JAVA 培训教程—Java 简介

Java 分为三个体系：

- JavaSE（J2SE）(Java2 Platform Standard Edition，java 平台标准版）
- JavaEE(J2EE)(Java 2 Platform,Enterprise Edition，java 平台企业版)
- JavaME(J2ME)(Java 2 Platform Micro Edition，java 平台微型版)。

2005 年 6 月，JavaOne 大会召开，SUN 公司公开 Java SE 6。此时，Java 的各种版本已经更名以取消其中的数字"2"：J2EE 更名为 Java EE，J2SE 更名为 Java SE，J2ME 更名为 Java ME。

主要特性

- **Java 语言是简单的：**

  Java 语言的语法与 C 语言和 C++语言很接近，使得大多数程序员很容易学习和使用。另一方面，Java 丢弃了 C++中很少使用的、很难理解的、令人迷惑的那些特性，如操作符重载、多继承、自动的强制类型转换。特别地，Java 语言不使用指针，而是引用。并提供了自动的废料收集，使得程序员不必为内存管理而担忧。

- **Java 语言是面向对象的：**

  Java 语言提供类、接口和继承等原语，为了简单起见，只支持类之间的单继承，但支持接口之间的多继承，并支持类与接口之间的实现机制（关键字为 implements）。Java 语言全面支持动态绑定，而 C++语言只对虚函数使用动态绑定。总之，Java 语言是一个纯的面向对象程序设计语言。

- **Java 语言是分布式的：**

  Java 语言支持 Internet 应用的开发，在基本的 Java 应用编程接口中有一个网络应用编程接口（java net），它提供了用于网络应用编程的类库，包括 URL、URLConnection、Socket、ServerSocket 等。Java 的 RMI（远程方法激活）机制也是开发分布式应用的重要手段。

- **Java 语言是健壮的：**

  Java 的强类型机制、异常处理、垃圾的自动收集等是 Java 程序健壮性的重要保证。对指针的丢弃是 Java 的明智选择。Java 的安全检查机制使得 Java 更具健壮性。

- **Java 语言是安全的：**

  Java 通常被用在网络环境中，为此，Java 提供了一个安全机制以防恶意代码的攻击。除了 Java 语言具有的许多安全特性以外，Java 对通过网络下载的类具有一个安全防范机制（类 ClassLoader），如分配不同的名字空间以防替代本地的同名类、字节代码检查，并提供安全管理机制（类 SecurityManager）让 Java 应用设置安全哨兵。

- **Java 语言是体系结构中立的：**

  Java 程序（后缀为 java 的文件）在 Java 平台上被编译为体系结构中立的字节码格式（后缀为 class 的文件），然后可以在实现这个 Java 平台的任何系统中运行。这种途径适合于异构的网络环境和软件的分发。

- **Java 语言是可移植的：**

  这种可移植性来源于体系结构中立性，另外，Java 还严格规定了各个基本数据类型的长度。Java 系统本身也具有很强的可移植性，Java 编译器是用 Java 实现的，Java 的运行环境是用 ANSI C 实现的。

- **Java 语言是解释型的：**

  如前所述，Java 程序在 Java 平台上被编译为字节码格式，然后可以在实现这个 Java 平台的任何系统中运行。在运行时，Java 平台中的 Java 解释器对这些字节码进行解释执行，执行过程中需要的类在联接阶段被载入到运行环境中。

- **Java 是高性能的：**

与那些解释型的高级脚本语言相比，Java 的确是高性能的。事实上，Java 的运行速度随着 JIT(Just-In-Time）编译器技术的发展越来越接近于 C++。

- **Java 语言是多线程的：**

在 Java 语言中，线程是一种特殊的对象，它必须由 Thread 类或其子（孙）类来创建。通常有两种方法来创建线程：其一，使用型构为 Thread(Runnable)的构造子将一个实现了 Runnable 接口的对象包装成一个线程，其二，从 Thread 类派生出子类并重写 run 方法，使用该子类创建的对象即为线程。值得注意的是 Thread 类已经实现了 Runnable 接口，因此，任何一个线程均有它的 run 方法，而 run 方法中包含了线程所要运行的代码。线程的活动由一组方法来控制。Java 语言支持多个线程的同时执行，并提供多线程之间的同步机制（关键字为 synchronized）。

- **Java 语言是动态的：**

Java 语言的设计目标之一是适应于动态变化的环境。Java 程序需要的类能够动态地被载入到运行环境，也可以通过网络来载入所需要的类。这也有利于软件的升级。另外，Java 中的类有一个运行时刻的表示，能进行运行时刻的类型检查。

发展历史

- 1995 年 5 月 23 日，Java 语言诞生
- 1996 年 1 月，第一个 JDK-JDK1.0 诞生
- 1996 年 4 月，10 个最主要的操作系统供应商申明将在其产品中嵌入 JAVA 技术
- 1996 年 9 月，约 8.3 万个网页应用了 JAVA 技术来制作
- 1997 年 2 月 18 日，JDK1.1 发布
- 1997 年 4 月 2 日，JavaOne 会议召开，参与者逾一万人，创当时全球同类会议规模之纪录
- 1997 年 9 月，JavaDeveloperConnection 社区成员超过十万
- 1998 年 2 月，JDK1.1 被下载超过 2,000,000 次
- 1998 年 12 月 8 日，JAVA2 企业平台 J2EE 发布

- 1999 年 6 月，SUN 公司发布 Java 的三个版本：标准版（JavaSE,以前是 J2SE）、企业版（JavaEE 以前是 J2EE）和微型版（JavaME，以前是 J2ME）
- 2000 年 5 月 8 日，JDK1.3 发布
- 2000 年 5 月 29 日，JDK1.4 发布
- 2001 年 6 月 5 日，NOKIA 宣布，到 2003 年将出售 1 亿部支持 Java 的手机
- 2001 年 9 月 24 日，J2EE1.3 发布
- 2002 年 2 月 26 日，J2SE1.4 发布，自此 Java 的计算能力有了大幅提升
- 2004 年 9 月 30 日 18:00PM，J2SE1.5 发布，成为 Java 语言发展史上的又一里程碑。为了表示该版本的重要性，J2SE1.5 更名为 Java SE 5.0
- 2005 年 6 月，JavaOne 大会召开，SUN 公司公开 Java SE 6。此时，Java 的各种版本已经更名，以取消其中的数字"2"：J2EE 更名为 Java EE，J2SE 更名为 Java SE，J2ME 更名为 Java ME
- 2006 年 12 月，SUN 公司发布 JRE6.0
- 2009 年 04 月 20 日，甲骨文 74 亿美元收购 Sun。取得 java 的版权。
- 2010 年 11 月,由于甲骨文对于 Java 社区的不友善,因此 Apache 扬言将退出 JCP[4]。
- 2011 年 7 月 28 日，甲骨文发布 java7.0 的正式版。

Java 开发工具

Java 语言尽量保证系统内存在 1G 以上，其他工具如下所示：

- Linux 系统或者 Windows 95/98/2000/XP，WIN 7/8 系统
- Java JDK 7
- Notepad 编辑器或者其他编辑器。
- IDE：Eclipse

安装好以上的工具后，我们就可以输出 Java 的第一个程序"Hello World!"

```
1   public  class  MyFirstJavaProgram {
2
3           public  static  void  main(String []args) {
```

```
4                   System.out.println("Hello World");
5               }
6   }
```

JAVA 培训教程—Java 开发环境配置

首先我们需要下载 java 开发工具包 JDK，下载后 JDK 的安装根据提示进行，还有安装 JDK 的时候也会安装 JRE，一并安装就可以了。

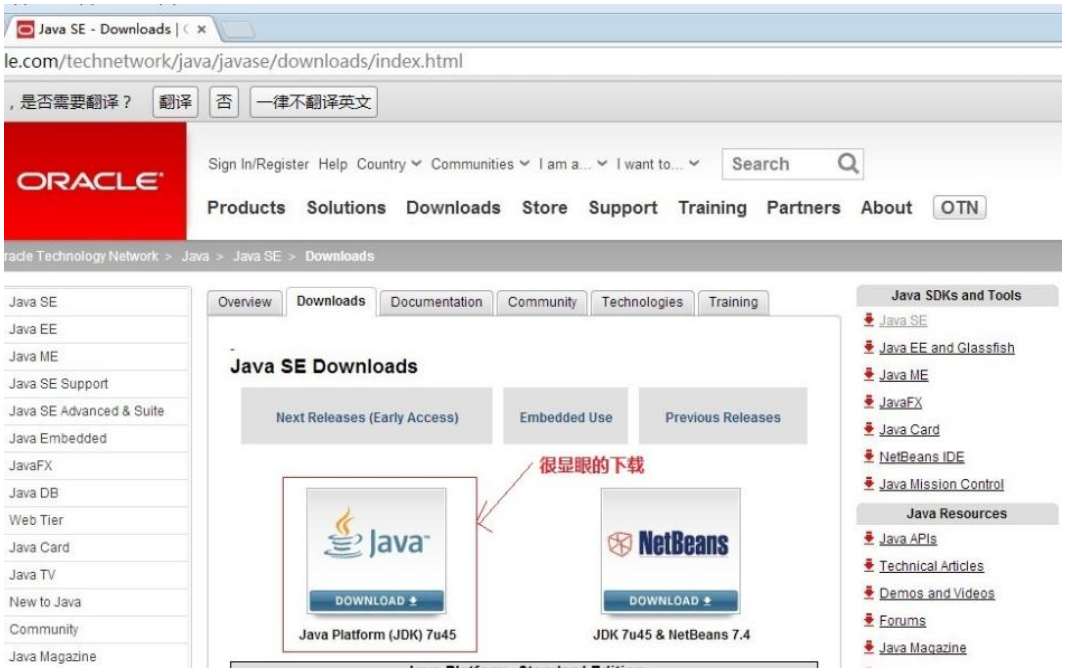
安装 JDK，安装过程中可以自定义安装目录等信息，例如我们选择安装目录为 C:\Program Files\Java\jdk1.7.0 。

配置环境变量

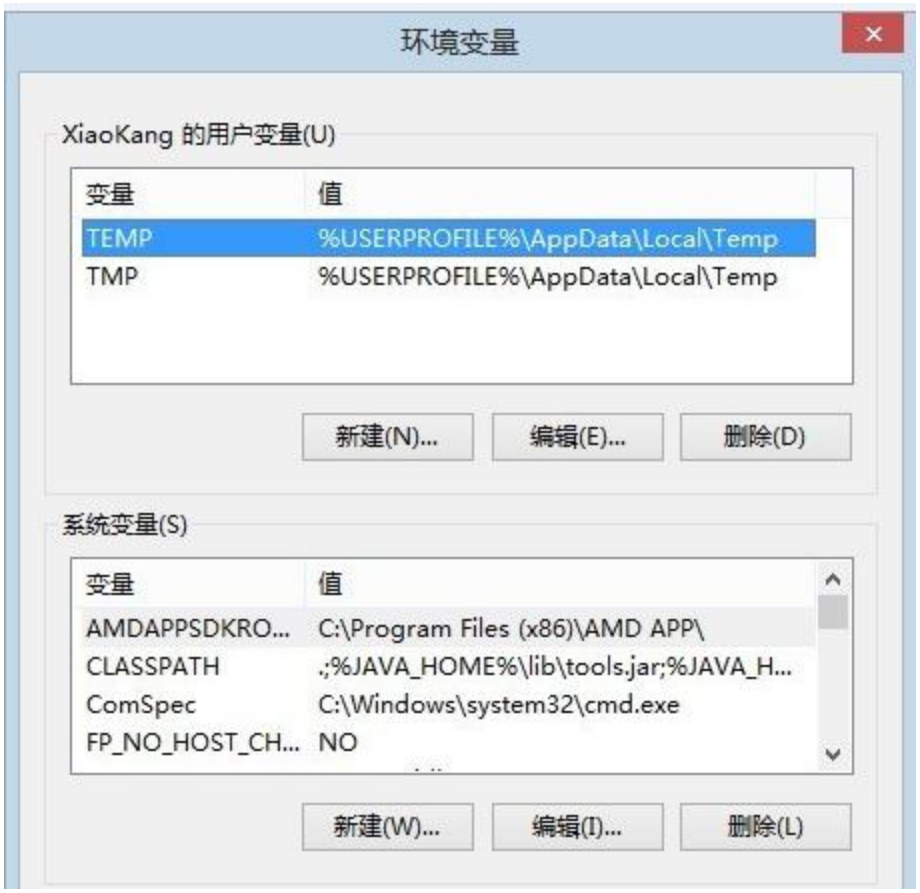
1.安装完成后，右击"我的电脑"，点击"属性"；

2.选择"高级"选项卡，点击"环境变量"；

然后就会出现如下图所示的画面



在"系统变量"中设置 3 项属性，JAVA_HOME,PATH,CLASSPATH(大小写无所谓),若已存在则点击"编辑"，不存在则点击"新建"。



变量设置

- 变量名：JAVA_HOME
- 变量值：C:\Program Files\Java\jdk1.7.0

//这里是你 JDK 的安装路径，可以更换

- 变量名：CLASSPATH
- 变量值：.;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar; //记得前面有个"."

- 变量名：Path
- 变量值：%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin;

这是 java 的环境配置，配置完成后直接启动 eclipse，它会自动完成 java 环境的配置。

测试 JDK 是否安装成功

1、"开始"->;"运行"，键入"cmd"；

2、键入命令"java -version"，"java"，"javac"几个命令，出现画面，说明环境变量配置成功；

JAVA 培训教程—Java 基础语法

简要介绍下类、对象、方法和实例变量的概念。

对象：对象是类的一个实例，有状态和行为。例如，一条狗是一个对象，它的状态有：颜色、名字、品种；行为有：摇尾巴、叫、吃等。

类：类是一个模板，它描述一类对象的行为和状态。

方法：方法就是行为，一个类可以有很多方法。逻辑运算、数据修改以及所有动作都是在方法中完成的。

实例变量：每个对象都有独特的实例变量，对象的状态由这些实例变量的值决定。

第一个 Java 程序

下面看一个简单的 Java 程序，它将打印字符串 *Hello World*

```
public class MyFirstJavaProgram {
        /* 第一个 Java 程序.
         * 它将打印字符串 Hello World
         */
        public static void main(String[] args) {
                System.out.println("Hello World"); // 打印 Hello World
        }
}
```

下面将逐步介绍如何保存、编译以及运行这个程序：

- 打开 Notepad，把上面的代码添加进去；
- 把文件名保存为：MyFirstJavaProgram.java；

- 打开 cmd 命令窗口，进入目标文件所在的位置，假设是 C:\
- 在命令行窗口键入 javac MyFirstJavaProgram.java 按下 enter 键编译代码。如果代码没有错误，cmd 命令提示符会进入下一行。（假设环境变量都设置好了）。
- 再键入 java MyFirstJavaProgram 按下 Enter 键就可以运行程序了

你将会在窗口看到 Hello World

```
C : > javac MyFirstJavaProgram.java

C : > java MyFirstJavaProgram

Hello World
```

基本语法

编写 Java 程序时，应注意以下几点：

- **大小写敏感**：Java 是大小写敏感的，这就意味着标识符 Hello 与 hello 是不同的。
- **类名**：对于所有的类来说，类名的首字母应该大写。如果类名由若干单词组成，那么每个单词的首字母应该大写，例如 MyFirstJavaClass 。
- **方法名**：所有的方法名都应该以小写字母开头。如果方法名含有若干单词，则后面的每个单词首字母大写。
- **源文件名**：源文件名必须和类名相同。当保存文件的时候，你应该使用类名作为文件名保存（切记 Java 是大小写敏感的），文件名的后缀为.java。（如果文件名和类名不相同则会导致编译错误）。
- **主方法入口**：所有的 Java 程序由 **public static void main(String args[])** 方法开始执行。

Java 标识符

Java 所有的组成部分都需要名字。类名、变量名以及方法名都被称为标识符。

关于 Java 标识符，有以下几点需要注意：

- 所有的标识符都应该以字母（A-Z 或者 a-z），美元符（$）、或者下划线（_）开始
- 首字符之后可以是任何字符的组合
- 关键字不能用作标识符
- 标识符是大小写敏感的
- 合法标识符举例：age、$salary、_value、__1_value
- 非法标识符举例：123abc、-salary

Java 修饰符

像其他语言一样，Java 可以使用修饰符来修饰类中方法和属性。主要有两类修饰符：

- 可访问修饰符 ： default, public , protected, private
- 不可访问修饰符 ： final, abstract, strictfp

在后面的章节中我们会深入讨论 Java 修饰符。

Java 变量

Java 中主要有如下几种类型的变量

- 局部变量
- 类变量（静态变量）
- 成员变量（非静态变量）

Java 数组

数组是储存在堆上的对象，可以保存多个同类型变量。在后面的章节中，我们将会学到如何声明、构造以及初始化一个数组。

Java 枚举

Java 5.0 引入了枚举，枚举限制变量只能是预先设定好的值。使用枚举可以减少代码中的 bug。

例如，我们为果汁店设计一个程序，它将限制果汁为小杯、中杯、大杯。这就意味着它不允许顾客点除了这三种尺寸外的果汁。

实例

```
1    class  FreshJuice  {
2          enum  FreshJuiceSize{  SMALL,  MEDUIM,  LARGE  }
3          FreshJuiceSize  size;
4    }
5
6    public  class  FreshJuiceTest  {
7          public  static  void  main(String[]  args){
8                FreshJuice  juice  =  new  FreshJuice();
9                juice.size  =  FreshJuice.  FreshJuiceSize.MEDUIM  ;
10         }
11   }
```

注意：枚举可以单独声明或者声明在类里面。方法、变量、构造函数也可以在枚举中定义。

Java 关键字

下面列出了 Java 保留字。这些保留字不能用于常量、变量、和任何标识符的名称。

关键字	描述
abstract	抽象方法，抽象类的修饰符
assert	断言条件是否满足
boolean	布尔数据类型
break	跳出循环或者 label 代码段
byte	8-bit 有符号数据类型
case	switch 语句的一个条件
catch	和 try 搭配扑捉异常信息
char	16-bit Unicode 字符数据类型
class	定义类

const	未使用
continue	不执行循环体剩余部分
default	switch 语句中的默认分支
do	循环语句，循环体至少会执行一次
double	64-bit 双精度浮点数
else	if 条件不成立时执行的分支
enum	枚举类型
extends	表示一个类是另一个类的子类
final	表示一个值在初始化之后就不能再改变了 表示方法不能被重写，或者一个类不能有子类
finally	为了完成执行的代码而设计的，主要是为了程序的健壮性和完整性，无论有没有异常发生都执行代码。
float	32-bit 单精度浮点数

for	for 循环语句
goto	未使用
if	条件语句
implements	表示一个类实现了接口
import	导入类
instanceof	测试一个对象是否是某个类的实例
int	32 位整型数
interface	接口，一种抽象的类型，仅有方法和常量的定义
long	64 位整型数
native	表示方法用非 java 代码实现
new	分配新的类实例
package	一系列相关类组成一个包

private	表示私有字段，或者方法等，只能从类内部访问
protected	表示字段只能通过类或者其子类访问 子类或者在同一个包内的其他类
public	表示共有属性或者方法
return	方法返回值
short	16 位数字
static	表示在类级别定义，所有实例共享的
strictfp	浮点数比较使用严格的规则
super	表示基类
switch	选择语句
synchronized	表示同一时间只能由一个线程访问的代码块
this	表示调用当前实例 或者调用另一个构造函数

throw	抛出异常
throws	定义方法可能抛出的异常
transient	修饰不要序列化的字段
try	表示代码块要做异常处理或者和 finally 配合表示是否抛出异常都执行 finally 中的代码
void	标记方法不返回任何值
volatile	标记字段可能会被多个线程同时访问，而不做同步
while	while 循环

Java 注释

类似于 C/C++，Java 也支持单行以及多行注释。注释中的字符将被 Java 编译器忽略。

```
1    public  class  MyFirstJavaProgram{
2          /*  这是第一个 Java 程序
3            *它将打印 Hello  World
4            *  这是一个多行注释的示例
5            */
6            public  static  void  main(String[]  args){
```

```
7                   //  这是单行注释的示例
8                   /*  这个也是单行注释的示例  */
9                   System.out.println("Hello  World");
10              }
11      }
```

Java 空行

空白行，或者有注释的的行，Java 编译器都会忽略掉。

继承

在 Java 中，一个类可以由其他类派生。如果你要创建一个类，而且已经存在一个类具有你所需要的属性或方法，那么你可以将新创建的类继承该类。

利用继承的方法，可以重用已存在类的方法和属性，而不用重写这些代码。被继承的类称为超类（super class），派生类称为子类（subclass）。

接口

在 Java 中，接口可理解为对象间相互通信的协议。接口在继承中扮演着很重要的角色。

接口只定义派生要用到的方法，但是方法的具体实现完全取决于派生类。

JAVA 培训教程—Java 文档注释

Java 只是三种注释方式。前两种分别是// 和/**/，第三种被称作说明注释，它以/**开始，以 */结束。

说明注释允许你在程序中嵌入关于程序的信息。你可以使用 javadoc 工具软件来生成信息，并输出到 HTML 文件中。

说明注释，是你更加方面的记录你的程序的信息。

javadoc 标签

javadoc 工具软件识别以下标签：

标签	描述	示例
@author	标识一个类的作者	@author description
@deprecated	指名一个过期的类或成员	@deprecated description
{@docRoot}	指明当前文档根目录的路径	Directory Path
@exception	标志一个类抛出的异常	@exception exception-name explanation
{@inheritDoc}	从直接父类继承的注释	Inherits a comment from the immediate superclass.

{@link}	插入一个到另一个主题的链接	{@link name text}
{@linkplain}	插入一个到另一个主题的链接，但是该链接显示纯文本字体	Inserts an in-line link to another topic.
@param	说明一个方法的参数	@param parameter-name explanation
@return	说明返回值类型	@return explanation
@see	指定一个到另一个主题的链接	@see anchor
@serial	说明一个序列化属性	@serial description
@serialData	说明通过 writeObject( ) 和 writeExternal( )方法写的数据	@serialData description
@serialField	说明一个ObjectStreamField组件	@serialField name type description
@since	标记当引入一个特定的变化时	@since release
@throws	和 @exception 标签一样.	The @throws tag has the same meaning as the @exception tag.

{@value}	显示常量的值，该常量必须是 static 属性。	Displays the value of a constant, which must be a static field.
@version	指定类的版本	@version info

文档注释

在开始的/**之后，第一行或几行是关于类、变量和方法的主要描述.

之后，你可以包含一个或多个何种各样的@标签。每一个@标签必须在一个新行的开始或者在一行的开始紧跟星号(*).

多个相同类型的标签应该放成一组。例如，如果你有三个@see 标签，可以将它们一个接一个的放在一起。

下面是一个类的说明注释的示例：

```
1    /*** This class draws a bar chart.
2    * @author Zara Ali
3    * @version 1.2
4    */
```

javadoc 输出什么

javadoc 工具将你 Java 程序的源代码作为输入,输出一些包含你程序注释的 HTML 文件。

每一个类的信息将在独自的 HTML 文件里。javadoc 也可以输出继承的树形结构和索引。

由于 javadoc 的实现不同，工作也可能不同，你需要检查你的 Java 开发系统的版本等细节，选择合适的 Javadoc 版本。

实例

下面是一个使用说明注释的简单实例。注意每一个注释都在它描述的项目的前面。

在经过 javadoc 处理之后，SquareNum 类的注释将在 SquareNum.html 中找到。

```
1    import  java.io.*;
2
3    /**
4    * This class demonstrates documentation comments.
5    * @author Ayan Amhed
6    * @version 1.2
7    */
8    public  class  SquareNum {
9          /**
10         * This method returns the square of num.
11         * This is a multiline description. You can use
12         * as many lines as you like.
13         * @param num The value to be squared.
14         * @return num squared.
15         */
16         public  double  square(double  num) {
17               return  num * num;
18         }
19         /**
20         * This method inputs a number from the user.
21         * @return The value input as a double.
22         * @exception IOException On input error.
23         * @see IOException
24         */
25         public  double  getNumber()  throws  IOException {
26               InputStreamReader isr =  new  InputStreamReader(System.in);
```

```
27              BufferedReader inData =  new  BufferedReader(isr);
28              String str;
29              str = inData.readLine();
30              return  (new  Double(str)).doubleValue();
31          }
32          /**
33          * This method demonstrates square().
34          * @param args Unused.
35          * @return Nothing.
36          * @exception IOException On input error.
37          * @see IOException
38          */
39          public  static  void  main(String args[])  throws  IOException
40          {
41              SquareNum ob =  new  SquareNum();
42              double  val;
43              System.out.println("Enter value to be squared: ");
44              val = ob.getNumber();
45              val = ob.square(val);
46              System.out.println("Squared value is "  + val);
47          }
48      }
```

如下，使用 javadoc 工具处理 SquareNum.java 文件：

```
1     $ javadoc SquareNum.java
2     Loading source file SquareNum.java...
3     Constructing Javadoc information...
4     Standard Doclet version  1.5.0_13
5     Building tree  for  all the packages and classes...
```

```
6    Generating SquareNum.html...
7    SquareNum.java:39: warning -  @return  tag cannot be used\
8                                                  in method with  void  return  type.
9    Generating  package-frame.html...
10   Generating  package-summary.html...
11   Generating  package-tree.html...
12   Generating constant-values.html...
13   Building index  for  all the packages and classes...
14   Generating overview-tree.html...
15   Generating index-all.html...
16   Generating deprecated-list.html...
17   Building index  for  all classes...
18   Generating allclasses-frame.html...
19   Generating allclasses-noframe.html...
20   Generating index.html...
21   Generating help-doc.html...
22   Generating stylesheet.css...
23   1  warning
24   $
```

JAVA 培训教程—Java 变量类型

在 Java 语言中，所有的变量在使用前必须声明。声明变量的基本格式如下：

```
type identifier [ = value][, identifier [= value] ...] ;
```

格式说明：type 为 Java 数据类型。identifier 是变量名。可以使用逗号隔开来声明多个同类型变量。

以下列出了一些变量的声明实例。注意有些包含了初始化过程。

```
int  a, b, c;                   // 声明三个 int 型整数：a、 b、c。
int  d = 3, e, f = 5;  // d 声明三个整数并赋予初值。
byte  z = 22;                    // 声明并初始化 z。
double  pi = 3.14159;  // 声明了 pi。
char  x = 'x';                  // 变量 x 的值是字符'x'。
```

Java 语言支持的变量类型有：

- 局部变量
- 成员变量
- 类变量

Java 局部变量

- 局部变量声明在方法、构造方法或者语句块中；
- 局部变量在方法、构造方法、或者语句块被执行的时候创建，当它们执行完成后，变量将会被销毁；
- 访问修饰符不能用于局部变量；
- 局部变量只在声明它的方法、构造方法或者语句块中可见；
- 局部变量是在栈上分配的。
- 局部变量没有默认值，所以局部变量量被声明后，必须经过初始化，才可以使用。

实例 1

在以下实例中 age 是一个局部变量。定义在 pubAge()方法中，它的作用域就限制在这个方法中。

```
1    public  class  Test{
2          public  void  pupAge(){
3                int  age =  0;
4                age = age +  7;
5                System.out.println("Puppy age is : "  + age);
6          }
7
8          public  static  void  main(String args[]){
9                Test test =  new  Test();
10               test.pupAge();
11         }
12   }
```

以上实例编译运行结果如下：

```
1   Puppy age is:  7
```

实例 2

在下面的例子中 age 变量没有初始化，所以在编译时出错。

```
1    public  class  Test{
2          public  void  pupAge(){
3                int  age;
4                age = age +  7;
5                System.out.println("Puppy age is : "  + age);
6          }
```

```
7
8           public  static  void  main(String args[]){
9                   Test test =  new  Test();
10                  test.pupAge();
11          }
12  }
```

以上实例编译运行结果如下:

```
1   Test.java:4:variable number might not have been initialized
2   age = age +  7;
3                       ^
4   1  error
```

实例变量

- 实例变量声明在一个类中，但在方法、构造方法和语句块之外；
- 当一个对象被实例化之后，每个实例变量的值就跟着确定；
- 实例变量在对象创建的时候创建，在对象被销毁的时候销毁；
- 实例变量的值应该至少被一个方法、构造方法或者语句块引用，使得外部能够通过这些方式获取实例变量信息；
- 实例变量可以声明在使用前或者使用后；
- 访问修饰符可以修饰实例变量；
- 实例变量对于类中的方法、构造方法或者语句块是可见的。一般情况下应该把实例变量设为私有。通过使用访问修饰符可以使实例变量对子类可见；
- 实例变量具有默认值。数值型变量的默认值是 0，布尔型变量的默认值是 false，引用类型变量的默认值是 null。变量的值可以在声明时指定，也可以在构造方法中指定；

- 实例变量可以直接通过变量名访问。但在静态方法以及其他类中，就应该使用完全限定名：ObejectReference.VariableName。

实例：

```
1    import  java.io.*;
2    public  class  Employee{
3          // 这个成员变量对子类可见
4          public  String name;
5          // 私有变量，仅在该类可见
6          private  double  salary;
7          //在构造器中对 name 赋值
8          public  Employee (String empName){
9                name = empName;
10         }
11         //设定 salary 的值
12         public  void  setSalary(double  empSal){
13               salary = empSal;
14         }
15         // 打印信息
16         public  void  printEmp(){
17               System.out.println("name   : "  + name );
18               System.out.println("salary :"  + salary);
19         }
20
21         public  static  void  main(String args[]){
22               Employee empOne =  new  Employee("Ransika");
23               empOne.setSalary(1000);
24               empOne.printEmp();
25         }
```

```
26    }
```

以上实例编译运行结果如下:

```
1    name    : Ransika
2    salary :1000.0
```

类变量（静态变量）

- 类变量也称为静态变量，在类中以 static 关键字声明，但必须在方法构造方法和语句块之外。
- 无论一个类创建了多少个对象，类只拥有类变量的一份拷贝。
- 静态变量除了被声明为常量外很少使用。常量是指声明为 publc/private，final 和 static 类型的变量。常量初始化后不可改变。
- 静态变量储存在静态存储区。经常被声明为常量，很少单独使用 static 声明变量。
- 静态变量在程序开始时创建，在程序结束时销毁。
- 与实例变量具有相似的可见性。但为了对类的使用者可见，大多数静态变量声明为 public 类型。
- 默认值和实例变量相似。数值型变量默认值是 0，布尔型默认值是 false，引用类型默认值是 null。变量的值可以在声明的时候指定，也可以在构造方法中指定。此外，静态变量还可以在静态语句块中初始化。
- 静态变量可以通过：*ClassName.VariableName* 的方式访问。
- 类变量被声明为 public static final 类型时，类变量名称必须使用大写字母。如果静态变量不是 public 和 final 类型，其命名方式与实例变量以及局部变量的命名方式一致。

实例：

```
1    import  java.io.*;
2    public  class  Employee{
```

```
3           //salary 是静态的私有变量
4           private  static  double  salary;
5           // DEPARTMENT 是一个常量
6           public  static  final  String DEPARTMENT =  "Development ";
7           public  static  void  main(String args[]){
8                 salary =  1000;
9                 System.out.println(DEPARTMENT+"average salary:"+salary);
10          }
11    }
```

以上实例编译运行结果如下:

```
1    Development average salary:1000
```

JAVA 培训教程—Java 基础数据类型

Java 的两大数据类型:

- 内置数据类型
- 引用数据类型

内置数据类型

Java 语言提供了八种基本类型。六种数字类型（四个整数型，两个浮点型），一种字符类型，还有一种布尔型。

byte：

- byte 数据类型是 8 位、有符号的，以二进制补码表示的整数；
- 最小值是-128（-2^7）；
- 最大值是 127（2^7-1）；
- 默认值是 0；
- byte 类型用在大型数组中节约空间，主要代替整数，因为 byte 变量占用的空间只有 int 类型的四分之一；
- 例子：byte a = 100，byte b = -50。

short：

- short 数据类型是 16 位、有符号的以二进制补码表示的整数
- 最小值是-32768（-2^15）；
- 最大值是 32767（2^15 - 1）；
- Short 数据类型也可以像 byte 那样节省空间。一个 short 变量是 int 型变量所占空间的二分之一；
- 默认值是 0；
- 例子：short s = 1000，short r = -20000。

int：

- int 数据类型是 32 位、有符号的以二进制补码表示的整数；
- 最小值是-2,147,483,648（-2^31）；
- 最大值是 2,147,485,647（2^31 - 1）；
- 一般地整型变量默认为 int 类型；
- 默认值是 0；
- 例子：int a = 100000, int b = -200000。

long：

- long 数据类型是 64 位、有符号的以二进制补码表示的整数；
- 最小值是-9,223,372,036,854,775,808（-2^63）；
- 最大值是 9,223,372,036,854,775,807（2^63 -1）；
- 这种类型主要使用在需要比较大整数的系统上；
- 默认值是 0L；
- 例子： long a = 100000L，int b = -200000L。

float：

- float 数据类型是单精度、32 位、符合 IEEE 754 标准的浮点数；
- float 在储存大型浮点数组的时候可节省内存空间；
- 默认值是 0.0f；
- 浮点数不能用来表示精确的值，如货币；
- 例子：float f1 = 234.5f。

double：

- double 数据类型是双精度、64 位、符合 IEEE 754 标准的浮点数；
- 浮点数的默认类型为 double 类型；
- double 类型同样不能表示精确的值，如货币；
- 默认值是 0.0f；
- 例子：double d1 = 123.4。

boolean：

- boolean 数据类型表示一位的信息；
- 只有两个取值：true 和 false；
- 这种类型只作为一种标志来记录 true/false 情况；
- 默认值是 false；
- 例子：boolean one = true。

char：

- char 类型是一个单一的 16 位 Unicode 字符；
- 最小值是’\u0000’（即为 0）；
- 最大值是’\uffff’（即为 65,535）；
- char 数据类型可以储存任何字符；
- 例子：char letter = ‘A’。

