 1(Eager Initialization)

public class MySingleton {

private static MySingleton \_instance = new MySingleton();

private MySingleton() {

// construct object . . .

}

public static MySingleton getInstance() {

return \_instance;

}

}

Listing 2(Lazy Initialization)

public class MySingleton {

private static MySingleton \_instance;

private MySingleton() {

// construct object . . .

}

// For lazy initialization

public static synchronized MySingleton getInstance() {

if (\_instance==null) {

\_instance = new MySingleton();

}

return \_instance;

}

}

两种实现方式其实都不支持子类化操作，因为其和Singleton操作有关的函数和变量都是static，不能实现多型。

1. 如果一个class同时在多个JVM上运行，那么各个JVM上都会分别有一个实例，这导致Singleton失败，实际中这种情况是有的，比如在那些用到了EJB、JINI和RMI等技术的分布系统中。

2. 当多个class Loader装载同一个实现了Singleton的类时，实际上拥有了这个类的多个拷贝，每个拷贝都可能创建自己的Singleton实例。这在某些Servlet引擎中存在这个问题。另外，当浏览器从网络上装载Applet时，可能对不同的服务使用不同的class Loader。

3. 当没有对象持有Singleton实例的引用时，Singleton实例会被GC释放，然后在后面需要的时候又重新装载，显然，重新装载后的实例已经不是原来的实例，实例的所有属性都被重新初始化了。

4. 除了上面的被GC释放后重新装载外，还有被程序本身主动请求释放的，比如Servlet允许Servlet引擎随时释放servlet的类（通过调用destroy()方法），在后面需要的时候再重新装载。

5. 程序在处理多线程同步不妥当时导致多个Singleton实例被创建（在lazy initialization中）当多个线程同时调用getInstance()方法时，由于同步问题，将可能导致多个Singleton实例被创建。

Listing 3 // error, no synchronization on method

public static MySingleton getInstance() {

if (\_instance==null) {

\_instance = new MySingleton();

}

return \_instance;

}

Listing 4// Also an error, synchronization does not prevent two calls of constructor.

public static MySingleton getInstance() {

if (\_instance==null) {

synchronized (MySingleton.class) {

\_instance = new MySingleton();

}

}

return \_instance;

}

正确的解决方法是让getInstance() 成为一个synchronized方法:

Listing 5 // correct solution

public static synchronized MySingleton getInstance() {

// . . . continue as in Listing 3

6. DCL(Double-checked locking) 在C++中的实现是可以解决这个问题，但是在JAVA中这个方法失效了，这和JAVA的内存模型有关，具体内容已经超出了Pattern的范畴，是JAVA规范的范畴。

Listing 6 // Double-checked locking -- don't use

public static MySingleton getInstance() {

if (\_instance==null) {

synchronized (MySingleton.class) {

if (\_instance==null) {

\_instance = new MySingleton();

}

}

}

}

7. 当子类化Singleton时，可能导致多个Singleton实例产生。Singleton模式通过控制对象创建过程来实现，但是象子类化这种手段却不在其控制之列。除非通过将构造函数声明为private。如果想允许子类化，又不能这样作。

8. 上面实现Singleton的操作和变量都是static的，这样Singleton类的弹性太差，不利于后面做修改。另外一种实现方式是通过工厂来控制，这样Singleton类就不用用静态方法和静态变量来控制了。在Servlet中有两种方式：Single ThreadModel和Multiple ThreadModel，在单线程模式下，Servlet创建一个对象处理所有的事务。如果你习惯于通常的Singleton模式，你很可能忽略了Servlet还有多线程模式的情况，产生在多线程模式下不正常的问题。（这问题应该不是产生了多个实例的问题，只是说产生的线程问题。）

9. 采用反序列操作可能导致多个Singleton实例的出现。多次反序列化一个被序列化的Singleton实例，将产生多个Singleton实例

10. 如果用工厂的方法实现Singleton，那么这个工厂类可能没有这种Singleton的保护机制，这时候可能产生多实例的问题。

### 用私有构造函数强化不可实例化的能力

在某些场合需要仅由静态方法和静态属性组成的类。这种类有它的用处，用它们可以把有关原始类型值和数组的方法组织到一起，如java.lang.Math及java.util.Arrays；也可以用它们把那些实现了特定接口的对象中的静态方法组织在一起，如java.util.Collections；还可以把final类里的方法组织到一起，替代扩展类。

这样的工具类（utility class）不希望被实例化：对它们进行实例化没有意义。然而在类中缺少显示地构造函数的时候，Java会自动地提供一个公共的无参数的缺省的构造函数（default congstuctor）。对用户而言，这个构造函数与其它的没什么区别。在公共API中，出现无意地对类进行了实例化地情形并不少见。

试图通过将类抽象化来强化类的不可实例化能力是行不通的。这是因为类可以被子类化，而子类可以被实例化。并且这种做法还会误导用户，以为这种类的目的是用来实现继承的。幸好有一种简单的用法可以解决这一问题。用于缺省的构造函数仅在类不包含显示的构造函数时才会生成，我们可以在类中包含显示的私有类型构造函数来实现类的不可实例化特性。

因为声明的构造函数是私有的，所以它在类的外部不可访问。假设构造函数不会被类自身从内部调用，即保证类永远不会被实例化。这种用法稍微违反了人的直观印象：构造函数被显式地提供，但又不可以被调用。所以，明知的做法是在声明中包含进对构造函数用途的描述性说明。

作为一种副作用，这种用法还使类不能被子类化。所有的构造函数必须要显式地或隐含地调用可访问的超类的构造函数，而在这种情形下，子类将无法调用可访问的构造函数。

**// Noninstantiable utility class**

public class UtilityClass {

**// Suppress default constructor for noninstantiability**

private UtilityClass() {

// This constructor will never be invoked

}

// ... // Remainder omitted

}

用私有构造函数强化不可实例化的能力，也可以这样理解：如果类所有的方法和属性都是静态(static)的，那么要定义缺省的(default)构造器，并且应该是私有(private)的，从而提高类的封装性。

**【错误示例】**

public class UCC {

public UCC() {} //违例

public static String getS() {

return s;

}

public static String s = "foo";

}

**【改正示例】**

public class UCC {

private UCC() {} //正确

public static String getS() {

return s;

}

public static String s = "foo";

}

### 避免创建重复对象

重用一个对象比每次都创建功能等同的新对象通常更合适。重用方式即快速也更加时尚。如果对象是非可变的，那么它永远可以被重用。在可以重用现有对象的情况下，尽量不要创建新的对象，例如：

String s = new String(“silly”); //每次执行都会在内存中创建新的String实例

String s = ”No longer silly”; //执行时，只在缓冲池中创建一个单一的String实例

如果非可变类同时提供了静态工厂方法和构造函数，我们选用前者（静态工厂）通常可以避免重复地创建相同的对象。构造函数在每次被调用时创建了一个新对象，而静态工厂方法从不这样做。

除了对非可变对象可以重用，我们也可以重用可变（mutable）对象，只有知道它们不会发生变化。有一个更复杂、更常见的例子，其中出现了可变对象，但它们的值在计算后就不再改变。不要使用下面的做法：

mport java.util.\*;

public class Person {

private final Date birthDate;

public Person(Date birthDate) {

this.birthDate = birthDate;

}

**// DON'T DO THIS!**

public boolean isBabyBoomer() {

Calendar gmtCal =

Calendar.getInstance(TimeZone.getTimeZone("GMT"));

gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);

Date boomStart = gmtCal.getTime();

gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);

Date boomEnd = gmtCal.getTime();

return birthDate.compareTo(boomStart) >= 0 &&

birthDate.compareTo(boomEnd) < 0;

}

}

在每次调用时，方法isBabyBoomer都要创建一个新的Calendar、TimeZone和两个Date实例。下面的版本是通过使用静态的初始化方法避免了这样的低效做法。

import java.util.\*;

class Person {

private final Date birthDate;

public Person(Date birthDate) {

this.birthDate = birthDate;

}

/\*\*

\* The starting and ending dates of the baby boom.

\*/

private static final Date BOOM\_START;

private static final Date BOOM\_END;

static {

Calendar gmtCal =

Calendar.getInstance(TimeZone.getTimeZone("GMT"));

gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);

BOOM\_START = gmtCal.getTime();

gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);

BOOM\_END = gmtCal.getTime();

}

public boolean isBabyBoomer() {

return birthDate.compareTo(BOOM\_START) >= 0 &&

birthDate.compareTo(BOOM\_END) < 0;

}

}

改进的版本仅在初始化阶段创建了一次Calendar、TimeZone和Date实例，而不是在每次isBabyBoomer调用时都去创建它们。如果此方法被频繁地调用，改进地做法会获得相当多地性能增益（大约1000倍）。改进后，不仅性能获得提高，而且定义也更加清楚了。

不要错误地认为以上内容暗示对象的创建消耗很高，要予以避免。相反，小对象的构造函数所做的微不足道的工作，尤其在现代的JVM实现上，对它们的创建和回收是容易实现的。如果创建额外的对象能使程序明晰、简洁，那么这样做对程序通常是有益的。

反之，通过维护自己的对象池（object pool）来避免对象的创建并不是好的想法，除非对象是极端重量级的。典型的合理使用对象池的例子是数据库连接。由于建立连接的代价太高，使得重用这样的对象是有意义的。然而一般而言，维护自己的对象池会扰乱代码，增加内存容量，损害性能。当代JVM实现大大地优化了垃圾回收单元，使轻量级对象在性能上可以轻而易举地超过对象池。

### 消除对过期对象的引用

Java语言虽然具有垃圾回收的功能，但同样需要考虑内存的管理问题。请看下面的栈实现的简单例子（存在一个内存泄漏）：

import java.util.\*;

// Can you spot the "memory leak"?

public class Stack {

private Object[] elements;

private int size = 0;

public Stack(int initialCapacity) {

this.elements = new Object[initialCapacity];

}

public void push(Object e) {

ensureCapacity();

elements[size++] = e;

}

public Object pop() {

if (size==0)

throw new EmptyStackException();

return elements[--size];

}

/\*\*

\* Ensure space for at least one more element, roughly

\* doubling the capacity each time the array needs to grow.

\*/

private void ensureCapacity() {

if (elements.length == size) {

Object[] oldElements = elements;

elements = new Object[2 \* elements.length + 1];

System.arraycopy(oldElements, 0, elements, 0, size);

}

}

public static void main(String[] args) {

Stack s = new Stack(0);

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

s.push(args[i]);

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

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

}

}

程序中的内存泄漏在哪里呢？如果栈先增长在收缩，那么被栈弹出的对象即使在程序不再引用它们时，也不会被垃圾回收单元回收。这是由于栈维护着这些对象的过期引用。所谓过期引用（obsolete reference）是指永远不会被解除引用的引用。在本例中，任何在元素队列的活动部分之外的引用都是过期的，活动部分由索引值小于size的元素构成。

在具有垃圾回收功能的语言中（称为无意识对象保持unintentional object retentions更合适）内存泄漏是难以察觉的。一旦一个对象的引用被不小心地保留，不仅这个对象被排除在垃圾回收之外，而且所有被这个对象引用地对象也是这样，并顺延下去。因此，即使仅有几个对象的引用被无意识地保留，也会有许多的对象被排除在垃圾回收之外，对性能造成潜在而可怕的影响。

解决这种问题的方法很简单：一旦它们过期，清除掉对它们的引用。对于例子中Stack类，只要条目被弹出栈，对它的引用即过期。pop方法的正确版本为：

public Object pop() {

if (size==0)

throw new EmptyStackException();

Object result = elements[--size];

elements[size] = null; // Eliminate obsolete reference

return result;

}

清除掉过期引用的额外好处是：如果它们随后被错误地解除引用，程序将会由于报出异常NullPointerException而退出，不会继续错误地运行下去。尽可能快地发现错误永远是有益的。

对此的过激反映是：只要程序与对象打完交道，就把所以对象的引用都清除掉。这既不必要也不是我们所期望的，因为它会不必要地扰乱程序，不可避免地降低性能。清除掉对对象的引用是异常而不是规范的行为。清除过期引用的最好办法是在对象的作用域内对它重用或者将它销毁。应该意识到，对目前的JVM而言，仅仅退出定义变量的程序块是不够的，要使引用消失，必须彻底退出包含该变量的方法。