实例

对于数值类型的基本类型的取值范围，我们无需强制去记忆，因为它们的值都已经以常量的形式定义在对应的包装类中了。请看下面的例子：

```
1    public  class  PrimitiveTypeTest {
2            public  static  void  main(String[] args) {
3                    // byte
4                    System.out.println("基本类型：byte 二进制位数：" + Byte.SIZE);
5                    System.out.println("包装类：java.lang.Byte");
6                    System.out.println("最小值：Byte.MIN_VALUE=" + Byte.MIN_VALUE);
7                    System.out.println("最大值：Byte.MAX_VALUE=" + Byte.MAX_VALUE);
8                    System.out.println();
9
10                   // short
11                   System.out.println("基本类型：short 二进制位数：" + Short.SIZE);
```

```
12                  System.out.println("包装类：java.lang.Short");
13                  System.out.println("最小值：Short.MIN_VALUE=" + Short.MIN_VALUE);
14                  System.out.println("最大值：Short.MAX_VALUE=" + Short.MAX_VALUE);
15                  System.out.println();
16
17                  // int
18                  System.out.println("基本类型：int 二进制位数：" + Integer.SIZE);
19                  System.out.println("包装类：java.lang.Integer");
20                  System.out.println("最小值：Integer.MIN_VALUE=" + Integer.MIN_VALUE);
21                  System.out.println("最大值：Integer.MAX_VALUE=" + Integer.MAX_VALUE);
22                  System.out.println();
23
24                  // long
25                  System.out.println("基本类型：long 二进制位数：" + Long.SIZE);
26                  System.out.println("包装类：java.lang.Long");
27                  System.out.println("最小值：Long.MIN_VALUE=" + Long.MIN_VALUE);
28                  System.out.println("最大值：Long.MAX_VALUE=" + Long.MAX_VALUE);
29                  System.out.println();
30
31                  // float
32                  System.out.println("基本类型：float 二进制位数：" + Float.SIZE);
33                  System.out.println("包装类：java.lang.Float");
34                  System.out.println("最小值：Float.MIN_VALUE=" + Float.MIN_VALUE);
35                  System.out.println("最大值：Float.MAX_VALUE=" + Float.MAX_VALUE);
36                  System.out.println();
37
38                  // double
39                  System.out.println("基本类型：double 二进制位数：" + Double.SIZE);
40                  System.out.println("包装类：java.lang.Double");
```

```
41                  System.out.println("最小值：Double.MIN_VALUE="  + Double.MIN_VALUE);
42                  System.out.println("最大值：Double.MAX_VALUE="  + Double.MAX_VALUE);
43                  System.out.println();
44
45                  // char
46                  System.out.println("基本类型：char 二进制位数："  + Character.SIZE);
47                  System.out.println("包装类：java.lang.Character");
48                  // 以数值形式而不是字符形式将 Character.MIN_VALUE 输出到控制台
49                  System.out.println("最小值：Character.MIN_VALUE="
50                                  + (int) Character.MIN_VALUE);
51                  // 以数值形式而不是字符形式将 Character.MAX_VALUE 输出到控制台
52                  System.out.println("最大值：Character.MAX_VALUE="
53                                  + (int) Character.MAX_VALUE);
54          }
55  }
```

编译以上代码输出结果如下所示：

```
1   基本类型：byte  二进制位数：8
2   包装类：java.lang.Byte
3   最小值：Byte.MIN_VALUE=-128
4   最大值：Byte.MAX_VALUE=127
5
6   基本类型：short  二进制位数：16
7   包装类：java.lang.Short
8   最小值：Short.MIN_VALUE=-32768
9   最大值：Short.MAX_VALUE=32767
10
11  基本类型：int  二进制位数：32
12  包装类：java.lang.Integer
```

```
13    最小值：Integer.MIN_VALUE=-2147483648
14    最大值：Integer.MAX_VALUE=2147483647
15
16    基本类型：long  二进制位数：64
17    包装类：java.lang.Long
18    最小值：Long.MIN_VALUE=-9223372036854775808
19    最大值：Long.MAX_VALUE=9223372036854775807
20
21    基本类型：float  二进制位数：32
22    包装类：java.lang.Float
23    最小值：Float.MIN_VALUE=1.4E-45
24    最大值：Float.MAX_VALUE=3.4028235E38
25
26    基本类型：double  二进制位数：64
27    包装类：java.lang.Double
28    最小值：Double.MIN_VALUE=4.9E-324
29    最大值：Double.MAX_VALUE=1.7976931348623157E308
30
31    基本类型：char  二进制位数：16
32    包装类：java.lang.Character
33    最小值：Character.MIN_VALUE=0
34    最大值：Character.MAX_VALUE=65535
```

Float 和 Double 的最小值和最大值都是以科学记数法的形式输出的，结尾的"E+数字"表示 E 之前的数字要乘以 10 的多少倍。比如 3.14E3 就是 3.14×1000=3140，3.14E-3 就是 3.14/1000=0.00314。

实际上，JAVA 中还存在另外一种基本类型 void，它也有对应的包装类 java.lang.Void，不过我们无法直接对它们进行操作。

引用类型

- 引用类型变量由类的构造函数创建，可以使用它们访问所引用的对象。这些变量在声明时被指定为一个特定的类型，比如 Employee、Pubby 等。变量一旦声明后，类型就不能被改变了。
- 对象、数组都是引用数据类型。
- 所有引用类型的默认值都是 null。
- 一个引用变量可以用来引用与任何与之兼容的类型。
- 例子：Animal animal = new Animal(“giraffe”)。

Java 常量

常量就是一个固定值。它们不需要计算，直接代表相应的值。

常量指不能改变的量。 在 Java 中用 final 标志，声明方式和变量类似：

```
1    final  double  PI =  3.1415927;
```

虽然常量名也可以用小写，但为了便于识别，通常使用大写字母表示常量。

字面量可以赋给任何内置类型的变量。例如：

```
1    byte  a =  68;
2    char  a =  'A'
```

byte、int、long、和 short 都可以用十进制、16 进制以及 8 进制的方式来表示。

当使用常量的时候，前缀 o 表明是 8 进制，而前缀 0x 代表 16 进制。例如：

```
1    int  decimal =  100;
2    int  octal =  0144;
3    int  hexa =    0x64;
```

和其他语言一样，Java 的字符串常量也是包含在两个引号之间的字符序列。下面是字符串型字面量的例子：

```
1    "Hello World"
2    "two\nlines"
3    "\"This is in quotes\""
```

字符串常量和字符常量都可以包含任何 Unicode 字符。例如：

```
1    char  a =  '\u0001';
2    String a =  "\u0001";
```

Java 语言支持一些特殊的转义字符序列。

符号	字符含义
\n	换行 (0x0a)
\r	回车 (0x0d)
\f	换页符(0x0c)
\b	退格 (0x08)
\s	空格 (0x20)
\t	制表符

\″	双引号
\’	单引号
\\	反斜杠
\ddd	八进制字符 (ddd)
\uxxxx	16 进制 Unicode 字符 (xxxx)

JAVA 培训教程—Java 运算符

计算机的最基本用途之一就是执行数学运算，作为一门计算机语言，Java 也提供了一套丰富的运算符来操纵变量。我们可以把运算符分成以下几组：

- 算术运算符
- 关系运算符
- 位运算符
- 逻辑运算符
- 赋值运算符
- 其他运算符

算术运算符

算术运算符用在数学表达式中，它们的作用和在数学中的作用一样。下表列出了所有的算术运算符。

表格中的实例假设整数变量 A 的值为 10，变量 B 的值为 20：

操作符	描述	例子
+	加法 - 相加运算符两侧的值	A + B 等于 30
-	减法 - 左操作数减去右操作数	A - B 等于-10
*	乘法 - 相乘操作符两侧的值	A * B 等于 200
/	除法 - 左操作数除以右操作数	B / A 等于 2

%	取模 - 右操作数除左操作数的余数	B%A 等于 0
+ +	自增 - 操作数的值增加 1	B + +等于 21
-	自减 - 操作数的值减少 1	B - -等于 19

实例

下面的简单示例程序演示了算术运算符。复制并粘贴下面的 Java 程序并保存为 Test.java 文件，然后编译并运行这个程序：

```
1    public class Test {
2
3        public static void main(String args[]) {
4            int a = 10;
5            int b = 20;
6            int c = 25;
7            int d = 25;
8            System.out.println("a + b = " + (a + b) );
9            System.out.println("a - b = " + (a - b) );
10           System.out.println("a * b = " + (a * b) );
11           System.out.println("b / a = " + (b / a) );
12           System.out.println("b % a = " + (b % a) );
13           System.out.println("c % a = " + (c % a) );
14           System.out.println("a++ = " + (a++) );
15           System.out.println("b-- = " + (a--) );
16           // Check the difference in d++ and ++d
17           System.out.println("d++ = " + (d++) );
```

```
18              System.out.println("++d      =  "  +    (++d)  );
19          }
20      }
```

以上实例编译运行结果如下：

```
1       a  +  b  =  30
2       a  -  b  =  -10
3       a  *  b  =  200
4       b  /  a  =  2
5       b  %  a  =  0
6       c  %  a  =  5
7       a++      =  10
8       b--      =  11
9       d++      =  25
10      ++d      =  27
```

关系运算符

下表为 Java 支持的关系运算符

表格中的 **例子** 整数变量 A=10，变量 B=20：

运算符	**描述**	**例子**
==	检查如果两个操作数的值是否相等，如果相等则条件为真。	（A == B）为假(非真)
!=	检查如果两个操作数的值是否相等，如果值不相等则条件为真。	（A != B）为真

>	检查左操作数的值是否大于右操作数的值，如果是那么条件为真。	（A > B）非真
<	检查左操作数的值是否小于右操作数的值，如果是那么条件为真。	（A < B）为真
>=	检查左操作数的值是否大于或等于右操作数的值，如果是那么条件为真。	（A >= B）为假
<=	检查左操作数的值是否小于或等于右操作数的值，如果是那么条件为真。	（A <= B）为真

实例

下面的简单示例程序演示了关系运算符。复制并粘贴下面的 Java 程序并保存为 Test.java 文件，然后编译并运行这个程序：

```
1    public  class  Test  {
2
3        public  static  void  main(String  args[])  {
4              int  a  =  10;
5              int  b  =  20;
6              System.out.println("a  ==  b  =  "  +  (a  ==  b)  );
7              System.out.println("a  !=  b  =  "  +  (a  !=  b)  );
8              System.out.println("a  >  b  =  "  +  (a  >  b)  );
9              System.out.println("a  <  b  =  "  +  (a  <  b)  );
10             System.out.println("b  >=  a  =  "  +  (b  >=  a)  );
11             System.out.println("b  <=  a  =  "  +  (b  <=  a)  );
```

```
12        }
13    }
```

以上实例编译运行结果如下：

```
1    a  ==  b  =  false
2    a  !=  b  =  true
3    a  >  b  =  false
4    a  <  b  =  true
5    b  >=  a  =  true
6    b  <=  a  =  false
```

位运算符

Java 定义了位运算符，应用于整数类型(int)，长整型(long)，短整型(short)，字符型(char)，和字节型(byte)等类型。

位运算符作用在所有的位上，并且按位运算。假设 a = 60，和 b = 13;它们的二进制格式表示将如下：

```
1    A  =  0011  1100
2    B  =  0000  1101
3    -----------------
4    A&b  =  0000  1100
5    A  |  B  =  0011  1101
6    ^  B  =  0011  0001
7    ~A=  1100  0011
```

下表列出了位运算符的基本运算,假设整数变量 A 的值为 60 和变量 B 的值为 13：

操作符	描述	例子

&	按位与操作符，当且仅当两个操作数的某一位都非 0 时候结果的该位才为 1。	（A&B），得到 12，即 0000 1100
|	按位或操作符，只要两个操作数的某一位有一个非 0 时候结果的该位就为 1。	（A | B）得到 61，即 0011 1101
^	按位异或操作符，两个操作数的某一位不相同时候结果的该位就为 1。	（A ^ B）得到 49，即 0011 0001
～	按位补运算符翻转操作数的每一位。	（～A）得到-60，即 1100 0011
<<	按位左移运算符。左操作数按位左移右操作数指定的位数。	A << 2 得到 240，即 1111 0000
>>	按位右移运算符。左操作数按位右移右操作数指定的位数。	A >> 2 得到 15 即 1111
>>>	按位右移补零操作符。左操作数的值按右操作数指定的位数右移，移动得到的空位以零填充。	A>>>2 得到 15 即 0000 1111

实例

下面的简单示例程序演示了位运算符。复制并粘贴下面的 Java 程序并保存为 Test.java 文件，然后编译并运行这个程序：

```
1    public  class  Test  {
```

```
2       public static void main(String args[]) {
3             int a = 60; /* 60 = 0011 1100 */
4             int b = 13; /* 13 = 0000 1101 */
5             int c = 0;
6             c = a & b;             /* 12 = 0000 1100 */
7             System.out.println("a & b = " + c );
8
9             c = a | b;             /* 61 = 0011 1101 */
10            System.out.println("a | b = " + c );
11
12            c = a ^ b;             /* 49 = 0011 0001 */
13            System.out.println("a ^ b = " + c );
14
15            c = ~a;                 /*-61 = 1100 0011 */
16            System.out.println("~a = " + c );
17
18            c = a << 2;          /* 240 = 1111 0000 */
19            System.out.println("a << 2 = " + c );
20
21            c = a >> 2;          /* 215 = 1111 */
22            System.out.println("a >> 2  = " + c );
23
24            c = a >>> 2;          /* 215 = 0000 1111 */
25            System.out.println("a >>> 2 = " + c );
26      }
27  }
```

以上实例编译运行结果如下：

```
1   a & b = 12
```

```
2   a | b = 61
3   a ^ b = 49
4   ~a = -61
5   a << 2 = 240
6   a >> 15
7   a >>> 15
```

逻辑运算符

下表列出了逻辑运算符的基本运算，假设布尔变量 A 为真，变量 B 为假

操作符	描述	例子
&&	称为逻辑与运算符。当且仅当两个操作数都为真，条件才为真。	（A && B）为假。
||	称为逻辑或操作符。如果任何两个操作数任何一个为真，条件为真。	（A || B）为真。
!	称为逻辑非运算符。用来反转操作数的逻辑状态。如果条件为 true，则逻辑非运算符将得到 false。	!（A && B）为真。

实例

下面的简单示例程序演示了逻辑运算符。复制并粘贴下面的 Java 程序并保存为 Test.java 文件，然后编译并运行这个程序：

```
1   public class Test {
```

```
2       public  static  void  main(String  args[])  {
3             boolean  a  =  true;
4             boolean  b  =  false;
5             System.out.println("a  &&  b  =  "  +  (a&&b));
6             System.out.println("a  ||  b  =  "  +  (a||b)  );
7             System.out.println("!(a  &&  b)  =  "  +  !(a  &&  b));
8       }
9   }
```

以上实例编译运行结果如下：

```
1   a  &&  b  =  false
2   a  ||  b  =  true
3   !(a  &&  b)  =  true
```

赋值运算符

下面是 Java 语言支持的赋值运算符：

操作符	描述	例子
=	简单的赋值运算符，将右操作数的值赋给左侧操作数	C = A + B 将把 A + B 得到的值赋给 C
+ =	加和赋值操作符，它把左操作数和右操作数相加赋值给左操作数	C + = A 等价于 C = C + A
- =	减和赋值操作符，它把左操作数和右操作数相减赋值给左操作数	C - = A 等价于 C = C - A

* =	乘和赋值操作符，它把左操作数和右操作数相乘赋值给左操作数	C * = A 等价于 C = C * A
/ =	除和赋值操作符，它把左操作数和右操作数相除赋值给左操作数	C / = A 等价于 C = C / A
（%）=	取模和赋值操作符，它把左操作数和右操作数取模后赋值给左操作数	C%= A 等价于 C = C% A
<< =	左移位赋值运算符	C << = 2 等价于 C = C << 2
>> =	右移位赋值运算符	C >> = 2 等价于 C = C >> 2
&=	按位与赋值运算符	C&= 2 等价于 C = C &2
^ =	按位异或赋值操作符	C ^ = 2 等价于 C = C ^ 2
| =	按位或赋值操作符	C | = 2 等价于 C = C | 2

实例

面的简单示例程序演示了赋值运算符。复制并粘贴下面的 Java 程序并保存为 Test.java 文件，然后编译并运行这个程序：

```
1    public  class  Test  {
2        public  static  void  main(String  args[])  {
3            int  a  =  10;
4            int  b  =  20;
5            int  c  =  0;
6            c  =  a  +  b;
7            System.out.println("c  =  a  +  b  =  "  +  c  );
8            c  +=  a  ;
9            System.out.println("c  +=  a    =  "  +  c  );
10           c  -=  a  ;
11           System.out.println("c  -=  a  =  "  +  c  );
12           c  *=  a  ;
13           System.out.println("c  *=  a  =  "  +  c  );
14           a  =  10;
15           c  =  15;
16           c  /=  a  ;
17           System.out.println("c  /=  a  =  "  +  c  );
18           a  =  10;
19           c  =  15;
20           c  %=  a  ;
21           System.out.println("c  %=  a    =  "  +  c  );
22           c  <<=  2  ;
23           System.out.println("c  <<=  2  =  "  +  c  );
24           c  >>=  2  ;
25           System.out.println("c  >>=  2  =  "  +  c  );
26           c  >>=  2  ;
27           System.out.println("c  >>=  a  =  "  +  c  );
```

```
28              c  &=  a  ;
29              System.out.println("c  &=  2    =  "  +  c  );
30              c  ^=  a  ;
31              System.out.println("c  ^=  a      =  "  +  c  );
32              c  |=  a  ;
33              System.out.println("c  |=  a      =  "  +  c  );
34          }
35      }
```

以上实例编译运行结果如下：

```
1     c  =  a  +  b  =  30
2     c  +=  a    =  40
3     c  -=  a  =  30
4     c  *=  a  =  300
5     c  /=  a  =  1
6     c  %=  a    =  5
7     c  <<=  2  =  20
8     c  >>=  2  =  5
9     c  >>=  2  =  1
10    c  &=  a    =  0
11    c  ^=  a      =  10
12    c  |=  a      =  10
```

条件运算符（?:）

条件运算符也被称为三元运算符。该运算符有 3 个操作数，并且需要判断布尔表达式的值。该运算符的主要是决定哪个值应该赋值给变量。

```
1    variable  x  =  (expression)  ?  value  if  true  :  value  if  false
```

实例

```
1    public  class  Test  {
2          public  static  void  main(String  args[]){
3                int  a  ,  b;
4                a  =  10;
5    b  =  (a  ==  1)  ?  20:  30;
6    System.out.println(  "Value  of  b  is  :  "  +    b  );
7                b  =  (a  ==  10)  ?  20:  30;
8              System.out.println(  "Value  of  b  is  :  "  +  b  );
9          }
10   }
```

以上实例编译运行结果如下：

```
1   Value  of  b  is  :  30
2   Value  of  b  is  :  20
```

instanceOf 运算符

该运算符用于操作对象实例，检查该对象是否是一个特定类型（类类型或接口类型）。

instanceof 运算符使用格式如下：

```
1   (  Object  reference  variable  )  instanceOf    (class/interface  type)
```

如果运算符左侧变量所指的对象，是操作符右侧类或接口(class/interface)的一个对象，那么结果为真。

下面是一个例子：

```
1   String  name  =  'James';
2   boolean  result  =  name  instanceOf  String;  //  由于 name 是 Strine 类型，所以返回真
```

如果被比较的对象兼容于右侧类型,该运算符仍然返回 true。

看下面的例子：

```
1   class  Vehicle  {}
2
3   public  class  Car  extends  Vehicle  {
4         public  static  void  main(String  args[]){
5               Vehicle  a  =  new  Car();
6               boolean  result  =    a  instanceof  Car;
7               System.out.println(  result);
8         }
9   }
```

以上实例编译运行结果如下：

```
1   true
```

Java 运算符优先级

当多个运算符出现在一个表达式中，谁先谁后呢？这就涉及到运算符的优先级别的问题。在一个多运算符的表达式中，运算符优先级不同会导致最后得出的结果差别甚大。

例如，（1+3）＋（3+2）*2，这个表达式如果按加号最优先计算，答案就是 18，如果按照乘号最优先，答案则是 14。

再如，x = 7 + 3 * 2;这里 x 得到 13，而不是 20，因为乘法运算符比加法运算符有较高的优先级，所以先计算 3 * 2 得到 6，然后再加 7。

下表中具有最高优先级的运算符在的表的最上面，最低优先级的在表的底部。

类别	操作符	关联性

后缀	() [] . (点操作符)	左到右
一元	+ + - ! ～	从右到左
乘性	* /%	左到右
加性	+ -	左到右
移位	>> >>> <<	左到右
关系	>> = << =	左到右
相等	== !=	左到右
按位与	&	左到右
按位异或	^	左到右
按位或	|	左到右
逻辑与	&&	左到右
逻辑或	||	左到右

条件	？：	从右到左
赋值	= + = - = * = / =%= >> = << =&= ^ = | =	从右到左
逗号	,	左到右

JAVA 培训教程—Java 分支结构

Java 有两种分支结构：

- if 语句
- switch 语句

if 语句

一个 if 语句包含一个布尔表达式和一条或多条语句。

语法

If 语句的用语法如下：

```
1    if(布尔表达式)
2    {
3          //如果布尔表达式为 true 将执行的语句
4    }
```

如果布尔表达式的值为 true，则执行 if 语句中的代码块。否则执行 If 语句块后面的代码。

```
1     public  class  Test {
2
3           public  static  void  main(String args[]){
4                 int  x =  10;
5
6                 if( x <  20  ){
7                       System.out.print("这是 if 语句");
8                 }
9           }
10    }
```

以上代码编译运行结果如下：

```
1    这是  if  语句
```

if...else 语句

if 语句后面可以跟 else 语句，当 if 语句的布尔表达式值为 false 时，else 语句块会被执行。

语法

if…else 的用法如下：

```
1    if(布尔表达式){
2          //如果布尔表达式的值为 true
3    }else{
4          //如果布尔表达式的值为 false
5    }
```

实例

```
1     public  class  Test {
2
3           public  static  void  main(String args[]){
4                 int  x =  30;
5
6                 if( x <  20  ){
7                       System.out.print("这是 if 语句");
8                 }else{
9                       System.out.print("这是 else 语句");
10                }
11          }
12    }
```

以上代码编译运行结果如下：

```
1    这是  else  语句
```

if...else if...else 语句

if 语句后面可以跟 elseif…else 语句，这种语句可以检测到多种可能的情况。

使用 if，else if，else 语句的时候，需要注意下面几点：

- if 语句至多有 1 个 else 语句，else 语句在所有的 elseif 语句之后。
- If 语句可以有若干个 elseif 语句，它们必须在 else 语句之前。
- 一旦其中一个 else if 语句检测为 true，其他的 else if 以及 else 语句都将跳过执行。

语法

if...else 语法格式如下:

```
1   if(布尔表达式  1){
2         //如果布尔表达式 1 的值为 true 执行代码
3   }else  if(布尔表达式  2){
4         //如果布尔表达式 2 的值为 true 执行代码
5   }else  if(布尔表达式  3){
6         //如果布尔表达式 3 的值为 true 执行代码
7   }else  {
8         //如果以上布尔表达式都不为 true 执行代码
9   }
```

实例

```
1    public  class  Test {
2
3          public  static  void  main(String args[]){
```

```
4                int  x =  30;
5
6                if( x ==  10  ){
7                      System.out.print("Value of X is 10");
8                }else  if( x ==  20  ){
9                      System.out.print("Value of X is 20");
10               }else  if( x ==  30  ){
11                     System.out.print("Value of X is 30");
12               }else{
13                     System.out.print("This is else statement");
14               }
15         }
16   }
```

以上代码编译运行结果如下：

```
1   Value of X is  30
```

嵌套的 if…else 语句

使用嵌套的 if-else 语句是合法的。也就是说你可以在另一个 if 或者 elseif 语句中使用 if 或者 elseif 语句。

语法

嵌套的 if…else 语法格式如下：

```
1   if(布尔表达式  1){
2          ////如果布尔表达式 1 的值为 true 执行代码
3          if(布尔表达式  2){
4                ////如果布尔表达式 2 的值为 true 执行代码
5          }
```

```
6    }
```

你可以像 *if* 语句一样嵌套 *else if...else*。

实例

```
1    public  class  Test {
2
3          public  static  void  main(String args[]){
4                int  x =  30;
5                int  y =  10;
6
7                if( x ==  30  ){
8                      if( y ==  10  ){
9                            System.out.print("X = 30 and Y = 10");
10                       }
11                 }
12           }
13   }
```

以上代码编译运行结果如下：

```
1    X =  30  and Y =  10
```

switch 语句

switch 语句判断一个变量与一系列值中某个值是否相等，每个值称为一个分支。

语法

switch 语法格式如下：

```
1    switch(expression){
2            case  value :
3                  //语句
4                  break;  //可选
5            case  value :
6                  //语句
7                  break;  //可选
8            //你可以有任意数量的 case 语句
9            default  :  //可选
10                 //语句
11   }
```

switch 语句有如下规则：

- switch 语句中的变量类型只能为 byte、short、int 或者 char。
- switch 语句可以拥有多个 case 语句。每个 case 后面跟一个要比较的值和冒号。
- case 语句中的值的数据类型必须与变量的数据类型相同，而且只能是常量或者字面常量。
- 当变量的值与 case 语句的值相等时，那么 case 语句之后的语句开始执行，直到 break 语句出现才会跳出 switch 语句。3
- 当遇到 break 语句时，switch 语句终止。程序跳转到 switch 语句后面的语句执行。case 语句不必须要包含 break 语句。如果没有 break 语句出现，程序会继续执行下一条 case 语句，直到出现 break 语句。
- switch 语句可以包含一个 default 分支，该分支必须是 switch 语句的最后一个分支。default 在没有 case 语句的值和变量值相等的时候执行。default 分支不需要 break 语句。

实例

```
1    public  class  Test {
2
```

```
3          public  static  void  main(String args[]){
4                //char grade = args[0].charAt(0);
5                char  grade =  'C';
6
7                switch(grade)
8                {
9                      case  'A'  :
10                           System.out.println("Excellent!");
11                           break;
12                     case  'B'  :
13                     case  'C'  :
14                           System.out.println("Well done");
15                           break;
16                     case  'D'  :
17                           System.out.println("You passed");
18                     case  'F'  :
19                           System.out.println("Better try again");
20                           break;
21                     default  :
22                           System.out.println("Invalid grade");
23               }
24               System.out.println("Your grade is "  + grade);
25         }
26   }
```

以上代码编译运行结果如下：

```
1   $ java Test
2   Well done
3   Your grade is a C
```

4 $

JAVA 培训教程—Java 循环结构

Java 中有三种主要的循环结构：

- while 循环
- do…while 循环
- for 循环

在 Java5 中引入了一种主要用于数组的增强型 for 循环。

while 循环

while 是最基本的循环，它的结构为：

```
while( 布尔表达式 ) {
        //循环内容
}
```

只要布尔表达式为 true，循环体会一直执行下去。

实例

```
public class Test {
        public static void main(String args[]) {
                int x = 10;
                while( x < 20 ) {
                        System.out.print("value of x : " + x );
                        x++;
                        System.out.print("\n");
                }
        }
}
```

以上实例编译运行结果如下：

```
value of x :  10
value of x :  11
value of x :  12
value of x :  13
value of x :  14
value of x :  15
value of x :  16
value of x :  17
value of x :  18
value of x :  19
```

do…while 循环

对于 while 语句而言，如果不满足条件，则不能进入循环。但有时候我们需要即使不满足条件，也至少执行一次。

do…while 循环和 while 循环相似，不同的是，do…while 循环至少会执行一次。

```
do  {
                //代码语句
}while(布尔表达式);
```

注意：布尔表达式在循环体的后面，所以语句块在检测布尔表达式之前已经执行了。 如果布尔表达式的值为 true，则语句块一直执行，直到布尔表达式的值为 false。

实例

```
public  class  Test {

        public  static  void  main(String args[]){
                int  x =  10;
```

```
5
6                do{
7                        System.out.print("value of x : "  + x );
8                        x++;
9                        System.out.print("\n");
10               }while( x <  20  );
11          }
12   }
```

以上实例编译运行结果如下：

```
1    value of x :  10
2    value of x :  11
3    value of x :  12
4    value of x :  13
5    value of x :  14
6    value of x :  15
7    value of x :  16
8    value of x :  17
9    value of x :  18
10   value of x :  19
```

for 循环

虽然所有循环结构都可以用 while 或者 do...while 表示，但 Java 提供了另一种语句 ——for 循环，使一些循环结构变得更加简单。

for 循环执行的次数是在执行前就确定的。语法格式如下

```
1    for(初始化; 布尔表达式; 更新) {
2            //代码语句
3    }
```

关于 for 循环有以下几点说明：

- 最先执行初始化步骤。可以声明并初始化一个或多个循环控制变量，也可以是空语句。
- 然后，检测布尔表达式的值。如果为 true，循环体被执行。如果为 false，循环终止，开始执行循环体后面的语句。
- 执行一次循环后，更新循环控制变量。
- 再次检测布尔表达式。循环执行上面的过程。

实例

```
1    public  class  Test {
2
3          public  static  void  main(String args[]) {
4
5                for(int  x =  10; x <  20; x = x+1) {
6                      System.out.print("value of x : "  + x );
7                      System.out.print("\n");
8                }
9          }
10   }
```

以上实例编译运行结果如下：

```
1    value of x :  10
2    value of x :  11
3    value of x :  12
4    value of x :  13
5    value of x :  14
6    value of x :  15
7    value of x :  16
```

```
8     value of x :  17
9     value of x :  18
10    value of x :  19
```

Java 增强 for 循环

Java5 引入了一种主要用于数组的增强型 for 循环。

Java 增强 for 循环语法格式如下:

```
1    for(声明语句 : 表达式)
2    {
3          //代码句子
4    }
```

声明语句：声明新的局部变量，该变量的类型必须和数组元素的类型匹配。其作用域限定在循环语句块，其值与此时数组元素的值相等。

表达式：表达式是要访问的数组名，或者是返回值为数组的方法。

实例

```
1     public  class  Test {
2
3           public  static  void  main(String args[]){
4                 int  [] numbers = {10,  20,  30,  40,  50};
5
6                 for(int  x : numbers ){
7                       System.out.print( x );
8                       System.out.print(",");
9                 }
10                System.out.print("\n");
11                String [] names ={"James",  "Larry",  "Tom",  "Lacy"};
```

```
12                for( String name : names ) {
13                      System.out.print( name );
14                      System.out.print(",");
15                }
16          }
17    }
```

以上实例编译运行结果如下：

```
1     10,20,30,40,50,
2     James,Larry,Tom,Lacy,
```

break 关键字

break 主要用在循环语句或者 switch 语句中，用来跳出整个语句块。

break 跳出最里层的循环，并且继续执行该循环下面的语句。

语法

break 的用法很简单，就是循环结构中的一条语句：

```
1    break;
```

实例

```
1     public  class  Test {
2
3           public  static  void  main(String args[]) {
4                 int  [] numbers = {10,  20,  30,  40,  50};
5
6                 for(int  x : numbers ) {
7                       if( x ==  30  ) {
8                         break;
```

```
9                       }
10                      System.out.print( x );
11                      System.out.print("\n");
12              }
13          }
14  }
```

以上实例编译运行结果如下：

```
1   10
2   20
```

continue 关键字

continue 适用于任何循环控制结构中。作用是让程序立刻跳转到下一次循环的迭代。

在 for 循环中，continue 语句使程序立即跳转到更新语句。

在 while 或者 do…while 循环中，程序立即跳转到布尔表达式的判断语句。

语法

continue 就是循环体中一条简单的语句：

```
1   continue;
```

实例

```
1   public  class  Test {
2
3           public  static  void  main(String args[]) {
4                   int  [] numbers = {10,  20,  30,  40,  50};
5
6                   for(int  x : numbers ) {
```

```
7                       if( x ==  30  ) {
8                         continue;
9                       }
10                      System.out.print( x );
11                      System.out.print("\n");
12                  }
13          }
14   }
```

以上实例编译运行结果如下：

```
1    10
2    20
3    40
4    50
```

JAVA 培训教程—Java 数组

数组对于每一门编辑应语言来说都是重要的数据结构之一，当然不同语言对数组的实现及处理也不尽相同。

Java 语言中提供的数组是用来存储固定大小的同类型元素。

你可以声明一个数组变量，如 numbers[100]来代替直接声明 100 个独立变量 number0，number1，....，number99。

本教程将为大家介绍 Java 数组的声明、创建和初始化，并给出其对应的代码。

声明数组变量

首先必须声明数组变量，才能在程序中使用数组。下面是声明数组变量的语法：

```
1    dataType[] arrayRefVar;       // 首选的方法
2
3    或
4
5    dataType arrayRefVar[];     // 效果相同，但不是首选方法
```

注意： 建议使用 dataType[] arrayRefVar 的声明风格声明数组变量。 dataType arrayRefVar[] 风格是来自 C/C++ 语言 ，在 Java 中采用是为了让 C/C++ 程序园能够快速理解 java 语言。

实例

下面是这两种语法的代码示例：

```
1    double[] myList;                   // 首选的方法
2
3    或
```

```
4
5    double  myList[];                    //    效果相同，但不是首选方法
```

创建数组

Java 语言使用 new 操作符来创建数组，语法如下：

```
1    arrayRefVar =  new  dataType[arraySize];
```

上面的语法语句做了两件事：

- 一、使用 dataType[arraySize]创建了一个数组。
- 二、把新创建的数组的引用赋值给变量 arrayRefVar。

数组变量的声明，和创建数组可以用一条语句完成，如下所示：

```
1    dataType[] arrayRefVar =  new  dataType[arraySize];
```

另外，你还可以使用如下的方式创建数组。

```
1    dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

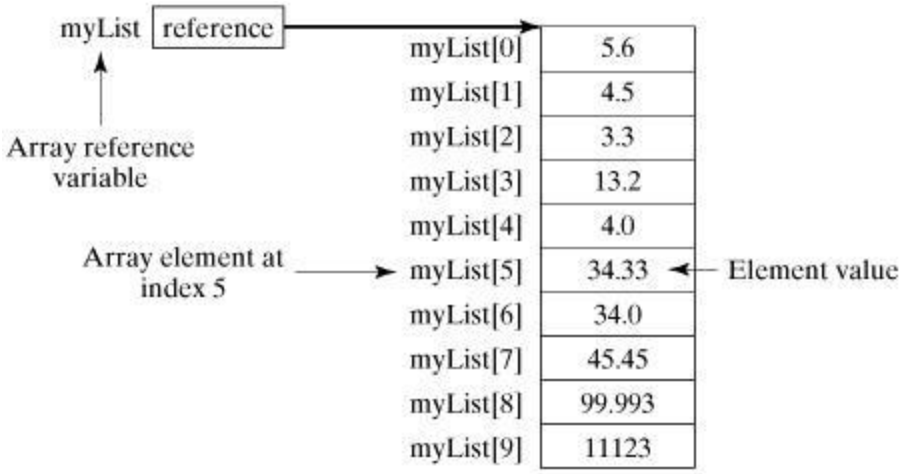
数组的元素是通过索引访问的。数组索引从 0 开始，所以索引值从 0 到 arrayRefVar.length-1。

实例

下面的语句首先声明了一个数组变量 myList，接着创建了一个包含 10 个 double 类型元素的数组，并且把它的引用赋值给 myList 变量。

```
1    double[] myList =  new  double[10];
```

下面的图片描绘了数组 myList。这里 myList 数组里有 10 个 double 元素，它的下标从 0 到 9。



处理数组

数组的元素类型和数组的大小都是确定的，所以当处理数组元素时候，我们通常使用基本循环或者 foreach 循环。

示例

该实例完整地展示了如何创建、初始化和操纵数组：

```
1    public  class  TestArray {
2
3          public  static  void  main(String[] args) {
4                double[] myList = {1.9,  2.9,  3.4,  3.5};
5
6                // 打印所有数组元素
7                for  (int  i =  0; i < myList.length; i++) {
8                      System.out.println(myList[i] +  " ");
9                }
10               // 计算所有元素的总和
11               double  total =  0;
12               for  (int  i =  0; i < myList.length; i++) {
13                     total += myList[i];
```

```
14              }
15              System.out.println("Total is "  + total);
16              // 查找最大元素
17              double  max = myList[0];
18              for  (int  i =  1; i < myList.length; i++) {
19                    if  (myList[i] > max) max = myList[i];
20              }
21              System.out.println("Max is "  + max);
22          }
23    }
```

以上实例编译运行结果如下：

```
1    1.9
2    2.9
3    3.4
4    3.5
5    Total is  11.7
6    Max is  3.5
```

foreach 循环

JDK 1.5 引进了一种新的循环类型，被称为 foreach 循环或者加强型循环，它能在不使用下标的情况下遍历数组。

示例

该实例用来显示数组 myList 中的所有元素：

```
1    public  class  TestArray {
2
3          public  static  void  main(String[] args) {
```

```
4                double[] myList = {1.9,  2.9,  3.4,  3.5};
5
6                // 打印所有数组元素
7                for  (double  element: myList) {
8                      System.out.println(element);
9                }
10          }
11    }
```

以上实例编译运行结果如下：

```
1    1.9
2    2.9
3    3.4
4    3.5
```

数组作为函数的参数

数组可以作为参数传递给方法。例如，下面的例子就是一个打印 int 数组中元素的方法。

```
1    public  static  void  printArray(int[] array) {
2        for  (int  i =  0; i < array.length; i++) {
3            System.out.print(array[i] +  " ");
4        }
5    }
```

下面例子调用 printArray 方法打印出 3，1，2，6，4 和 2：

```
1    printArray(new  int[]{3,  1,  2,  6,  4,  2});
```

数组作为函数的返回值

```
1    public  static  int[] reverse(int[] list) {
```

```
2        int[] result =  new  int[list.length];
3
4        for  (int  i =  0, j = result.length -  1; i < list.length; i++, j--) {
5            result[j] = list[i];
6        }
7        return  result;
8    }
```

以上实例中 result 数组作为函数的返回值。

Arrays 类

java.util.Arrays 类能方便地操作数组，它提供的所有方法都是静态的。具有以下功能：

- 给数组赋值：通过 fill 方法。
- 对数组排序：通过 sort 方法,按升序。
- 比较数组：通过 equals 方法比较数组中元素值是否相等。
- 查找数组元素：通过 binarySearch 方法能对排序好的数组进行二分查找法操作。

具体说明请查看下表：

序号	方法和说明
1	**public static int binarySearch(Object[] a, Object key)** 用二分查找算法在给定数组中搜索给定值的对象(Byte,Int,double 等)。数组在调用前必须排序好的。如果查找值包含在数组中，则返回搜索键的索引；否则返回 (-(*插入点*) - 1)。
2	**public static boolean equals(long[] a, long[] a2)** 如果两个指定的 long 型数组彼此*相等*，则返回 true。如果两个数组包含相同

数量的元素，并且两个数组中的所有相应元素对都是相等的，则认为这两个数组是相等的。换句话说，如果两个数组以相同顺序包含相同的元素，则两个数组是相等的。同样的方法适用于所有的其他基本数据类型（Byte，short，Int 等）。

3	**public static void fill(int[] a, int val)** 将指定的 int 值分配给指定 int 型数组指定范围中的每个元素。同样的方法适用于所有的其他基本数据类型（Byte，short，Int 等）。
4	**public static void sort(Object[] a)** 对指定对象数组根据其元素的自然顺序进行升序排列。同样的方法适用于所有的其他基本数据类型（Byte，short，Int 等）。

JAVA 培训教程—Java 数据结构

Java 工具包提供了强大的数据结构。在 Java 中的数据结构主要包括以下几种接口和类：

1、枚举（Enumeration）

2、位集合（BitSet）

3、向量（Vector）

4、栈（Stack）

5、字典（Dictionary）

6、哈希表（Hashtable）

7、属性（Properties）

以上这些类是传统遗留的，在 Java2 中引入了一种新的框架-集合框架(Collection)，我们再讨论。

枚举（Enumeration）

枚举（Enumeration）接口虽然它本身不属于数据结构,但它在其他数据结构的范畴里应用很广。 枚举（The Enumeration）接口定义了一种从数据结构中取回连续元素的方式。

例如，枚举定义了一个叫 nextElement 的方法，该方法用来得到一个包含多元素的数据结构的下一个元素。

位集合（BitSet）

位集合类实现了一组可以单独设置和清除的位或标志。

该类在处理一组布尔值的时候非常有用，你只需要给每个值赋值一"位"，然后对位进行适当的设置或清除，就可以对布尔值进行操作了。

向量（Vector）

向量（Vector）类和传统数组非常相似，但是 Vector 的大小能根据需要动态的变化。

和数组一样，Vector 对象的元素也能通过索引访问。

使用 Vector 类最主要的好处就是在创建对象的时候不必给对象指定大小，它的大小会根据需要动态的变化。

栈（Stack）

栈（Stack）实现了一个后进先出（LIFO）的数据结构。

你可以把栈理解为对象的垂直分布的栈，当你添加一个新元素时，就将新元素放在其他元素的顶部。

当你从栈中取元素的时候，就从栈顶取一个元素。换句话说，最后进栈的元素最先被取出。

字典（Dictionary）

字典（Dictionary） 类是一个抽象类，它定义了键映射到值的数据结构。

当你想要通过特定的键而不是整数索引来访问数据的时候，这时候应该使用 Dictionary。

由于 Dictionary 类是抽象类，所以它只提供了键映射到值的数据结构，而没有提供特定的实现。

哈希表（Hashtable）

Hashtable 类提供了一种在用户定义键结构的基础上来组织数据的手段。

例如，在地址列表的哈希表中，你可以根据邮政编码作为键来存储和排序数据，而是通过人的名字。

哈希表键的具体含义完全取决于哈希表的使用情景和它包含的数据。

属性（Properties）

Properties 继承于 Hashtable.Properties 类表示了一个持久的属性集.属性列表中每个键及其对应值都是一个字符串。

Properties 类被许多 Java 类使用。例如，在获取环境变量时它就作为 System.getProperties()方法的返回值。

JAVA 培训教程—Java 集合框架

在 Java2 之前，Java 就提供了特设类。比如:Dictionary, Vector, Stack, 和 Properties 这些类用来存储和操作对象组。

虽然这些类都非常有用，但是它们缺少一个核心的，统一的主题。由于这个原因，使用 Vector 类的方式和使用 Properties 类的方式有着很大不同。

集合框架被设计成要满足以下几个目标。

- 该框架必须是高性能的。基本集合（动态数组，链表，树，哈希表）的实现也必须是高效的。
- 该框架允许不同类型的集合，以类似的方式工作，具有高度的互操作性。
- 对一个集合的扩展和适应必须是简单的。

为此，整个集合框架就围绕一组标准接口而设计。你可以直接使用这些接口的标准实现，诸如： LinkedList, HashSet, 和 TreeSet 等,除此之外你也可以通过这些接口实现自己的集合。

集合框架是一个用来代表和操纵集合的统一架构。所有的集合框架都包含如下内容：

- **接口：**是代表集合的抽象数据类型。接口允许集合独立操纵其代表的细节。在面向对象的语言，接口通常形成一个层次。
- **实现（类）：**是集合接口的具体实现。从本质上讲，它们是可重复使用的数据结构。
- **算法：**是实现集合接口的对象里的方法执行的一些有用的计算，例如：搜索和排序。这些算法被称为多态，那是因为相同的方法可以在相似的接口上有着不同的实现。

除了集合，该框架也定义了几个 Map 接口和类。Map 里存储的是键/值对。尽管 Map 不是 collections，但是它们完全整合在集合中。

集合接口

集合框架定义了一些接口。本节提供了每个接口的概述：

序号	接口描述
1	Collection 接口 允许你使用一组对象，是 Collection 层次结构的根接口。
2	List 接口 继承于 **Collection** 和一个 List 实例存储一个有序集合的元素。
3	Set 继承于 **Collection，是**一个不包含重复元素的集合。
4	SortedSet 继承于 Set 保存有序的集合。
5	Map 将唯一的键映射到值。
6	Map.Entry 描述在一个 Map 中的一个元素（键/值对）。是一个 Map 的内部类。
7	SortedMap 继承于 Map，使 Key 保持在升序排列。
8	Enumeration 这是一个传统的接口和定义的方法，通过它可以枚举（一次获得一个）对象集合中

的元素。这个传统接口已被迭代器取代。

集合类

Java 提供了一套实现了 Collection 接口的标准集合类。其中一些是具体类，这些类可以直接拿来使用，而另外一些是抽象类，提供了接口的部分实现。

标准集合类汇总于下表：

序号	类描述
1	**AbstractCollection** 实现了大部分的集合接口。
2	**AbstractList** 继承于 AbstractCollection 并且实现了大部分 List 接口。
3	**AbstractSequentialList** 继承于 AbstractList ，提供了对数据元素的链式访问而不是随机访问。
4	LinkedList 继承于 AbstractSequentialList，实现了一个链表。
5	ArrayList 通过继承 AbstractList，实现动态数组。

6 **AbstractSet**

继承于 AbstractCollection 并且实现了大部分 Set 接口。

7 HashSet

继承了 AbstractSet，并且使用一个哈希表。

8 LinkedHashSet

具有可预知迭代顺序的 Set 接口的哈希表和链接列表实现。

9 TreeSet

继承于 AbstractSet，使用元素的自然顺序对元素进行排序.

10 **AbstractMap**

实现了大部分的 Map 接口。

11 HashMap

继承了 HashMap，并且使用一个哈希表。

12 TreeMap

继承了 AbstractMap，并且使用一颗树。

13 WeakHashMap

继承 AbstractMap 类，使用弱密钥的哈希表。

14 LinkedHashMap

继承于 HashMap，使用元素的自然顺序对元素进行排序.

15	IdentityHashMap 继承 AbstractMap 类，比较文档时使用引用相等。

在前面的教程中已经讨论通过 java.util 包中定义的类，如下所示：

序号	类描述
1	Vector Vector 类实现了一个动态数组。和 ArrayList 和相似，但是两者是不同的。
2	Stack 栈是 Vector 的一个子类，它实现了一个标准的后进先出的栈。
3	Dictionary Dictionary 类是一个抽象类，用来存储键/值对，作用和 Map 类相似。
4	Hashtable Hashtable 是原始的 java.util 的一部分， 是一个 Dictionary 具体的实现 。
5	Properties Properties 继承于 Hashtable.表示一个持久的属性集.属性列表中每个键及其对应值都是一个字符串。

6	BitSet 一个 Bitset 类创建一种特殊类型的数组来保存位值。BitSet 中数组大小会随需要增加。

一个 Bitset 类创建一种特殊类型的数组来保存位值。BitSet 中数组大小会随需要增加。

集合算法

集合框架定义了几种算法，可用于集合和映射。这些算法被定义为集合类的静态方法。

在尝试比较不兼容的类型时，一些方法能够抛出 ClassCastException 异常。当试图修改一个不可修改的集合时，抛出 UnsupportedOperationException 异常。

集合定义三个静态的变量：EMPTY_SET EMPTY_LIST，EMPTY_MAP 的。这些变量都不可改变。

序号	算法描述
1	Collection Algorithms 这里是一个列表中的所有算法实现。

如何使用迭代器

通常情况下，你会希望遍历一个集合中的元素。例如，显示集合中的每个元素。

做到这一点最简单的方法是采用一个迭代器，它是一个对象，实现了 Iterator 接口或 ListIterator 接口。

迭代器，使你能够通过循环来得到或删除集合的元素。ListIterator 继承了 Iterator，以允许双向遍历列表和修改元素。

这里通过实例列出 Iterator 和 listIterator 接口提供的所有方法。

如何使用比较器

TreeSet 和 TreeMap 的按照排序顺序来存储元素. 然而，这是通过比较器来精确定义按照什么样的排序顺序。

这个接口可以让我们以不同的方式来排序一个集合。

序号	比较器方法描述
1	使用 Java Comparator 这里通过实例列出 Comparator 接口提供的所有方法

总结

Java 集合框架为程序员提供了预先包装的数据结构和算法来操纵他们。集合是一个对象，可容纳其他对象的引用。集合接口声明对每一种类型的集合可以执行的操作。集合框架的类和接口均在 java.util 包中。

JAVA 培训教程—Java String 类

字符串广泛应用在 Java 编程中，在 Java 中字符串属于对象，Java 提供了 String 类来创建和操作字符串。

创建字符串

创建字符串最简单的方式如下:

```
1    String greeting =  "Hello world!";
```

在代码中遇到字符串常量时，这里的值是"Hello world!"，编译器会使用该值创建一个 String 对象。

和其它对象一样，可以使用关键字和构造方法来创建 String 对象。

String 类有 11 种构造方法，这些方法提供不同的参数来初始化字符串，比如提供一个字符数组参数:

```
1    public  class  StringDemo{
2
3          public  static  void  main(String args[]){
4                char[] helloArray = {  'h',  'e',  'l',  'l',  'o',  '.'};
5                String helloString =  new  String(helloArray);
6                System.out.println( helloString );
7          }
8    }
```

以上实例编译运行结果如下：

```
1    hello.
```

注意: String 类是不可改变的,所以你一旦创建了 String 对象,那它的值就无法改变了。如果需要对字符串做很多修改，那么应该选择使用 StringBuffer & StringBuilder 类。

字符串长度

用于获取有关对象的信息的方法称为访问器方法。

String 类的一个访问器方法是 length()方法，它返回字符串对象包含的字符数。

下面的代码执行后，len 变量等于 17:

```
1   public  class  StringDemo {
2
3         public  static  void  main(String args[]) {
4               String palindrome =  "Dot saw I was Tod";
5               int  len = palindrome.length();
6               System.out.println(  "String Length is : "  + len );
7         }
8   }
```

以上实例编译运行结果如下：

```
1   String Length is :  17
```

连接字符串

String 类提供了连接两个字符串的方法：

```
1   string1.concat(string2);
```

返回 string2 连接 string1 的新字符串。也可以对字符串常量使用 concat()方法，如：

```
1   "My name is ".concat("Zara");
```

更常用的是使用'+'操作符来连接字符串，如：

```
1   "Hello," + " world" + "!"
```

结果如下:

```
1   "Hello, world!"
```

下面是一个例子:

```
1   public class StringDemo {
2         public static void main(String args[]) {
3         String string1 = "saw I was ";
4         System.out.println("Dot " + string1 + "Tod");
5   }
6   }
```

以上实例编译运行结果如下：

```
1   Dot saw I was Tod
```

创建格式化字符串

我们知道输出格式化数字可以使用 printf()和 format()方法。String 类使用静态方法 format()返回一个 String 对象而不是 PrintStream 对象。

String 类的静态方法 format()能用来创建可复用的格式化字符串，而不仅仅是用于一次打印输出。如下所示：

```
1   System.out.printf("The value of the float variable is " +
2   "%f, while the value of the integer " +
3   "variable is %d, and the string " +
4   "is %s", floatVar, intVar, stringVar);
```

也可以这样写

```
1   String fs;
2   fs = String.format("The value of the float variable is " +
3   "%f, while the value of the integer " +
4   "variable is %d, and the string " +
5   "is %s", floatVar, intVar, stringVar);
6   System.out.println(fs);
```

String 方法

下面是 String 类支持的方法:

SN(序号)	方法描述
1	char charAt(int index) 返回指定索引处的 char 值。
2	int compareTo(Object o) 把这个字符串和另一个对象比较。
3	int compareTo(String anotherString) 按字典顺序比较两个字符串。
4	int compareToIgnoreCase(String str) 按字典顺序比较两个字符串，不考虑大小写。
5	String concat(String str) 将指定字符串连接到此字符串的结尾。
6	boolean contentEquals(StringBuffer sb) 当且仅当字符串与指定的 StringButter 有相同顺序的字符时候返回真。

7	static String copyValueOf(char[] data) 返回指定数组中表示该字符序列的 String。
8	static String copyValueOf(char[] data, int offset, int count) 返回指定数组中表示该字符序列的 String。
9	boolean endsWith(String suffix) 测试此字符串是否以指定的后缀结束。
10	boolean equals(Object anObject) 将此字符串与指定的对象比较。
11	boolean equalsIgnoreCase(String anotherString) 将此 String 与另一个 String 比较，不考虑大小写。
12	byte[] getBytes() 使用平台的默认字符集将此 String 编码为 byte 序列，并将结果存储到一
13	byte[] getBytes(String charsetName) 使用指定的字符集将此 String 编码为 byte 序列，并将结果存储到一个新的
14	void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) 将字符从此字符串复制到目标字符数组。
15	int hashCode() 返回此字符串的哈希码。
16	int indexOf(int ch) 返回指定字符在此字符串中第一次出现处的索引。
17	int indexOf(int ch, int fromIndex) 返回在此字符串中第一次出现指定字符处的索引，从指定的索引开始搜索。

18 int indexOf(String str)

返回指定子字符串在此字符串中第一次出现处的索引。

19 int indexOf(String str, int fromIndex)

返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始。

20 String intern()

返回字符串对象的规范化表示形式。

21 int lastIndexOf(int ch)

返回指定字符在此字符串中最后一次出现处的索引。

22 int lastIndexOf(int ch, int fromIndex)

返回指定字符在此字符串中最后一次出现处的索引，从指定的索引处开始进行

23 int lastIndexOf(String str)

返回指定子字符串在此字符串中最右边出现处的索引。

24 int lastIndexOf(String str, int fromIndex)

返回指定子字符串在此字符串中最后一次出现处的索引，从指定的索引开始

25 int length()

返回此字符串的长度。

26 boolean matches(String regex)

告知此字符串是否匹配给定的正则表达式。

27 boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)

28 boolean regionMatches(int toffset, String other, int ooffset, int len)

测试两个字符串区域是否相等。

29	String replace(char oldChar, char newChar) 返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有
30	String replaceAll(String regex, String replacement 使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符
31	String replaceFirst(String regex, String replacement) 使用给定的 replacement 替换此字符串匹配给定的正则表达式的第一个子
32	String[] split(String regex) 根据给定正则表达式的匹配拆分此字符串。
33	String[] split(String regex, int limit) 根据匹配给定的正则表达式来拆分此字符串。
34	boolean startsWith(String prefix) 测试此字符串是否以指定的前缀开始。
35	boolean startsWith(String prefix, int toffset) 测试此字符串从指定索引开始的子字符串是否以指定前缀开始。
36	CharSequence subSequence(int beginIndex, int endIndex) 返回一个新的字符序列，它是此序列的一个子序列。
37	String substring(int beginIndex) 返回一个新的字符串，它是此字符串的一个子字符串。
38	String substring(int beginIndex, int endIndex) 返回一个新字符串，它是此字符串的一个子字符串。
39	char[] toCharArray() 将此字符串转换为一个新的字符数组。

40 String toLowerCase()

使用默认语言环境的规则将此 String 中的所有字符都转换为小写。

41 String toLowerCase(Locale locale)

使用给定 Locale 的规则将此 String 中的所有字符都转换为小写。

42 String toString()

返回此对象本身（它已经是一个字符串!）。

43 String toUpperCase()

使用默认语言环境的规则将此 String 中的所有字符都转换为大写。

44 String toUpperCase(Locale locale)

使用给定 Locale 的规则将此 String 中的所有字符都转换为大写。

45 String trim()

返回字符串的副本，忽略前导空白和尾部空白。

46 static String valueOf(primitive data type x)

返回给定 data type 类型 x 参数的字符串表示形式。

JAVA 培训教程—Java StringBuffer 和 StringBuilder 类

当对字符串进行修改的时候，需要使用 StringBuffer 和 StringBuilder 类。

和 String 类不同的是，StringBuffer 和 StringBuilder 类的对象能够被多次的修改，并且不产生新的未使用对象。

StringBuilder 类在 Java 5 中被提出，它和 StringBuffer 之间的最大不同在于 StringBuilder 的方法不是线程安全的（不能同步访问）。

由于 StringBuilder 相较于 StringBuffer 有速度优势，所以多数情况下建议使用 StringBuilder 类。然而在应用程序要求线程安全的情况下，则必须使用 StringBuffer 类。

实例

```
1  public  class  Test{
2
3          public  static  void  main(String args[]){
4                StringBuffer sBuffer =  new  StringBuffer(" test");
5                sBuffer.append(" String Buffer");
6                System.ou.println(sBuffer);
7        }
8  }
```

以上实例编译运行结果如下：

```
1    test String Buffer
```

StringBuffer 方法

以下是 StringBuffer 类支持的主要方法：

序号	方法描述
1	public StringBuffer append(String s) 将指定的字符串追加到此字符序列。
2	public StringBuffer reverse() 将此字符序列用其反转形式取代。
3	public delete(int start, int end) 移除此序列的子字符串中的字符。
4	public insert(int offset, int i) 将 int 参数的字符串表示形式插入此序列中。
5	replace(int start, int end, String str) 使用给定 String 中的字符替换此序列的子字符串中的字符。

下面的列表里的方法和 String 类的方法类似：

序号	方法描述
1	int capacity() 返回当前容量。
2	char charAt(int index) 返回此序列中指定索引处的 char 值。
3	void ensureCapacity(int minimumCapacity) 确保容量至少等于指定的最小值。

4	void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) 将字符从此序列复制到目标字符数组 dst。
5	int indexOf(String str) 返回第一次出现的指定子字符串在该字符串中的索引。
6	int indexOf(String str, int fromIndex) 从指定的索引处开始，返回第一次出现的指定子字符串在该字符串中的索引。
7	int lastIndexOf(String str) 返回最右边出现的指定子字符串在此字符串中的索引。
8	int lastIndexOf(String str, int fromIndex) 返回最后一次出现的指定子字符串在此字符串中的索引。
9	int length() 返回长度（字符数）。
10	void setCharAt(int index, char ch) 将给定索引处的字符设置为 ch。
11	void setLength(int newLength) 设置字符序列的长度。
12	CharSequence subSequence(int start, int end) 返回一个新的字符序列，该字符序列是此序列的子序列。
13	String substring(int start) 返回一个新的 String，它包含此字符序列当前所包含的字符子序列。
14	String substring(int start, int end) 返回一个新的 String，它包含此序列当前所包含的字符子序列。

15	String toString() 返回此序列中数据的字符串表示形式。

JAVA 培训教程—Java Character 类

使用字符时，我们通常使用的是内置数据类型 char。

实例

```
char  ch =  'a';

// Unicode for uppercase Greek omega character
char  uniChar =  '\u039A';

// 字符数组
char[] charArray ={  'a',  'b',  'c',  'd',  'e'  };
```

然而，在实际开发过程中，我们经常会遇到需要使用对象，而不是内置数据类型的情况。为了解决这个问题，Java 语言为内置数据类型 char 提供了包装类 Character 类。

Character 类提供了一系列方法来操纵字符。你可以使用 Character 的构造方法创建一个 Character 类对象，例如：

```
Character ch =  new  Character('a');
```

在某些情况下，Java 编译器会自动创建一个 Character 对象。

例如，将一个 char 类型的参数传递给需要一个 Character 类型参数的方法时，那么编译器会自动地将 char 类型参数转换为 Character 对象。 这种特征称为装箱，反过来称为拆箱。

实例

```
// Here following primitive char 'a'
// is boxed into the Character object ch
Character ch =  'a';
```

```
    // Here primitive 'x' is boxed for method test,
    // return is unboxed to char 'c'
    char  c = test('x');
```

转义序列

前面有反斜杠（\）的字符代表转义字符，它对编译器来说是有特殊含义的。

下面列表展示了 Java 的转义序列：

转义序列	描述
\t	在文中该处插入一个 tab 键
\b	在文中该处插入一个后退键
\n	在文中该处换行
\r	在文中该处插入回车
\f	在文中该处插入换页符
\'	在文中该处插入单引号
\"	在文中该处插入双引号

\\	在文中该处插入反斜杠

实例

当打印语句遇到一个转义序列时，编译器可以正确地对其进行解释。

```
public class Test {

    public static void main(String args[]) {
        System.out.println("She said \"Hello!\" to me.");
    }
}
```

以上实例编译运行结果如下：

```
She said "Hello!" to me.
```

Character 方法

下面是 Character 类的方法：

序号	方法与描述
1	isLetter() 是否是一个字母
2	isDigit() 是否是一个数字字符
3	isWhitespace() 是否一个空格

4 isUpperCase()

是否是大写字母

5 isLowerCase()

是否是小写字母

6 toUpperCase()

指定字母的大写形式

7 toLowerCase()

指定字母的小写形式

8 toString()

返回字符的字符串形式，字符串的长度仅为 1

JAVA 培训教程—Java Number 类

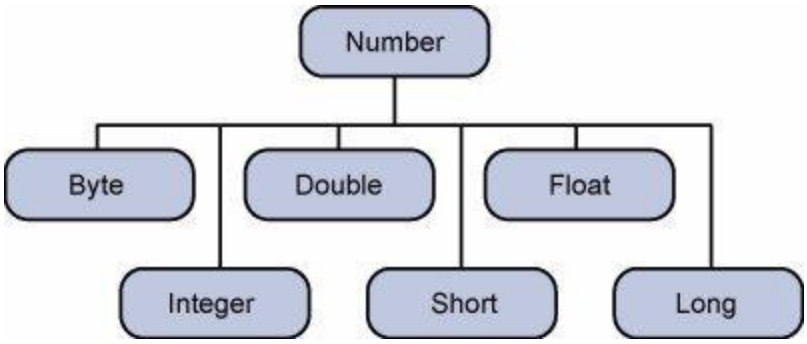
一般地，当需要使用数字的时候，我们通常使用内置数据类型，如：byte、int、long、double 等。

实例

```
1    int  i =  5000;
2    float  gpa =  13.65;
3    byte  mask =  0xaf;
```

然而，在实际开发过程中，我们经常会遇到需要使用对象，而不是内置数据类型的情形。为了解决这个问题，Java 语言为每一个内置数据类型提供了对应的包装类。

所有的包装类（Integer、Long、Byte、Double、Float、Short）都是抽象类 Number 的子类。



这种由编译器特别支持的包装称为装箱，所以当内置数据类型被当作对象使用的时候，编译器会把内置类型装箱为包装类。相似的，编译器也可以把一个对象拆箱为内置类型。Number 类属于 java.lang 包。

下面是一个装箱与拆箱的例子：

```
1    public  class  Test{
2
3          public  static  void  main(String args[]){
4                Integer x =  5;  // boxes int to an Integer object
```

```
5                x =   x +  10;       // unboxes the Integer to a int
6                System.out.println(x);
7          }
8    }
```

以上实例编译运行结果如下：