那么应该什么时候清除掉引用呢？Stack类的哪些特点使它容易受内存泄漏的影响呢？简单地说，就是Stack类自己管理自己的内存。存储池（storage pool）由items数组（对象引用单元，不是对象自身）的元素组成。数组活动部分的元素被分配了，数组其余部分还没有。垃圾回收单元没有办法知道这一点：对它而言，items数组的所有对象引用都是等同的。仅仅使程序员知道数组的非活动部分是不重要的。只要它们成为非活动部分，程序员就要通过手工清空数组元素的方式，及时地把实际情况通知给垃圾回收单元。

一般而言，只要类是自己管理内存，程序员就要注意内存泄漏问题。只要元素被释放，包含在其中的对象引用都应该被清除掉。

另一个内存泄漏的常见来源是缓存。一旦对象的引用被放入缓存，它就很容易被遗忘，从而使得在它不被使用时仍保留在原来的位置。对这个问题有两种解决办法。

1．如果你碰巧要实现这样的缓存（只要有来自缓存外面的对某条目关键字的引用，该条目就会关联），那么就可以使用WeakHashMap来描述缓存。这样条目只要过期，就会被自动地删除。

2．而更一般的情况，缓存条目关联的周期难以确定，并且随着时间的增加，条目存在的价值会降低。在这种情况下，缓存应该不断地清除无用的条目。这样的清除工作可以由后台线程完成（可能要通过java.util.Timer API），也可以利用向缓存增加新条目的副作用来实现。Java.util.LinkedHashMap类的removeEldestEntry方法可以很容易的用来实现后面的方法。

内存泄漏难以发现，只有通过对代码的仔细检查或借助于调试工具如heap profiler的帮助，才能发现它们。

### 避免使用终结程序

终结程序（finalizer）的行为是不可预测的，而且是危险的，通常也不必要。使用它们会造成不稳定，破坏性能，带来移植问题等。终结程序有时也有可用之处，但作为一个原则，对终结程序的使用应该被避免。

例如：一个未被捕获的异常在终结过程中引发，那么异常将被忽略，而对象的终结过程被终止。未被捕获的异常将使对象处于崩溃状态。如果试图使用这个处于崩溃状态的对象，将会出现无法确知的行为。正常情况下，未被捕获的异常应该结束线程，并打印堆栈踪迹。但如果是在终结程序中出现，情形不会这样：它甚至连警告都不会给出。

在Java语言中，在对象不可用时，垃圾回收单元会回收与其相关的资源，不需要程序员在这方面做专门的工作。如果需要回收，一般用try-finally结构做类似的工作。

终结程序无法保证被及时地执行。从对象不可用地时刻到终结程序地执行时刻之间的时间间隔是不确定的。这意味着不能用终结程序来处理时间关键（time-critical）性的操作。例如，依赖终结程序去关闭打开的文件是一个严重的错误。

及时地安排终结程序的执行是垃圾回收算法的基本功能之一，在JVM的不同实现上，这种算法差别很大。依赖于终结程序的及时执行的程序，其表现也截然不同。一个程序在你的JVM上测试执行得非常好，但换到其它得JVM上去执行却变得非常糟糕，这种情况完全可能出现。

*Java Language Specification* (JLS)不仅不保证终结程序的执行，它甚至不会保证终结程序的执行。完全有可能在程序结束的时候，某些不可用对象的终结程序还没有执行。作为一个结论，永远不要依赖终结程序去更新关键的持续状态（persistent state）。例如：不要依赖终结程序释放一个共享资源如数据库上的持续锁。

不要被System.gc和System.runFinalization的方法所欺骗，它们可能增加了终结程序获得执行的机会，但不会保证它一定对运行。唯一对终结程序作出承诺的是两种天生具有缺陷且已经摒弃不用的方法System.runFinalizersOnExit和Runtime.runFinalizersOnExit。

如果类封装的资源（如文件或线程）需要回收，用什么来替代类的终结程序呢？为类提供一个显示声明的终止（termination）方法。然后客户在类的每个实例不再需要这些资源时调用这个方法，就可满足上面的要求。需要指出的是实例必须要追踪它是否已经被终止：那个显示的终止方法需要在对象的私有域中记录该对象不再有效，其它的方法必须检查这个域。如果在对象已经被终止之后，在调用它们，则要引发IllegalStateException异常。

使用这种显示的终止方法的典型例子是InputStream和OutputStream的close方法。另一个例子是java.util.Timer的cancel方法，java.util.Timer需要做必要的状态改变操作，以使一个与Timer实例关联的线程终止自己。其它的例子还有java.awt中的方法Graphics.dispose和Window.dispose。这些方法通常由于性能不好而被忽略。还有Image.flush，它解除所有与Image实例相关的资源，但把它保留在可用状态，以便必要时在为它重新分配资源。

显示的终止方法通常与try-finally结构结合使用以保证及时的终止。在finally里调用显式的终止方法将保证它在对象的使用过程中即使引发异常也能被执行。

**//try finally block guarantees execution of termination method**

Foo foo = new Foo(…)

try {

//Do what must be done with foo

} finally{

foo.terminate(); //Explicit termination method

}

那么终结程序的用处有那些呢？有两种适合它使用的场合：

1．当对象的属主忘记了调用前面段落中建议过的显式的终止方法的情况下，它可以作为安全过滤网（safety net）。尽管终结程序不保证能被及时地调用，但推迟释放临界资源总比不释放资源要好，尤其是（希望这种情形不会发生）当客户没有通过调用显式地终止方法提出终止交易地情况下。前面例子里，提供了显式地终止方法的3个类（InputStream、OutputStream和Timer），在它们的终止方法没有被调用的情况下，就是让终结程序用作安全过滤网。

2．适合终止程序使用的场合与对象的固有对等体（native peer）有关。一个固有对等体是一个固有对象（native object），这个对象由一个正常的对象通过固有方法（native method）代表。因为固有对等体不是正常的对象，垃圾回收单元不认识它，因此当对象本生被回收时，它无法被回收。假设固有对等体没有保持临界资源，那么终结程序是完成这项工作的理想工具。如果固有对等体持有需要及时终止的资源，应该象前面所述，为类提供一个显式的终止方法。终止方法将执行释放临界资源的所有工作。终止方法可以是固有方法或者是其它可调用的形式。

很重要的一点要被注意：终结程序链（finalizer chaining）不能被自动执行。如果类（不是对象）有一个终结程序，子类重载了它，那么子类的终结程序必须要手工调用超类的终结程序。应该在try块中终结子类，并在对应的finally块中调用超类的终结程序。这样能保证即使子类终结过程中引发异常，超类终结程序也能执行，反之亦然。

**//manual finalizer chaining**

protected void finalize() throw Throwable{

try{

//Finalize subclass state

} finally{

super.finalize();

}

}

如果子类的实现程序重载了超类的终结程序但忘记了手工调用超类的终结程序，那么超类的终结程序将永远不会被调用。为了防范这样大意的子类，经常需要为每一个要终结的对象创建一个额外的对象。因此，不要在需要终止的类中放置终止程序，而要把终结程序放到一个匿名类中。使用这个匿名类的唯一目的是用来终结封装它的类的所有实例。该匿名类的唯一实例称为终结程序守卫者（finalizer guardian），为每一个封装类的实例而创建。封装类的实例在它的私有实例域中存储着对守卫者的唯一引用，以便守卫者在封装类的实例之前可以被终结。当守卫者终结的时候，它执行封装类实例所期望的终结活动，仿佛匿名类的终结程序是封装类中的方法一样。

//Finalizer Guardian idiom

public class Foo{

//Sole purpose of this object is to finalize outer Foo object

private final Object finalizerGuardian = new Object(){

protected void finalize() throw Throwable{

//Finalize outer Foo object

}

};

}

请注意，公共类Foo没有终结程序（除了它从Object中继承的无关紧要的一个），所以无论子类终结程序是否调用super.finalize，都不会造成实际影响。每一种带有终结程序的非final的公共类都应该考虑使用这种技术。

总而言之，除了作为安全过滤网或为了终止非临界（noncritical）的固有资源以外，不要使用终结程序。一旦使用了终结程序，不要忘记嗲用super.finalize方法。最后，如果需要把终结程序和一个公共的非final的类关联起来，可以考虑使用终结程序守卫者，以保证终结程序能被执行，这样做，即使子类终结程序没有调用super.finalize也能保证终结程序被执行。

## 对象的通用方法

尽管Object是一种具体的类，它的主要目的是用来实现扩展。它的所有非final方法（clone、equals、hasCode、toString、finalize等等）都有明确的通用约定（general contract），因为它们要被重载。任何类重载这些方法时都要遵循这种约定，否则将会妨碍它们与那些依赖这些约定的其它类的正常工作。

### 重写equals时要遵守通用约定

避免错误重载equals的最简单的方法是不要重载它，在这种情况下，每一个实例只能等于它自身。如果下面的条件满足，就不要重载equals：

* **每个类实例本质上是唯一的**。对于表示活动实体而非数值类，情形是这样的。例如：Thread。Object提供的equals实现对这些类的正确的处理行为。
* **不关系类是否提供了逻辑意义的等同（logical equality）测试**。例如java.util.Random本来可以重载equals方法，用以检查两个Random实例是否会产生相同的随机数序列，但设计者不认为客户会需要这种功能。这种情况下，使用从Object继承的equals实现就足够了。
* **超类已经重载类equals，而从超类继承的行为适合该类**。例如，大多数的Set实现从AbstractSet继承了equals实现。List继承了AbstractList的equals实现，Map继承了AbstractMap的equals实现。
* **类是私有的或包内私有（package private）的，而且可以确定它的equals方法永远不会被调用。**

那么什么时候重载Object.equals方法合适呢？当类有逻辑上的等同意义而不仅仅是对象意义上的等同，而且超类没有重载equals方法以实现期望的行为，这时需要重载。这种情形通常是数据类型的类（value class），例如Integer或Date。程序员使用equals方法比较数值类型对象时，希望知道它们是否是逻辑上的相等，而不是它们是否引用相同的对象。

不需要重载equals方法的一种数值类型的类是类型安全枚举类。因为类型安全枚举类保证对每一个值最多只能有一个对象存在。对这种类而言，Object.equals方法等同于逻辑意义上的equals方法。

重载equals方法，必须坚持equals方法的一般约定。下面5个约定拷贝自java.lang.Object的规范。

■ **自反性（reflexive）**：对于任意的引用值x，x.equals(x)总是返回true。第一个条件要求对象要等于自身。如果违背了这个条件，并在集合中增加该类的一个实例，那么调用集合的方法contains时，它会告诉你，集合中不包含刚才加入的实例。

■ **对称性（Symmetry）**：对于任意的引用值x、y，如果y.equals(x)返回true，x.equals(y)总返回true。第二个条件要求两个对象对它们是否相等这个问题要保持一致。

考虑下面的例子（请查阅instanceof的用法）：

/\*\*

\* Case-insensitive string. Case of the original string is

\* preserved by toString, but ignored in comparisons.