```
1
```

当 x 被赋为整型值时，由于 x 是一个对象，所以编译器要对 x 进行装箱。然后，为了使 x 能进行加运算，所以要对 x 进行拆箱。

Number 类的成员方法

下面的表中列出的是 Number 类的方法：

序号	方法与描述
1	xxxValue() 将 number 对象转换为 xxx 数据类型的值并返回。
2	compareTo() 将 number 对象与参数比较。
3	equals() 判断 number 对象是否与参数相等。
4	valueOf() 返回一个 Integer 对象指定的内置数据类型

5	toString()	以字符串形式返回值。
6	parseInt()	将字符串解析为 int 类型。
7	abs()	返回参数的绝对值。
8	ceil()	对整形变量向左取整，返回类型为 double 型。
9	floor()	对整型变量向右取整。返回类型为 double 类型。
10	rint()	返回与参数最接近的整数。返回类型为 double。
11	round()	返回一个最接近的 int、long 型值。
12	min()	返回两个参数中的最小值。
13	max()	

返回两个参数中的最大值。

14 exp()

返回自然数底数 e 的参数次方。

15 log()

返回参数的自然数底数的对数值。

16 pow()

返回第一个参数的第二个参数次方。

17 sqrt()

求参数的算术平方根。

18 sin()

求指定 double 类型参数的正弦值。

19 cos()

求指定 double 类型参数的余弦值。

20 tan()

求指定 double 类型参数的正切值。

21 asin()

求指定 double 类型参数的反正弦值。

22	acos() 求指定 double 类型参数的反余弦值。
23	atan() 求指定 double 类型参数的反正切值。
24	atan2() 将笛卡尔坐标转换为极坐标，并返回极坐标的角度值。
25	toDegrees() 将参数转化为角度。
26	toRadians() 将角度转换为弧度。
27	random() 返回一个随机数。

JAVA 培训教程—Java 异常处理

异常是程序中的一些错误，但并不是所有的错误都是异常，并且错误有时候是可以避免的。

比如说，你的代码少了一个分号，那么运行出来结果是提示是错误 java.lang.Error；如果你用 System.out.println(11/0)，那么你是因为你用 0 做了除数，会抛出 java.lang.ArithmeticException 的异常。

异常发生的原因有很多，通常包含以下几大类：

- 用户输入了非法数据。
- 要打开的文件不存在。
- 网络通信时连接中断，或者 JVM 内存溢出。

这些异常有的是因为用户错误引起，有的是程序错误引起的，还有其它一些是因为物理错误引起的。-

要理解 Java 异常处理是如何工作的，你需要掌握以下三种类型的异常：

- **检查性异常：**最具代表的检查性异常是用户错误或问题引起的异常，这是程序员无法预见的。例如要打开一个不存在文件时，一个异常就发生了，这些异常在编译时不能被简单地忽略。
- **运行时异常：** 运行时异常是可能被程序员避免的异常。与检查性异常相反，运行时异常可以在编译时被忽略。
- **错误：** 错误不是异常，而是脱离程序员控制的问题。错误在代码中通常被忽略。例如，当栈溢出时，一个错误就发生了，它们在编译也检查不到的。

Exception 类的层次

所有的异常类是从 java.lang.Exception 类继承的子类。

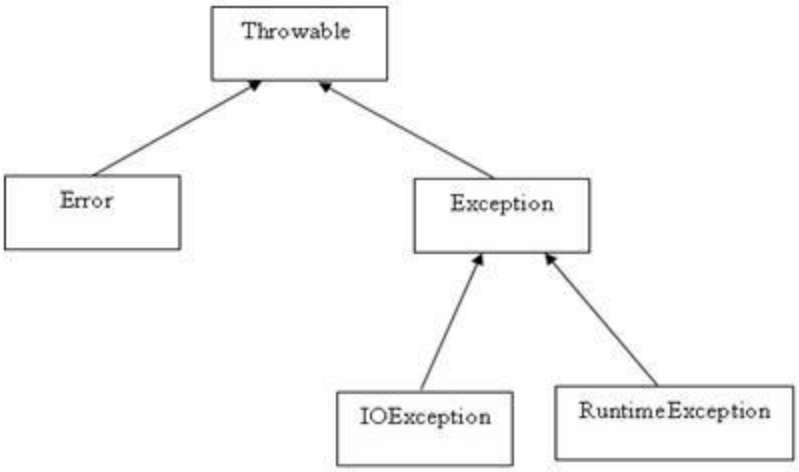
Exception 类是 Throwable 类的子类。除了 Exception 类外，Throwable 还有一个子类 Error 。

Java 程序通常不捕获错误。错误一般发生在严重故障时，它们在 Java 程序处理的范畴之外。

Error 用来指示运行时环境发生的错误。

例如，JVM 内存溢出。一般地，程序不会从错误中恢复。

异常类有两个主要的子类：IOException 类和 RuntimeException 类。



在 Java 内置类中(接下来会说明)，有大部分常用检查性和非检查性异常。

Java 内置异常类

Java 语言定义了一些异常类在 java.lang 标准包中。

标准运行时异常类的子类是最常见的异常类。由于 java.lang 包是默认加载到所有的 Java 程序的，所以大部分从运行时异常类继承而来的异常都可以直接使用。

Java 根据各个类库也定义了一些其他的异常，下面的表中列出了 Java 的非检查性异常。

异常	描述
ArithmeticException	当出现异常的运算条件时，抛出此异常。例如，一个整数"除以零"时，抛出此类的一个实例。
ArrayIndexOutOfBoundsException	用非法索引访问数组时抛出的异常。如果索引为负或大于等于数组大小，则该索引为非法索引。
ArrayStoreException	试图将错误类型的对象存储到一个对象数组时抛出的异常。
ClassCastException	当试图将对象强制转换为不是实例的子类时，抛出该异常。
IllegalArgumentException	抛出的异常表明向方法传递了一个不合法或不正确的参数。
IllegalMonitorStateException	抛出的异常表明某一线程已经试图等待对象的监视器，或者试图通知其他正在等待对象的监视器而本身没有指定监视器的线程。
IllegalStateException	在非法或不适当的时间调用方法时产生的信号。换句话说，即 Java 环境或 Java 应用程序没有处于请求操作所要求的适当状态下。

IllegalThreadStateException	线程没有处于请求操作所要求的适当状态时抛出的异常。
IndexOutOfBoundsException	指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。
NegativeArraySizeException	如果应用程序试图创建大小为负的数组，则抛出该异常。
NullPointerException	当应用程序试图在需要对象的地方使用 `null` 时，抛出该异常
NumberFormatException	当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。
SecurityException	由安全管理器抛出的异常，指示存在安全侵犯。
StringIndexOutOfBoundsException	此异常由 `String` 方法抛出，指示索引或者为负，或者超出字符串的大小。
UnsupportedOperationException	当不支持请求的操作时，抛出该异常。

下面的表中列出了 Java 定义在 java.lang 包中的检查性异常类。

异常	描述
ClassNotFoundException	应用程序试图加载类时，找不到相应的类，抛出该异常。
CloneNotSupportedException	当调用 Object 类中的 clone 方法克隆对象，但该对象的类无法实现 Cloneable 接口时，抛出该异常。
IllegalAccessException	拒绝访问一个类的时候，抛出该异常。
InstantiationException	当试图使用 Class 类中的 newInstance 方法创建一个类的实例，而指定的类对象因为是一个接口或是一个抽象类而无法实例化时，抛出该异常。
InterruptedException	一个线程被另一个线程中断，抛出该异常。
NoSuchFieldException	请求的变量不存在
NoSuchMethodException	请求的方法不存在

异常方法

下面的列表是 Throwable 类的主要方法:

序号　方法及说明

1 public String getMessage()

返回关于发生的异常的详细信息。这个消息在 Throwable 类的构造函数中初始化了。

2 public Throwable getCause()

返回一个 Throwable 对象代表异常原因。

3 public String toString()

使用 getMessage()的结果返回类的串级名字。

4 public void printStackTrace()

打印 toString()结果和栈层次到 System.err，即错误输出流。

5 public StackTraceElement [] getStackTrace()

返回一个包含堆栈层次的数组。下标为 0 的元素代表栈顶，最后一个元素代表方法调用堆栈的栈底。

6 public Throwable fillInStackTrace()

用当前的调用栈层次填充 Throwable 对象栈层次，添加到栈层次任何先前信息中。

捕获异常

使用 try 和 catch 关键字可以捕获异常。try/catch 代码块放在异常可能发生的地方。

try/catch 代码块中的代码称为保护代码，使用 try/catch 的语法如下：

```
1  try
2  {
```

```
3          // 程序代码
4    }catch(ExceptionName e1)
5    {
6          //Catch 块
7    }
```

Catch 语句包含要捕获异常类型的声明。当保护代码块中发生一个异常时，try 后面的 catch 块就会被检查。

如果发生的异常包含在 catch 块中，异常会被传递到该 catch 块，这和传递一个参数到方法是一样。

实例

下面的例子中声明有两个元素的一个数组，当代码试图访问数组的第三个元素的时候就会抛出一个异常。

```
1    // 文件名 : ExcepTest.java
2    import  java.io.*;
3    public  class  ExcepTest{
4
5          public  static  void  main(String args[]){
6                try{
7                      int  a[] =  new  int[2];
8                      System.out.println("Access element three :"  + a[3]);
9                }catch(ArrayIndexOutOfBoundsException e){
10                     System.out.println("Exception thrown   :"  + e);
11               }
12               System.out.println("Out of the block");
13         }
14   }
```

以上代码编译运行输出结果如下：

```
1    Exception thrown    :java.lang.ArrayIndexOutOfBoundsException:  3
2    Out of the block
```

多重捕获块

一个 try 代码块后面跟随多个 catch 代码块的情况就叫多重捕获。

多重捕获块的语法如下所示：

```
1    try{
2            // 程序代码
3    }catch(异常类型 1   异常的变量名 1){
4            // 程序代码
5    }catch(异常类型 2   异常的变量名 2){
6            // 程序代码
7    }catch(异常类型 2   异常的变量名 2){
8            // 程序代码
9    }
```

上面的代码段包含了 3 个 catch 块。

可以在 ry 语句后面添加任意数量的 catch 块。

如果保护代码中发生异常，异常被抛给第一个 catch 块。

如果抛出异常的数据类型与 ExceptionType1 匹配，它在这里就会被捕获。

如果不匹配，它会被传递给第二个 catch 块。

如此，直到异常被捕获或者通过所有的 catch 块。

实例

该实例展示了怎么使用多重 try/catch。

```
1    try
2    {
3            file =  new  FileInputStream(fileName);
4            x = (byte) file.read();
5    }catch(IOException i)
6    {
7            i.printStackTrace();
8            return  -1;
9    }catch(FileNotFoundException f)  //Not valid!
10   {
11           f.printStackTrace();
12           return  -1;
13   }
```

throws/throw 关键字：

如果一个方法没有捕获一个检查性异常，那么该方法必须使用 throws 关键字来声明。throws 关键字放在方法签名的尾部。

也可以使用 throw 关键字抛出一个异常，无论它是新实例化的还是刚捕获到的。

下面方法的声明抛出一个 RemoteException 异常：

```
1    import  java.io.*;
2    public  class  className
3    {
```

```
4          public  void  deposit(double  amount)  throws  RemoteException
5          {
6                // Method implementation
7                throw  new  RemoteException();
8          }
9          //Remainder of class definition
10   }
```

一个方法可以声明抛出多个异常，多个异常之间用逗号隔开。

例如，下面的方法声明抛出 RemoteException 和 InsufficientFundsException：

```
1    import  java.io.*;
2    public  class  className
3    {
4          public  void  withdraw(double  amount)  throws  RemoteException,
5                                                          InsufficientFundsException
6          {
7                  // Method implementation
8          }
9          //Remainder of class definition
10   }
```

finally 关键字

finally 关键字用来创建在 try 代码块后面执行的代码块。

无论是否发生异常，finally 代码块中的代码总会被执行。

在 finally 代码块中，可以运行清理类型等收尾善后性质的语句。

finally 代码块出现在 catch 代码块最后，语法如下：

```
try{
        // 程序代码
}catch(异常类型 1  异常的变量名 1){
        // 程序代码
}catch(异常类型 2  异常的变量名 2){
        // 程序代码
}finally{
        // 程序代码
}
```

实例

```
 public  class  ExcepTest{

        public  static  void  main(String args[]){
                int  a[] =  new  int[2];
                try{
                        System.out.println("Access element three :"  + a[3]);
                }catch(ArrayIndexOutOfBoundsException e){
                        System.out.println("Exception thrown   :"  + e);
                }
                finally{
                        a[0] =  6;
                        System.out.println("First element value: "  +a[0]);
                        System.out.println("The finally statement is executed");
                }
        }
}
```

以上实例编译运行结果如下：

```
1    Exception thrown    :java.lang.ArrayIndexOutOfBoundsException:  3
2    First element value:  6
3    The  finally  statement is executed
```

注意下面事项：

- catch 不能独立于 try 存在。
- 在 try/catch 后面添加 finally 块并非强制性要求的。
- try 代码后不能既没 catch 块也没 finally 块。
- try, catch, finally 块之间不能添加任何代码。

声明自定义异常

在 Java 中你可以自定义异常。编写自己的异常类时需要记住下面的几点。

- 所有异常都必须是 Throwable 的子类。
- 如果希望写一个检查性异常类，则需要继承 Exception 类。
- 如果你想写一个运行时异常类，那么需要继承 RuntimeException 类。

可以像下面这样定义自己的异常类：

```
1    class  MyException  extends  Exception{
2    }
```

只继承 Exception 类来创建的异常类是检查性异常类。

下面的 InsufficientFundsException 类是用户定义的异常类，它继承自 Exception。

一个异常类和其它任何类一样，包含有变量和方法。

实例

```
1     // 文件名 InsufficientFundsException.java
2     import  java.io.*;
```

```
3
4    public  class  InsufficientFundsException  extends  Exception
5    {
6          private  double  amount;
7          public  InsufficientFundsException(double  amount)
8          {
9                this.amount = amount;
10         }
11         public  double  getAmount()
12         {
13               return  amount;
14         }
15   }
```

为了展示如何使用我们自定义的异常类，

在下面的 CheckingAccount 类中包含一个 withdraw()方法抛出一个 InsufficientFundsException 异常。

```
1    // 文件名称 CheckingAccount.java
2    import  java.io.*;
3
4    public  class  CheckingAccount
5    {
6          private  double  balance;
7          private  int  number;
8          public  CheckingAccount(int  number)
9          {
10               this.number = number;
11         }
```

```
12          public  void  deposit(double  amount)
13          {
14                  balance += amount;
15          }
16          public  void  withdraw(double  amount)  throws
17                                                          InsufficientFundsException
18          {
19                  if(amount <= balance)
20                  {
21                          balance -= amount;
22                  }
23                  else
24                  {
25                          double  needs = amount - balance;
26                          throw  new  InsufficientFundsException(needs);
27                  }
28          }
29          public  double  getBalance()
30          {
31                  return  balance;
32          }
33          public  int  getNumber()
34          {
35                  return  number;
36          }
37  }
```

下面的 BankDemo 程序示范了如何调用 CheckingAccount 类的 deposit() 和 withdraw()方法。

```
1    //文件名称 BankDemo.java
2    public  class  BankDemo
3    {
4          public  static  void  main(String [] args)
5          {
6                CheckingAccount c =  new  CheckingAccount(101);
7                System.out.println("Depositing $500...");
8                c.deposit(500.00);
9                try
10               {
11                     System.out.println("\nWithdrawing $100...");
12                     c.withdraw(100.00);
13                     System.out.println("\nWithdrawing $600...");
14                     c.withdraw(600.00);
15               }catch(InsufficientFundsException e)
16               {
17                     System.out.println("Sorry, but you are short $"
18                                                                    + e.getAmount());
19                     e.printStackTrace();
20               }
21          }
22   }
```

编译上面三个文件，并运行程序 BankDemo，得到结果如下所示：

```
1    Depositing $500...
2
3    Withdrawing $100...
4
5    Withdrawing $600...
```

```
6    Sorry, but you are  short  $200.0
7    InsufficientFundsException
8                    at CheckingAccount.withdraw(CheckingAccount.java:25)
9                    at BankDemo.main(BankDemo.java:13)
```

通用异常

在 Java 中定义了两种类型的异常和错误。

- **JVM(Java 虚拟机)异常：**由 JVM 抛出的异常或错误。例如：NullPointerException 类，ArrayIndexOutOfBoundsException 类，ClassCastException 类。
- **程序级异常：**由程序或者 API 程序抛出的异常。例如 IllegalArgumentException 类，IllegalStateException 类。

JAVA 培训教程—Java 对象和类

Java 作为一种面向对象语言。支持以下基本概念：

1、多态

2、继承

3、封装

4、抽象

5、类

6、对象

7、实例

8、方法

9、消息解析

本节我们重点研究对象和类的概念。

对象：对象是类的一个实例，有状态和行为。例如，一条狗是一个对象，它的状态有：颜色、名字、品种；行为有：摇尾巴、叫、吃等。

类：类是一个模板，它描述一类对象的行为和状态。

Java 中的对象

现在让我们深入了解什么是对象。看看周围真实的世界，会发现身边有很多对象，车，狗，人等等。所有这些对象都有自己的状态和行为。

拿一条狗来举例，它的状态有：名字、品种、颜色，行为有：叫、摇尾巴和跑。

对比现实对象和软件对象，它们之间十分相似。

软件对象也有状态和行为。软件对象的状态就是属性，行为通过方法体现。

在软件开发中，方法操作对象内部状态的改变，对象的相互调用也是通过方法来完成。

Java 中的类

类可以看成是创建 Java 对象的模板。

通过下面一个简单的类来理解下 Java 中类的定义：

```
1    public  class  Dog{
2          String breed;
3          int  age;
4          String color;
5          void  barking(){
6          }
7
8          void  hungry(){
9          }
10
11         void  sleeping(){
12         }
13   }
```

一个类可以包含以下类型变量：

局部变量：在方法、构造方法或者语句块中定义的变量被称为局部变量。变量声明和初始化都是在方法中，方法结束后，变量就会自动销毁。

成员变量：成员变量是定义在类中，方法体之外的变量。这种变量在创建对象的时候实例化。成员变量可以被类中方法、构造方法和特定类的语句块访问。

类变量：类变量也声明在类中，方法体之外，但必须声明为 static 类型。

一个类可以拥有多个方法，在上面的例子中：barking()、hungry()和 sleeping()都是 Dog 类的方法。

构造方法

每个类都有构造方法。如果没有显式地为类定义构造方法，Java 编译器将会为该类提供一个默认构造方法。

在创建一个对象的时候，至少要调用一个构造方法。构造方法的名称必须与类同名，一个类可以有多个构造方法。

下面是一个构造方法示例：

```
public  class  Puppy{
        public  puppy(){
        }

        public  puppy(String name){
                // 这个构造器仅有一个参数：name
        }
}
```

创建对象

对象是根据类创建的。在 Java 中，使用关键字 new 来创建一个新的对象。创建对象需要以下三步：

声明：声明一个对象，包括对象名称和对象类型。

实例化：使用关键字 new 来创建一个对象。

初始化：使用 new 创建对象时，会调用构造方法初始化对象。

下面是一个创建对象的例子：

```
1    public  class  Puppy{
2          public  Puppy(String name){
3                //这个构造器仅有一个参数：name
4                System.out.println("Passed Name is :"  + name );
5          }
6          public  static  void  main(String []args){
7                // 下面的语句将创建一个 Puppy 对象
8                Puppy myPuppy =  new  Puppy(  "tommy"  );
9          }
10   }
```

编译并运行上面的程序，会打印出下面的结果：

```
Passed Name is :tommy
```

访问实例变量和方法

通过已创建的对象来访问成员变量和成员方法，如下所示：

```
/* 实例化对象 */
ObjectReference = new Constructor();
/* 访问其中的变量 */
ObjectReference.variableName;
/* 访问类中的方法 */
ObjectReference.MethodName();
```

实例

下面的例子展示如何访问实例变量和调用成员方法：

```
1    public  class  Puppy{
```

```
2          int  puppyAge;
3          public  Puppy(String name){
4                // 这个构造器仅有一个参数：name
5                System.out.println("Passed Name is :"  + name );
6          }
7
8          public  void  setAge(  int  age ){
9                  puppyAge = age;
10         }
11
12         public  int  getAge( ){
13                 System.out.println("Puppy's age is :"  + puppyAge );
14                 return  puppyAge;
15         }
16
17         public  static  void  main(String []args){
18               /* 创建对象 */
19               Puppy myPuppy = new Puppy( "tommy" );
20               /* 通过方法来设定 age */
21               myPuppy.setAge( 2 );
22               /* 调用另一个方法获取 age */
23               myPuppy.getAge( );
24               /*你也可以像下面这样访问成员变量 */
25               System.out.println("Variable Value :"  + myPuppy.puppyAge );
26         }
27   }
```

编译并运行上面的程序，产生如下结果：

```
Passed Name is :tommy
```

```
Puppy's age is :2
Variable Value :2
```

源文件声明规则

在本节的最后部分，我们将学习源文件的声明规则。当在一个源文件中定义多个类，并且还有 import 语句和 package 语句时，要特别注意这些规则。

1、一个源文件中只能有一个 public 类

2、一个源文件可以有多个非 public 类

3、源文件的名称应该和 public 类的类名保持一致。例如：源文件中 public 类的类名是 Employee，那么源文件应该命名为 Employee.java。

4、如果一个类定义在某个包中，那么 package 语句应该在源文件的首行。

5、如果源文件包含 import 语句，那么应该放在 package 语句和类定义之间。如果没有 package 语句，那么 import 语句应该在源文件中最前面。

6、import 语句和 package 语句对源文件中定义的所有类都有效。在同一源文件中，不能给不同的类不同的包声明。

类有若干种访问级别，并且类也分不同的类型：抽象类和 final 类等。这些将在访问控制章节介绍。

除了上面提到的几种类型，Java 还有一些特殊的类，如：内部类、匿名类。

Java 包

包主要用来对类和接口进行分类。当开发 Java 程序时，可能编写成百上千的类，因此很有必要对类和接口进行分类。

Import 语句

在 Java 中，如果给出一个完整的限定名，包括包名、类名，那么 Java 编译器就可以很容易地定位到源代码或者类。Import 语句就是用来提供一个合理的路径，使得编译器可以找到某个类。

例如，下面的命令行将会命令编译器载入 java_installation/java/io 路径下的所有类

```
import  java.io.*;
```

JAVA 培训教程—Java 方法

Java 方法是语句的集合，它们在一起执行一个功能。

- 方法是解决一类问题的步骤的有序组合
- 方法包含于类或对象中
- 方法在程序中被创建，在其他地方被引用

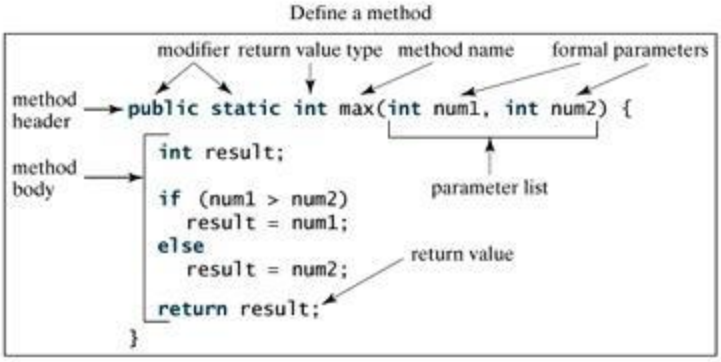
方法的定义

一般情况下，定义一个方法包含以下语法：

```
1    修饰符 返回值类型 方法名 (参数类型 参数名){
2            ...
3            方法体
4            ...
5            return   返回值;
6    }
```

方法包含一个方法头和一个方法体。下面是一个方法的所有部分：

- **修饰符：**修饰符，这是可选的，告诉编译器如何调用该方法。定义了该方法的访问类型。
- **返回值类型** ：方法可能会返回值。returnValueType 是方法返回值的数据类型。有些方法执行所需的操作，但没有返回值。在这种情况下，returnValueType 是关键字 void。
- **方法名：**是方法的实际名称。方法名和参数表共同构成方法签名。
- **参数类型：**参数像是一个占位符。当方法被调用时，传递值给参数。这个值被称为实参或变量。参数列表是指方法的参数类型、顺序和参数的个数。参数是可选的，方法可以不包含任何参数。
- **方法体：**方法体包含具体的语句，定义该方法的功能。



如：

```
1    public  static  int  age(int  birthday){...}
```

参数可以有多个：

```
1    static  float  interest(float  principal,  int  year){...}
```

注意： 在一些其它语言中方法指过程和函数。一个返回非 void 类型返回值的方法称为函数；一个返回 void 类型返回值的方法叫做过程。

实例

下面的方法包含 2 个参数 num1 和 num2，它返回这两个参数的最大值。

```
1     /** 返回两个整型变量数据的较大值 */
2     public  static  int  max(int  num1,  int  num2) {
3           int  result;
4           if  (num1 > num2)
5                 result = num1;
6           else
7                 result = num2;
8
9           return  result;
10    }
```

方法调用

Java 支持两种调用方法的方式，根据方法是否返回值来选择。

当程序调用一个方法时，程序的控制权交给了被调用的方法。当被调用方法的返回语句执行或者到达方法体闭括号时候交还控制权给程序。

当方法返回一个值的时候，方法调用通常被当做一个值。例如：

```
1    int  larger = max(30,  40);
```

如果方法返回值是 void，方法调用一定是一条语句。例如，方法 println 返回 void。下面的调用是个语句：

```
1    System.out.println("Welcome to Java!");
```

示例

下面的例子演示了如何定义一个方法，以及如何调用它：

```
1    public  class  TestMax {
2          /** 主方法 */
3          public  static  void  main(String[] args) {
4                int  i =  5;
5                int  j =  2;
6                int  k = max(i, j);
7                System.out.println("The maximum between "  + i +
8                                          " and "  + j +  " is "  + k);
9          }
10
11         /** 返回两个整数变量较大的值 */
12         public  static  int  max(int  num1,  int  num2) {
13               int  result;
```

```
14                if  (num1 > num2)
15                      result = num1;
16                else
17                      result = num2;
18
19                return  result;
20          }
21    }
```

以上实例编译运行结果如下：

```
1    The maximum between  5  and  2  is  5
```

这个程序包含 main 方法和 max 方法。Main 方法是被 JVM 调用的，除此之外，main 方法和其它方法没什么区别。

main 方法的头部是不变的，如例子所示，带修饰符 public 和 static,返回 void 类型值，方法名字是 main,此外带个一个 String[]类型参数。String[]表明参数是字符串数组。

void 关键字

本节说明如何声明和调用一个 void 方法。

下面的例子声明了一个名为 printGrade 的方法，并且调用它来打印给定的分数。

示例

```
1     public  class  TestVoidMethod {
2
3           public  static  void  main(String[] args) {
4                 printGrade(78.5);
5           }
6
```

```
7          public  static  void  printGrade(double  score) {
8                if  (score >=  90.0) {
9                      System.out.println('A');
10               }
11               else  if  (score >=  80.0) {
12                     System.out.println('B');
13               }
14               else  if  (score >=  70.0) {
15                     System.out.println('C');
16               }
17               else  if  (score >=  60.0) {
18                     System.out.println('D');
19               }
20               else  {
21                     System.out.println('F');
22               }
23         }
24   }
```

以上实例编译运行结果如下：

```
1   C
```

这里 printGrade 方法是一个 void 类型方法，它不返回值。

一个 void 方法的调用一定是一个语句。 所以，它被在 main 方法第三行以语句形式调用。就像任何以分号结束的语句一样。

通过值传递参数

调用一个方法时候需要提供参数，你必须按照参数列表指定的顺序提供。

例如，下面的方法连续 n 次打印一个消息：

```
1   public  static  void  nPrintln(String message,  int  n) {
2       for  (int  i =  0; i < n; i++)
3           System.out.println(message);
4   }
```

示例

下面的例子演示按值传递的效果。

该程序创建一个方法，该方法用于交换两个变量。

```
1    public  class  TestPassByValue {
2
3         public  static  void  main(String[] args) {
4               int  num1 =  1;
5               int  num2 =  2;
6
7               System.out.println("Before swap method, num1 is "  +
8                                                   num1 +  " and num2 is "  + num2);
9
10              // 调用 swap 方法
11              swap(num1, num2);
12              System.out.println("After swap method, num1 is "  +
13                                                  num1 +  " and num2 is "  + num2);
14        }
15        /** 交换两个变量的方法 */
16        public  static  void  swap(int  n1,  int  n2) {
17              System.out.println("\tInside the swap method");
18              System.out.println("\t\tBefore swapping n1 is "  + n1
```

```
19                                                  +  " n2 is "  + n2);
20              // 交换 n1 与 n2 的值
21              int  temp = n1;
22              n1 = n2;
23              n2 = temp;
24
25              System.out.println("\t\tAfter swapping n1 is "  + n1
26                                                  +  " n2 is "  + n2);
27          }
28    }
```

以上实例编译运行结果如下：

```
1    Before swap method, num1 is  1  and num2 is  2
2                    Inside the swap method
3                                   Before swapping n1 is  1  n2 is  2
4                                   After swapping n1 is  2  n2 is  1
5    After swap method, num1 is  1  and num2 is  2
```

传递两个参数调用 swap 方法。有趣的是，方法被调用后，实参的值并没有改变。

方法的重载

上面使用的 max 方法仅仅适用于 int 型数据。但如果你想得到两个浮点类型数据的最大值呢？

解决方法是创建另一个有相同名字但参数不同的方法，如下面代码所示：

```
1    public  static  double  max(double  num1,  double  num2) {
2        if  (num1 > num2)
3            return  num1;
4        else
```

```
5           return  num2;
6   }
```

如果你调用 max 方法时传递的是 int 型参数，则 int 型参数的 max 方法就会被调用；

如果传递的事 double 型参数，则 double 类型的 max 方法体会被调用，这叫做方法重载；

就是说一个类的两个方法拥有相同的名字，但是有不同的参数列表。

Java 编译器根据方法签名判断哪个方法应该被调用。

方法重载可以让程序更清晰易读。执行密切相关任务的方法应该使用相同的名字。

重载的方法必须拥有不同的参数列表。你不能仅仅依据修饰符或者返回类型的不同来重载方法。

变量作用域

变量的范围是程序中该变量可以被引用的部分。

方法内定义的变量被称为局部变量。

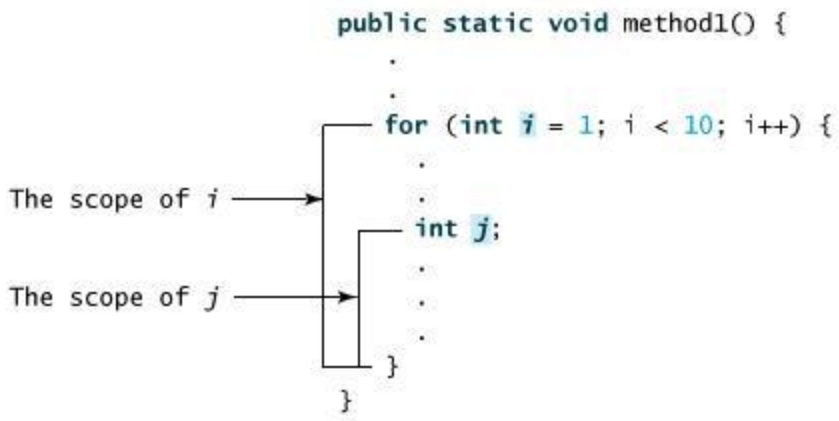
局部变量的作用范围从声明开始，直到包含它的块结束。

局部变量必须声明才可以使用。

方法的参数范围涵盖整个方法。参数实际上是一个局部变量。

for 循环的初始化部分声明的变量，其作用范围在整个循环。

但循环体内声明的变量其适用范围是从它声明到循环体结束。它包含如下所示的变量声明：



你可以在一个方法里，不同的非嵌套块中多次声明一个具有相同的名称局部变量，但你不能在嵌套块内两次声明局部变量。

命令行参数的使用

有时候你希望运行一个程序时候再传递给它消息。这要靠传递命令行参数给 main()函数实现。

命令行参数是在执行程序时候紧跟在程序名字后面的信息。

实例

下面的程序打印所有的命令行参数：

```
1   public  class  CommandLine {
2
3         public  static  void  main(String args[]){
4               for(int  i=0; i<args.length; i++){
5                     System.out.println("args[" + i + "]: " +
6                                                                    args[i
7               }
8         }
9   }
```

如下所示，运行这个程序：

```
1    java CommandLine  this  is a command line  200  -100
```

运行结果如下：

```
1    args[0]:  this
2    args[1]: is
3    args[2]: a
4    args[3]: command
5    args[4]: line
6    args[5]:  200
7    args[6]: -100
```

构造方法

当一个对象被创建时候，构造方法用来初始化该对象。构造方法和它所在类的名字相同，但构造方法没有返回值。

通常会使用构造方法给一个类的实例变量赋初值，或者执行其它必要的步骤来创建一个完整的对象。

不管你与否自定义构造方法，所有的类都有构造方法，因为 Java 自动提供了一个默认构造方法，它把所有成员初始化为 0。

一旦你定义了自己的构造方法，默认构造方法就会失效。

实例

下面是一个使用构造方法的例子：

```
1    // 一个简单的构造函数
2    class  MyClass {
3          int  x;
```

```
4
5         // 以下是构造函数
6         MyClass() {
7               x =  10;
8         }
9   }
```

你可以像下面这样调用构造方法来初始化一个对象：

```
1   public  class  ConsDemo {
2
3         public  static  void  main(String args[]) {
4               MyClass t1 =  new  MyClass();
5               MyClass t2 =  new  MyClass();
6               System.out.println(t1.x +  " "  + t2.x);
7         }
8   }
```

大多时候需要一个有参数的构造方法。

实例

下面是一个使用构造方法的例子：