\*/

public final class CaseInsensitiveString {

private String s;

public CaseInsensitiveString(String s) {

if (s == null)

throw new NullPointerException();

this.s = s;

}

**// Broken - violates symmetry!**

public boolean equals(Object o) {

if (o instanceof CaseInsensitiveString)

return s.equalsIgnoreCase(

((CaseInsensitiveString)o).s);

if (o instanceof String) **// One-way interoperability!**

return s.equalsIgnoreCase((String)o);

return false;

}

// Note that a hashCode method would be required

static void main(String[] args) {

CaseInsensitiveString cis = new CaseInsensitiveString("Polish");

String s = "polish";

System.out.println(cis.equals(s));

System.out.println(s.equals(cis));

}

}

显然：cis.equals(s)返回true；s.equals(cis)返回false。明显地违背了对称原则。为了解决这个问题重新定义equals()函数，使它只有一种返回值：

public boolean equals(Object o) {

return o instanceof CaseInsensitiveString &&

((CaseInsensitiveString)o).s.equalsIgnoreCase(s);

}

（感觉这个例子不恰当，应该是这个类本身的方法equals具有对称性）

■ **传递性（Transitivity）**：对于任意的引用值x、y、z，如果x.equals(y)返回true，y.equals(z)返回true，那么x.equals(z)一定返回true。第三个条件要求如果第一个对象等于第二个对象，第二个对象等于第三个对象，那么，第一个对象必须等于第三个对象。

考虑下面的例子（一个简单的非可变的二维点）：

public class Point {

private final int x;

private final int y;

public Point(int x, int y) {

this.x = x;

this.y = y;

}

public boolean equals(Object o) {

if (!(o instanceof Point))

return false;

Point p = (Point)o;

return p.x == x && p.y == y;

}

}

如果要扩展这个类，为类增加点颜色属性：

public class ColorPoint extends Point {

private Color color;

public ColorPoint(int x, int y, Color color) {

super(x, y);

this.color = color;

}

/\* COMMENTED OUT

**// Broken - violates symmetry!**

public boolean equals(Object o) {

if (!(o instanceof ColorPoint))

return false;

ColorPoint cp = (ColorPoint)o;

return super.equals(o) && cp.color == color;

}

\*/

**// Broken - violates transitivity.**

public boolean equals(Object o) {

if (!(o instanceof Point))

return false;

// If o is a normal Point, do a color-blind comparison

if (!(o instanceof ColorPoint))

return o.equals(this);

// o is a ColorPoint; do a full comparison

ColorPoint cp = (ColorPoint)o;

return super.equals(o) && cp.color == color;

}

public static void main(String[] args) {

**// Symmetry test**

Point p = new Point(1, 2);

ColorPoint cp = new ColorPoint(1, 2, Color.RED);

System.out.println(p.equals(cp)); //返回true

System.out.println(cp.equals(p)); //返回false

System.out.println();

**// Transitivity test**

ColorPoint p1 = new ColorPoint(1, 2, Color.RED);

Point p2 = new Point(1, 2);

ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);

System.out.println(p1.equals(p2)); //返回true

System.out.println(p2.equals(p3)); //返回true

System.out.println(p1.equals(p3)); //返回false

}

}

那么有什么解决方案呢？事实上，这是面向对象语言在判等关系上的一个根本难题。没有一个简单的办法可以保证在扩展一个可实例化的类并增加新特性的同时仍保持equals约定。面向对象编程的建议：不采用继承，而采用复合，可以很好的解决这个问题。不采用ColorPoint扩展Point，而是给ColorPoint增加一个私有的Point域和一个公共的view方法。实现如下：

**// Adds an aspect without violating the equals contract**

public class ColorPoint2 {

private Point point;

private Color color;

public ColorPoint2(int x, int y, Color color) {

point = new Point(x, y);

this.color = color;

}

/\*\*

\* Returns the point-view of this color point.

\*/

public Point asPoint() {

return point;

}

public boolean equals(Object o) {

if (!(o instanceof ColorPoint2))

return false;

ColorPoint2 cp = (ColorPoint2)o;

return cp.point.equals(point) && cp.color.equals(color);

}

// Note that a hashCode method would be required

public static void main(String[] args) {

**// Symmetry test**

Point p = new Point(1, 2);

ColorPoint2 cp = new ColorPoint2(1, 2, Color.RED);

System.out.println(p.equals(cp)); //返回false

System.out.println(cp.equals(p)); //返回false

System.out.println();

**// Transitivity test**

ColorPoint2 p1 = new ColorPoint2(1, 2, Color.RED);

Point p2 = new Point(1, 2);

ColorPoint2 p3 = new ColorPoint2(1, 2, Color.BLUE);

System.out.println(p1.equals(p2)); //返回false

System.out.println(p2.equals(p3)); //返回false

System.out.println(p1.equals(p3)); //返回false

}

}

在Java类库中有一些类，对可实例化（instantiable）类进行子类化并加入了新特性。例如，java.sql.Timestamp是java.util.Dtae的子类，增加了nanoseconds域。Timestamp的equals实现确实违反了对称性，如果Timestamp和Date对象被用在同一个集合中或者被混合使用，将会出现不可预知的行为。因此Timestamp类有一个声明，告诫程序员不要混合使用date和timestamp对象。

注意：我们可以为一个抽象（abstract）类的子类增加新特性，而不违背equals约定。即用类层次代替联合，例如：定义一个无特性的抽象Shape子类，然后在其中增加radius域生成Circle子类，增加length和with生成Rectangle子类。只要超类的实例无法创建，就不会发生刚刚提到的问题。

■ **一致性（Consistency）**：对于任意的引用值x、y，如果对象中用于equals比较的信息没有修改，那么对x.equals(y)的多个调用，要么一直为true，要么一直为false。第四个条件是如果两个对象相等，它们就应该永远相等，直到它们之中至少有一个发生变化。如果注意到这样的事实：可变对象可以在不同的时间等于不同的对象，而非可变对象不会这样。那么就可知道这个条件的表述不很合适。在设计类时，要认真考虑此类是否应该成为非可变的。如果可以确定这一点，就要确保自己的equals方法遵守这个约定：相等的对象永远相等，不等的对象永远不等。

■ **非空特性（Non-nullity）**：对于任何非空引用值x，x.equals(null)总为false。这个条件要求所有的对象都不等于null。尽管很难想象o.equals(null)调用会返回true，但让它引发异常NullPointerException同样很难。通用约定不允许这样做。许多具有equals方法的类通过显式地做null测试来防止异常出现：

public boolean equals(Object o){

if(o==null)

return false;

……

}

这种测试时不必要的。因为为了测试参数的等同性，equals方法首先要把参数映射为合适的类型，以使它的访问器能被调用或它的域能被访问。而在进行映射之前，方法需要使用操作符instaceof，检查它的参数是否是正确的类型：

public boolean equals(Object o){

if(!(o instanceof MyType))

return false;

……

}

一旦漏掉了类型检查，而传入equals方法的是错误类型的参数，那么equals方法将引发异常ClassCastException，这破坏了equals约定。但对于instanceof操作符，如果第一个操作数是null，它就会返回false，而不管随后出现的操作数类型。因此如果null被传递进来，类型检查将返回false，所以没有必要做单独的null检查。

综合在一起，为实现高质量的equals方法，请遵循以下原则：

**①** **用==操作符检查参数是否是对该对象的引用。**如果是，返回true。这仅仅是一种性能优化的策略，如果在比较操作上的开销高，就不值得这样做。

**②** **用instanceof操作符检查参数是否是正确的类型。**如果不是，返回false。正确的类型通常是此方法（equals）出现在其中的类，有时也可能是被这个类实现的接口。如果实现了接口的类增强了equals约定，允许在实现了这些接口的类之间进行比较，那么就使用这个接口作为正确的类型。集合接口Set、List、Map和Map.Entry等都有这样的特性。

**③ 把参数映射到正确的类型。**因为这种映射之前已经进行了instanceof测试，因此这种映射能够保证成功。

**④ 对类中每一个主要的（significant）域（相当于类的属性），检查参数中的域是否与对象中相应的域匹配。**如果所有测试都成功，返回true；否则，返回false。如果步骤②中的类型是接口，那么必须访问参数中的域；如果类型是类，那么也许可以直接方位这些域，这依赖于它们的可访问性。

对于类型不是float和double的原始类型域，使用==操作符；

对于对象引用类型域，可以递归地使用equals方法；

对于float类型的域的值，用Float.floatToIntBits方法将其转化为int值，然后用==操作符比较得到的int值。

对于double类型域的值，用Double.doubleToLongBits将其转化为Long型，再用==操作符比较Long类型的值。

对于float和double类型域，由于Float.NaN（-0.0f）的存在，要做特殊处理，细节请参阅Float.equals文档。

对于数组类型域，对其中每一个元素分别应用上述原则。

有些对象引用类型域包含null是合法的，为了避免NullPointerException的可能出现，使用下面的语法对这种域进行比较：

(field == null ? o.field == null : field.equals(o.field))

由于field和o.field通常是对相同对象的引用，因此下面的做法可能更快：

(field == o.field || (field != null && field.equals(o.field)))

equals方法的性能会受域的比较顺序的影响。为了获得最好的性能，要尽可能先比较那些不同的域，以及那些比较起来开销低的域，当然，最理想的情况是这两种条件都满足的域。切记，不要去比较不属于对象逻辑状态的域，如Object中用来同步操作的域。不要比较冗余的域（redundant fiels），它们可以通过主要域计算得到，但这样也有可能提高equals方法的性能。如果冗余域能够大致地描述对象全貌，那么如果最终的比较以不相等结束，使用这些域做比较要比使用实际数据节省开销。

**⑤ 完成自己的equals方法时，问自己3个问题：它是否是对称的、可传递的、一致的（其它两个属性一般会自行满足）。**如果答案是否定的，找出这些属性未能满足的原因，再修改相应代码。

下面是实现equals方法时应该注意的地方：

■ **在重载equals方法时，总要重载hashCode方法。**

■ **不要使自己聪明过头。**简单地测试域地等同性，不一定能保证会遵循equals约定，而过分追求等同也容易造成错误。把任何地同义形式考虑在比较的范围内一般使糟糕的想法，例如File类不应该与指向同一文件的符号链接进行比较。

■ **不要设计依赖于不可靠资源的equals方法。**否则，要满足一致性要求是及其困难的。例如：java.net.URL的equals方法依赖于被比较的该URL的主机IP地址，由于把主机名转化为IP地址需要进行网络访问，因而不能保证它在不同时刻产生相同的结果。除了少数几个例外，equals方法应该基于内存驻留对象（memory resident）进行确定的比较。

■ **不要将equals声明中的Object替换为其它类型。**程序员编写如下所示的equals方法并不少见，它会让人迷惑：所设计的方法为什么不能工作：

public boolean equals(MyClass o){

……

}

问题处在这个方法没有重写（override）参数为Object类型的Object.equals方法，而是重载（overload）了它。

### 重写equals时永远要重写hashCode

bug的一个通常来源是未重写hashCode方法。**一定要在每一个重写（覆盖）了equals的类中重写hashCode方法。**不这样做会违背Object.hashCode的一般约定，并导致你的类在与所有基于散列的集合（HashMap、HashSet和HashTable）一起作用时不能正常工作。

下面给出的约定拷贝自java.lang.Object的规范：

* 如果在应用程序执行期间，用于对象的equals方法中的信息如果没有被修改，那么对同一对象的多次调用，hashCode方法总是返回相同的整数。这个整数不要求在同一应用程序的不同执行期间保持一致。
* 如果两个对象按照equals(Object)方法是等同的，那么对这两个对象的任意一个调用hashCode方法，必须要产生同样的整数结果。
* 如果两个对象根据equals(Object)方法不相同，那么对这两个对象中的任意一个对象调用hashCode方法，不要求产生两个不同的整数结果。但程序员应该知道，为不相等的对象产生不同的整数结果可以提高hash表的性能。

**不重写hashCode方法违背的关键规定是第二条：相等的对象必须有相同的散列码。**如果不重写hashCode方法，两个截然不同实例根据类的equals方法也许逻辑上是等同的，但对于Object类中的hashCode方法，它们就是两个对象，仅此而已。因此对象的hashCode方法返回两个看上去是随机的数值，而不是约定中要求的相等的值。

看下面的例子：

import java.util.\*;

public final class PhoneNumber {

private final short areaCode;

private final short exchange;

private final short extension;

public PhoneNumber(int areaCode, int exchange, int extension) {

rangeCheck(areaCode, 999, "area code");

rangeCheck(exchange, 999, "exchange");

rangeCheck(extension, 9999, "extension");

this.areaCode = (short) areaCode;

this.exchange = (short) exchange;

this.extension = (short) extension;

}

private static void rangeCheck(int arg, int max, String name) {

if (arg < 0 || arg > max)

throw new IllegalArgumentException(name +": " + arg);

}

public boolean equals(Object o) {

if (o == this)

return true;

if (!(o instanceof PhoneNumber))

return false;

PhoneNumber pn = (PhoneNumber)o;

return pn.extension == extension && pn.exchange == exchange &&

pn.areaCode == areaCode;

}

// No hashCode method!

/\*

public int hashCode() {

int result = 17;

result = 37\*result + areaCode;

result = 37\*result + exchange;

result = 37\*result + extension;

return result;

}

\*/

/\* COMMENTED OUT

**// Lazily initialized, cached hashCode**

private volatile int hashCode = 0;

public int hashCode() {

if (hashCode == 0) {

int result = 17;

result = 37\*result + areaCode;

result = 37\*result + exchange;

result = 37\*result + extension;

hashCode = result;

}

return hashCode;

}

\*/

public static void main(String[] args) {

Map m = new HashMap();

m.put(new PhoneNumber(408, 867, 5309), "Jenny");

System.out.println(m.get(new PhoneNumber(408, 867, 5309))); // 输出null

}

}

注意到该例子中涉及到了两个PhoneNumber实例，一个用来插入到HashMap中，另一个等同的实例用来做检索用。由于PhoneNumber类没有重载hashCode，从而导致了两个相同的实例得到了不同的hash码，破坏了hashCode约定。因此，get方法是在不同于电话号码被put方法放入的hash表中寻找它，因此输出为null。改正问题的方法是为PhoneNumber类提供一个正确的hashCode方法。

那么hashCode该是什么样的？编写一个合乎规定但不好用的hashCode的方法非常容易。例如：

**// The worst possible legal hashCode function – never use!**

Public int hashCode() { return 888;}

例子保证了相等的对象会有相同的hash码，因而它是满足规定的。而另一方面，它为所有对象提供了完全相同的hash码，所以它是拙劣的。

好的hash函数倾向于为不相等的对象生成不相等的hash码。这正是hashCode约定第三条想要说的。理想情况下，hash函数应该把所有不相等实例的合理集合均匀地分布到所有可能的hash值上去。达到理想状态很困难，达到一种相对合适的状态并不困难，下面是一个简单的方法：

① 保存某个非0常数（如8888）到名为result的int类型变量中；

② 对对象中的每一个主要域f（每个域由equals方法负责）， 做下面的工作：

* 1. 为域计算int型的hsah码c：
     1. 如果域是boolean型，计算（f?0:1）；
     2. 如果域是byte型、char型、short型或int型，计算(int)f；
     3. 如果域是long型，计算(int)(f^(f>>>32))；
     4. 如果域是float型，计算Float.floatToBits(f)；
     5. 如果域是double型，计算Double.doubleToLongBits(f)，然后如②.a.3所示，对long型结果进一步处理。
     6. 如果域是对象引用，而且这个类的equals方法又递归调用了equals方法对域进行比较，那么对这个域递归地调用hashCode方法。如果需要一种更复杂的比较方式，那么先为这个域计算出范式表示，然后在该范式表示上调用hashCode方法。如果域为mull，则返回0（其它常数亦可，但0更常见）。
     7. 如果域是数组，则把每一个元素作为分离的域对待。即递归地使用这些规则，为每个注意元素计算hash码。然后用②.b所示方法复合这些值。
  2. 把步骤a中计算出地hash码c按照如下方式与result复合：

rusult = 37\*result+c;

③ 返回result；

④ 完成hashCode方法后，测试是否相同的实例会有相同的hash码。

对hash码的计算排除冗余域是可接受的。即如果某些域的值可以从其它参与计算的域中计算出来，则允许将这些域排除在处理之外。同时哪些不进行等于比较的域要被排除去。否则，可能会导致破坏hashCode约定的第二条规定。

在步骤①中使用了一个非0的初始值，这样hash值会被步骤②.a中计算hash值为0的初始域影响。如果在步骤①中用0作为初始值，所有的hash值都不会被这样的初始域影响，但这样会增加冲突的可能。值8888是任意选的。

步骤②中中的乘法部分使hash值依赖于域的顺序，在类中含有多个相似的域的情况下，这样做会获得更好的散列效果。例如，按照上面介绍的方法设计的String散列函数，如果乘法部分被省略掉，那么所有的片语，只要由相同的字母组成，它们的hash码就会相同。用37作为乘数是因为它是一个奇素数，如果用偶数，那么在乘法溢出时，信息会丢失，因为乘2的效果与移位相同。使用素数做乘数没有一个明确的理由，但这是传统用法。

这样实现的（上面代码的颜色）hash函数对PhoneNumber来说已经相等完美。它的实现简单，速度快，很合适把不同的电话号码分散到不同的hash表中。

如果类是可变的，而计算hash码的开销又高，我们可以考虑把hash码缓存到对象中，而不是每次调用时，都要重新计算。如果确定这种类型的大多数对象将被用作hash关键字，那么应该在实例被创建时计算hash码。否则，可以在hashCode第一次被调用时用惰性初始化方法（lazily initialize）来处理它。例子中的PhoneNumber类不一定从这种处理方式中获得好处，这里只是通过它来说明这种方式是如何实现的，

尽管该处理方法可以产生很好的散列功能，但它不能产生出完美的散列函数。如何编写散列函数是一个活跃的研究课题，应该留给数学家和计算机理论家去完成。

**不要试图从hash码计算中排除主要部分以提高性能。**

### 永远要重写toString

java.lang.Object提供的toString方法返回的表示实例对象的字符串是由类名字、紧随其后的@符号和hash码的无符号的十六进制表示组成。相当于实现了

getClass().getName() + '@' + Integer.toHexString(hashCode());如PhoneNumber@163b91。

toString的一般约定指出，返回的字符串应该是使人容易阅读的、简洁、翔实的形式。toString的约定进一步指出：推荐所有的子类重写（override）这个方法。

尽管不象equals和hashCode约定那样重要，**但为类提供一个好的toString实现可以使类使用起来更加赏心悦目；实际实用中toString应该返回包含在对象中的所有令人感兴趣的信息。**当对象被传递给println方法、串的连接操作符（+）或assert方法时（Java1.4），方法toString会被自动调用。例如，可以生成调试信息：

System.out.println(“My phoneNumber is” + phoneNumber);

在实现toString方法时，我们要做的一个重要决定是在文档中是否指明要返回值的格式。指明格式的好处是它可以用做一种标准的。明确的对象表现形式。例如：

/\*\*

\* Returns the string representation of this phone number. The string consists of

\* fourteen characters whose format is "(XXX) YYY-ZZZZ", where XXX is the area

\* code,YYY is the extension, and ZZZZ is the exchange. (Each of the capital

\* letters represents a single decimal digit.)

\*

\* If any of the three parts of this phone number is too small to fill up its field, the

\* field is padded with leading zeros. For example, if the value of the exchange is

\* 123, the last four characters of the string representation will be "0123".

\*

\* Note that there is a single space separating the closing parenthesis after the

\* area code from the first digit of the exchange.

\*/

public String toString() {

return "(" + toPaddedString(areaCode, 3) + ") " +

toPaddedString(exchange, 3) + "-" +

toPaddedString(extension, 4);

}

/\*\*

\* Translates an int to a string of the specified length, padded with leading zeros.

\* Assumes i >= 0, 1 <= length <= 10, and Integer.toString(i).length() <= length.

\*/

private static String toPaddedString(int i, int length) {

String s = Integer.toString(i);

return ZEROS[length - s.length()] + s;

}

private static String[] ZEROS =

{"", "0", "00", "000", "0000", "00000",

"000000", "0000000", "00000000", "000000000"};

System.out.println("Failed to connect: " +

new PhoneNumber(408, 867, 5309));

实际的输出为：Failed to connect: (408) 867-5309

无论是否指明格式，对所有包含在由toString方法返回的值中的信息，提供一种可用程序访问的方法，总是好的思路。

### 谨慎地重写clone

Cloneable接口的目的是作为对象的mixin（混合类型）接口，表明可以对这样的对象进行克隆，但是它没有实现这个目的，主要缺陷是缺少clone方法，而Object方法的clone方法是受保护的。如果不使用反射（reflection），则不能因为对象实现了Cloneable接口，就可以调用对象的clone方法。即使使用了反射方法，也可能会失败，因为不能保证那个对象有一个可访问的clone方法。但是，尽管有很多缺陷，这种用法还是被广泛地使用着，因此有了解它的必要。下面介绍如何去实现一个表现良好的clone方法，讨论什么时候这样做合适，并简要介绍一种替代clone的方法。

假如Cloneable接口没有包含任何方法，它会起什么作用呢？它决定了Object的受保护的clone实现的行为：如果类实现了Cloneable接口，Object的clone方法就能返回对对象的逐域拷贝，或者引发CloneNotSupportedException异常。这是接口一种很不规范的用法，不要去模仿这种行为。正常情况下，实现一个接口意味着类能为它的客户做某些工作，然而在Cloneable的例子里，它修改了超类里受保护clone方法的行为。

为了实现Cloneable接口，使它在类上产生某些效果，它和所有的超类必须要遵循一个相当复杂的、非强制性的并且基本无正式说明的协议。由此会产生一种奇怪的机制：不通过调用构造函数却创建了一个对象。

Clone方法的一般约定的约束是弱的。下面的约定摘自java.lang.Object规范：

Clone方法创建和返回该对象的拷贝，拷贝的精确涵义依赖于对象所属的类。它的一般意义如下：

对任意对象x，表达式x.clone()!=x;为true，并且表达式x.clone().getClass()==x.getClass();为true，但这些不是绝对必须的。尽管的典型的表达式x.clone().equals(x)为true，这也不是绝对必需的。对象的拷贝操作一般会导致一个新的类实例的创建，但同时可能需要拷贝内部数据结构。在这个过程中，构造函数不被调用。

这个约定带来许多问题，构造函数不被调用的规定要求太强。一个行为良好的clone方法能通过构造函数为克隆出的对象创建内部结构。如果类是final的，clone是指能返回一个由构造函数创建的对象。

另一方面，x.clone().getClass()等同于x.getClass()的规定又太弱。实际上，程序会认为如果扩展了一个类，然后从子类中调用了super.clone方法，那么返回的应该是子类的实例。超类能提供这种功能的唯一途径是返回一个（子类）通过调用super.clone方法获得的对象。如果clone方法返回了一个由构造函数创建的对象，它将得到错误的类。因而，如果要重写非final类中的clone方法，那么应该返回一个通过调用super.colne方法获得的对象。如果所有类的超类都遵循这一原则，那么对super.clone的调用将最终调用到Object的clone方法，从而创建出正确的类的实例。这种机制类似于自动地调用构造函数链，但它不是强制要求的。

实际中，实现Cloneable接口的类被期望提供一个具有正确功能的公共的clone方法。实现这一点需要所有的超类都提供了一个行为良好的clone实现（无论是public还是protected的），否则一般无法做到。

假设你想在类中实现Cloneable接口，而且该类的超类提供了表现良好的clone方法。从super.clone()得到的对象不一定与你最终要返回的对象相似，这依赖于类的自身属性。从超类的观点来看，这个对象是初始对象的功能的完整克隆，在你的类中声明的域（如果有的话）将与被克隆的初始对象中相应的与有相同的值。

如果任何一个域都包含一个原始类型或是对非可变对象的引用，那么返回的对象可能正是你所需要的。在这种情况下，不需要再做进一步的处理。例如PhoneNumber类的例子就是这种情况，此时所有要做的工作是对Object的受保护的clone方法提供可公共访问的能力。

public Object clone(){

try{

return super.clone();

} catch (CloneNotSupportException e){

throw new Error(“Assertion failure”); //Can’t happen

}

}

然而，如果对象中包含有对可变对象的引用域，使用上面这样的clone实现可能会是灾难性的。以Stack类为例（在消除对过期对象的引用中已经使用过）：

import java.util.\*;

public class Stack implements Cloneable {

private Object[] elements;

private int size = 0;

public Stack(int initialCapacity) {

this.elements = new Object[initialCapacity];

}

public void push(Object e) {

ensureCapacity();

elements[size++] = e;

}

public Object pop() {

if (size == 0)

throw new EmptyStackException();

Object result = elements[--size];

elements[size] = null; // Eliminate obsolete reference

return result;

}

// Ensure space for at least one more element.

private void ensureCapacity() {

if (elements.length == size) {

Object oldElements[] = elements;

elements = new Object[2 \* elements.length + 1];

System.arraycopy(oldElements, 0, elements, 0, size);

}

}

/\* COMMENTED OUT

**// Won't work for this class!!**

public Object clone() throws CloneNotSupportedException {

return super.clone();

}

\*/

public Object clone() throws CloneNotSupportedException {

Stack result = (Stack) super.clone();

result.elements = (Object[]) elements.clone();

return result;

}

public static void main(String[] args) {

Stack s = new Stack(0);

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

s.push(args[i]);

Stack s2 = null;

try {

s2 = (Stack) s.clone();

} catch(CloneNotSupportedException e) {

throw new Error("Assertion failure"); // Can't happen

}

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

System.out.print(s.pop()+" ");

System.out.println();

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

System.out.print(s2.pop()+" ");