```
1   // 一个简单的构造函数
2   class  MyClass {
3         int  x;
4
5         // 以下是构造函数
6         MyClass(int  i ) {
7               x = i;
8         }
```

```
9    }
```

你可以像下面这样调用构造方法来初始化一个对象：

```
1    public  class  ConsDemo {
2
3          public  static  void  main(String args[]) {
4                MyClass t1 =  new  MyClass(  10  );
5                MyClass t2 =  new  MyClass(  20  );
6                System.out.println(t1.x +  " "  + t2.x);
7          }
8    }
```

运行结果如下：

```
1    10  20
```

可变参数

JDK 1.5 开始，Java 支持传递同类型的可变参数给一个方法。

方法的可变参数的声明如下所示：

```
1    typeName... parameterName
```

在方法声明中，在指定参数类型后加一个省略号(...) 。

一个方法中只能指定一个可变参数，它必须是方法的最后一个参数。任何普通的参数必须在它之前声明。

实例

```
1     public  class  VarargsDemo {
2
3           public  static  void  main(String args[]) {
```

```
4                 // 调用可变参数的方法
5                 printMax(34,  3,  3,  2,  56.5);
6                 printMax(new  double[]{1,  2,  3});
7           }
8
9           public  static  void  printMax(  double... numbers) {
10          if  (numbers.length ==  0) {
11                System.out.println("No argument passed");
12                return;
13          }
14
15          double  result = numbers[0];
16
17          for  (int  i =  1; i <   numbers.length; i++)
18                if  (numbers[i] >   result)
19                result = numbers[i];
20                System.out.println("The max value is "  + result);
21          }
22    }
```

以上实例编译运行结果如下：

```
1    The max value is  56.5
2    The max value is  3.0
```

finalize() 方法

Java 允许定义这样的方法，它在对象被垃圾收集器析构(回收)之前调用，这个方法叫做 finalize( )，它用来清除回收对象。

例如，你可以使用 finalize()来确保一个对象打开的文件被关闭了。

在 finalize()方法里，你必须指定在对象销毁时候要执行的操作。

finalize()一般格式是：

```
1    protected  void  finalize()
2    {
3          // 在这里终结代码
4    }
```

关键字 protected 是一个限定符，它确保 finalize() 方法不会被该类以外的代码调用。

当然，Java 的内存回收可以由 JVM 来自动完成。如果你手动使用，则可以使用上面的方法。

实例

```
1     public  class  FinalizationDemo {
2             public  static  void  main(String[] args) {
3                     Cake c1 =  new  Cake(1);
4                     Cake c2 =  new  Cake(2);
5                     Cake c3 =  new  Cake(3);
6
7                     c2 = c3 =  null;
8                     System.gc();  //调用 Java 垃圾收集器
9             }
10    }
11
12    class  Cake  extends  Object {
13            private  int  id;
14            public  Cake(int  id) {
15                    this.id = id;
```

```
16                  System.out.println("Cake Object "  + id +  "is created");
17          }
18
19          protected  void  finalize()  throws  java.lang.Throwable {
20                  super.finalize();
21                  System.out.println("Cake Object "  + id +  "is disposed");
22          }
23  }
```

运行以上代码，输出结果如下：

```
1   C:\1>java FinalizationDemo
2   Cake Object 1is created
3   Cake Object 2is created
4   Cake Object 3is created
5   Cake Object 3is disposed
6   Cake Object 2is disposed
```

JAVA 培训教程—Java 修饰符

Java 语言提供了很多修饰符，主要分为以下两类：

- 访问修饰符
- 非访问修饰符

修饰符用来定义类、方法或者变量，通常放在语句的最前端。我们通过下面的例子来说明：

```
1   public  class  className {
2         // ...
3   }
4   private  boolean  myFlag;
5   static  final  double  weeks =  9.5;
6   protected  static  final  int  BOXWIDTH =  42;
7   public  static  void  main(String[] arguments) {
8         // 方法体
9   }
```

访问控制修饰符

Java 中，可以使用访问控制符来保护对类、变量、方法和构造方法的访问。Java 支持 4 种不同的访问权限。

默认的，也称为 default，在同一包内可见，不使用任何修饰符。

私有的，以 private 修饰符指定，在同一类内可见。

共有的，以 public 修饰符指定，对所有类可见。

受保护的，以 protected 修饰符指定，对同一包内的类和所有子类可见。

默认访问修饰符-不使用任何关键字

使用默认访问修饰符声明的变量和方法，对同一个包内的类是可见的。接口里的变量都隐式声明为 public static final,而接口里的方法默认情况下访问权限为 public。

实例：

如下例所示，变量和方法的声明可以不使用任何修饰符。

```
1   String version = "1.5.1";
2   boolean processOrder() {
3       return true;
4   }
```

私有访问修饰符-private

私有访问修饰符是最严格的访问级别，所以被声明为 private 的方法、变量和构造方法只能被所属类访问，并且类和接口不能声明为 private。

声明为私有访问类型的变量只能通过类中公共的 getter 方法被外部类访问。

Private 访问修饰符的使用主要用来隐藏类的实现细节和保护类的数据。

下面的类使用了私有访问修饰符：

```
1   public class Logger {
2       private String format;
3       public String getFormat() {
4           return this.format;
5       }
6       public void setFormat(String format) {
7           this.format = format;
8       }
```

```
9    }
```

实例中，Logger 类中的 format 变量为私有变量，所以其他类不能直接得到和设置该变量的值。为了使其他类能够操作该变量，定义了两个 public 方法：getFormat() （返回 format 的值）和 setFormat(String)（设置 format 的值）

公有访问修饰符-public

被声明为 public 的类、方法、构造方法和接口能够被任何其他类访问。

如果几个相互访问的 public 类分布在不用的包中，则需要导入相应 public 类所在的包。由于类的继承性，类所有的公有方法和变量都能被其子类继承。

以下函数使用了公有访问控制：

```
1    public  static  void  main(String[] arguments) {
2          // ...
3    }
```

Java 程序的 main() 方法必须设置成公有的，否则，Java 解释器将不能运行该类。

受保护的访问修饰符-protected

被声明为 protected 的变量、方法和构造器能被同一个包中的任何其他类访问，也能够被不同包中的子类访问。

Protected 访问修饰符不能修饰类和接口，方法和成员变量能够声明为 protected，但是接口的成员变量和成员方法不能声明为 protected。

子类能访问 Protected 修饰符声明的方法和变量，这样就能保护不相关的类使用这些方法和变量。

下面的父类使用了 protected 访问修饰符，子类重载了父类的 openSpeaker()方法。

```
1    class  AudioPlayer {
2          protected  boolean  openSpeaker(Speaker sp) {
3                // 实现细节
4          }
5    }
6
7    class  StreamingAudioPlayer {
8          boolean  openSpeaker(Speaker sp) {
9                // 实现细节
10         }
11   }
```

如果把 openSpeaker()方法声明为 private，那么除了 AudioPlayer 之外的类将不能访问该方法。如果把 openSpeaker()声明为 public，那么所有的类都能够访问该方法。如果我们只想让该方法对其所在类的子类可见，则将该方法声明为 protected。

访问控制和继承

请注意以下方法继承的规则：

- 父类中声明为 public 的方法在子类中也必须为 public。
- 父类中声明为 protected 的方法在子类中要么声明为 protected，要么声明为 public。不能声明为 private。
- 父类中默认修饰符声明的方法，能够在子类中声明为 private。
- 父类中声明为 private 的方法，不能够被继承。

非访问修饰符

为了实现一些其他的功能，Java 也提供了许多非访问修饰符。

static 修饰符，用来创建类方法和类变量。

Final 修饰符，用来修饰类、方法和变量，final 修饰的类不能够被继承，修饰的方法不能被继承类重新定义，修饰的变量为常量，是不可修改的。

Abstract 修饰符，用来创建抽象类和抽象方法。

Synchronized 和 volatile 修饰符，主要用于线程的编程。

Static 修饰符

- **静态变量：**

  Static 关键字用来声明独立于对象的静态变量，无论一个类实例化多少对象，它的静态变量只有一份拷贝。 静态变量也被成为类变量。局部变量能被声明为 static 变量。

- **静态方法：**

  Static 关键字用来声明独立于对象的静态方法。静态方法不能使用类的非静态变量。静态方法从参数列表得到数据，然后计算这些数据。

对类变量和方法的访问可以直接使用 classname.variablename 和 classname.methodname 的方式访问。

如下例所示，static 修饰符用来创建类方法和类变量。

```
1    public  class  InstanceCounter {
2          private  static  int  numInstances =  0;
3          protected  static  int  getCount() {
4                return  numInstances;
5          }
6
7          private  static  void  addInstance() {
8                numInstances++;
```

```
9           }
10
11          InstanceCounter() {
12                  InstanceCounter.addInstance();
13          }
14
15          public  static  void  main(String[] arguments) {
16                  System.out.println("Starting with "  +
17                  InstanceCounter.getCount() +  " instances");
18                  for  (int  i =  0; i <  500; ++i){
19                        new  InstanceCounter();
20                          }
21                  System.out.println("Created "  +
22                  InstanceCounter.getCount() +  " instances");
23          }
24    }
```

以上实例运行编辑结果如下:

```
1    Started with  0  instances
2    Created  500  instances
```

Final 修饰符

Final 变量:

Final 变量能被显式地初始化并且只能初始化一次。被声明为 final 的对象的引用不能指向不同的对象。但是 final 对象里的数据可以被改变。也就是说 final 对象的引用不能改变，但是里面的值可以改变。

Final 修饰符通常和 static 修饰符一起使用来创建类常量。

实例:

```
1    public  class  Test{
2        final  int  value =  10;
3        // 下面是声明常量的实例
4        public  static  final  int  BOXWIDTH =  6;
5        static  final  String TITLE =  "Manager";
6
7        public  void  changeValue(){
8              value =  12;  //将输出一个错误
9        }
10   }
```

Final 方法

类中的 Final 方法可以被子类继承，但是不能被子类修改。

声明 final 方法的主要目的是防止该方法的内容被修改。

如下所示，使用 final 修饰符声明方法。

```
1   public  class  Test{
2           public  final  void  changeName(){
3                 // 方法体
4           }
5   }
```

Final 类

Final 类能被继承，没有类能够继承 final 类的任何特性。

实例:

```
1   public  final  class  Test {
```

```
2          // 类体
3    }
```

Abstract 修饰符

抽象类：

抽象类不能用来实例化对象，声明抽象类的唯一目的是为了将来对该类进行扩充。

一个类不能同时被 abstract 和 final 修饰。如果一个类包含抽象方法，那么该类一定要声明为抽象类，否则将出现编译错误。

抽象类可以包含抽象方法和非抽象方法。

实例：

```
1    abstract  class  Caravan{
2          private  double  price;
3          private  String model;
4          private  String year;
5          public  abstract  void  goFast();  //抽象方法
6          public  abstract  void  changeColor();
7    }
```

抽象方法

抽象方法是一种没有任何实现的方法，该方法的的具体实现由子类提供。抽象方法不能被声明成 final 和 strict。

任何继承抽象类的子类必须实现父类的所有抽象方法，除非该子类也是抽象类。

如果一个类包含若干个抽象方法，那么该类必须声明为抽象类。抽象类可以不包含抽象方法。

抽象方法的声明以分号结尾，例如：public abstract sample();

实例：

```
1    public  abstract  class  SuperClass{
2            abstract  void  m();  //抽象方法
3    }
4
5    class  SubClass  extends  SuperClass{
6              //实现抽象方法
7                void  m(){
8                        .........
9                }
10   }
```

Synchronized 修饰符

Synchronized 关键字声明的方法同一时间只能被一个线程访问。Synchronized 修饰符可以应用于四个访问修饰符。

实例：

```
1   public  synchronized  void  showDetails(){
2   .......
3   }
```

Transient 修饰符

序列化的对象包含被 transient 修饰的实例变量时，java 虚拟机(JVM)跳过该特定的变量。

该修饰符包含在定义变量的语句中，用来预处理类和变量的数据类型。

实例：

```
1   public  transient  int  limit =  55;      // will not persist
2   public  int  b;  // will persist
```

volatile 修饰符

Volatile 修饰的成员变量在每次被线程访问时，都强迫从共享内存中重读该成员变量的值。而且，当成员变量发生变化时，强迫线程将变化值回写到共享内存。这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。一个 volatile 对象引用可能是 null。

实例：

```
1    public  class  MyRunnable  implements  Runnable
2    {
3            private  volatile  boolean  active;
4            public  void  run()
5            {
6                    active =  true;
7                    while  (active)  // line 1
8                    {
9                            // 代码
10                   }
11           }
12           public  void  stop()
13           {
14                   active =  false;  // line 2
15           }
16   }
```

一般地，在一个线程中调用 run()方法，在另一个线程中调用 stop()方法。如果 line 1 中的 active 位于缓冲区的值被使用，那么当把 line 2 中的 active 设置成 false 时，循环也不会停止。

JAVA 培训教程—Java 封装

在面向对象程式设计方法中，封装（英语：Encapsulation）是指，一种将抽象性函式接口的实作细节部份包装、隐藏起来的方法。

封装可以被认为是一个保护屏障，防止该类的代码和数据被外部类定义的代码随机访问。

要访问该类的代码和数据，必须通过严格的接口控制。

封装最主要的功能在于我们能修改自己的实现代码，而不用修改那些调用我们代码的程序片段。

适当的封装可以让程式码更容易理解与维护，也加强了程式码的安全性。

实例

让我们来看一个 java 封装类的例子：

```
1    /* 文件名: EncapTest.java */
2    public  class  EncapTest{
3
4          private  String name;
5          private  String idNum;
6          private  int  age;
7
8          public  int  getAge(){
9                return  age;
10         }
11
12         public  String getName(){
13               return  name;
```

```
14          }
15
16          public  String getIdNum(){
17                  return  idNum;
18          }
19
20          public  void  setAge(  int  newAge){
21                  age = newAge;
22          }
23
24          public  void  setName(String newName){
25                  name = newName;
26          }
27
28          public  void  setIdNum( String newId){
29                  idNum = newId;
30          }
31  }
```

以上实例中 public 方法是外部类访问该类成员变量的入口。

通常情况下，这些方法被称为 getter 和 setter 方法。

因此，任何要访问类中私有成员变量的类都要通过这些 getter 和 setter 方法。

通过如下的例子说明 EncapTest 类的变量怎样被访问：

```
1   /* F 文件名 : RunEncap.java */
2   public  class  RunEncap{
3
4           public  static  void  main(String args[]){
```

```
5                EncapTest encap =  new  EncapTest();
6                encap.setName("James");
7                encap.setAge(20);
8                encap.setIdNum("12343ms");
9
10               System.out.print("Name : "  + encap.getName()+
11                                                          " Age : "+ encap.getAge());
12           }
13   }
```

以上代码编译运行结果如下:

```
1    Name : James Age :  20
```

JAVA 培训教程—Java 继承

继承是 java 面向对象编程技术的一块基石，因为它允许创建分等级层次的类。继承可以理解为一个对象从另一个对象获取属性的过程。

如果类 A 是类 B 的父类，而类 B 是类 C 的父类，我们也称 C 是 A 的子类，类 C 是从类 A 继承而来的。在 Java 中，类的继承是单一继承，也就是说，一个子类只能拥有一个父类

继承中最常使用的两个关键字是 extends 和 implements。

这两个关键字的使用决定了一个对象和另一个对象是否是 IS-A(是一个)关系。

通过使用这两个关键字，我们能实现一个对象获取另一个对象的属性。

所有 Java 的类均是由 java.lang.Object 类继承而来的，所以 Object 是所有类的祖先类，而除了 Object 外，所有类必须有一个父类。

通过过 extends 关键字可以申明一个类是继承另外一个类而来的，一般形式如下：

```
1    // A.java
2    public  class  A {
3            private  int  i;
4            protected  int  j;
5
6            public  void  func() {
7
8            }
9    }
10
11   // B.java
12   public  class  B  extends  A {
13   }
```

以上的代码片段说明，B 由 A 继承而来的，B 是 A 的子类。而 A 是 Object 的子类，这里可以不显示地声明。

作为子类，B 的实例拥有 A 所有的成员变量，但对于 private 的成员变量 B 却没有访问权限，这保障了 A 的封装性。

IS-A 关系

IS-A 就是说:一个对象是另一个对象的一个分类。

下面是使用关键字 extends 实现继承。

```
1    public  class  Animal{
2    }
3
4    public  class  Mammal  extends  Animal{
5    }
6
7    public  class  Reptile  extends  Animal{
8    }
9
10   public  class  Dog  extends  Mammal{
11   }
```

基于上面的例子，以下说法是正确的：

- Animal 类是 Mammal 类的父类。
- Animal 类是 Reptile 类的父类。
- Mammal 类和 Reptile 类是 Animal 类的子类。
- Dog 类既是 Mammal 类的子类又是 Animal 类的子类。

分析以上示例中的 IS-A 关系，如下：

- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal

因此 ： Dog IS-A Animal

通过使用关键字 **extends**，子类可以继承父类的除 private 属性外所有的属性。

我们通过使用 instanceof 操作符，能够确定 Mammal IS-A Animal

实例

```
1    public  class  Dog  extends  Mammal{
2
3          public  static  void  main(String args[]){
4
5                Animal a =  new  Animal();
6                Mammal m =  new  Mammal();
7                Dog d =  new  Dog();
8
9                System.out.println(m  instanceof  Animal);
10               System.out.println(d  instanceof  Mammal);
11               System.out.println(d  instanceof  Animal);
12         }
13   }
```

以上实例编译运行结果如下：

```
1    true
```

```
2    true
3    true
```

介绍完 **extends** 关键字之后，我们再来看下 **implements** 关键字是怎样使用来表示 IS-A 关系。

Implements 关键字使用在类继承接口的情况下， 这种情况不能使用关键字 **extends**。

实例

```
1    public  interface  Animal {}
2
3    public  class  Mammal  implements  Animal{
4    }
5
6    public  class  Dog  extends  Mammal{
7    }
```

instanceof 关键字

可以使用 **instanceof** 运算符来检验 Mammal 和 dog 对象是否是 Animal 类的一个实例。

```
1     interface  Animal{}
2
3     class  Mammal  implements  Animal{}
4
5     public  class  Dog  extends  Mammal{
6           public  static  void  main(String args[]){
7
8                 Mammal m =  new  Mammal();
9                 Dog d =  new  Dog();
10
```

```
11              System.out.println(m  instanceof  Animal);
12              System.out.println(d  instanceof  Mammal);
13              System.out.println(d  instanceof  Animal);
14          }
15    }
```

以上实例编译运行结果如下：

```
1    true
2    true
3    true
```

HAS-A 关系

HAS-A 代表类和它的成员之间的从属关系。这有助于代码的重用和减少代码的错误。

例子

```
1    public  class  Vehicle{}
2    public  class  Speed{}
3    public  class  Van  extends  Vehicle{
4           private  Speed sp;
5    }
```

Van 类和 Speed 类是 HAS-A 关系(Van 有一个 Speed)，这样就不用将 Speed 类的全部代码粘贴到 Van 类中了，并且 Speed 类也可以重复利用于多个应用程序。

在面向对象特性中，用户不必担心类的内部怎样实现。

Van 类将实现的细节对用户隐藏起来，因此，用户只需要知道怎样调用 Van 类来完成某一功能，而不必知道 Van 类是自己来做还是调用其他类来做这些工作。

Java 只支持单继承，也就是说，一个类不能继承多个类。

下面的做法是不合法的：

```
1   public  class  extends  Animal, Mammal{}
```

Java 只支持单继承（继承基本类和抽象类），但是我们可以用接口来实现（多继承接口来实现）,脚本结构如：

```
1   public  class  Apple  extends  Fruit  implements  Fruit1, Fruit2{}
```

一般我们继承基本类和抽象类用 extends 关键字,实现接口类的继承用 implements 关键字。

JAVA 培训教程—Java 重写(Override)与重载(Overload)

重写(Override)重写是子类对父类的允许访问的方法的实现过程进行重新编写!返回值和形参都不能改变。即外壳不变，核心重写！重写的好处在于子类可以根据需要，定义特定于自己的行为。也就是说子类能够根据需要实现父类的方法。

在面向对象原则里，重写意味着可以重写任何现有方法。实例如下：

```
1    class  Animal{
2
3          public  void  move(){
4                System.out.println("动物可以移动");
5          }
6    }
7
8    class  Dog  extends  Animal{
9
10         public  void  move(){
11               System.out.println("狗可以跑和走");
12         }
13   }
14
15   public  class  TestDog{
16
17         public  static  void  main(String args[]){
18               Animal a =  new  Animal();  // Animal 对象
19               Animal b =  new  Dog();  // Dog 对象
20
21               a.move();// 执行 Animal 类的方法
22
```

```
23                b.move();//执行 Dog 类的方法
24          }
25    }
```

以上实例编译运行结果如下：

```
1     动物可以移动
2     狗可以跑和走
```

在上面的例子中可以看到，尽管 b 属于 Animal 类型，但是它运行的是 Dog 类的 move 方法。

这是由于在编译阶段，只是检查参数的引用类型。

然而在运行时，Java 虚拟机(JVM)指定对象的类型并且运行该对象的方法。

因此在上面的例子中，之所以能编译成功，是因为 Animal 类中存在 move 方法，然而运行时，运行的是特定对象的方法。

思考以下例子：

```
1     class  Animal{
2
3           public  void  move(){
4                 System.out.println("动物可以移动");
5           }
6     }
7
8     class  Dog  extends  Animal{
9
10          public  void  move(){
11                System.out.println("狗可以跑和走");
12          }
```

```
13          public  void  bark(){
14                  System.out.println("狗可以吠叫");
15          }
16    }
17
18    public  class  TestDog{
19
20          public  static  void  main(String args[]){
21                  Animal a =  new  Animal();  // Animal 对象
22                  Animal b =  new  Dog();  // Dog 对象
23
24                  a.move();// 执行 Animal 类的方法
25                  b.move();//执行 Dog 类的方法
26                  b.bark();
27          }
28    }
```

以上实例编译运行结果如下：

```
1    TestDog.java:30: cannot find symbol
2    symbol    : method bark()
3    location:  class  Animal
4                                       b.bark();
5                                         ^
```

该程序将抛出一个编译错误，因为 b 的引用类型 Animal 没有 bark 方法。

方写重写的规则

- 参数列表必须完全与被重写方法的相同；
- 返回类型必须完全与被重写方法的返回类型相同；

- 访问权限不能比父类中被重写的方法的访问权限更高。例如：如果父类的一个方法被声明为 public，那么在子类中重写该方法就不能声明为 protected。
- 父类的成员方法只能被它的子类重写。
- 声明为 final 的方法不能被重写。
- 声明为 static 的方法不能被重写，但是能够被再次声明。
- 如果一个方法不能被继承，那么该方法不能被重写。
- 子类和父类在同一个包中，那么子类可以重写父类所有方法，除了声明为 private 和 final 的方法。
- 子类和父类不在同一个包中，那么子类只能够重写父类的声明为 public 和 protected 的非 final 方法。
- 重写的方法能够抛出任何非强制异常，无论被重写的方法是否抛出异常。但是，重写的方法不能抛出新的强制性异常，或者比被重写方法声明的更广泛的强制性异常，反之则可以。
- 构造方法不能被重写。
- 如果不能继承一个方法，则不能重写这个方法。

Super 关键字的使用

当需要在子类中调用父类的被重写方法时，要使用 super 关键字。

```
1    class  Animal{
2
3          public  void  move(){
4                System.out.println("动物可以移动);
5          }
6    }
7
8    class  Dog  extends  Animal{
9
10         public  void  move(){
```

```
11                super.move();  // 应用 super 类的方法
12                System.out.println("狗可以跑和走");
13        }
14    }
15
16    public  class  TestDog{
17
18        public  static  void  main(String args[]){
19
20                Animal b =  new  Dog(); /
21                b.move();  //执行 Dog 类的方法
22
23        }
24    }
```

以上实例编译运行结果如下：

```
1    动物可以移动
2    狗可以跑和走
```

重载(Overload)

重载(overloading) 是在一个类里面，方法名字相同，而参数不同。返回类型呢？可以相同也可以不同。

每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。

只能重载构造函数

重载规则

- 被重载的方法必须改变参数列表；
- 被重载的方法可以改变返回类型；

- 被重载的方法可以改变访问修饰符；
- 被重载的方法可以声明新的或更广的检查异常；
- 方法能够在同一个类中或者在一个子类中被重载。

实例

```
1    public  class  Overloading {
2
3            public  int  test(){
4                    System.out.println("test1");
5                    return  1;
6            }
7
8            public  void  test(int  a){
9                    System.out.println("test2");
10           }
11
12           //以下两个参数类型顺序不同
13           public  String test(int  a,String s){
14                   System.out.println("test3");
15                   return  "returntest3";
16           }
17
18           public  String test(String s,int  a){
19                   System.out.println("test4");
20                   return  "returntest4";
21           }
22
23           public  static  void  main(String[] args){
24                   Overloading o =  new  Overloading();
```

```
25                  System.out.println(o.test());
26                  o.test(1);
27                  System.out.println(o.test(1,"test3"));
28                  System.out.println(o.test("test4",1));
29              }
```

重写与重载之间的区别

区别点	重载方法	重写方法
参数列表	必须修改	一定不能修改
返回类型	可以修改	一定不能修改
异常	可以修改	可以减少或删除，一定不能抛出新的或者更广的异常
访问	可以修改	一定不能做更严格的限制（可以降低限制）

JAVA 培训教程—Java 抽象类

在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。

抽象类除了不能实例化对象之外，类的其它功能依然存在，成员变量、成员方法和构造方法的访问方式和普通类一样。

由于抽象类不能实例化对象，所以抽象类必须被继承，才能被使用。也是因为这个原因，通常在设计阶段决定要不要设计抽象类。

父类包含了子类集合的常见的方法，但是由于父类本身是抽象的，所以不能使用这些方法。

抽象类

在 Java 语言中使用 abstract class 来定义抽象类。如下实例：

```
1    /* 文件名 : Employee.java */
2    public  abstract  class  Employee
3    {
4          private  String name;
5          private  String address;
6          private  int  number;
7          public  Employee(String name, String address,  int  number)
8          {
9                System.out.println("Constructing an Employee");
10               this.name = name;
11               this.address = address;
12               this.number = number;
13         }
```

```
14          public  double  computePay()
15          {
16              System.out.println("Inside Employee computePay");
17              return  0.0;
18          }
19          public  void  mailCheck()
20          {
21                System.out.println("Mailing a check to "  +  this.name
22                  +  " "  +  this.address);
23          }
24          public  String toString()
25          {
26                return  name +  " "  + address +  " "  + number;
27          }
28          public  String getName()
29          {
30                return  name;
31          }
32          public  String getAddress()
33          {
34                return  address;
35          }
36          public  void  setAddress(String newAddress)
37          {
38                address = newAddress;
39          }
40          public  int  getNumber()
41          {
42              return  number;
```

```
43          }
44    }
```

注意到该 Employee 类没有什么不同，尽管该类是抽象类，但是它仍然有 3 个成员变量，7 个成员方法和 1 个构造方法。 现在如果你尝试如下的例子：

```
1     /* 文件名 : AbstractDemo.java */
2     public class AbstractDemo
3     {
4           public static void main(String [] args)
5           {
6                 /* 以下是不允许的，会引发错误 */
7                 Employee e =  new  Employee("George W.",  "Houston, TX",  43);
8
9                 System.out.println("\n Call mailCheck using Employee reference--");
10                e.mailCheck();
11            }
12    }
```

当你尝试编译 AbstractDemo 类时，会产生如下错误：

```
1    Employee.java:46: Employee is  abstract; cannot be instantiated
2                 Employee e =  new  Employee("George W.",  "Houston, TX",  43);
3                                          ^
4    1  error
```

继承抽象类

我们能通过一般的方法继承 Employee 类：

```
1     /* 文件名 : Salary.java */
2     public  class  Salary  extends  Employee
```

```
3    {
4          private  double  salary;  //Annual salary
5          public  Salary(String name, String address,  int  number,  double
6                salary)
7          {
8                  super(name, address, number);
9                  setSalary(salary);
10         }
11         public  void  mailCheck()
12         {
13                 System.out.println("Within mailCheck of Salary class ");
14                 System.out.println("Mailing check to "  + getName()
15                 +  " with salary "  + salary);
16         }
17         public  double  getSalary()
18         {
19                 return  salary;
20         }
21         public  void  setSalary(double  newSalary)
22         {
23                 if(newSalary >=  0.0)
24                 {
25                       salary = newSalary;
26                 }
27         }
28         public  double  computePay()
29         {
30               System.out.println("Computing salary pay for "  + getName());
31               return  salary/52;
```

```
32          }
33    }
```

尽管我们不能实例化一个 Employee 类的对象，但是如果我们实例化一个 Salary 类对象，该对象将从 Employee 类继承 3 个成员变量和 7 个成员方法。

```
1     /* 文件名 : AbstractDemo.java */
2     public  class  AbstractDemo
3     {
4           public  static  void  main(String [] args)
5           {
6                 Salary s =  new  Salary("Mohd Mohtashim",  "Ambehta, UP",  3,  3600.00);
7                 Employee e =  new  Salary("John Adams",  "Boston, MA",  2,  2400.00);
8
9                 System.out.println("Call mailCheck using Salary reference --");
10                s.mailCheck();
11
12                System.out.println("\n Call mailCheck using Employee reference--");
13                e.mailCheck();
14            }
15    }
```

以上程序编译运行结果如下：

```
1    Constructing an Employee
2    Constructing an Employee
3    Call mailCheck using   Salary reference --
4    Within mailCheck of Salary  class
5    Mailing check to Mohd Mohtashim with salary  3600.0
6
7    Call mailCheck using Employee reference--
```

```
8    Within mailCheck of Salary  class
9    Mailing check to John Adams with salary  2400.
```

抽象方法

如果你想设计这样一个类，该类包含一个特别的成员方法，该方法的具体实现由它的子类确定，那么你可以在父类中声明该方法为抽象方法。

Abstract 关键字同样可以用来声明抽象方法，抽象方法只包含一个方法名，而没有方法体。

抽象方法没有定义，方法名后面直接跟一个分号，而不是花括号。

```
1    public  abstract  class  Employee
2    {
3          private  String name;
4          private  String address;
5          private  int  number;
6
7          public  abstract  double  computePay();
8
9          //其余代码
10   }
```

声明抽象方法会造成以下两个结果：

- 如果一个类包含抽象方法，那么该类必须是抽象类。
- 任何子类必须重写父类的抽象方法，或者声明自身为抽象类。

继承抽象方法的子类必须重载该方法。否则，该子类也必须声明为抽象类。最终，必须有子类实现该抽象方法，否则，从最初的父类到最终的子类都不能用来实例化对象。

如果 Salary 类继承了 Employee 类，那么它必须实现 computePay()方法：

```
1    /* 文件名 : Salary.java */
2    public  class  Salary  extends  Employee
3    {
4          private  double  salary;  // Annual salary
5
6          public  double  computePay()
7          {
8                System.out.println("Computing salary pay for "  + getName());
9                return  salary/52;
10         }
11
12         //其余代码
13   }
```

JAVA 培训教程—Java 接口

接口（英文：Interface），在 JAVA 编程语言中是一个抽象类型，是抽象方法的集合，接口通常以 interface 来声明。一个类通过继承接口的方式，从而来继承接口的抽象方法。

接口并不是类，编写接口的方式和类很相似，但是它们属于不同的概念。类描述对象的属性和方法。接口则包含类要实现的方法。

实现接口的类是抽象类，否则该类要定义接口中的所有方法。

接口无法被实例化，但是可以被实现。一个实现接口的类，必须实现接口内所描述的所有方法，否则就必须声明为抽象类。另外，在 Java 中，接口类型可用来声明一个变量，他们可以成为一个空指针，或是被绑定在一个以此接口实现的对象。

接口与类相似点：

- 一个接口可以有多个方法。
- 接口文件保存在.java 结尾的文件中，文件名使用接口名。
- 接口的字节码文件保存在.class 结尾的文件中。
- 接口相应的字节码文件必须在与包名称相匹配的目录结构中。

接口与类的区别：

- 接口不能用于实例化对象。
- 接口没有构造方法。
- 接口中所有的方法必须是抽象方法。
- 接口不能包含成员变量，除了 static 和 final 变量。
- 接口不是被类继承了，而是要被类实现。
- 接口支持多重继承。

接口的声明

接口的声明语法格式如下：

```
1   [可见度] interface 接口名称 [extends 其他的类名] {
2                   // 声明变量
3                   // 抽象方法
4   }
```

Interface 关键字用来声明一个接口。下面是接口声明的一个简单例子。