System.out.println();

}

}

假设你希望是这个类成为可克隆的，如果它的clone方法只返回super.clone()，那么得到的Stack实例在size域上会有正确的值，但它的elements域会与初始的Stack实例引用同一个数组。对初始数组的修改会破坏克隆对象中的变量，反之亦然。很快你就会发现程序产生了无意义的结果，或者引发ArrayIndexOutOfBoundsException异常。

当调用Stack类中的唯一构造函数时，这种局面将不会出现。事实上，clone方法充当了另一个构造函数的角色，必须要保证不伤害初始对象，同时正确地建立克隆上的约束。为了使Stack中的clone方法正常工作，必需要拷贝栈的内部值。做到这一点的最简单的方法是对数组里面的元素递归调用clone。

注意，如果某些域是final的，这种方式不能正常运行。因为此时clone方法禁止为该域分配新值。有一个基本的问题：clone这种结构与引用可变对象的final域的正常使用是不兼容的。除非该可变对象可以在对象和它的克隆体之间被安全地共享，因此为了使类成为可克隆的，有时可能需要从某些域中移走final修饰符。

递归调用clone方法的方式并不是总是可行的。例如，如果需要为hash表编写一个clone方法，hash表由桶（bucket）数组组成，每个桶指向一个键值队的链表的第一个条目，如果桶为空，其值为null。处于性能上的考虑，类实现了自己的轻量级单向链表，没有使用系统自身的java.util.LinkedList：

public class HashTable implements Cloneable {

private Entry[] buckets = new Entry[100];

private static class Entry {

Object key;

Object value;

Entry next;

Entry(Object key, Object value, Entry next) {

this.key = key;

this.value = value;

this.next = next;

}

}

**// Broken - results in shared internal state!**

public Object clone() throws CloneNotSupportedException {

HashTable result = (HashTable) super.clone();

result.buckets = (Entry[]) buckets.clone();

return result;

}

}

尽管克隆出的实例有自己的桶数组，但这个数组还是与初始实例引用同一个数组，这会很容易地在双方之间造成不确定的行为（相当与2级引用？）。为了修正这个问题，不得不为每一个桶单独地拷贝链表（每一个桶相当于一个链表）。下面的例子是一个常用的方法：

public class HashTable implements Cloneable {

private Entry[] buckets = new Entry[100];

private static class Entry {

Object key;

Object value;

Entry next;

Entry(Object key, Object value, Entry next) {

this.key = key;

this.value = value;

this.next = next;

}

**// Recursively copy the linked list headed by this Entry**

Entry deepCopy() {

return new Entry(key, value, next == null ? null : next.deepCopy());

}

/\* COMMENTED OUT - Iterative alternative to above method

**// Iteratively copy the linked list headed by this Entry**

Entry deepCopy() {

Entry result = new Entry(key, value, next);

for (Entry p = result; p.next != null; p = p.next)

p.next = new Entry(p.next.key, p.next.value, p.next.next);

return result;

}

\*/

}

public Object clone() throws CloneNotSupportedException {

HashTable result = (HashTable) super.clone();

result.buckets = new Entry[buckets.length];

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

if (buckets[i] != null)

result.buckets[i] = buckets[i].deepCopy();

return result;

}

}

在支持深度拷贝（deepCopy）方法的问题上，私有类HashTable.Entry一直在被讨论。HashTable的clone方法按照初始的桶数组分配一个新的、大小合适的桶数组，然后深度拷贝每个非空桶。Entry上的深度拷贝方法被递归地调用以拷贝该条目指向的全部链表。如果桶不是太长，这种方法很漂亮。但是克隆链表并不是好办法，因为它要为链表里的每一个元素消耗一个栈单元。如果链表很长，这样做会很容易导致栈溢出。为了防止这种局面发生，可以在深度拷贝中用循环迭代来代替递归。

解决该克隆复杂对象的最终方法是调用super.clone，将结果对象的所有域设置成原始状态（virgin），然后调用更高层的方法来重新生成对象的状态。在Hashtable的例子情况下，buckers域将被初始化成新的桶数组，之后被克隆的hash表中的每一个键值映射对调用put(key,value)方法。这种方法通常能产生简单、雅致的clone方法，但直接管理对象及由它克隆得到的对象内部结构的方法要比这种方法运行得快。

象构造函数一样，clone方法在构造过程中不能调用克隆对象中得任何非final方法。如果clone调用了一个被重载的方法，那么这个方法会在定义它的子类之前获得执行，因此有机会修改克隆对象中的状态。因此刚刚讨论过的put(key,value)方法应该是final或者是private的。

Object的clone方法被声明为会引发CloneNotSupportedException的，但重载的clone方法可以忽略这一点。final类的close方法应该忽略该声明，因为在这里不引发被检查的异常比引发它更好。如果一个会扩展类，特别是用来实现继承的可扩展类重写了clone方法，那么重写的clone方法应该包含引发CloneNotSupportedException异常的声明，这样做允许子类可以体面地放弃克隆能力。实现方法如下：

//clone method to guarantee that instance cannot be cloned

public final Object clone() throws CloneNotSupportedException{

throw new CloneNotSupportedException();

}

遵守前面的建议不是必须的，因为子类不想被克隆时，它的clone方法总是会引发非检查的异常，例如，如果子类重写的clone方法没有声明引发CloneNotSupportedException，clone会发出UnSupporedOperationException。

扼要的说，实现Cloneable接口的所有类应该用公共方法的形式重写clone。这个公共方法应该首先调用super.clone，然后修改需要修改的域。一般情况下，这意味着要拷贝所有组成被克隆对象内部具有深度结构（deep structure）的可变对象，然后用指向拷贝出的对象引用来替换原来对这些对象的引用。尽管这些内部结构能通过递归的地调用clone方法生成，但这样做并不总是最好的办法。如果类仅仅包含原始类型域或是非可变对象的引用，那么所有的域都不需要被修改。但是，这条原则存在例外的情况。例如：代表序列号或其它的唯一ID号的域，以及代表对象的创建时间的域，尽管它们是基本数据类型或是非可变的，但它们需要被修改。

所有这些复杂的问题都是必要的吗？很少是。如果扩展了一个实现了Cloneable接口的类，那么除了实现一个表现良好的clone方法之外，没有其它的选择。或者提供对象拷贝的某种可替换的方法或者根本不提供这种功能，这样做可能是更好的选择。例如，对于非可变类，支持对象拷贝没有太大的意义，因为拷贝与原始对象事实上没有区别。

实现对象拷贝的精巧的方法是提供一个拷贝构造函数（copy constructure）。拷贝构造函数是一个简单的构造函数，使用唯一的参数，参数的类型即是包含该构造函数的类。例如：

public Yum(Yum yum);

public static Yum newInstance(Yum yum);

拷贝 构造函数及其它的静态工厂变形与Cloneable/clone方法相比有很多好处：它们不依赖与那种有风险的蹩脚的对象创建机制；不需要遵守由糟糕文档规范的约束；不会与final域的正常使用产生冲突；不要求客户不必要地捕捉被检查的异常；给客户提供了一种类型化的对象。尽管不可能把拷贝构造函数和静态工厂放入接口中，但是Cloneable由于缺乏public的clone方法，同样也没有实现接口功能，因此使用拷贝构造函数取代clone方法并没有放弃接口功能。

此外，拷贝构造函数（和静态工厂）可以使用参数，其参数类型就是该类实现的接口。例如，所有通用目的的集合实现按照惯例都提供了一个拷贝函数，其参数就是Collection或Map类型。基于接口的拷贝构造函数允许客户选择具体的拷贝实现，而不是强迫客户接受初始的实现。例如，现在有一个LinkedList llist，但你希望把它拷贝成ArrayList，Clone方法没有提供这种能力，但使用拷贝构造函数很容易实现：new ArrayList(llist)。

了解了与Cloneable相关的所有问题，可以确定地说，其它的接口不应该去扩展它，为实现继承而设计的类也不应该去扩展它。因为它有诸多的缺点，除了使用它的廉价拷贝数组能力外，一些有经验的程序员通常从不去重写clone方法，从不调用clone方法。必须清楚：对于为实现继承而设计的类，如果没有提供至少一种表现良好的受保护的clone方法，它的子类就不可能实现Clonealbe接口。

### 实现Comparable的问题

compareTo是java.lang.Comparable接口中唯一的方法。它允许进行简单的相等比较也允许其它顺序比较，除此之外它与Object的equals方法有类似的特性。类如果实现了Comparable接口，即表明它的实例具有内在的排序功能（natural ordering）。给一个实现Comparable接口的数组对象排列，可以简单地如下所示：Arrays.sort(a)。

Comparable对象的查找、极值计算及有序集合的自动维护都同样简单。类String实现了Comparable，下面的例子实现了给命令行参数剔除重复字符，然后按字符顺序打印列表：

import java.util.\*;

public class WordList {

public static void main(String[] args) {

Set s = new TreeSet();

s.addAll(Arrays.asList(args));

System.out.println(s);

}

}

运行java WordList asd ddf sdg 123 123 456 dfddf

输出结果为：[123, 456, asd, ddf, dfddf, sdg]

如果类实现了Comparable，就可以使它和许多依赖于这个接口的通用算法、集合实现互操作，就可以以很小的代价获得巨大的能力。事实上，Java类库中所有的值类型的类都实现了Comparable。如果正在设计一个具有自然排序特性的类，例如按照字母顺序、数字顺序、年代顺序等，那么应该认真考虑实现该接口。下面将告诉你该如何去做。

compareTo方法的一般约定与equals方法相似。下面是Comparable的规范：

将一个对象与指定的对象进行比较，如果对象小于、等于、大于指定的对象，对应地返回负整数、0或正整数。如果指定的对象类型无法与该对象进行比较，则引发ClassCastException。sgn(expression)实现了数学意义上的signum函数功能，它根据expression的值为负、0和正，定义返回-1、0或1。compareTo方法实现程序必须保证：

1．实现程序必须保证：对于所有x和y，sgn(x.compareTo(y))== -sgn(y.compareTo(x))，这意味着当且仅当y.compareTo(x)引发异常时，x.compareTo(y)引发异常。

2．实现程序必须保证关系是可传递的：( x.compareTo(y)>0 && y.compareTo(z)>0 )，意味着x.compareTo(z)>0。

3．实现程序必须保证：如果 x.compareTo(y)==0，则意味着对所有z，sgn(x.compareTo(z)) == sgn(y.compareTo(z))。

4．强烈推荐保证(x.compareTo(y)==0) == (x.equals(y))成立，但这不是必需的。一般来说，实现了Comparable接口但不满足本条件的类应明确地表明这个事实。在文档中，推荐这样的语句：注意该类有与等同关系不一致的自排序能力。

在类内，任何合理的顺序关系都可以满足compareTo约定，但与equals方法不同，在类之间compareTo方法不能工作：如果被比较的两个对象引用指向不同类的对象，compareTo允许引发ClassCastException异常。在这种条件下，通常这正是compareTo所应该做的。尽管约定没有排除类之间的比较，但在1.4版中，Java类库还不支持这种能力。

就像违背了hashCode约定的类会破坏其它的依赖于散列的类一样，违背了compareTo约定的类同样能破坏其它的依赖于比较的类。依赖比较关系的类包括排序集合：TreeSet和TreeMap，实用类Collections和Arrays，它们中包含有查找和排序算法。

compareTo的第1、2和3条约定说明其必须遵守约束规则：自反性、对程序、传递性和非空特性（non-nullity）。因此，没有简单的方法能够做到在增加新特性扩展类实例的同时，仍能使它保持compareTo约定。因此同样，如果给实现了Compareable接口的类增加了一个主要特征，则不能在扩展该类；应该使这个类作为另一个类的域，再提供一个view方法返回需要的域。这样做，可以在第二个类上自由地实现所需要地compareTo方法，同时在需要时，使客户可以把后面类的实例看做前面类的实例。

compareTo的第4条约定只是一个强烈建议，而不是真正的规定，它要求compareTo方法施加相等测试一般应该返回与equals方法相同的结果。如果遵守这一条，那么compareTo方法产生的排序与equals方法产生的一致，否则将不一致。如果不一致，类仍然可以工作，但包含该类元素的顺序集合也许将不会遵守相应的集合接口（Collection、Set和Map）的一般约定。因为接口的一般约定是按照equals方法定义的，但顺序集合使用了compareTo而不是equals方法施加相等测试。尽管这种情况的出现不会造成灾难，但应该有所了解。

以Float为例，它的compareTo方法与equals方法不一致。如果创建了HashSet，然后增加了new Float(-0.0f)和new Float(0.0f)，那么集合会包含两个元素，因为增加到集合中的两个实例使用equals方法比较时不相等。然而如果用TreeSet代替HashSet执行相同的过程，集合中将存在唯一的一个元素，因为这两个Float实例用compareTo方法比较时是相等的。

设计compareTo方法与equals方法基本相似，但也存在几点关键差异。在映射之前不需要对参数做类型检查，如果参数不是合适的类型，compareTo方法会引发ClassCaseException异常。如果参数为null，compareTo方法会引发NullPointerException异常。这正是当你仅把参数映射成正确的类型，然后试图访问它的成员时，应该出现的结果。

域的比较本质上是顺序比较，而不是等值比较。对对象引用类型的域比较，可以通过递归地调用compareTo方法来实现。如果域没有实现Compareable或者需要使用非标准地顺序，那么可以使用显式的Comparator。

/\*\*

\* Case-insensitive string. Case of the original string is preserved by toString, but

\* ignored in comparisons. This version implements Comparable.

\*/

import java.util.\*;

public final class CaseInsensitiveString implements Comparable {

private String s;

public CaseInsensitiveString(String s) {

if (s == null)

throw new NullPointerException();

this.s = s;

}

public boolean equals(Object o) {

return o instanceof CaseInsensitiveString &&

((CaseInsensitiveString)o).s.equalsIgnoreCase(s);

}

// Lazily initialized, cached hashCode

private volatile int hashCode = 0;

public int hashCode() {

if (hashCode == 0)

hashCode = s.toUpperCase().hashCode();

return hashCode;

}

public int compareTo(Object o) {

CaseInsensitiveString cis = (CaseInsensitiveString)o;

return String.CASE\_INSENSITIVE\_ORDER.compare(s, cis.s);

}

public String toString() {

return s;

}

static void main(String[] args) {

// Print arguments in order

CaseInsensitiveString ciArgs[] = new CaseInsensitiveString[args.length];

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

ciArgs[i] = new CaseInsensitiveString(args[i]);

Arrays.sort(ciArgs);

System.out.println(Arrays.asList(ciArgs));

// Print arguments with duplicates eliminated

Set s = new HashSet();

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

s.add(ciArgs[i]);

System.out.println(s);

}

}

运行：java CaseInsensitiveString asdfg sdf dfg dfg ffrr 123 456 456 123

输出结果：[123, 123, 456, 456, asdfg, dfg, dfg, ffrr, sdf]

[asdfg, dfg, 123, sdf, 456, ffrr]

原始类型的比较使用关系操作符，如””<”和””>””，数组类型域的比较通过对每一个元素分别使用上述规则的方法实现。如果类有多个主要域，那么比较的顺序很关键。应该从最重要的域开始进行比较工作，如果比较产生了非零（0代表相等），比较完成，返回比较结果即可；如果对最重要的域的比较结果为零，对次要的域进行比较，依此类推。如果对所有的域的比较结果都相等，那么对象相等，返回零。

// PhoneNumber class

import java.util.\*;

public final class PhoneNumber implements Comparable {

private final short areaCode;

private final short exchange;

private final short extension;

public PhoneNumber(int areaCode, int exchange, int extension) {

rangeCheck(areaCode, 999, "area code");

rangeCheck(exchange, 999, "exchange");

rangeCheck(extension, 9999, "extension");

this.areaCode = (short) areaCode;

this.exchange = (short) exchange;

this.extension = (short) extension;

}

private static void rangeCheck(int arg, int max, String name) {

if (arg < 0 || arg > max)

throw new IllegalArgumentException(name +": " + arg);

}

public boolean equals(Object o) {

if (o == this)

return true;

if (!(o instanceof PhoneNumber))

return false;

PhoneNumber pn = (PhoneNumber)o;

return pn.extension == extension &&

pn.exchange == exchange &&

pn.areaCode == areaCode;

}

public int hashCode() {

int result = 17;

result = 37\*result + areaCode;

result = 37\*result + exchange;

result = 37\*result + extension;

return result;

}

/\* COMMENTED OUT

public int compareTo(Object o) {

PhoneNumber pn = (PhoneNumber)o;

// Compare area codes

if (areaCode < pn.areaCode)

return -1;

if (areaCode > pn.areaCode)

return 1;

// Area codes are equal, compare exchanges

if (exchange < pn.exchange)

return -1;

if (exchange > pn.exchange)

return 1;

// Area codes and exchanges are equal, compare extensions

if (extension < pn.extension)

return -1;

if (extension > pn.extension)

return 1;

return 0; // All fields are equal

}

\*/

public int compareTo(Object o) {

PhoneNumber pn = (PhoneNumber)o;

// Compare area codes

int areaCodeDiff = areaCode - pn.areaCode;

if (areaCodeDiff != 0)

return areaCodeDiff;

// Area codes are equal, compare exchanges

int exchangeDiff = exchange - pn.exchange;

if (exchangeDiff != 0)

return exchangeDiff;

// Area codes and exchanges are equal, compare extensions

return extension - pn.extension;

}

public String toString() {

return "(" + toPaddedString(areaCode, 3) + ") " +

toPaddedString(exchange, 3) + "-" +

toPaddedString(extension, 4);

}

private static String toPaddedString(int i, int length) {

String s = Integer.toString(i);

return ZEROS[length - s.length()] + s;

}

private static String[] ZEROS =

{"", "0", "00", "000", "0000", "00000",

"000000", "0000000", "00000000", "000000000"};

public static void main(String args[]) {

int n = Integer.parseInt(args[0]);

Random rnd = new Random();

PhoneNumber[] a = new PhoneNumber[n];

for (int i=0; i<n; i++)

a[i] = new PhoneNumber(rnd.nextInt(1000), rnd.nextInt(1000),

rnd.nextInt(10000));

Arrays.sort(a);

System.out.println(Arrays.asList(a));

}

}

虽然该颜色的方法compareTo可以很好的工作，但它还有改进的余地。因为对compareTo方法的约定没有规定返回值的大小，只是规定了符号。因此可以利用这一点，简化代码，并可能提高速度。

这个技巧在本例运行得很好，但使用时应该特别小心。除非可以确定类中得域不为负值，或者域可能取的最大值和最小值之差不大于Integer.MAX\_VALUE(2^31-1)，否则不要使用这个方法。此技巧不能成为一般用法的原因时有符号32位整数不能大到足以表示任意两个有符号32位整数的差。如果i是一个大的正整数，j是一个很小的负数，那么(I-j)会溢出，返回负值，compareTo的结果将失去意义。对某些参数它返回无意义的结果，也可能会破坏compareTo约定的前两条。这不仅仅是理论上的问题，而且会造成真实系统的失效。这样的失效很难调试，由于出问题的compareTo方法对许多输入仍能正常工作。

## 类和接口

类和接口在Java程序设计语言中处于核心地位，它们构成了Java抽象的基本单元。

### 最小化类和成员的可访问能力

Java语言通过访问控制（access control）来决定类、接口和成员的可访问能力。实体的可访问能力由实体声明的位置、实体声明中的访问控制修饰符（private、protected、public）决定。正确地使用这些修饰符对于实现信息的隐藏是必须的。

经验规则指出应该使每一个类或成员尽可能地不被外部访问。换句话说，在设计程序时，应根据软件功能，使用允许的最低的访问等级。

对于顶层（非嵌套的）类和接口，只有两个可能的访问级：包内私有的（package private）和公有的（public）。如果声明了一个具有public修饰符的顶层类或接口，那么它是公有的；否则它是包内私有的。如果顶层类或接口可以成为包内私有的，就应该使它成为包内私有的。如果是包内私有的，那么它就是包实现的一部分，而不是导出的API。这样在以后的版本中，可以对它修改、替换或撤销，而不用担心会伤害现有的用户。如果它成为public，就被迫要永远支持它，以保持兼容性。

如果包内私有的顶层类或接口仅在唯一的类中使用，可以考虑使它成为这个类或接口的私有嵌套类（接口）。这进一步降低了可访问性。然而，这样做不像使一个必要的公共类成为包内私有类那样更有意义，因为包内私有类已经是包实现的一部分而不是它的API。

对于成员（域、方法、嵌套类和嵌套接口）有四种可能的访问级，下面按照访问性递增的顺序列出：

* 私有（private）：这种类型的成员仅在它被声明的顶层类内部可以访问。
* 包内私有的（package private）：这种类型的成员对声明它的包内所有类可访问。在技术上称之为缺省访问（default access）。如果成员没有指定修饰符，它就属于这种访问级。
* 受保护的（protected）：这种类型的成员对于声明它的类的子类及包内所有的类可访问。（对第一种情况，存在一些限制）
* 公共的（public）：成员的访问不受限制。

在认真的设计了类的公共API后，下一步工作应该是使其它的成员私有。只有当同一包里的其它类确实需要访问该成员的时候，才可以为它去除private修饰符，使成员成为包内私有的。如果发现自己在频繁地做这项工作，那么应该重新检查设计，决定是否能分解产生新的类，以减少相互作用。可以说，私有的和包内私有的成员是类实现的一部分，一般不会影响它的API。但是如果类实现了Serializable接口，这些域可能会泄露（leak）到导出的API中。

对于公共类的成员，当访问等级从包内私有变成受保护时，会出现访问性上的巨大增加。受保护的成员是类的导出API的一部分，必须一直被支持。而且导出类的受保护成员对类的内部实现细节而言，相当于做出了与公共类型相同的承诺。受保护类型的成员应该尽量减少。

有一条规则限制了我们给方法降低可访问的能力：如果方法要重写超类中的方法，那么不允许子类中该方法的访问等级低于超类中的访问等级。为了保证在超类的实例可用的地方，子类的实例也都可以用，这样的规则是必须的。如果违背了这条规则，在编译子类的时候，编译器会产生一个错误。这条规则的一个特殊情形是，如果类实现了一个接口，那么类中对应接口中的方法必须要被声明为公共的。这是因为接口中所有方法隐含地声明为公共的。

公共类中基本上不应该存在公共域（不同于公共的方法）。原因如下：如果类中的域是非final的，或者是可变对象的fianl引用，那么一旦它成为公共的，就失去了对存储在这个域中的值进行约束的能力；也将失去当域被修改时采取行动的能力。这样的一个简单后果就是具有公共的可变域的类不是线程安全的。即使域是final的并且不是对可变对象的引用，如果它成为公共的，也将失去把域切换到新的内部数据表示的灵活性。

对”公共类不应该有公共域的这条规则”存在一个例外。类允许通过公共的静态的final域暴露常量。按照习惯，这种类的名字由大写字母开头，后随下划线再加其它字符。关键的一点是这些域应该只能包含或者是原始类型值或者是非可变对象的引用。包含有可变对象引用的final域具有非fianl域的所有缺点。尽管引用不能被修改，但被引用的对象可以被修改，这会带来灾难性的结果。

注意：零长度的数组总是可变的，所以类中出现公共的静态final数组的域时差不多永远是错误的。如果类中有这样的域，客户将可以修改数组的内容。这通常是安全漏洞的根源：

**// Potential security hole!**

public static final Type[] VALUES = { …… };

此公共数组应该使用一个私有数组和一个公共可变列表来代替。

private static final Type[] PRIVATE\_VALUES = { …… };

public static final List VALUES =

Collections.unmodifiableList(Arrays.asList(PRIVATE\_VALUES));

并且，如果你需要获得编译时的类型安全，且愿意损失一些性能的话，可以用返回私有数组的拷贝公共方法来替代此公共数组：

private static final Type[] PRIVATE\_VALUES = { …… };

public static final Type[] values(){

return (Type[]) PRIVATE\_VALUES.clone();

}

总的来说，应该尽可能减少类或接口的可访问性。在认真地设计了最小的公共API之后，应该防止任何杂散（stray）类、接口或者成员成为API的一部分。除了公共的静态final域的例外情况，公共类不应该有公共域。要保证被公共静态final域引用的对象是非可变的。