```
1   /* 文件名 : NameOfInterface.java */
2   import java.lang.*;
3   //引入包
4
5   public interface NameOfInterface
6   {
7         //任何类型 final, static 字段
8         //抽象方法
9   }
```

接口有以下特性：

- 接口是隐式抽象的，当声明一个接口的时候，不必使用 **abstract** 关键字。
- 接口中每一个方法也是隐式抽象的，声明时同样不需要 **abstract** 关键子。
- 接口中的方法都是公有的。

实例

```
1   /* 文件名 : Animal.java */
2   interface Animal {
3
4         public void eat();
5         public void travel();
6   }
```

接口的实现

当类实现接口的时候，类要实现接口中所有的方法。否则，类必须声明为抽象的类。

类使用 implements 关键字实现接口。在类声明中，Implements 关键字放在 class 声明后面。

实现一个接口的语法，可以使用这个公式：

```
...  implements  接口名称[, 其他接口, 其他接口..., ...] ...
```

实例

```
/* 文件名 : MammalInt.java */
public  class  MammalInt  implements  Animal{

        public  void  eat(){
                System.out.println("Mammal eats");
        }

        public  void  travel(){
                System.out.println("Mammal travels");
        }

        public  int  noOfLegs(){
                return  0;
        }

        public  static  void  main(String args[]){
                MammalInt m =  new  MammalInt();
                m.eat();
                m.travel();
        }
}
```

以上实例编译运行结果如下:

```
1    Mammal eats
2    Mammal travels
```

重写接口中声明的方法时，需要注意以下规则：

- 类在实现接口的方法时，不能抛出强制性异常，只能在接口中，或者继承接口的抽象类中抛出该强制性异常。
- 类在重写方法时要保持一致的方法名，并且应该保持相同或者相兼容的返回值类型。
- 如果实现接口的类是抽象类，那么就没必要实现该接口的方法。

在实现接口的时候，也要注意一些规则：

- 一个类可以同时实现多个接口。
- 一个类只能继承一个类，但是能实现多个接口。
- 一个接口能继承另一个接口，这和类之间的继承比较相似。

接口的继承

一个接口能继承另一个接口，和类之间的继承方式比较相似。接口的继承使用 extends 关键字，子接口继承父接口的方法。

下面的 Sports 接口被 Hockey 和 Football 接口继承：

```
1    // 文件名: Sports.java
2    public  interface  Sports
3    {
4          public  void  setHomeTeam(String name);
5          public  void  setVisitingTeam(String name);
6    }
7
8    // 文件名: Football.java
```

```
9    public  interface  Football  extends  Sports
10   {
11         public  void  homeTeamScored(int  points);
12         public  void  visitingTeamScored(int  points);
13         public  void  endOfQuarter(int  quarter);
14   }
15
16   // 文件名: Hockey.java
17   public  interface  Hockey  extends  Sports
18   {
19         public  void  homeGoalScored();
20         public  void  visitingGoalScored();
21         public  void  endOfPeriod(int  period);
22         public  void  overtimePeriod(int  ot);
23   }
```

Hockey 接口自己声明了四个方法，从 Sports 接口继承了两个方法，这样，实现 Hockey 接口的类需要实现六个方法。

相似的，实现 Football 接口的类需要实现五个方法，其中两个来自于 Sports 接口。

接口的多重继承

在 Java 中，类的多重继承是不合法，但接口允许多重继承，。

在接口的多重继承中 extends 关键字只需要使用一次，在其后跟着继承接口。 如下所示：

```
1    public  interface  Hockey  extends  Sports, Event
```

以上的程序片段是合法定义的子接口，与类不同的是，接口允许多重继承，而 Sports 及 Event 可能定义或是继承相同的方法

标记接口

最常用的继承接口是没有包含任何方法的接口。

标识接口是没有任何方法和属性的接口.它仅仅表明它的类属于一个特定的类型,供其他代码来测试允许做一些事情。

标识接口作用：简单形象的说就是给某个对象打个标（盖个戳），使对象拥有某个或某些特权。

例如：java.awt.event 包中的 MouseListener 接口继承的 java.util.EventListener 接口定义如下：

```
1    package  java.util;
2    public  interface  EventListener
3    {}
```

没有任何方法的接口被称为标记接口。标记接口主要用于以下两种目的：

- **建立一个公共的父接口：**

  正如 EventListener 接口，这是由几十个其他接口扩展的 Java API，你可以使用一个标记接口来建立一组接口的父接口。例如：当一个接口继承了 EventListener 接口，Java 虚拟机(JVM)就知道该接口将要被用于一个事件的代理方案。

- **向一个类添加数据类型：**

  这种情况是标记接口最初的目的,实现标记接口的类不需要定义任何接口方法(因为标记接口根本就没有方法)，但是该类通过多态性变成一个接口类型。

JAVA 培训教程—Java 多态

多态是同一个行为具有多个不同表现形式或形态的能力。多态性是对象多种表现形式的体现。比如我们说"宠物"这个对象，它就有很多不同的表达或实现，比如有小猫、小狗、蜥蜴等等。那么我到宠物店说"请给我一只宠物"，服务员给我小猫、小狗或者蜥蜴都可以，我们就说"宠物"这个对象就具备多态性。

接下来让我们通过实例来了解 Java 的多态。

例子

```
1   public  interface  Vegetarian{}
2   public  class  Animal{}
3   public  class  Deer  extends  Animal  implements  Vegetarian{}
```

因为 Deer 类具有多重继承，所以它具有多态性。以上实例解析如下：

- 一个 Deer IS-A（是一个） Animal
- 一个 Deer IS-A（是一个） Vegetarian
- 一个 Deer IS-A（是一个） Deer
- 一个 Deer IS-A（是一个）Object

在 Java 中，所有的对象都具有多态性，因为任何对象都能通过 IS-A 测试的类型和 Object 类。访问一个对象的唯一方法就是通过引用型变量。引用型变量只能有一种类型，一旦被声明，引用型变量的类型就不能被改变了。引用型变量不仅能够被重置为其他对象，前提是这些对象没有被声明为 final。还可以引用和它类型相同的或者相兼容的对象。它可以声明为类类型或者接口类型。

当我们将引用型变量应用于 Deer 对象的引用时，下面的声明是合法的：

```
1   Deer d =  new  Deer();
2   Animal a = d;
3   Vegetarian v = d;
```

```
4    Object o = d;
```

所有的引用型变量 d,a,v,o 都指向堆中相同的 Deer 对象。

虚方法

我们将介绍在 Java 中，当设计类时，被重载的方法的行为怎样影响多态性。已经讨论了方法的重载，也就是子类能够重载父类的方法。当子类对象调用重载的方法时，调用的是子类的方法，而不是父类中被重载的方法。

要想调用父类中被重载的方法，则必须使用关键字 super。

```
1    /* 文件名 : Employee.java */
2    public  class  Employee
3    {
4          private  String name;
5          private  String address;
6          private  int  number;
7          public  Employee(String name, String address,  int  number)
8          {
9                System.out.println("Constructing an Employee");
10               this.name = name;
11               this.address = address;
12               this.number = number;
13         }
14         public  void  mailCheck()
15         {
16               System.out.println("Mailing a check to "  +  this.name
17                 +  " "  +  this.address);
18         }
19         public  String toString()
```

```
20          {
21                return  name +  " "  + address +  " "  + number;
22          }
23          public  String getName()
24          {
25                return  name;
26          }
27          public  String getAddress()
28          {
29                return  address;
30          }
31          public  void  setAddress(String newAddress)
32          {
33                address = newAddress;
34          }
35          public  int  getNumber()
36          {
37              return  number;
38          }
39    }
```

假设下面的类继承 Employee 类：

```
1     /* 文件名 : Salary.java */
2     public  class  Salary  extends  Employee
3     {
4           private  double  salary;  //Annual salary
5           public  Salary(String name, String address,  int  number,  double
6                 salary)
7           {
```

```
8                   super(name, address, number);
9                   setSalary(salary);
10          }
11          public  void  mailCheck()
12          {
13                  System.out.println("Within mailCheck of Salary class ");
14                  System.out.println("Mailing check to "  + getName()
15                  +  " with salary "  + salary);
16          }
17          public  double  getSalary()
18          {
19                  return  salary;
20          }
21          public  void  setSalary(double  newSalary)
22          {
23                  if(newSalary >=  0.0)
24                  {
25                        salary = newSalary;
26                  }
27          }
28          public  double  computePay()
29          {
30                System.out.println("Computing salary pay for "  + getName());
31                return  salary/52;
32          }
33    }
```

现在我们仔细阅读下面的代码，尝试给出它的输出结果：

```
1     /* 文件名 : VirtualDemo.java */
```

```
2    public  class  VirtualDemo
3    {
4          public  static  void  main(String [] args)
5          {
6                Salary s =  new  Salary("Mohd Mohtashim",  "Ambehta, UP",  3,  3600.00);
7                Employee e =  new  Salary("John Adams",  "Boston, MA",  2,  2400.00);
8                System.out.println("Call mailCheck using Salary reference --");
9                s.mailCheck();
10               System.out.println("\n Call mailCheck using Employee reference--");
11               e.mailCheck();
12           }
13   }
```

以上实例编译运行结果如下：

```
1    Constructing an Employee
2    Constructing an Employee
3    Call mailCheck using Salary reference --
4    Within mailCheck of Salary  class
5    Mailing check to Mohd Mohtashim with salary  3600.0
6
7    Call mailCheck using Employee reference--
8    Within mailCheck of Salary  class
9    Mailing check to John Adams with salary  2400.0
```

例子中,我们实例化了两个 Salary 对象。一个使用 Salary 引用 s,另一个使用 Employee 引用。编译时,编译器检查到 mailCheck()方法在 Salary 类中的声明。在调用 s.mailCheck()时，Java 虚拟机(JVM)调用 Salary 类的 mailCheck()方法。因为 e 是 Employee 的引用，所以调用 e 的 mailCheck()方法则有完全不同的结果。当编译器检查 e.mailCheck()方法时，编译器检查到 Employee 类中的 mailCheck()方法。

在编译的时候，编译器使用 Employee 类中的 mailCheck()方法验证该语句， 但是在运行的时候，Java 虚拟机(JVM)调用的是 Salary 类中的 mailCheck()方法。该行为被称为虚拟方法调用，该方法被称为虚拟方法。

Java 中所有的方法都能以这种方式表现，借此，重写的方法能在运行时调用，不管编译的时候源代码中引用变量是什么数据类型。

JAVA 培训教程—Java 包(package)

为了更好地组织类，Java 提供了包机制，用于区别类名的命名空间。

包的作用

1、把功能相似或相关的类或接口组织在同一个包中，方便类的查找和使用。

2、如同文件夹一样，包也采用了树形目录的存储方式。同一个包中的类名字是不同的，不同的包中的类的名字是可以相同的，当同时调用两个不同包中相同类名的类时，应该加上包名加以区别。因此，包可以避免名字冲突。

3、包也限定了访问权限，拥有包访问权限的类才能访问某个包中的类。

Java 使用包（package）这种机制是为了防止命名冲突，访问控制，提供搜索和定位类（class）、接口、枚举（enumerations）和注释（annotation）等。

包语句的语法格式为：

```
1    package  pkg1[. pkg2[. pkg3…]];
```

例如,一个 Something.java 文件它的内容

```
1    package  net.java.util
2    public  class  Something{
3          ...
4    }
```

那么它的路径应该是 net/java/Something.java 这样保存的。 package(包)的作用是把不同的 java 程序分类保存，更方便的被其他 java 程序调用。

一个包（package）可以定义为一组相互联系的类型（类、接口、枚举和注释），为这些类型提供访问保护和命名空间管理的功能。

以下是一些 Java 中的包：

- java.lang-打包基础的类
- java.io-包含输入输出功能的函数

开发者可以自己把一组类和接口等打包，并定义自己的 package。而且在实际开发中这样做是值得提倡的，当你自己完成类的实现之后，将相关的类分组，可以让其他的编程者更容易地确定哪些类、接口、枚举和注释等是相关的。

由于 package 创建了新的命名空间（namespace），所以不会跟其他 package 中的任何名字产生命名冲突。使用包这种机制，更容易实现访问控制，并且让定位相关类更加简单。

创建包

创建 package 的时候，你需要为这个 package 取一个合适的名字。之后，如果其他的一个源文件包含了这个包提供的类、接口、枚举或者注释类型的时候，都必须将这个 package 的声明放在这个源文件的开头。

包声明应该在源文件的第一行，每个源文件只能有一个包声明，这个文件中的每个类型都应用于它。

如果一个源文件中没有使用包声明，那么其中的类，函数，枚举，注释等将被放在一个无名的包（unnamed package）中。

例子

让我们来看一个例子，这个例子创建了一个叫做 animals 的包。通常使用小写的字母来命名避免与类、接口名字的冲突。

在 animals 包中加入一个接口（interface）：

```
1   /* 文件名: Animal.java */
2   package  animals;
3
```

```
4   interface  Animal {
5         public  void  eat();
6         public  void  travel();
7   }
```

接下来，在同一个包中加入该接口的实现：

```
1    package  animals;
2
3    /* 文件名 : MammalInt.java */
4    public  class  MammalInt  implements  Animal{
5
6          public  void  eat(){
7                System.out.println("Mammal eats");
8          }
9
10         public  void  travel(){
11               System.out.println("Mammal travels");
12         }
13
14         public  int  noOfLegs(){
15               return  0;
16         }
17
18         public  static  void  main(String args[]){
19               MammalInt m =  new  MammalInt();
20               m.eat();
21               m.travel();
22         }
23   }
```

然后，编译这两个文件，并把他们放在一个叫做 animals 的子目录中。 用下面的命令来运行：

```
1    $ mkdir animals
2    $ cp Animal.class    MammalInt.class  animals
3    $ java animals/MammalInt
4    Mammal eats
5    Mammal travel
```

import 关键字

为了能够使用某一个包的成员，我们需要在 Java 程序中明确导入该包。使用"import"语句可完成此功能。

在 java 源文件中 import 语句应位于 package 语句之后，所有类的定义之前，可以没有，也可以有多条，其语法格式为：

```
1    import  package1[.package2…].(classname|*);
```

如果在一个包中，一个类想要使用本包中的另一个类，那么该包名可以省略。

例子

下面的 payroll 包已经包含了 Employee 类，接下来向 payroll 包中添加一个 Boss 类。Boss 类引用 Employee 类的时候可以不用使用 payroll 前缀，Boss 类的实例如下。

```
1    package  payroll;
2
3    public  class  Boss
4    {
5          public  void  payEmployee(Employee e)
6          {
7                e.mailCheck();
```

```
8           }
9    }
```

如果 Boss 类不在 payroll 包中又会怎样？Boss 类必须使用下面几种方法之一来引用其他包中的类。

使用类全名描述，例如：

```
1    payroll.Employee
```

用 import 关键字引入，使用通配符"*"

```
1    import  payroll.*;
```

使用 import 关键字引入 Employee 类

```
1    import  payroll.Employee;
```

注意：

类文件中可以包含任意数量的 import 声明。import 声明必须在包声明之后，类声明之前。

package 的目录结构

类放在包中会有两种主要的结果：

- 包名成为类名的一部分，正如我们前面讨论的一样。
- 包名必须与相应的字节码所在的目录结构相吻合。

下面是管理自己 java 中文件的一种简单方式：

将类、接口等类型的源码放在一个文本中，这个文件的名字就是这个类型的名字，并以.java 作为扩展名。例如：

```
1    // 文件名 :   Car.java
```

```
2
3   package  vehicle;
4
5   public  class  Car {
6         // 类实现
7   }
```

接下来，把源文件放在一个目录中，这个目录要对应类所在包的名字。

```
1   ....\vehicle\Car.java
```

现在，正确的类名和路径将会是如下样子：

- 类名 -> vehicle.Car
- 路径名 -> vehicle\Car.java (in windows)

通常，一个公司使用它互联网域名的颠倒形式来作为它的包名.例如：互联网域名是 apple.com，所有的包名都以 com.apple 开头。包名中的每一个部分对应一个子目录。

例如：这个公司有一个 com.apple.computers 的包，这个包包含一个叫做 Dell.java 的源文件，那么相应的，应该有如下面的一连串子目录：

```
1   ....\com\apple\computers\Dell.java
```

编译的时候，编译器为包中定义的每个类、接口等类型各创建一个不同的输出文件，输出文件的名字就是这个类型的名字，并加上.class 作为扩展后缀。 例如：

```
1   // 文件名: Dell.java
2
3   package  com.apple.computers;
4   public  class  Dell{
5
6   }
```

```
7    class  Ups{
8
9    }
```

现在，我们用-d 选项来编译这个文件，如下：

```
1    $javac -d . Dell.java
```

这样会像下面这样放置编译了的文件：

```
1    .\com\apple\computers\Dell.class.\com\apple\computers\Ups.class
```

你可以像下面这样来导入所有 \com\apple\computers\中定义的类、接口等：

```
1    import  com.apple.computers.*;
```

编译之后的.class 文件应该和.java 源文件一样，它们放置的目录应该跟包的名字对应起来。但是，并不要求.class 文件的路径跟相应的.java 的路径一样。你可以分开来安排源码和类的目录。

```
1    <path-one>\sources\com\apple\computers\Dell.java
2    <path-two>\classes\com\apple\computers\Dell.class
```

这样，你可以将你的类目录分享给其他的编程人员，而不用透露自己的源码。用这种方法管理源码和类文件可以让编译器和 java 虚拟机(JVM)可以找到你程序中使用的所有类型。

类目录的绝对路径叫做 class path。设置在系统变量 CLASSPATH 中。编译器和 java 虚拟机通过将 package 名字加到 class path 后来构造.class 文件的路径。

<path- two>\classes 是 class path，package 名字是 com.apple.computers,而编译器和 JVM 会在 <path-two>\classes\com\apple\compters 中找.class 文件。

一个 class path 可能会包含好几个路径。多路径应该用分隔符分开。默认情况下，编译器和 JVM 查找当前目录。JAR 文件按包含 Java 平台相关的类，所以他们的目录默认放在了 class path 中。

设置 CLASSPATH 系统变量

用下面的命令显示当前的 CLASSPATH 变量：

- Windows 平台（DOS 命令行下）-> C:\> set CLASSPATH
- UNIX 平台（Bourne shell 下）-> % echo $CLASSPATH

删除当前 CLASSPATH 变量内容：

- Windows 平台（DOS 命令行下）-> C:\> set CLASSPATH=
- UNIX 平台（Bourne shell 下）-> % unset CLASSPATH; export CLASSPATH

设置 CLASSPATH 变量:

- Windows 平台（DOS 命令行下）-> set CLASSPATH=C:\users\jack\java\classes
- UNIX 平台（Bourne shell 下）-> % CLASSPATH=/home/jack/java/classes; export CLASSPATH

JAVA 培训教程—Java 泛型

Java 泛型方法和泛型类支持程序员使用一个方法指定一组相关方法，或者使用一个类指定一组相关的类型。

Java 泛型（generics）是 JDK 5 中引入的一个新特性,泛型提供了编译时类型安全检测机制，该机制允许程序员在编译时检测到非法的类型。

使用 Java 泛型的概念，我们可以写一个泛型方法来对一个对象数组排序。然后，调用该泛型方法来对整型数组、浮点数数组、字符串数组等进行排序。

泛型方法

你可以写一个泛型方法，该方法在调用时可以接收不同类型的参数。根据传递给泛型方法的参数类型，编译器适当地处理每一个方法调用。

下面是定义泛型方法的规则：

1、所有泛型方法声明都有一个类型参数声明部分（由尖括号分隔），该类型参数声明部分在方法返回类型之前（在下面例子中的<E>）。

2、每一个类型参数声明部分包含一个或多个类型参数，参数间用逗号隔开。一个泛型参数，也被称为一个类型变量，是用于指定一个泛型类型名称的标识符。

3、类型参数能被用来声明返回值类型，并且能作为泛型方法得到的实际参数类型的占位符。

4、泛型方法方法体的声明和其他方法一样。注意类型参数只能代表引用型类型，不能是原始类型（像 int,double,char 的等）。

实例

下面的例子演示了如何使用泛型方法打印不同字符串的元素：

```
1   public  class  GenericMethodTest
2   {
3         // 泛型方法 printArray
4         public  static  < E >  void  printArray( E[] inputArray )
5         {
6               // 输出数组元素
7                     for  ( E element : inputArray ){
8                           System.out.printf(  "%s ", element );
9                     }
10                    System.out.println();
11          }
12
13          public  static  void  main( String args[] )
14          {
15                  // 创建不同类型数组： Integer, Double 和 Character
16                  Integer[] intArray = {  1,  2,  3,  4,  5  };
17                  Double[] doubleArray = {  1.1,  2.2,  3.3,  4.4  };
18                  Character[] charArray = {  'H',  'E',  'L',  'L',  'O'  };
19
20                  System.out.println(  "Array integerArray contains:"  );
21                  printArray( intArray   );  // 传递一个整型数组
22
23                  System.out.println(  "\nArray doubleArray contains:"  );
24                  printArray( doubleArray );  // 传递一个双精度型数组
25
26                  System.out.println(  "\nArray characterArray contains:"  );
27                  printArray( charArray );  // 传递一个字符型型数组
28          }
29  }
```

编译以上代码，运行结果如下所示：

```
Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4

Array characterArray contains:
H E L L O
```

有界的类型参数:

可能有时候，你会想限制那些被允许传递到一个类型参数的类型种类范围。例如，一个操作数字的方法可能只希望接受 Number 或者 Number 子类的实例。这就是有界类型参数的目的。

要声明一个有界的类型参数，首先列出类型参数的名称，后跟 extends 关键字，最后紧跟它的上界。

实例

下面的例子演示了"extends"如何使用在一般意义上的意思"extends"（类）或者"implements"（接口）。该例子中的泛型方法返回三个可比较对象的最大值。

```
1   public  class  MaximumTest
2   {
3         // 比较三个值并返回最大值
4         public  static  <T  extends  Comparable<T>> T maximum(T x, T y, T z)
5         {
6               T max = x;  // 假设 x 是初始最大值
7               if  ( y.compareTo( max ) >  0  ){
```

```
8                       max = y;  //y 更大
9                 }
10                if  ( z.compareTo( max ) >  0  ){
11                      max = z;  // 现在 z 更大
12                }
13                return  max;  // 返回最大对象
14          }
15          public  static  void  main( String args[] )
16          {
17                System.out.printf(  "Max of %d, %d and %d is %d\n\n",
18                                          3,  4,  5, maximum(  3,  4,  5  ) );
19
20                System.out.printf(  "Maxm of %.1f,%.1f and %.1f is %.1f\n\n",
21                                          6.6,  8.8,  7.7, maximum(  6.6,  8.8,  7.7  ) );
22
23                System.out.printf(  "Max of %s, %s and %s is %s\n","pear",
24                      "apple",  "orange", maximum(  "pear",  "apple",  "orange"  ) );
25          }
26    }
```

编译以上代码，运行结果如下所示：

```
Maximum of  3,  4  and  5  is  5

Maximum of  6.6,  8.8  and  7.7  is  8.8

Maximum of pear, apple and orange is pear
```

泛型类

泛型类的声明和非泛型类的声明类似，除了在类名后面添加了类型参数声明部分。

和泛型方法一样，泛型类的类型参数声明部分也包含一个或多个类型参数，参数间用逗号隔开。一个泛型参数，也被称为一个类型变量，是用于指定一个泛型类型名称的标识符。因为他们接受一个或多个参数，这些类被称为参数化的类或参数化的类型。

实例

如下实例演示了我们如何定义一个泛型类:

```
1    public  class  Box<T> {
2
3        private  T t;
4
5        public  void  add(T t) {
6            this.t = t;
7        }
8
9        public  T get() {
10           return  t;
11       }
12
13       public  static  void  main(String[] args) {
14             Box<Integer> integerBox =  new  Box<Integer>();
15             Box<String> stringBox =  new  Box<String>();
16
17             integerBox.add(new  Integer(10));
18             stringBox.add(new  String("Hello World"));
19
20             System.out.printf("Integer Value :%d\n\n", integerBox.get());
21             System.out.printf("String Value :%s\n", stringBox.get());
22       }
```

```
23      }
```

编译以上代码，运行结果如下所示：

```
:     Integer Value :10
;
;     String Value :Hello World
```

JAVA 培训教程—Java 流(Stream)、文件(File)和 IO

Java.io 包几乎包含了所有操作输入、输出需要的类。所有这些流类代表了输入源和输出目标。

Java.io 包中的流支持很多种格式，比如：基本类型、对象、本地化字符集等等。

一个流可以理解为一个数据的序列。输入流表示从一个源读取数据，输出流表示向一个目标写数据。

Java 为 I/O 提供了强大的而灵活的支持，使其更广泛地应用到文件传输和网络编程中。

但本节讲述最基本的和流与 I/O 相关的功能。我们将通过一个个例子来学习这些功能。

读取控制台输入

Java 的控制台输入由 Sysem.in 完成。

为了获得一个绑定到控制台的字符流，你可以把 System.in 包装在一个 BufferedReader 对象中来创建一个字符流。

下面是创建 BufferedReader 的基本语法：

```
1    BufferedReader br =  new  BufferedReader(new
2                                          InputStreamReader(System.in));
```

BufferedReader 对象创建后，我们便可以使用 read()方法从控制台读取一个字符，或者用 readLine()方法读取一个字符串。

从控制台读取多字符输入

从 BufferedReader 对象读取一个字符要使用 read()方法，它的语法如下：

```
1    int  read( )  throws  IOException
```

每次调用 read()方法，它从输入流读取一个字符并把该字符作为整数值返回。 当流结束的时候返回-1。该方法抛出 IOException。

下面的程序示范了用 read()方法从控制台不断读取字符直到用户输入"q"。

```
1    // 使用 BufferedReader 在控制台读取字符
2
3    import  java.io.*;
4
5    public  class  BRRead {
6          public  static  void  main(String args[])  throws  IOException
7          {
8                char  c;
9                // 使用 System.in 创建 BufferedReader
10               BufferedReader br =  new  BufferedReader(new
11                                                   InputStreamReader(System.in));
12               System.out.println("Enter characters, 'q' to quit.");
13               // 读取字符
14               do  {
15                     c = (char) br.read();
16                     System.out.println(c);
17               }  while(c !=  'q');
18         }
19   }
```

以上实例编译运行结果如下:

```
1    Enter characters,  'q'  to quit.
2    123abcq
```

3 1

4 2

5 3

6 a

7 b

8 c

9 q

从控制台读取字符串

从标准输入读取一个字符串需要使用 BufferedReader 的 readLine()方法。

它的一般格式是：

```
1   String readLine( )  throws  IOException
```

下面的程序读取和显示字符行直到你输入了单词"end"。

```
1 // 使用 BufferedReader 在控制台读取字符
2 import  java.io.*;
3 public  class  BRReadLines {
4       public  static  void  main(String args[])  throws  IOException
5       {
6             // 使用 System.in 创建 BufferedReader
7             BufferedReader br =  new  BufferedReader(new
8                                                          InputStreamReader
9 (System.in));
1             String str;
0             System.out.println("Enter lines of text.");
1             System.out.println("Enter 'end' to quit.");
1             do  {
1                   str = br.readLine();
```

```
2                   System.out.println(str);
1             }  while(!str.equals("end"));
3       }
1 }
4
1
5
1
6
1
7
```

以上实例编译运行结果如下:

```
1   Enter lines of text.
2   Enter 'end' to quit.
3   This is line one
4   This is line one
5   This is line two
6   This is line two
7   end
8   end
```

控制台输出

在此前已经介绍过，控制台的输出由 print( ) 和 println( )完成。这些方法都由类 PrintStream 定义，System.out 是该类对象的一个引用。

PrintStream 继承了 OutputStream 类，并且实现了方法 write()。这样，write()也可以用来往控制台写操作。

PrintStream 定义 write()的最简单格式如下所示：

```
1    void  write(int  byteval)
```

该方法将 byteval 的低八位字节写到流中。

实例

下面的例子用 write()把字符"A"和紧跟着的换行符输出到屏幕：

```
1     import  java.io.*;
2
3     // 演示 System.out.write().
4     public  class  WriteDemo {
5           public  static  void  main(String args[]) {
6                 int  b;
7                 b =  'A';
8                 System.out.write(b);
9                 System.out.write('\n');
10          }
11    }
```

运行以上实例在输出窗口输出"A"字符

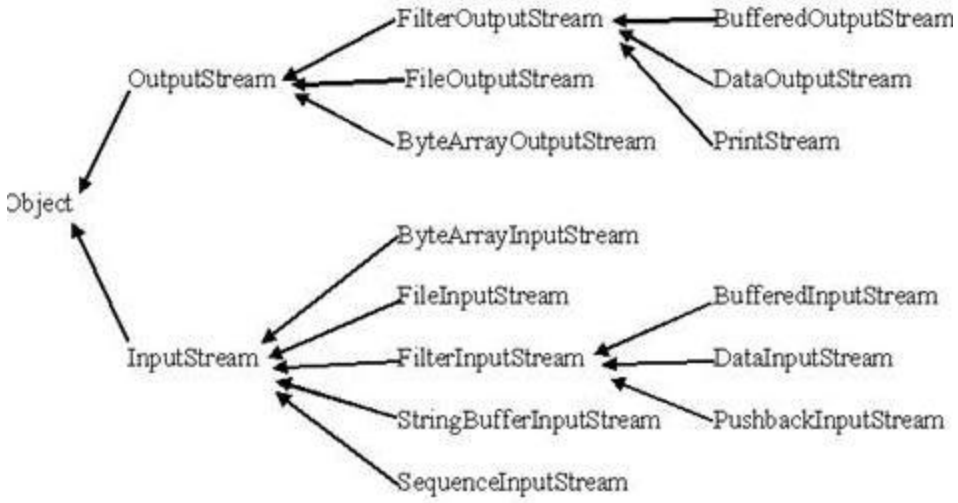
```
1    A
```

注意：write()方法不经常使用，因为 print()和 println()方法用起来更为方便。

读写文件

如前所述，一个流被定义为一个数据序列。输入流用于从源读取数据，输出流用于向目标写数据。

下图是一个描述输入流和输出流的类层次图。



下面将要讨论的两个重要的流是 FileInputStream 和 FileOutputStream：

FileInputStream

该流用于从文件读取数据，它的对象可以用关键字 new 来创建。

有多种构造方法可用来创建对象。

可以使用字符串类型的文件名来创建一个输入流对象来读取文件：

```
1    InputStream f =  new  FileInputStream("C:/java/hello");
```

也可以使用一个文件对象来创建一个输入流对象来读取文件。我们首先得使用 File()方法来创建一个文件对象：

```
1    File f =  new  File("C:/java/hello");
2    InputStream f =  new  FileInputStream(f);
```

创建了 InputStream 对象，就可以使用下面的方法来读取流或者进行其他的流操作。

序号　方法及描述

1 `public void close() throws IOException{}`

关闭此文件输入流并释放与此流有关的所有系统资源。抛出 IOException 异常。

2 `protected void finalize()throws IOException {}`

这个方法清除与该文件的连接。确保在不再引用文件输入流时调用其 close 方法。抛出 IOException 异常。

3 `public int read(int r)throws IOException{}`

这个方法从 InputStream 对象读取指定字节的数据。返回为整数值。返回下一字节数据，如果已经到结尾则返回-1。

4 `public int read(byte[] r) throws IOException{}`

这个方法从输入流读取 r.length 长度的字节。返回读取的字节数。如果是文件结尾则返回-1。

5 `public int available() throws IOException{}`

返回下一次对此输入流调用的方法可以不受阻塞地从此输入流读取的字节数。返回一个整数值。

除了 InputStream 外，还有一些其他的输入流，更多的细节参考下面链接：

- ByteArrayInputStream
- DataInputStream

FileOutputStream

该类用来创建一个文件并向文件中写数据。

如果该流在打开文件进行输出前，目标文件不存在，那么该流会创建该文件。

有两个构造方法可以用来创建 FileOutputStream 对象。

使用字符串类型的文件名来创建一个输出流对象：

```
1    OutputStream f =  new  FileOutputStream("C:/java/hello")
```

也可以使用一个文件对象来创建一个输出流来写文件。我们首先得使用 File()方法来创建一个文件对象：

```
1    File f =  new  File("C:/java/hello");
2    OutputStream f =  new  FileOutputStream(f);
```

创建 OutputStream 对象完成后，就可以使用下面的方法来写入流或者进行其他的流操作。

序号	方法及描述
1	**public void close() throws IOException{}** 关闭此文件输入流并释放与此流有关的所有系统资源。抛出 IOException 异常。
2	**protected void finalize()throws IOException {}** 这个方法清除与该文件的连接。确保在不再引用文件输入流时调用其 close 方法。抛出 IOException 异常。
3	**public void write(int w)throws IOException{}** 这个方法把指定的字节写到输出流中。
4	**public void write(byte[] w)**

把指定数组中 w.length 长度的字节写到 OutputStream 中。

除了 OutputStream 外，还有一些其他的输出流，更多的细节参考下面链接：

- ByteArrayOutputStream
- DataOutputStream

实例

下面是一个演示 InputStream 和 OutputStream 用法的例子：