### 倾向于非可变性

非可变类就是类实例不能被修改的类。包含在每个实例中的所有信息在实例创建时提供，然后在对象的整个生命周期中是固定的。Java类库中包含有许多非可变类，如String、原始类型的包装类、BigInteger和BigDecimal。非可变类有许多好的存在理由：与可变类相比，非可变类更容易设计、实现和使用；不容易犯错误；更安全。

为了使类成为非可变的，要遵循下面5条原则：

* **不要提供任何修改对象的方法。**（被称为变动器mutator）
* **保证没有可以被重写的方法。**这防止了粗心的或恶意的子类危害对象的不可变行为。防止方法被重写的一般方法是使类成为final的。但还有其它选择。
* **使所有的域都成为final的。**借助于系统的强制方式，可以清楚地表达出你的意图。而且在不使用同步机制的情况下，当新创建对象的引用从一个线程传递到另一个线程时，为了保证对象行为的正确性：依赖于重新设计中的内存模型（memory model），这一条也是必需的。
* **使所有的域都是私有的（private）。**这可以防止客户直接修改域。尽管非可变类具有包含原始类型值或不可变对象引用的公共final属性的域，从技术上是允许的，但实际中不推荐这样使用，因为这会使对以后版本内部表示的改变成为不可能。
* **保证对任何可变组件的互斥访问。**如果类中存在指向可变对象的引用，要保证类的客户不能获得对这些对象的引用。既不要用客户提供的对象引用初始化这样的域，也不要从访问器（accessor）中返回对象的引用。在构造函数、访问器和readObject中使用保护性拷贝方法（defensive copy）。

给出一个稍微复杂的例子（该例仅说明非可变性，它不是实际意义上得复数实现）：

public final class Complex {

private final float re;

private final float im;

public Complex(float re, float im) {

this.re = re;

this.im = im;

}

**// Accessors with no corresponding mutators**

public float realPart() { return re; }

public float imaginaryPart() { return im; }

public Complex add(Complex c) {

return new Complex(re + c.re, im + c.im);

}

public Complex subtract(Complex c) {

return new Complex(re - c.re, im - c.im);

}

public Complex multiply(Complex c) {

return new Complex(re\*c.re - im\*c.im, re\*c.im + im\*c.re);

}

public Complex divide(Complex c) {

float tmp = c.re\*c.re + c.im\*c.im;

return new Complex((re\*c.re + im\*c.im)/tmp, (im\*c.re - re\*c.im)/tmp);

}

public boolean equals(Object o) {

if (o == this)

return true;

if (!(o instanceof Complex))

return false;

Complex c = (Complex)o;

return (Float.floatToIntBits(re) == Float.floatToIntBits(c.re)) &&

(Float.floatToIntBits(im) == Float.floatToIntBits(c.im));

}

public int hashCode() {

int result = 17 + Float.floatToIntBits(re);

result = 37\*result + Float.floatToIntBits(im);

return result;

}

public String toString() {

return "(" + re + " + " + im + "i)";

}

**// Public constants**

public static final Complex ZERO = new Complex(0, 0);

public static final Complex ONE = new Complex(1, 0);

public static final Complex I = new Complex(0, 1);

public static void main(String args[]) {

Complex x = new Complex(2, 3);

Complex y = new Complex(2,-3);

System.out.println(x + " + " + y + " = " + x.add(y));

System.out.println(x + " - " + y + " = " + x.subtract(y));

System.out.println(x + " \* " + y + " = " + x.multiply(y));

System.out.println(x + " / " + y + " = " + x.divide(y));

System.out.println(x.divide(y).multiply(y));

}

}

该类表示复数（complex number，具有实部和虚部）。注意到算术运算操作创建和返回一个新的复数实例，而不是修改了这个实例。这种模式被用在一些重要的非可变类中，被成为函数方式（functional），因为方法返回了对操作数使用函数产生的结果。与此相对应的更通用的实过程方式（procedure），在这种方式下，方法在操作数上使用了一个过程，造成状态改变。

采用函数方式可以实现非可变性，具有许多优点。首先，非可变对象简单。非可变对象可以仅存在于一种状态中，即它被创建时的状态。如果能够保证构造函数建立了所有的常量，那么就保证了这些常量将永远不变，不需要你及使用它们的程序员为保证这一点做额外的工作。另一方面，可变对象具有任意的复杂的状态空间。如果文档没有对变动器（mutator）进行状态转换提供精确的描述，那么可信地使用一个可变对象是困难地甚至是不大可能的。

非可变对象本质上是线程安全的，不需要同步机制。它不会由于多线程的并发方法而被破坏。这是获得线程安全的最简单的途径。事实上，线程不会看到其它线程对非可变对象施加的影响。因而，非可变对象可以被自由地共享。做到这一点的一个简单的方法是为频繁使用的值提供公共的静态final属性常数。

这种方法可以更进一步。非可变对象可以提供一个静态工厂方法，将经常被请求的实例缓存起来，避免在被请求的实例存在时，重复地创建新的实例。BigInteger和Boolean类都具有这种静态工厂方法。

非可变对象可以被自由地共享，这一特性地结果时永远不必使用保护性拷贝。事实上，根本不必做任何拷贝，因为任何拷贝都与初始对象相同，因此不必要也不应该对非可变类提供拷贝构造函数和clone方法。这一点在Java出现的早期不容易理解，所以尽管String有一个拷贝构造函数，但它很少被使用。

不仅可以共享非可变对象，而且还可以共享它们内部的信息。例如，BigInteger类内部使用了一个有符号值形式，符号部分用int类型的数表示，数值部分用int数组表示。Negate方法产生一个同样的数值部分和一个相反符号部分。它不需要拷贝数组：将新创建的BigInteger指向初始的内部数组即可。

非可变对象为其它对象----无论是可变的还是非可变的，构件了大量的构造块。如果知道构建复杂对象的组件组件对象不会变，那么维护此复杂对象的约束会更容易。这一原则的特别例子是使用了大量映射键和集合元素的非可变对象，一旦它们进入映射或集合，就不必担心这些对象的值会改变----这样的改变会破坏映射或集合的约束。

非可变对象的真正也是唯一的缺点是对每个不同的值要求有一个单独的对象。创建这种对象需要付出开销，尤其在对象比较大时。

如果要执行多步操作，每一步产生一个新的对象，但最终只保留最后的结果，那么性能问题就显现出来了。有两种方法可以解决这个问题，第一种是先假想那些多步操作会被普遍需要，就把它们作为基本单元提供。如果一个多步操作已经被作为基本单元提供，那么非可变类就不必每一步都创建一个单独的对象。如果能够准确地预测出客户想要在你的非可变类上做哪些复杂的多步操作，那么这种方法会运行的很好。否则，最好的赌注是提供一个公共的可变伙伴类。在Java类库中，使用这种方法最有名的例子是String类以及可变的伙伴类StringBuffer。

为了保证非可变性，必须要使类的任何方法不能被重写。使类成为final可以保证这一点。除此之外还有其它两种方法可以保证这一点。一种方法是使类中所有的方法成为final的。这种方式的唯一好处是允许程序员在已有类的基础上增加新的方法，实现类的扩展。这种方式因为与在单独的、不可实例化的实用类中提供新方法作为静态方法的效果相同，因此不推荐使用。

另一种可选择的方法是使类的所有构造函数成为私有的或包内私有的，然后增加公共的静态工厂代替公共的构造函数。

**// Immutable class with static factories of constructors**

public class Complex {

private final float re;

private final float im;

**private** Complex(float re, float im) {

this.re = re;

this.im = im;

}

**public static Complex valueOf(float re, float im)** {

return new Complex(re, im);

}

…… //Remainder

}

尽管这种方式没有被普遍使用，却通常是三种方式中最好的一个选择。它是最灵活的方式，因为这样允许多个包内私有的类实现的使用。对包外的客户，非可变类实际上是fianl的，因为扩展来自其它包的、没有公共的或受保护的构造函数的类是不可能的。除此之外，这种方式可以在以后的版本中，通过提高静态工厂的对象缓存能力，来改善程序的性能。

静态工厂与构造函数相比有许多好处。例如，假设Complex类需要提供一种基于极坐标创建复数的方式。使用构造函数会很麻烦，因为它本身的构造函数与将要提供的构造函数具有同样的签名：complex(float, float)。但使用静态工厂方法会使这个问题简单：增加一个静态工厂方法，并使它使用可以表示该功能的名字即可：

public static Complex valueOfPolar(float r, float theta) {

return new Complex((float) (r \* Math.cos(theta)), (float) (r \* Math.sin(theta)));

}

但是，在BigInteger和BigDecimal设计的时候，是非可变类成为final这一点还没有被广泛接受。所以，它们的所有方法都可以被重写。不幸的是，为了保持向上兼容，这一问题将不会纠正。如果你写了一个类，它的安全依赖于不可信用户提供的BigInteger和BigDecimal参数的非可变性，那么，你需要检查参数是否是BigInteger和BigDecimal的实例，以避免是不可信的子类的实例。如果出现后面的情况，就必须要在参数可能是可变的假设下，做保护性的拷贝。

public void foo(BigInteger b){

if(b.gerClass() != BigInteger.class)

b = new BigInteger(b.toByteArray());

……

}

在本节开始有关非可变类的一系列规则指出不允许有方法可以修改对象，而且所有的域都应该是fianl的。事实上，这些规则优点过于严格，为了提高性能可以放松限制。实际使用中，没有哪种方法能对对象状态产生外部可见的改变。然而，许多非可变类定义一个或多个非final的冗余域，用来缓存高开销计算在首次被请求时所得的结果。如果以后再进行同样的计算，就返回缓存值，节省重新计算的开销。这种技巧能够正常地工作，因为对象是非可变的。这种非可变性保证了计算在重新执行时会产生同样的结果。

例如PhoneNumber的hashCode方法在第一次被调用时，会计算hash码，然后把结果缓存起来，以免它再次被需要。这种技术----一种典型的惰性初始化方法，在String类中已被采用。它不需要同步机制，由于hash值被计算几次都不会产生问题。下面时非可变对象的缓存的、惰性初始化的函数一般用法：

**// Cached, lazily initialized function of an immutable object**

private volatile Foo cachedFooVal = UNLIKELY\_FOO\_VALUE;

public Foo foo() {

Foo result = cachedFooVal;

if (result == UNLIKELY\_FOO\_VALUE)

result = cachedFooVal = fooVal();

}

// Private helper function to calculate our foo valur

pribate Foo fooVal() {……}

有关串行化的问题需要在此做以说明。如果需要使用非可变类实现Serializable，而类中包含有几个对可变对象的引用，那么需要显式地提供readObject或readResplve方法，即使缺省的串行化形式是可以接受的。缺省的readObject方法会允许攻击者从非可变类中创建可变的实例。

概括地说，我们反对对每一个get方法都提供相应的ser方法的做法。除非有足够的理由使类变成可变的，否则应该使类做为非可变的类出现。非可变类具有许多优点，而它唯一的缺点是在某些情况下的潜在的性能问题。应该总是使较小的对象成为非可变的，例如Complexh和PhoneNumber（在Java中，有几种类应该成为非可变的，确没有这样做：如java.util.Date和 java.awt.Point）。也应该认真地考虑使较大的对象成为非可变的，如String和BigInteger。只有在确实需要考虑性能问题时才可以考虑为非可变类提供公共的可变伙伴类的方式。

对有些类，非可变性对它们是不实际的。这样的类包括”线程类”，如Thread和TimerTask。如果类不能成为非可变的，那么就要尽可能地限制它们的可变性。减少对象可以存在的状态数，可以更容易地分析对象，降低出错的可能性。所以，构造函数应该创建对象的常量，建立完全初始化的对象。构造函数不应该向其它方法传递没有构造完全的实例。除非有绝对的理由，否则不能在构造函数之外提供公共的初始化方法。同样，不能提供”再初始化”方法，它会使对象好像被使用了不同的初始状态重新被构造。与增加了的复杂性，再初始化方法通常不会对性能的提高有多少益处。

TimerTask类例证了这些原则。它是可变的，但它的状态空间被设计得很小。实例创建后，可以对它进行调度、执行或者取消。一旦一个计时任务运行结束，或者被取消，就不允许被再次调用。

### 组合优于继承

继承是实现代码重用的有力途径，但它不总是完成这项工作的最好工具。如果被不适当地使用，会产生脆弱地软件。在包内使用继承是安全的，由于这时子类和超类的实现是在同一个程序员的控制下。对于专门设计并且具有文档说明的用做扩展的类，使用继承也是安全的。然而，跨越包的边界，从普通的、具体的类实现继承是危险的。