```
1    import  java.io.*;
2
3    public  class  fileStreamTest{
4
5          public  static  void  main(String args[]){
6
7          try{
8                byte  bWrite [] = {11,21,3,40,5};
9                OutputStream os =  new  FileOutputStream("test.txt");
10               for(int  x=0; x < bWrite.length ; x++){
11                     os.write( bWrite[x] );  // writes the bytes
12               }
13               os.close();
14
15               InputStream is =  new  FileInputStream("test.txt");
16               int  size = is.available();
17
18               for(int  i=0; i< size; i++){
19                     System.out.print((char)is.read() +  "   ");
```

```
20                }
21                is.close();
22          }catch(IOException e){
23                System.out.print("Exception");
24          }
25          }
26    }
```

上面的程序首先创建文件 test.txt，并把给定的数字以二进制形式写进该文件，同时输出到控制台上。

文件和 I/O

还有一些关于文件和 I/O 的类，我们也需要知道：

- File Class(类)
- FileReader Class(类)
- FileWriter Class(类)

Java 中的目录

创建目录：

File 类中有两个方法可以用来创建文件夹：

- **mkdir()**方法创建一个文件夹，成功则返回 true，失败则返回 false。失败表明 File 对象指定的路径已经存在，或者由于整个路径还不存在，该文件夹不能被创建。
- **mkdirs()**方法创建一个文件夹和它的所有父文件夹。

下面的例子创建 "/tmp/user/java/bin"文件夹：

```
1     import   java.io.File;
2
```

```
3    public  class  CreateDir {
4          public  static  void  main(String args[]) {
5                String dirname =  "/tmp/user/java/bin";
6                File d =  new  File(dirname);
7                // 现在创建目录
8                d.mkdirs();
9        }
10   }
```

编译并执行上面代码来创建目录"/tmp/user/java/bin"。

注意：Java 在 UNIX 和 Windows 自动按约定分辨文件路径分隔符。如果你在 Windows 版本的 Java 中使用分隔符(/) ，路径依然能够被正确解析。

读取目录

一个目录其实就是一个 File 对象，它包含其他文件和文件夹。

如果创建一个 File 对象并且它是一个目录，那么调用 isDirectory( )方法会返回 true。

可以通过调用该对象上的 list()方法，来提取它包含的文件和文件夹的列表。

下面展示的例子说明如何使用 list()方法来检查一个文件夹中包含的内容：

```
1   import  java.io.File;
2
3   public  class  DirList {
4         public  static  void  main(String args[]) {
5               String dirname =  "/tmp";
6               File f1 =  new  File(dirname);
7               if  (f1.isDirectory()) {
8                     System.out.println(  "Directory of "  + dirname);
```

```
9                   String s[] = f1.list();
10                  for  (int  i=0; i < s.length; i++) {
11                        File f =  new  File(dirname +  "/"  + s[i]);
12                        if  (f.isDirectory()) {
13                              System.out.println(s[i] +  " is a directory");
14                        }  else  {
15                              System.out.println(s[i] +  " is a file");
16                        }
17                  }
18            }  else  {
19                  System.out.println(dirname +  " is not a directory");
20        }
21    }
22 }
```

以上实例编译运行结果如下：

```
1  Directory of /tmp
2  bin is a directory
3  lib is a directory
4  demo is a directory
5  test.txt is a file
6  README is a file
7  index.html is a file
8  include is a directory
```

JAVA 培训教程—Java 日期时间

java.util 包提供了 Date 类来封装当前的日期和时间。Date 类提供两个构造函数来实例化 Date 对象。

第一个构造函数使用当前日期和时间来初始化对象。

```
1    Date( )
```

第二个构造函数接收一个参数，该参数是从 1970 年 1 月 1 日起的微秒数。

```
1    Date(long  millisec)
```

Date 对象创建以后，可以调用下面的方法。

序号	方法和描述
1	**boolean after(Date date)** 若当调用此方法的 Date 对象在指定日期之后返回 true,否则返回 false。
2	**boolean before(Date date)** 若当调用此方法的 Date 对象在指定日期之前返回 true,否则返回 false。
3	**Object clone( )** 返回此对象的副本。
4	**int compareTo(Date date)** 比较当调用此方法的 Date 对象和指定日期。两者相等时候返回 0。调用对象在指定日期之前则返回负数。调用对象在指定日期之后则返回正数。

5	int compareTo(Object obj) 若 obj 是 Date 类型则操作等同于 compareTo(Date) 。否则它抛出 ClassCastException。
6	boolean equals(Object date) 当调用此方法的 Date 对象和指定日期相等时候返回 true,否则返回 false。
7	long getTime( ) 返回自 1970 年 1 月 1 日 00:00:00 GMT 以来此 Date 对象表示的毫秒数。
8	int hashCode( ) 返回此对象的哈希码值。
9	void setTime(long time) 用自 1970 年 1 月 1 日 00:00:00 GMT 以后 time 毫秒数设置时间和日期。
10	String toString( ) 转换 Date 对象为 String 表示形式，并返回该字符串。

获取当前日期时间

Java 中获取当前日期和时间很简单，使用 Date 对象的 toString()方法来打印当前日期和时间，如下所示：

```
1    import  java.util.Date;
2
```

```
3    public  class  DateDemo {
4          public  static  void  main(String args[]) {
5                  // 初始化 Date 对象
6                  Date date =  new  Date();
7
8                  // 使用 toString() 函数显示日期时间
9                  System.out.println(date.toString());
10         }
11   }
```

以上实例编译运行结果如下:

```
1    Mon May  04  09:51:52  CDT  2013
```

日期比较

Java 使用以下三种方法来比较两个日期：

- 使用 getTime( ) 方法获取两个日期（自 1970 年 1 月 1 日经历的微妙数值），然后比较这两个值。
- 使用方法 before()，after()和 equals()。例如，一个月的 12 号比 18 号早，则 new Date(99, 2, 12).before(new Date (99, 2, 18))返回 true。
- 使用 compareTo()方法，它是由 Comparable 接口定义的，Date 类实现了这个接口。

使用 SimpleDateFormat 格式化日期

SimpleDateFormat 是一个以语言环境敏感的方式来格式化和分析日期的类。

SimpleDateFormat 允许你选择任何用户自定义日期时间格式来运行。例如：

```
1    import  java.util.*;
2    import  java.text.*;
3
```

```
4    public  class  DateDemo {
5          public  static  void  main(String args[]) {
6
7                Date dNow =  new  Date( );
8                SimpleDateFormat ft =
9                new  SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");
10
11               System.out.println("Current Date: "  + ft.format(dNow));
12         }
13   }
```

以上实例编译运行结果如下:

```
1   Current Date: Sun  2004.07.18  at  04:14:09  PM PDT
```

简单的 DateFormat 格式化编码

时间模式字符串用来指定时间格式。在此模式中，所有的 ASCII 字母被保留为模式字母，定义如下：

字母	描述	示例
G	纪元标记	AD
y	四位年份	2001
M	月份	July or 07

d	一个月的日期	10
h	A.M./P.M. (1~12)格式小时	12
H	一天中的小时 (0~23)	22
m	分钟数	30
s	秒数	55
S	微妙数	234
E	星期几	Tuesday
D	一年中的日子	360
F	一个月中第几周的周几	2 (second Wed. in July)
w	一年中第几周	40
W	一个月中第几周	1
a	A.M./P.M. 标记	PM

k	一天中的小时(1~24)	24
K	A.M./P.M. (0~11)格式小时	10
z	时区	Eastern Standard Time
'	文字定界符	Delimiter
"	单引号	`

使用 printf 格式化日期

printf 方法可以很轻松地格式化时间和日期。使用两个字母格式，它以 t 开头并且以下面表格中的一个字母结尾。例如：

```
1    import  java.util.Date;
2
3    public  class  DateDemo {
4
5        public  static  void  main(String args[]) {
6             // 初始化 Date 对象
7             Date date =  new  Date();
8
9             // 使用 toString()显示日期和时间
10            String str = String.format("Current Date/Time : %tc", date );
11
12            System.out.printf(str);
```

```
13        }
14    }
```

以上实例编译运行结果如下:

```
1    Current Date/Time : Sat Dec  15  16:37:57  MST  2012
```

如果你需要重复提供日期，那么利用这种方式来格式化它的每一部分就有点复杂了。因此，可以利用一个格式化字符串指出要被格式化的参数的索引。

索引必须紧跟在%后面，而且必须以$结束。例如：

```
1     import  java.util.Date;
2
3     public  class  DateDemo {
4
5           public  static  void  main(String args[]) {
6                   // 初始化 Date 对象
7                   Date date =  new  Date();
8
9                   // 使用 toString()显示日期和时间
10                  System.out.printf("%1$s %2$tB %2$td, %2$tY",
11                                                    "Due date:", date);
12          }
13    }
```

以上实例编译运行结果如下:

```
1    Due date: February  09,  2004
```

或者，你可以使用<标志。它表明先前被格式化的参数要被再次使用。例如：

```
1     import  java.util.Date;
```

```
2
3    public  class  DateDemo {
4
5          public  static  void  main(String args[]) {
6                  // 初始化 Date 对象
7                  Date date =  new  Date();
8
9                  // 显示格式化时间
10                 System.out.printf("%s %tB %<te, %<tY",
11                                                    "Due date:", date);
12         }
13   }
```

以上实例编译运行结果如下:

```
1    Due date: February  09,  2004
```

日期和时间转换字符

字符	描述	例子
c	完整的日期和时间	Mon May 04 09:51:52 CDT 2009
F	ISO 8601 格式日期	2004-02-09
D	U.S. 格式日期 (月/日/年)	02/09/2004

T	24 小时时间	18:05:19
r	12 小时时间	06:05:19 pm
R	24 小时时间，不包含秒	18:05
Y	4 位年份(包含前导 0)	2004
y	年份后 2 位(包含前导 0)	04
C	年份前 2 位(包含前导 0)	20
B	月份全称	February
b	月份简称	Feb
n	2 位月份(包含前导 0)	02
d	2 位日子(包含前导 0)	03
e	2 位日子(不包含前导 0)	9
A	星期全称	Monday

a	星期简称	Mon
j	3 位年份(包含前导 0)	069
H	2 位小时(包含前导 0)，00 到 23	18
k	2 位小时(不包含前导 0)，　0 到 23	18
I	2 位小时(包含前导 0)，01 到 12	06
l	2 位小时(不包含前导 0)，　1 到 12	6
M	2 位分钟(包含前导 0)	05
S	2 位秒数(包含前导 0)	19
L	3 位毫秒(包含前导 0)	047
N	9 位纳秒(包含前导 0)	047000000
P	大写上下午标志	PM
p	小写上下午标志	pm

z	从 GMT 的 RFC 822 数字偏移	-0800
Z	时区	PST
s	自 1970-01-01 00:00:00 GMT 的秒数	1078884319
Q	自 1970-01-01 00:00:00 GMT 的毫妙	1078884319047

还有其他有用的日期和时间相关的类。对于更多的细节，你可以参考到 Java 标准文档。

解析字符串为时间

SimpleDateFormat 类有一些附加的方法，特别是 parse()，它试图按照给定的 SimpleDateFormat 对象的格式化存储来解析字符串。例如：

```
1    import  java.util.*;
2    import  java.text.*;
3
4    public  class  DateDemo {
5
6          public  static  void  main(String args[]) {
7                SimpleDateFormat ft =  new  SimpleDateFormat ("yyyy-MM-dd");
8
9                String input = args.length ==  0  ?  "1818-11-11"  : args[0];
10
11               System.out.print(input +  " Parses as ");
12
13               Date t;
```

```
14
15              try  {
16                      t = ft.parse(input);
17                      System.out.println(t);
18              }  catch  (ParseException e) {
19                      System.out.println("Unparseable using "  + ft);
20              }
21          }
22   }
```

以上实例编译运行结果如下:

```
1   $ java DateDemo
2   1818-11-11  Parses as Wed Nov  11  00:00:00  GMT  1818
3   $ java DateDemo  2007-12-01
4   2007-12-01  Parses as Sat Dec  01  00:00:00  GMT  2007
```

Java 休眠(sleep)

你可以让程序休眠一毫秒的时间或者到您的计算机的寿命长的任意段时间。例如，下面的程序会休眠 10 秒：

```
1    import  java.util.*;
2
3    public  class  SleepDemo {
4          public  static  void  main(String args[]) {
5                try  {
6                      System.out.println(new  Date( ) +  "\n");
7                      Thread.sleep(5*60*10);
8                      System.out.println(new  Date( ) +  "\n");
9                }  catch  (Exception e) {
```

```
10                      System.out.println("Got an exception!");
11              }
12          }
13      }
```

以上实例编译运行结果如下:

```
1    Sun May  03  18:04:41  GMT  2009
2
3    Sun May  03  18:04:51  GMT  2009
```

测量时间

下面的一个例子表明如何测量时间间隔（以毫秒为单位）：

```
1     import  java.util.*;
2
3     public  class  DiffDemo {
4
5           public  static  void  main(String args[]) {
6                 try  {
7                       long  start = System.currentTimeMillis( );
8                       System.out.println(new  Date( ) +  "\n");
9                       Thread.sleep(5*60*10);
10                      System.out.println(new  Date( ) +  "\n");
11                      long  end = System.currentTimeMillis( );
12                      long  diff = end - start;
13                      System.out.println("Difference is : "  + diff);
14                } catch  (Exception e) {
15                      System.out.println("Got an exception!");
16                }
```

```
17          }
18    }
```

以上实例编译运行结果如下:

```
1    Sun May  03  18:16:51  GMT  2009
2
3    Sun May  03  18:16:57  GMT  2009
4
5    Difference is :  5993
```

Calendar 类

我们现在已经能够格式化并创建一个日期对象了,但是我们如何才能设置和获取日期数据的特定部分呢,比如说小时,日,或者分钟? 我们又如何在日期的这些部分加上或者减去值呢? 答案是使用 Calendar 类。

Calendar 类的功能要比 Date 类强大很多，而且在实现方式上也比 Date 类要复杂一些。

Calendar 类是一个抽象类，在实际使用时实现特定的子类的对象，创建对象的过程对程序员来说是透明的，只需要使用 getInstance 方法创建即可。

创建一个代表系统当前日期的 Calendar 对象

```
1    Calendar c = Calendar.getInstance();//默认是当前日期
```

创建一个指定日期的 Calendar 对象

使用 Calendar 类代表特定的时间，需要首先创建一个 Calendar 的对象，然后再设定该对象中的年月日参数来完成。

```
1    //创建一个代表 2009 年 6 月 12 日的 Calendar 对象
2    Calendar c1 = Calendar.getInstance();
3    c1.set(2009,  6  -  1,  12);
```

Calendar 类对象字段类型

Calendar 类中用一下这些常量表示不同的意义，jdk 内的很多类其实都是采用的这种思想

常量	描述
Calendar.YEAR	年份
Calendar.MONTH	月份
Calendar.DATE	日期
Calendar.DAY_OF_MONTH	日期，和上面的字段意义完全相同
Calendar.HOUR	12 小时制的小时
Calendar.HOUR_OF_DAY	24 小时制的小时
Calendar.MINUTE	分钟
Calendar.SECOND	秒
Calendar.DAY_OF_WEEK	星期几

Calendar 类对象信息的设置

Set 设置

如：

```
1    Calendar c1 = Calendar.getInstance();
```

调用：

```
1    public  final  void  set(int  year,int  month,int  date)
1    c1.set(2009,  6  -  1,  12);//把 Calendar 对象 c1 的年月日分别设这为：2009、6、12
```

利用字段类型设置

如果只设定某个字段，例如日期的值，则可以使用如下 set 方法：

```
1    public  void  set(int  field,int  value)
```

把 c1 对象代表的日期设置为 10 号，其它所有的数值会被重新计算

```
1    c1.set(Calendar.DATE,10);
```

把 c1 对象代表的年份设置为 2008 年，其他的所有数值会被重新计算

```
1    c1.set(Calendar.YEAR,2008);
```

其他字段属性 set 的意义以此类推

Add 设置

```
1    Calendar c1 = Calendar.getInstance();
```

把 c1 对象的日期加上 10，也就是 c1 所表的日期的 10 天后的日期，其它所有的数值会被重新计算

```
1    c1.add(Calendar.DATE,  10);
```

把 c1 对象的日期加上 10，也就是 c1 所表的日期的 10 天前的日期，其它所有的数值会被重新计算

```
1    <pre>c1.add(Calendar.DATE, -10);
```

其他字段属性的 add 的意义以此类推

Calendar 类对象信息的获得

```
1     Calendar c1 = Calendar.getInstance();
2     // 获得年份
3     int  year = c1.get(Calendar.YEAR);
4     // 获得月份
5     int  month = c1.get(Calendar.MONTH) +  1;
6     // 获得日期
7     int  date = c1.get(Calendar.DATE);
8     // 获得小时
9     int  hour = c1.get(Calendar.HOUR_OF_DAY);
10    // 获得分钟
11    int  minute = c1.get(Calendar.MINUTE);
12    // 获得秒
13    int  second = c1.get(Calendar.SECOND);
14    // 获得星期几（注意（这个与 Date 类是不同的）：1 代表星期日、2 代表星期 1、3 代表星期二，以此类
15    int  day = c1.get(Calendar.DAY_OF_WEEK);
```

GregorianCalendar 类

Calendar 类实现了公历日历，GregorianCalendar 是 Calendar 类的一个具体实现。

Calendar 的 getInstance（）方法返回一个默认用当前的语言环境和时区初始化的 GregorianCalendar 对象。GregorianCalendar 定义了两个字段：AD 和 BC。这些代表公历定义的两个时代。

下面列出 GregorianCalendar 对象的几个构造方法：

序号	构造函数和说明
1	GregorianCalendar() 在具有默认语言环境的默认时区内使用当前时间构造一个默认的 GregorianCalendar。
2	GregorianCalendar(int year, int month, int date) 在具有默认语言环境的默认时区内构造一个带有给定日期设置的 GregorianCalendar
3	GregorianCalendar(int year, int month, int date, int hour, int minute) 为具有默认语言环境的默认时区构造一个具有给定日期和时间设置的 GregorianCalendar。
4	GregorianCalendar(int year, int month, int date, int hour, int minute, int second) 为具有默认语言环境的默认时区构造一个具有给定日期和时间设置的 GregorianCalendar。
5	GregorianCalendar(Locale aLocale) 在具有给定语言环境的默认时区内构造一个基于当前时间的 GregorianCalendar。
6	GregorianCalendar(TimeZone zone) 在具有默认语言环境的给定时区内构造一个基于当前时间的 GregorianCalendar。

7	GregorianCalendar(TimeZone zone, Locale aLocale) 在具有给定语言环境的给定时区内构造一个基于当前时间的 GregorianCalendar。

这里是 GregorianCalendar 类提供的一些有用的方法列表：

序号	方法和说明
1	void add(int field, int amount) 根据日历规则，将指定的（有符号的）时间量添加到给定的日历字段中。
2	protected void computeFields() 转换 UTC 毫秒值为时间域值
3	protected void computeTime() 覆盖 Calendar ，转换时间域值为 UTC 毫秒值
4	boolean equals(Object obj) 比较此 GregorianCalendar 与指定的 Object。
5	int get(int field) 获取指定字段的时间值
6	int getActualMaximum(int field) 返回当前日期，给定字段的最大值

7 int getActualMinimum(int field)

返回当前日期，给定字段的最小值

8 int getGreatestMinimum(int field)

返回此 GregorianCalendar 实例给定日历字段的最高的最小值。

9 Date getGregorianChange()

获得格里高利历的更改日期。

10 int getLeastMaximum(int field)

返回此 GregorianCalendar 实例给定日历字段的最低的最大值

11 int getMaximum(int field)

返回此 GregorianCalendar 实例的给定日历字段的最大值。

12 Date getTime()

获取日历当前时间。

13 long getTimeInMillis()

获取用长整型表示的日历的当前时间

14 TimeZone getTimeZone()

获取时区。

15 int getMinimum(int field)

返回给定字段的最小值。

16 `int hashCode()`

重写 hashCode.

17 `boolean isLeapYear(int year)`

确定给定的年份是否为闰年。

18 `void roll(int field, boolean up)`

在给定的时间字段上添加或减去（上/下）单个时间单元，不更改更大的字段。

19 `void set(int field, int value)`

用给定的值设置时间字段。

20 `void set(int year, int month, int date)`

设置年、月、日的值。

21 `void set(int year, int month, int date, int hour, int minute)`

设置年、月、日、小时、分钟的值。

22 `void set(int year, int month, int date, int hour, int minute, int second)`

设置年、月、日、小时、分钟、秒的值。

23 `void setGregorianChange(Date date)`

设置 GregorianCalendar 的更改日期。

24 **void setTime(Date date)**

用给定的日期设置 Calendar 的当前时间。

25 **void setTimeInMillis(long millis)**

用给定的 long 型毫秒数设置 Calendar 的当前时间。

26 **void setTimeZone(TimeZone value)**

用给定时区值设置当前时区。

27 **String toString()**

返回代表日历的字符串。

实例

```
1    import  java.util.*;
2
3    public  class  GregorianCalendarDemo {
4
5          public  static  void  main(String args[]) {
6                String months[] = {
7                "Jan",  "Feb",  "Mar",  "Apr",
8                "May",  "Jun",  "Jul",  "Aug",
9                "Sep",  "Oct",  "Nov",  "Dec"};
10
11               int  year;
12               // 初始化 Gregorian 日历
13               // 使用当前时间和日期
14               // 默认为本地时间和时区
```

```
15              GregorianCalendar gcalendar =  new  GregorianCalendar();
16              // 显示当前时间和日期的信息
17              System.out.print("Date: ");
18              System.out.print(months[gcalendar.get(Calendar.MONTH)]);
19              System.out.print(" "  + gcalendar.get(Calendar.DATE) +  " ");
20              System.out.println(year = gcalendar.get(Calendar.YEAR));
21              System.out.print("Time: ");
22              System.out.print(gcalendar.get(Calendar.HOUR) +  ":");
23              System.out.print(gcalendar.get(Calendar.MINUTE) +  ":");
24              System.out.println(gcalendar.get(Calendar.SECOND));
25
26              // 测试当前年份是否为闰年
27              if(gcalendar.isLeapYear(year)) {
28                    System.out.println("当前年份是闰年");
29              }
30              else  {
31                    System.out.println("当前年份不是闰年");
32              }
33        }
34   }
```

以上实例编译运行结果如下：

```
1   Date: Apr  22  2009
2   Time:  11:25:27
3   当前年份不是闰年
```

JAVA 培训教程—Java 正则表达式

正则表达式定义了字符串的模式；正则表达式可以用来搜索、编辑或处理文本；正则表达式并不仅限于某一种语言，但是在每种语言中有细微的差别。Java 正则表达式和 Perl 的是最为相似的。

java.util.regex 包主要包括以下三个类：

- **Pattern 类：**

  pattern 对象是一个正则表达式的编译表示。Pattern 类没有公共构造方法。要创建一个 Pattern 对象，你必须首先调用其公共静态编译方法，它返回一个 Pattern 对象。该方法接受一个正则表达式作为它的第一个参数。

- **Matcher 类：**

  Matcher 对象是对输入字符串进行解释和匹配操作的引擎。与 Pattern 类一样，Matcher 也没有公共构造方法。你需要调用 Pattern 对象的 matcher 方法来获得一个 Matcher 对象。

- **PatternSyntaxException：**

  PatternSyntaxException 是一个非强制异常类，它表示一个正则表达式模式中的语法错误。

捕获组

捕获组是把多个字符当一个单独单元进行处理的方法，它通过对括号内的字符分组来创建。

例如，正则表达式(dog) 创建了单一分组，组里包含"d"，"o"，和"g"。

捕获组是通过从左至右计算其开括号来编号。例如，在表达式（（A）（B（C））），有四个这样的组：

- ((A)(B(C)))
- (A)
- (B(C))
- (C)

可以通过调用 matcher 对象的 groupCount 方法来查看表达式有多少个分组。groupCount 方法返回一个 int 值，表示 matcher 对象当前有多个捕获组。

还有一个特殊的组（组 0），它总是代表整个表达式。该组不包括在 groupCount 的返回值中。

实例

下面的例子说明如何从一个给定的字符串中找到数字串：

```
1    import  java.util.regex.Matcher;
2    import  java.util.regex.Pattern;
3
4    public  class  RegexMatches
5    {
6            public  static  void  main( String args[] ){
7
8                // 按指定模式在字符串查找
9                String line =  "This order was placed for QT3000! OK?";
10               String pattern =  "(.*)(\\d+)(.*)";
11
12               // 创建 Pattern 对象
13               Pattern r = Pattern.compile(pattern);
14
15               // 现在创建 matcher 对象
16               Matcher m = r.matcher(line);
```

```
17                if  (m.find( )) {
18                      System.out.println("Found value: "  + m.group(0) );
19                      System.out.println("Found value: "  + m.group(1) );
20                      System.out.println("Found value: "  + m.group(2) );
21                } else  {
22                      System.out.println("NO MATCH");
23                }
24          }
25    }
```

以上实例编译运行结果如下：

```
1    Found value: This order was placed  for  QT3000! OK?
2    Found value: This order was placed  for  QT300
3    Found value:  0
```

正则表达式语法

字符	说明
\	将下一字符标记为特殊字符、文本、反向引用或八进制转义符。例如，"n"匹配字符"n"。"\n"匹配换行符。序列"\\"匹配"\"，"\("匹配"("。
^	匹配输入字符串开始的位置。如果设置了 RegExp 对象的 Multiline 属性，^ 还会与"\n"或"\r"之后的位置匹配。
$	匹配输入字符串结尾的位置。如果设置了 RegExp 对象的 Multiline 属性，$ 还会与"\n"或"\r"之前的位置匹配。
*	零次或多次匹配前面的字符或子表达式。例如，zo* 匹配"z"和"zoo"。* 等效于 {0,}。
+	一次或多次匹配前面的字符或子表达式。例如，"zo+"与"zo"和"zoo"匹配，

	但与"z"不匹配。+ 等效于 {1,}。
?	零次或一次匹配前面的字符或子表达式。例如，"do(es)?"匹配"do"或"does"中的"do"。? 等效于 {0,1}。
{*n*}	*n* 是非负整数。正好匹配 *n* 次。例如，"o{2}"与"Bob"中的"o"不匹配，但与"food"中的两个"o"匹配。
{*n*,}	*n* 是非负整数。至少匹配 *n* 次。例如，"o{2,}"不匹配"Bob"中的"o"，而匹配"foooood"中的所有 o。"o{1,}"等效于"o+"。"o{0,}"等效于"o*"。
{*n*,*m*}	*M* 和 *n* 是非负整数，其中 *n* <= *m*。匹配至少 *n* 次，至多 *m* 次。例如，"o{1,3}"匹配"fooooood"中的头三个 o。'o{0,1}' 等效于 'o?'。注意：您不能将空格插入逗号和数字之间。
?	当此字符紧随任何其他限定符（*、+、?、{*n*}、{*n*,}、{*n*,*m*}）之后时，匹配模式是"非贪心的"。"非贪心的"模式匹配搜索到的、尽可能短的字符串，而默认的"贪心的"模式匹配搜索到的、尽可能长的字符串。例如，在字符串"oooo"中，"o+?"只匹配单个"o"，而"o+"匹配所有"o"。
.	匹配除"\n"之外的任何单个字符。若要匹配包括"\n"在内的任意字符，请使用诸如"[\s\S]"之类的模式。
(*pattern*)	匹配 *pattern* 并捕获该匹配的子表达式。可以使用 **$0…$9** 属性从结果"匹配"集合中检索捕获的匹配。若要匹配括号字符 ( )，请使用"\("或者"\)"。
(?:*pattern*)	匹配 *pattern* 但不捕获该匹配的子表达式，即它是一个非捕获匹配，不存储供以后使用的匹配。这对于用"or"字符 (|) 组合模式部件的情况很有用。例如，'industr(?:y|ies) 是比 'industry|industries' 更经济的表达式。
(?=*pattern*)	执行正向预测先行搜索的子表达式，该表达式匹配处于匹配 *pattern* 的字符串的起始点的字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹

	配。例如，'Windows (?=95|98|NT|2000)' 匹配"Windows 2000"中的"Windows"，但不匹配"Windows 3.1"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。
(?!*pattern*)	执行反向预测先行搜索的子表达式,该表达式匹配不处于匹配 *pattern* 的字符串的起始点的搜索字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?!95|98|NT|2000)' 匹配"Windows 3.1"中的"Windows"，但不匹配"Windows 2000"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。
x|*y*	匹配 *x* 或 *y*。例如,'z|food' 匹配"z"或"food"。'(z|f)ood' 匹配"zood"或"food"。
[*xyz*]	字符集。匹配包含的任一字符。例如，"[abc]"匹配"plain"中的"a"。
[^*xyz*]	反向字符集。匹配未包含的任何字符。例如，"[^abc]"匹配"plain"中的"p"。
[*a-z*]	字符范围。匹配指定范围内的任何字符。例如，"[a-z]"匹配"a"到"z"范围内的任何小写字母。
[^*a-z*]	反向范围字符。匹配不在指定的范围内的任何字符。例如，"[^a-z]"匹配任何不在"a"到"z"范围内的任何字符。
\b	匹配一个字边界，即字与空格间的位置。例如，"er\b"匹配"never"中的"er"，但不匹配"verb"中的"er"。
\B	非字边界匹配。"er\B"匹配"verb"中的"er"，但不匹配"never"中的"er"。
\c*x*	匹配 *x* 指示的控制字符。例如，\cM 匹配 Control-M 或回车符。*x* 的值必须在 A-Z 或 a-z 之间。如果不是这样，则假定 c 就是"c"字符本身。
\d	数字字符匹配。等效于 [0-9]。

\D	非数字字符匹配。等效于 [^0-9]。
\f	换页符匹配。等效于 \x0c 和 \cL。
\n	换行符匹配。等效于 \x0a 和 \cJ。
\r	匹配一个回车符。等效于 \x0d 和 \cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等。与 [\f\n\r\t\v] 等效。
\S	匹配任何非空白字符。与 [^ \f\n\r\t\v] 等效。
\t	制表符匹配。与 \x09 和 \cI 等效。
\v	垂直制表符匹配。与 \x0b 和 \cK 等效。
\w	匹配任何字类字符，包括下划线。与"[A-Za-z0-9_]"等效。
\W	与任何非单词字符匹配。与"[^A-Za-z0-9_]"等效。
\x*n*	匹配 *n*，此处的 *n* 是一个十六进制转义码。十六进制转义码必须正好是两位数长。例如，"\x41"匹配"A"。"\x041"与"\x04"&"1"等效。允许在正则表达式中使用 ASCII 代码。
num	匹配 *num*，此处的 *num* 是一个正整数。到捕获匹配的反向引用。例如，"(.)\1"匹配两个连续的相同字符。
n	标识一个八进制转义码或反向引用。如果 *n* 前面至少有 *n* 个捕获子表达式，那么 *n* 是反向引用。否则，如果 *n* 是八进制数 (0-7)，那么 *n* 是八进制转义码。
nm	标识一个八进制转义码或反向引用。如果 *nm* 前面至少有 *nm* 个捕获子表达式，那么 *nm* 是反向引用。如果 *nm* 前面至少有 *n* 个捕获，则 *n* 是反向引用，后面跟有字符 *m*。如果两种前面的情况都不存在，则 *nm* 匹配八进制值 *nm*，其中 *n* 和 *m* 是八进制数字 (0-7)。

\nml	当 *n* 是八进制数 (0-3)，*m* 和 *l* 是八进制数 (0-7) 时，匹配八进制转义码 *nml*。
\u*n*	匹配 *n*,其中 *n* 是以四位十六进制数表示的 Unicode 字符。例如,\u00A9 匹配版权符号 (©)。

Mather 类的方法

索引方法

索引方法提供了有用的索引值，精确表明输入字符串中在哪能找到匹配：

序号 方法及说明

1	**public int start()** 返回以前匹配的初始索引。
2	**public int start(int group)** 返回在以前的匹配操作期间，由给定组所捕获的子序列的初始索引
3	**public int end()** 返回最后匹配字符之后的偏移量。
4	**public int end(int group)** 返回在以前的匹配操作期间，由给定组所捕获子序列的最后字符之后的偏移量。

研究方法

研究方法用来检查输入字符串并返回一个布尔值，表示是否找到该模式：

序号	方法及说明
1	public boolean lookingAt() 尝试将从区域开头开始的输入序列与该模式匹配。
2	public boolean find() 尝试查找与该模式匹配的输入序列的下一个子序列。
3	public boolean find(int start) 重置此匹配器，然后尝试查找匹配该模式、从指定索引开始的输入序列的下一个子序列。
4	public boolean matches() 尝试将整个区域与模式匹配。

替换方法

替换方法是替换输入字符串里文本的方法：

序号	方法及说明
1	public Matcher appendReplacement(StringBuffer sb, String replacement) 实现非终端添加和替换步骤。
2	public StringBuffer appendTail(StringBuffer sb)

实现终端添加和替换步骤。

3 public String replaceAll(String replacement)

替换模式与给定替换字符串相匹配的输入序列的每个子序列。

4 public String replaceFirst(String replacement)

替换模式与给定替换字符串匹配的输入序列的第一个子序列。

5 public static String quoteReplacement(String s)

返回指定字符串的字面替换字符串。这个方法返回一个字符串，就像传递给 Matcher 类的 appendReplacement 方法一个字面字符串一样工作。

start 和 end 方法

下面是一个对单词"cat"出现在输入字符串中出现次数进行计数的例子：