与方法调用不同，继承打破了封装性。换句话说，子类的特有的功能，依赖于它的超类的实现细节。作为继承的后果，子类必然会与超类关联在一起，除非超类的设计者把它设计成专门用来实现扩展的，并且提供了文档说明。

### 设计和文档化继承

### 接口优于抽象类

### 接口只能用来定义类型

### 静态成员优于非静态的

# 思考问题

## Java与C++有什么区别？

Java去掉了C++中的指针运算、结构、typedefs、#define和需要释放内存，而且Java的整个解释器只需215K的RAM。

## 你在编程中出过什么样的错误？

## 你知道对象是如何在内存中组织的吗？Java真的不用管理内存分配吗？

## 有哪些方法可以优化Java程序的性能？如何定位性能瓶颈？

该问题涉及的内容比较广，可从下面给予考虑：

* + 1. 对象的生成和大小的调整

如：String、StringBuffer；+和append的合理选择。

* + 1. 慎用异常类

异常的频繁产生会生成大量的异常类对象。

* + 1. 同步的合理使用
    2. I/O优化

使用缓冲，如：BufferedInputStream等。

* + 1. JDBC优化

选择合理的驱动器程序类型。（在独立性和性能上做权衡）

* + 1. 合理使用关键词

如：采用final进行代码的内联等。

* + 1. 局部变量

调用方法时传递的参数以及在调用中创建的临时变量都保存在栈（Stack）中，速度较快。其它变量，如静态变量、实例变量等，都在堆（Heap）中创建，速度较慢。

* + 1. 合理选择数据结构(SET/LIST/MAP等系列)等。

如：正确选用ArrayList,LinkedList, Vector等。

## 什么是内存泄漏？JVM一般在什么情况下开始垃圾回收？

内存泄漏的本质是：

使用后的对象没有被回收，至少有一个活动块保持着对象的应用，而此时程序处理逻辑不能访问该对象。因此，内存泄漏主要是由编程引起的，应确认当一个对象不再使用时，指向它的所有引用均应置为空。

当指向这个对象的所有引用均被置为null时，该对象就符合回收的条件，但不一定马上开始回收。

## Swing主要采用什么设计模式？存在哪些线程安全问题？

MVC模式

Swing不保证线程安全性，它的大多数方法不是被同步的方法。不能在自己的线程中调用非线程安全的Swing方法。

# 编码规则

## Java规则

* 模块（指可重用的软件组件，范围涉及单一方法及包含有多个包的复杂系统）要使用户清楚它的行为；
* 模块应该尽可能地简洁，但不是越小越好；
* 代码应该被重用而不是被拷贝；
* 模块间的依赖性应该降到最低；
* 错误应该尽早地被检测到，理想的情况是在编译期间。

不要求你盲目地遵循这些原则，但要尽可能少地打破这些原则，并要有足够充分的理由。编程艺术的学习，像大多数其它训练一样，首先要学会使用原则的用法，然后才可以尝试打破这些原则。

首先关心如何编写清晰、正确、可用、健壮、伸缩性好和可维护的程序，然后再去考虑系统的性能问题。

## Java编程30条建议

(1) 类名首字母应该大写。字段、方法以及对象（句柄）的首字母应小写。对于所有标识符，其中包含的所有单词都应紧靠在一起，而且大写中间单词的首字母。例如：

ThisIsAClassName

thisIsMethodOrFieldName

若在定义中出现了常数初始化字符，则大写static final基本类型标识符中的所有字母。这样便可标志出它们属于编译期的常数。

Java包（Package）属于一种特殊情况：它们全都是小写字母，即便中间的单词亦是如此。对于域名扩展名称，如com，org，net或者edu等，全部都应小写（这也是Java 1.1和Java 1.2的区别之一）。

(2) 为了常规用途而创建一个类时，请采取"经典形式"，并包含对下述元素的定义：

equals()

hashCode()

toString()

clone()（implement Cloneable）

implement Serializable

(3) 对于自己创建的每一个类，都考虑置入一个main()，其中包含了用于测试那个类的代码。为使用一个项目中的类，我们没必要删除测试代码。若进行了任何形式的改动，可方便地返回测试。这些代码也可作为如何使用类的一个示例使用。

(4) 应将方法设计成简要的、功能性单元，用它描述和实现一个不连续的类接口部分。理想情况下，方法应简明扼要。若长度很大，可考虑通过某种方式将其分割成较短的几个方法。这样做也便于类内代码的重复使用（有些时候，方法必须非常大，但它们仍应只做同样的一件事情）。

(5) 设计一个类时，请设身处地为客户程序员考虑一下（类的使用方法应该是非常明确的）。然后，再设身处地为管理代码的人考虑一下（预计有可能进行哪些形式的修改，想想用什么方法可把它们变得更简单）。

(6) 使类尽可能短小精悍，而且只解决一个特定的问题。下面是对类设计的一些建议：

■一个复杂的开关语句：考虑采用"多形"机制

■数量众多的方法涉及到类型差别极大的操作：考虑用几个类来分别实现

■许多成员变量在特征上有很大的差别：考虑使用几个类

(7) 让一切东西都尽可能地"私有"--private。可使库的某一部分"公共化"（一个方法、类或者一个字段等等），就永远不能把它拿出。若强行拿出，就可能破坏其他人现有的代码，使他们不得不重新编写和设计。若只公布自己必须公布的，就可放心大胆地改变其他任何东西。在多线程环境中，隐私是特别重要的一个因素--只有private字段才能在非同步使用的情况下受到保护。

(8) 谨惕"巨大对象综合症"。对一些习惯于顺序编程思维、且初涉OOP领域的新手，往往喜欢先写一个顺序执行的程序，再把它嵌入一个或两个巨大的对象里。根据编程原理，对象表达的应该是应用程序的概念，而非应用程序本身。

(9) 若不得已进行一些不太雅观的编程，至少应该把那些代码置于一个类的内部。

(10) 任何时候只要发现类与类之间结合得非常紧密，就需要考虑是否采用内部类，从而改善编码及维护工作

(11) 尽可能细致地加上注释，并用javadoc注释文档语法生成自己的程序文档。

(12) 避免使用"魔术数字"，这些数字很难与代码很好地配合。如以后需要修改它，无疑会成为一场噩梦，因为根本不知道"100"到底是指"数组大小"还是"其他全然不同的东西"。所以，我们应创建一个常数，并为其使用具有说服力的描述性名称，并在整个程序中都采用常数标识符。这样可使程序更易理解以及更易维护。

(13) 涉及构建器和异常的时候，通常希望重新丢弃在构建器中捕获的任何异常--如果它造成了那个对象的创建失败。这样一来，调用者就不会以为那个对象已正确地创建，从而盲目地继续。

(14) 当客户程序员用完对象以后，若你的类要求进行任何清除工作，可考虑将清除代码置于一个良好定义的方法里，采用类似于cleanup()这样的名字，明确表明自己的用途。除此以外，可在类内放置一个boolean（布尔）标记，指出对象是否已被清除。在类的finalize()方法里，请确定对象已被清除，并已丢弃了从RuntimeException继承的一个类（如果还没有的话），从而指出一个编程错误。在采取象这样的方案之前，请确定finalize()能够在自己的系统中工作（可能需要调用System.runFinalizersOnExit(true)，从而确保这一行为）。

(15) 在一个特定的作用域内，若一个对象必须清除（非由垃圾收集机制处理），请采用下述方法：初始化对象；若成功，则立即进入一个含有finally从句的try块，开始清除工作。

(16) 若在初始化过程中需要覆盖（取消）finalize()，请记住调用super.finalize()（若Object属于我们的直接超类，则无此必要）。在对finalize()进行覆盖的过程中，对super.finalize()的调用应属于最后一个行动，而不应是第一个行动，这样可确保在需要基础类组件的时候它们依然有效。

(17) 创建大小固定的对象集合时，请将它们传输至一个数组（若准备从一个方法里返回这个集合，更应如此操作）。这样一来，我们就可享受到数组在编译期进行类型检查的好处。此外，为使用它们，数组的接收者也许并不需要将对象"造型"到数组里。

(18) 尽量使用interfaces，不要使用abstract类。若已知某样东西准备成为一个基础类，那么第一个选择应是将其变成一个interface（接口）。只有在不得不使用方法定义或者成员变量的时候，才需要将其变成一个abstract（抽象）类。接口主要描述了客户希望做什么事情，而一个类则致力于（或允许）具体的实施细节。

(19) 在构建器内部，只进行那些将对象设为正确状态所需的工作。尽可能地避免调用其他方法，因为那些方法可能被其他人覆盖或取消，从而在构建过程中产生不可预知的结果（参见第7章的详细说明）。

(20) 对象不应只是简单地容纳一些数据；它们的行为也应得到良好的定义。

(21) 在现成类的基础上创建新类时，请首先选择"新建"或"创作"。只有自己的设计要求必须继承时，才应考虑这方面的问题。若在本来允许新建的场合使用了继承，则整个设计会变得没有必要地复杂。

(22) 用继承及方法覆盖来表示行为间的差异，而用字段表示状态间的区别。一个非常极端的例子是通过对不同类的继承来表示颜色，这是绝对应该避免的：应直接使用一个"颜色"字段。

(23) 为避免编程时遇到麻烦，请保证在自己类路径指到的任何地方，每个名字都仅对应一个类。否则，编译器可能先找到同名的另一个类，并报告出错消息。若怀疑自己碰到了类路径问题，请试试在类路径的每一个起点，搜索一下同名的.class文件。

(24) 在Java 1.1 AWT中使用事件"适配器"时，特别容易碰到一个陷阱。若覆盖了某个适配器方法，同时拼写方法没有特别讲究，最后的结果就是新添加一个方法，而不是覆盖现成方法。然而，由于这样做是完全合法的，所以不会从编译器或运行期系统获得任何出错提示--只不过代码的工作就变得不正常了。

(25) 用合理的设计方案消除"伪功能"。也就是说，假若只需要创建类的一个对象，就不要提前限制自己使用应用程序，并加上一条"只生成其中一个"注释。请考虑将其封装成一个"独生子"的形式。若在主程序里有大量散乱的代码，用于创建自己的对象，请考虑采纳一种创造性的方案，将些代码封装起来。

(26) 警惕"分析瘫痪"。请记住，无论如何都要提前了解整个项目的状况，再去考察其中的细节。由于把握了全局，可快速认识自己未知的一些因素，防止在考察细节的时候陷入"死逻辑"中。

(27) 警惕"过早优化"。首先让它运行起来，再考虑变得更快--但只有在自己必须这样做、而且经证实在某部分代码中的确存在一个性能瓶颈的时候，才应进行优化。除非用专门的工具分析瓶颈，否则很有可能是在浪费自己的时间。性能提升的隐含代价是自己的代码变得难于理解，而且难于维护。

(28) 请记住，阅读代码的时间比写代码的时间多得多。思路清晰的设计可获得易于理解的程序，但注释、细致的解释以及一些示例往往具有不可估量的价值。无论对你自己，还是对后来的人，它们都是相当重要的。如对此仍有怀疑，那么请试想自己试图从联机Java文档里找出有用信息时碰到的挫折，这样或许能将你说服。

(29) 如认为自己已进行了良好的分析、设计或者实施，那么请稍微更换一下思维角度。试试邀请一些外来人士--并不一定是专家，但可以是来自本公司其他部门的人。请他们用完全新鲜的眼光考察你的工作，看看是否能找出你一度熟视无睹的问题。采取这种方式，往往能在最适合修改的阶段找出一些关键性的问题，避免产品发行后再解决问题而造成的金钱及精力方面的损失。

(30) 良好的设计能带来最大的回报。简言之，对于一个特定的问题，通常会花较长的时间才能找到一种最恰当的解决方案。但一旦找到了正确的方法，以后的工作就轻松多了，再也不用经历数小时、数天或者数月的痛苦挣扎。我们的努力工作会带来最大的回报（甚至无可估量）。而且由于自己倾注了大量心血，最终获得一个出色的设计方案，成功的快感也是令人心动的。坚持抵制草草完工的诱惑--那样做往往得不偿失

# 部分小知识

Given an ActionEvent, how to identify the affected component? 用getSource()方法，记住！