```
1  import  java.util.regex.Matcher;
2  import  java.util.regex.Pattern;
3
4  public  class  RegexMatches
5  {
6          private  static  final  String REGEX =  "\\bcat\\b";
7          private  static  final  String INPUT =
8                                                                  "cat
9  cat cat cattie cat";
10
11         public  static  void  main( String args[] ){
12               Pattern p = Pattern.compile(REGEX);
13               Matcher m = p.matcher(INPUT);  // 获取 matcher 对象
```

```
14                int  count =  0;
15
16                while(m.find()) {
17                    count++;
18                    System.out.println("Match number "+count);
19                    System.out.println("start(): "+m.start());
20                    System.out.println("end(): "+m.end());
21                }
22            }
    }
```

以上实例编译运行结果如下：

```
1    Match number  1
2    start():  0
3    end():  3
4    Match number  2
5    start():  4
6    end():  7
7    Match number  3
8    start():  8
9    end():  11
10   Match number  4
11   start():  19
12   end():  22
```

可以看到这个例子是使用单词边界，以确保字母 "c" "a" "t" 并非仅是一个较长的词的子串。它也提供了一些关于输入字符串中匹配发生位置的有用信息。

Start 方法返回在以前的匹配操作期间，由给定组所捕获的子序列的初始索引，end 方法最后一个匹配字符的索引加 1。

matches 和 lookingAt 方法

matches 和 lookingAt 方法都用来尝试匹配一个输入序列模式。它们的不同是 matcher 要求整个序列都匹配，而 lookingAt 不要求。

这两个方法经常在输入字符串的开始使用。

我们通过下面这个例子，来解释这个功能：

```
1    import  java.util.regex.Matcher;
2    import  java.util.regex.Pattern;
3
4    public  class  RegexMatches
5    {
6            private  static  final  String REGEX =  "foo";
7            private  static  final  String INPUT =  "fooooooooooooooooo";
8            private  static  Pattern pattern;
9            private  static  Matcher matcher;
10
11           public  static  void  main( String args[] ){
12                   pattern = Pattern.compile(REGEX);
13                   matcher = pattern.matcher(INPUT);
14
15                   System.out.println("Current REGEX is: "+REGEX);
16                   System.out.println("Current INPUT is: "+INPUT);
17
18                   System.out.println("lookingAt(): "+matcher.lookingAt());
19                   System.out.println("matches(): "+matcher.matches());
20           }
21   }
```

以上实例编译运行结果如下：

```
Current REGEX is: foo
Current INPUT is: fooooooooooooooooo
lookingAt():  true
matches():  false
```

replaceFirst 和 replaceAll 方法

replaceFirst 和 replaceAll 方法用来替换匹配正则表达式的文本。不同的是，replaceFirst 替换首次匹配，replaceAll 替换所有匹配。

下面的例子来解释这个功能：

```
import  java.util.regex.Matcher;
import  java.util.regex.Pattern;

public  class  RegexMatches
{
        private  static  String REGEX =  "dog";
        private  static  String INPUT =  "The dog says meow. "  +
                                                        "All dogs say meow.
        private  static  String REPLACE =  "cat";

        public  static  void  main(String[] args) {
                Pattern p = Pattern.compile(REGEX);
                // get a matcher object
                Matcher m = p.matcher(INPUT);
                INPUT = m.replaceAll(REPLACE);
                System.out.println(INPUT);
        }
```

```
18    }
```

以上实例编译运行结果如下：

```
1    The cat says meow. All cats say meow.
```

appendReplacement 和 appendTail 方法

Matcher 类也提供了 appendReplacement 和 appendTail 方法用于文本替换：

看下面的例子来解释这个功能：

```
1     import  java.util.regex.Matcher;
2     import  java.util.regex.Pattern;
3
4     public  class  RegexMatches
5     {
6           private  static  String REGEX =  "a*b";
7           private  static  String INPUT =  "aabfooaabfooabfoob";
8           private  static  String REPLACE =  "-";
9           public  static  void  main(String[] args) {
10                Pattern p = Pattern.compile(REGEX);
11                // 获取 matcher 对象
12                Matcher m = p.matcher(INPUT);
13                StringBuffer sb =  new  StringBuffer();
14                while(m.find()){
15                      m.appendReplacement(sb,REPLACE);
16                }
17                m.appendTail(sb);
18                System.out.println(sb.toString());
19          }
20    }
```

以上实例编译运行结果如下：

```
1    -foo-foo-foo-
```

PatternSyntaxException 类的方法

PatternSyntaxException 是一个非强制异常类，它指示一个正则表达式模式中的语法错误。

PatternSyntaxException 类提供了下面的方法来帮助我们查看发生了什么错误。

序号	方法及说明
1	public String getDescription() 获取错误的描述。
2	public int getIndex() 获取错误的索引。
3	public String getPattern() 获取错误的正则表达式模式。
4	public String getMessage() 返回多行字符串，包含语法错误及其索引的描述、错误的正则表达式模式和模式中错误索引的可视化指示。

JAVA 培训教程—Java 多线程编程

Java 给多线程编程提供了内置的支持。一个多线程程序包含两个或多个能并发运行的部分。程序的每一部分都称作一个线程，并且每个线程定义了一个独立的执行路径。

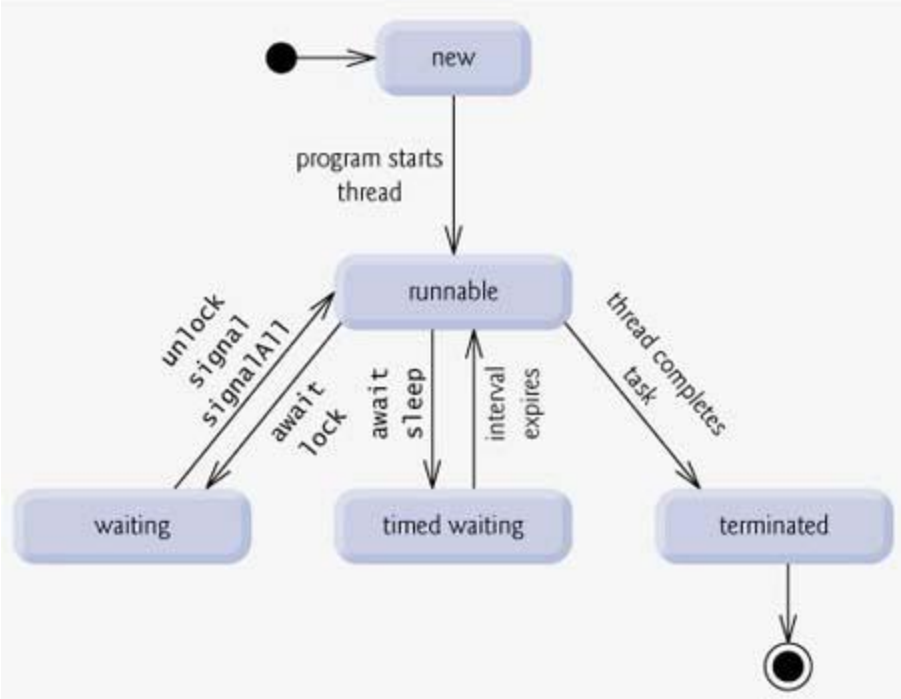
多线程是多任务的一种特别的形式。多线程比多任务需要更小的开销。

这里定义和线程相关的另一个术语：进程：一个进程包括由操作系统分配的内存空间，包含一个或多个线程。一个线程不能独立的存在，它必须是进程的一部分。一个进程一直运行，直到所有的非守候线程都结束运行后才能结束。

多线程能满足程序员编写非常有效率的程序来达到充分利用 CPU 的目的，因为 CPU 的空闲时间能够保持在最低限度。

一个线程的生命周

线程经过其生命周期的各个阶段。下图显示了一个线程完整的生命周期。



- **新状态：** 一个新产生的线程从新状态开始了它的生命周期。它保持这个状态知道程序 start 这个线程。

- **运行状态:**当一个新状态的线程被 start 以后，线程就变成可运行状态，一个线程在此状态下被认为是开始执行其任务
- **就绪状态:**当一个线程等待另外一个线程执行一个任务的时候，该线程就进入就绪状态。当另一个线程给就绪状态的线程发送信号时，该线程才重新切换到运行状态。
- **休眠状态:** 由于一个线程的时间片用完了，该线程从运行状态进入休眠状态。当时间间隔到期或者等待的时间发生了，该状态的线程切换到运行状态。
- **终止状态:** 一个运行状态的线程完成任务或者其他终止条件发生，该线程就切换到终止状态。

线程的优先级

每一个 Java 线程都有一个优先级，这样有助于操作系统确定线程的调度顺序。Java 优先级在 MIN_PRIORITY（1）和 MAX_PRIORITY（10）之间的范围内。默认情况下，每一个线程都会分配一个优先级 NORM_PRIORITY（5）。

具有较高优先级的线程对程序更重要，并且应该在低优先级的线程之前分配处理器时间。然而，线程优先级不能保证线程执行的顺序，而且非常依赖于平台。

创建一个线程

Java 提供了两种创建线程方法：

- 通过实现 Runable 接口；
- 通过继承 Thread 类本身。

通过实现 Runnable 接口来创建线程

创建一个线程，最简单的方法是创建一个实现 Runnable 接口的类。

为了实现 Runnable，一个类只需要执行一个方法调用 run()，声明如下：

```
1    publicvoid run()
```

你可以重写该方法，重要的是理解的 run()可以调用其他方法，使用其他类，并声明变量，就像主线程一样。

在创建一个实现 Runnable 接口的类之后，你可以在类中实例化一个线程对象。

Thread 定义了几个构造方法，下面的这个是我们经常使用的：

```
1    Thread(Runnable threadOb,String threadName);
```

这里，threadOb 是一个实现 Runnable 接口的类的实例，并且 threadName 指定新线程的名字。

新线程创建之后，你调用它的 start()方法它才会运行。

```
1    void  start();
```

实例

下面是一个创建线程并开始让它执行的实例：

```
1     // 创建一个新的线程
2     class  NewThread  implements  Runnable {
3           Thread t;
4           NewThread() {
5                 // 创建第二个新线程
6                 t =  new  Thread(this,  "Demo Thread");
7                 System.out.println("Child thread: "  + t);
8                 t.start();  // 开始线程
9           }
10
11          // 第二个线程入口
12          public  void  run() {
13                try  {
```

```
14                      for(int  i =  5; i >  0; i--) {
15                              System.out.println("Child Thread: "  + i);
16                              // 暂停线程
17                              Thread.sleep(50);
18                      }
19              }  catch  (InterruptedException e) {
20                      System.out.println("Child interrupted.");
21              }
22              System.out.println("Exiting child thread.");
23          }
24  }
25
26  public  class  ThreadDemo {
27          public  static  void  main(String args[]) {
28                  new  NewThread();  // 创建一个新线程
29                  try  {
30                          for(int  i =  5; i >  0; i--) {
31                              System.out.println("Main Thread: "  + i);
32                              Thread.sleep(100);
33                          }
34                  }  catch  (InterruptedException e) {
35                          System.out.println("Main thread interrupted.");
36                  }
37                  System.out.println("Main thread exiting.");
38          }
39  }
```

编译以上程序运行结果如下：

```
1    Child thread: Thread[Demo Thread,5,main]
```

```
2    Main Thread:  5
3    Child Thread:  5
4    Child Thread:  4
5    Main Thread:  4
6    Child Thread:  3
7    Child Thread:  2
8    Main Thread:  3
9    Child Thread:  1
10   Exiting child thread.
11   Main Thread:  2
12   Main Thread:  1
13   Main thread exiting.
```

通过继承 Thread 来创建线程

创建一个线程的第二种方法是创建一个新的类，该类继承 Thread 类，然后创建一个该类的实例。

继承类必须重写 run()方法，该方法是新线程的入口点。它也必须调用 start()方法才能执行。

实例

```
1    // 通过继承 Thread 创建线程
2    class  NewThread  extends  Thread {
3          NewThread() {
4                // 创建第二个新线程
5                super("Demo Thread");
6                System.out.println("Child thread: "  +  this);
7                start();  // 开始线程
8          }
```

```
9
10          // 第二个线程入口
11          public  void  run() {
12                try  {
13                      for(int  i =  5; i >  0; i--) {
14                            System.out.println("Child Thread: "  + i);
15                                                                  // 让线程休眠一会
16                            Thread.sleep(50);
17                      }
18                }  catch  (InterruptedException e) {
19                      System.out.println("Child interrupted.");
20                }
21                System.out.println("Exiting child thread.");
22          }
23    }
24
25    public  class  ExtendThread {
26          public  static  void  main(String args[]) {
27                new  NewThread();  // 创建一个新线程
28                try  {
29                      for(int  i =  5; i >  0; i--) {
30                            System.out.println("Main Thread: "  + i);
31                            Thread.sleep(100);
32                      }
33                }  catch  (InterruptedException e) {
34                      System.out.println("Main thread interrupted.");
35                }
36                System.out.println("Main thread exiting.");
37          }
```

```
38    }
```

编译以上程序运行结果如下：

```
1     Child thread: Thread[Demo Thread,5,main]
2     Main Thread:  5
3     Child Thread:  5
4     Child Thread:  4
5     Main Thread:  4
6     Child Thread:  3
7     Child Thread:  2
8     Main Thread:  3
9     Child Thread:  1
10    Exiting child thread.
11    Main Thread:  2
12    Main Thread:  1
13    Main thread exiting.
```

Thread 方法

下表列出了 Thread 类的一些重要方法：

序号	方法描述
1	**public void start()** 使该线程开始执行；Java 虚拟机调用该线程的 run 方法。
2	**public void run()** 如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对

象的 run 方法；否则，该方法不执行任何操作并返回。

3	public final void setName(String name) 改变线程名称，使之与参数 name 相同。
4	public final void setPriority(int priority) 更改线程的优先级。
5	public final void setDaemon(boolean on) 将该线程标记为守护线程或用户线程。
6	public final void join(long millisec) 等待该线程终止的时间最长为 millis 毫秒。
7	public void interrupt() 中断线程。
8	public final boolean isAlive() 测试线程是否处于活动状态。

测试线程是否处于活动状态。上述方法是被 Thread 对象调用的。下面的方法是 Thread 类的静态方法。

序号	方法描述

1	**public static void yield()** 暂停当前正在执行的线程对象，并执行其他线程。
2	**public static void sleep(long millisec)** 在指定的毫秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。
3	**public static boolean holdsLock(Object x)** 当且仅当当前线程在指定的对象上保持监视器锁时，才返回 true。
4	**public static Thread currentThread()** 返回对当前正在执行的线程对象的引用。
5	**public static void dumpStack()** 将当前线程的堆栈跟踪打印至标准错误流。

实例

如下的 ThreadClassDemo 程序演示了 Thread 类的一些方法：

```
1    // 文件名 : DisplayMessage.java
2    // 通过实现 Runnable 接口创建线程
3    public  class  DisplayMessage  implements  Runnable
4    {
5          private  String message;
6          public  DisplayMessage(String message)
7          {
8                this.message = message;
```

```
9           }
10          public  void  run()
11          {
12                  while(true)
13                  {
14                          System.out.println(message);
15                  }
16          }
17  }
1   // 文件名 : GuessANumber.java
2   // 通过继承 Thread 类创建线程
3
4   public  class  GuessANumber  extends  Thread
5   {
6           private  int  number;
7           public  GuessANumber(int  number)
8           {
9                   this.number = number;
10          }
11          public  void  run()
12          {
13                  int  counter =  0;
14                  int  guess =  0;
15                  do
16                  {
17                          guess = (int) (Math.random() *  100  +  1);
18                          System.out.println(this.getName()
19                                                  +  " guesses "  + guess);
20                          counter++;
```

```
21              }while(guess != number);
22              System.out.println("** Correct! "  +  this.getName()
23                                                  +  " in "  + counter +  " guesses.**");
24          }
25      }
1       // 文件名 : ThreadClassDemo.java
2       public  class  ThreadClassDemo
3       {
4           public  static  void  main(String [] args)
5           {
6               Runnable hello =  new  DisplayMessage("Hello");
7               Thread thread1 =  new  Thread(hello);
8               thread1.setDaemon(true);
9               thread1.setName("hello");
10              System.out.println("Starting hello thread...");
11              thread1.start();
12
13              Runnable bye =  new  DisplayMessage("Goodbye");
14              Thread thread2 =  new  Thread(hello);
15              thread2.setPriority(Thread.MIN_PRIORITY);
16              thread2.setDaemon(true);
17              System.out.println("Starting goodbye thread...");
18              thread2.start();
19
20              System.out.println("Starting thread3...");
21              Thread thread3 =  new  GuessANumber(27);
22              thread3.start();
23              try
24              {
```

```
25                      thread3.join();
26                }catch(InterruptedException e)
27                {
28                      System.out.println("Thread interrupted.");
29                }
30                System.out.println("Starting thread4...");
31                Thread thread4 =  new  GuessANumber(75);
32
33                          thread4.start();
34                System.out.println("main() is ending...");
35          }
36    }
```

运行结果如下，每一次运行的结果都不一样。

```
1     Starting hello thread...
2     Starting goodbye thread...
3     Hello
4     Hello
5     Hello
6     Hello
7     Hello
8     Hello
9     Hello
10    Hello
11    Hello
12    Thread-2  guesses  27
13    Hello
14    ** Correct! Thread-2  in  102  guesses.**
15    Hello
```

```
16    Starting thread4...
17    Hello
18    Hello
19    ..........remaining result produced.
```

线程的几个主要概念:

在多线程编程时，你需要了解以下几个概念：

- 线程同步
- 线程间通信
- 线程死锁
- 线程控制：挂起、停止和恢复

多线程的使用

有效利用多线程的关键是理解程序是并发执行而不是串行执行的。例如：程序中有两个子系统需要并发执行，这时候就需要利用多线程编程。

通过对多线程的使用，可以编写出非常高效的程序。不过请注意，如果你创建太多的线程，程序执行的效率实际上是降低了，而不是提升了。

请记住，上下文的切换开销也很重要，如果你创建了太多的线程，CPU 花费在上下文的切换的时间将多于执行程序的时间！

JAVA 培训教程—Java 网络编程

网络编程是指编写运行在多个设备（计算机）的程序，这些设备都通过网络连接起来。

java.net 包中 J2SE 的 API 包含有类和接口，它们提供低层次的通信细节。你可以直接使用这些类和接口，来专注于解决问题，而不用关注通信细节。

java.net 包中提供了两种常见的网络协议的支持：

- **TCP**： TCP 是传输控制协议的缩写，它保障了两个应用程序之间的可靠通信。通常用于互联网协议，被称 TCP / IP。
- **UDP**:UDP 是用户数据报协议的缩写，一个无连接的协议。提供了应用程序之间要发送的数据的数据包。

本教程主要讲解以下两个主题。

- **Socket 编程**： 这是使用最广泛的网络概念，它已被解释地非常详细
- **URL 处理**： 这部分会在另外的篇幅里讲，点击这里更详细地了解在 Java 语言中的 URL 处理。

Socket 编程

套接字使用 TCP 提供了两台计算机之间的通信机制。 客户端程序创建一个套接字，并尝试连接服务器的套接字。

当连接建立时，服务器会创建一个 Socket 对象。客户端和服务器现在可以通过对 Socket 对象的写入和读取来进行进行通信。

java.net.Socket 类代表一个套接字，并且 java.net.ServerSocket 类为服务器程序提供了一种来监听客户端，并与他们建立连接的机制。

以下步骤在两台计算机之间使用套接字建立 TCP 连接时会出现：

- 服务器实例化一个 ServerSocket 对象，表示通过服务器上的端口通信。

- 服务器调用 ServerSocket 类 的 accept（）方法，该方法将一直等待，直到客户端连接到服务器上给定的端口。
- 服务器正在等待时，一个客户端实例化一个 Socket 对象，指定服务器名称和端口号来请求连接。
- Socket 类的构造函数试图将客户端连接到指定的服务器和端口号。如果通信被建立，则在客户端创建一个 Socket 对象能够与服务器进行通信。
- 在服务器端，accept()方法返回服务器上一个新的 socket 引用，该 socket 连接到客户端的 socket。

连接建立后，通过使用 I/O 流在进行通信。每一个 socket 都有一个输出流和一个输入流。客户端的输出流连接到服务器端的输入流，而客户端的输入流连接到服务器端的输出流。

TCP 是一个双向的通信协议，因此数据可以通过两个数据流在同一时间发送. 以下是一些类提供的一套完整的有用的方法来实现 sockets。

ServerSocket 类的方法

服务器应用程序通过使用 java.net.ServerSocket 类以获取一个端口，并且侦听客户端请求。

ServerSocket 类有四个构造方法：

序号	方法描述
1	**public ServerSocket(int port) throws IOException** 创建绑定到特定端口的服务器套接字。
2	**public ServerSocket(int port, int backlog) throws IOException** 利用指定的 backlog 创建服务器套接字并将其绑定到指定的本地端口号。

3	public ServerSocket(int port, int backlog, InetAddress address) throws IOException 使用指定的端口、侦听 backlog 和要绑定到的本地 IP 地址创建服务器。
4	public ServerSocket() throws IOException 创建非绑定服务器套接字。

创建非绑定服务器套接字。 如果 ServerSocket 构造方法没有抛出异常，就意味着你的应用程序已经成功绑定到指定的端口，并且侦听客户端请求。

这里有一些 ServerSocket 类的常用方法：

序号	方法描述
1	public int getLocalPort()返回此套接字在其上侦听的端口。
2	public Socket accept() throws IOException 侦听并接受到此套接字的连接。
3	public void setSoTimeout(int timeout)通过指定超时值启用/禁用 SO_TIMEOUT，以毫秒为单位。
4	public void bind(SocketAddress host, int backlog)将 ServerSocket 绑定到特定地址（IP 地址和端口号）。

Socket 类的方法

java.net.Socket 类代表客户端和服务器都用来互相沟通的套接字。客户端要获取一个 Socket 对象通过实例化 ，而 服务器获得一个 Socket 对象则通过 accept()方法的返回值。

Socket 类有五个构造方法.

序号	方法描述
1	public Socket(String host, int port) throws UnknownHostException, IOException. 创建一个流套接字并将其连接到指定主机上的指定端口号。
2	public Socket(InetAddress host, int port) throws IOException 创建一个流套接字并将其连接到指定 IP 地址的指定端口号。
3	public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException. 创建一个套接字并将其连接到指定远程主机上的指定远程端口。
4	public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException. 创建一个套接字并将其连接到指定远程地址上的指定远程端口。
5	public Socket() 通过系统默认类型的 SocketImpl 创建未连接套接字

当 Socket 构造方法返回，并没有简单的实例化了一个 Socket 对象，它实际上会尝试连接到指定的服务器和端口。

下面列出了一些感兴趣的方法，注意客户端和服务器端都有一个 Socket 对象，所以无论客户端还是服务端都能够调用这些方法。

序号	方法描述
1	public void connect(SocketAddress host, int timeout) throws IOException 将此套接字连接到服务器，并指定一个超时值。
2	public InetAddress getInetAddress() 返回套接字连接的地址。
3	public int getPort() 返回此套接字连接到的远程端口。
4	public int getLocalPort() 返回此套接字绑定到的本地端口。
5	public SocketAddress getRemoteSocketAddress() 返回此套接字连接的端点的地址，如果未连接则返回 null。
6	public InputStream getInputStream() throws IOException 返回此套接字的输入流。
7	public OutputStream getOutputStream() throws IOException 返回此套接字的输出流。
8	public void close() throws IOException 关闭此套接字。

InetAddress 类的方法

这个类表示互联网协议(IP)地址。下面列出了 Socket 编程时比较有用的方法：

序号	方法描述
1	static InetAddress getByAddress(byte[] addr) 在给定原始 IP 地址的情况下，返回 InetAddress 对象。
2	static InetAddress getByAddress(String host, byte[] addr) 根据提供的主机名和 IP 地址创建 InetAddress。
3	static InetAddress getByName(String host) 在给定主机名的情况下确定主机的 IP 地址。
4	String getHostAddress() 返回 IP 地址字符串（以文本表现形式）。
5	String getHostName() 获取此 IP 地址的主机名。
6	static InetAddress getLocalHost() 返回本地主机。
7	String toString() 将此 IP 地址转换为 String。

Socket 客户端实例

如下的 GreetingClient 是一个客户端程序，改程序通过 socket 连接到服务器并发送一个问候，然后等待一个响应。

```
1    // 文件名 GreetingClient.java
2    <pre>
3    import  java.net.*;
4    import  java.io.*;
5
6    public  class  GreetingClient
7    {
8          public  static  void  main(String [] args)
9          {
10               String serverName = args[0];
11               int  port = Integer.parseInt(args[1]);
12               try
13               {
14                     System.out.println("Connecting to "  + serverName
15                                                          +  " on port "  + port);
16                     Socket client =  new  Socket(serverName, port);
17                     System.out.println("Just connected to "
18                                                 + client.getRemoteSocketAddress());
19                     OutputStream outToServer = client.getOutputStream();
20                     DataOutputStream out =
21                                                   new  DataOutputStream(outToServer);
22
23                     out.writeUTF("Hello from "
24                                                 + client.getLocalSocketAddress());
25                     InputStream inFromServer = client.getInputStream();
```

```
26                      DataInputStream in =
27                                              new  DataInputStream(inFromServer);
28                      System.out.println("Server says "  + in.readUTF());
29                      client.close();
30              }catch(IOException e)
31              {
32                      e.printStackTrace();
33              }
34          }
35    }
```

Socket 服务器实例

如下的 GreetingServer 程序是一个服务器端应用程序，改程序使用 Socket 来监听一个指定的端口。

```
1    $ java GreetingServer  6066
2    Waiting  for  client on port  6066...
```

像下面一样开启客户端：

```
1    $ java GreetingClient localhost  6066
2    Connecting to localhost on port  6066
3    Just connected to localhost/127.0.0.1:6066
4    Server says Thank you  for  connecting to /127.0.0.1:6066
5    Goodbye!
```

JAVA 培训教程—Java 序列化

Java 提供了一种对象序列化的机制，该机制中，一个对象可以被表示为一个字节序列，该字节序列包括该对象的数据、有关对象的类型的信息和存储在对象中数据的类型。

将序列化对象写入文件之后，可以从文件中读取出来，并且对它进行反序列化，也就是说，对象的类型信息、对象的数据，还有对象中的数据类型可以用来在内存中新建对象。

整个过程都是 Java 虚拟机（JVM）独立的，也就是说，在一个平台上序列化的对象可以在另一个完全不同的平台上反序列化该对象。

类 ObjectInputStream 和 ObjectOutputStream 是高层次的数据流，它们包含序列化和反序列化对象的方法。

ObjectOutputStream 类包含很多写方法来写各种数据类型，但是一个特别的方法例外：

```
1    public  final  void  writeObject(Object x)  throws  IOException
```

上面的方法序列化一个对象，并将它发送到输出流。相似的 ObjectInputStream 类包含如下反序列化一个对象的方法：

```
1    public  final  Object readObject()  throws  IOException,
2    ClassNotFoundException
```

该方法从流中取出下一个对象，并将对象反序列化。它的返回值为 Object，因此，你需要将它转换成合适的数据类型。

为了演示序列化在 Java 中是怎样工作的，我将使用之前教程中提到的 Employee 类，假设我们定义了如下的 Employee 类，该类实现了 Serializable 接口。

```
1    public  class  Employee  implements  java.io.Serializable
2    {
3          public  String name;
```

```
4          public  String address;
5          public  transient  int  SSN;
6          public  int  number;
7          public  void  mailCheck()
8          {
9                System.out.println("Mailing a check to "  + name
10                                                    +  " "  + address);
11         }
12   }
```

请注意，一个类的对象要想序列化成功，必须满足两个条件：该类必须实现 java.io.Serializable 对象；该类的所有属性必须是可序列化的。如果有一个属性不是可序列化的，则该属性必须注明是短暂的。

如果你想知道一个 Java 标准类是否是可序列化的，请查看该类的文档。检验一个类的实例是否能序列化十分简答， 只需要查看该类有没有实现 java.io.Serializable 接口。

序列化对象

ObjectOutputStream 类用来序列化一个对象，如下的 SerializeDemo 例子实例化了一个 Employee 对象，并将该对象序列化到一个文件中。

该程序执行后，就创建了一个名为 employee.ser 文件。该程序没有任何输出，但是你可以通过代码研读来理解程序的作用。

注意： 当序列化一个对象到文件时， 按照 Java 的标准约定是给文件一个.ser 扩展名。

```
1    import  java.io.*;
2
3    public  class  SerializeDemo
4    {
5          public  static  void  main(String [] args)
```

```
6          {
7                Employee e =  new  Employee();
8                e.name =  "Reyan Ali";
9                e.address =  "Phokka Kuan, Ambehta Peer";
10               e.SSN =  11122333;
11               e.number =  101;
12               try
13               {
14                     FileOutputStream fileOut =
15                     new  FileOutputStream("/tmp/employee.ser");
16                     ObjectOutputStream out =  new  ObjectOutputStream(fileOut);
17                     out.writeObject(e);
18                     out.close();
19                     fileOut.close();
20                     System.out.printf("Serialized data is saved in /tmp/employee.ser");
21               }catch(IOException i)
22               {
23                       i.printStackTrace();
24               }
25         }
26   }
```

反序列化对象

下面的 DeserializeDemo 程序反序列化在 SerializeDemo 程序中创建 Employee 对象。

```
1    import  java.io.*;
2    public  class  DeserializeDemo
3    {
4          public  static  void  main(String [] args)
```

```
5          {
6                Employee e =  null;
7                try
8                {
9                      FileInputStream fileIn =  new  FileInputStream("/tmp/employee.ser");
10                     ObjectInputStream in =  new  ObjectInputStream(fileIn);
11                     e = (Employee) in.readObject();
12                     in.close();
13                     fileIn.close();
14               }catch(IOException i)
15               {
16                     i.printStackTrace();
17                     return;
18               }catch(ClassNotFoundException c)
19               {
20                     System.out.println("Employee class not found");
21                     c.printStackTrace();
22                     return;
23               }
24               System.out.println("Deserialized Employee...");
25               System.out.println("Name: "  + e.name);
26               System.out.println("Address: "  + e.address);
27               System.out.println("SSN: "  + e.SSN);
28               System.out.println("Number: "  + e.number);
29           }
30   }
```

以上程序编译运行结果如下所示：

```
1    Deserialized Employee...
```

```
2    Name: Reyan Ali
3    Address:Phokka Kuan, Ambehta Peer
4    SSN:  0
5    Number:101
```

这里要注意以下要点：

readObject() 方法中的 try/catch 代码块尝试捕获 ClassNotFoundException 异常。对于 JVM 可以反序列化对象，它必须是能够找到字节码的类。如果 JVM 在反序列化对象的过程中找不到该类，则抛出一个 ClassNotFoundException 异常。

注意，readObject()方法的返回值被转化成 Employee 引用。

当对象被序列化时，属性 SSN 的值为 111222333，但是因为该属性是短暂的，该值没有被发送到输出流。所以反序列化后 Employee 对象的 SSN 属性为 0。

JAVA 培训教程—Java 发送邮件

使用 Java 应用程序发送 E-mail 十分简单,但是首先你应该在你的机器上安装 JavaMail API 和 Java Activation Framework (JAF) 。

你可以在 JavaMail (Version 1.2) 下载最新的版本。你可以再 在 JAF (Version 1.1.1) 下载最新的版本。

下载并解压这些文件，最上层文件夹你会发现很多的 jar 文件。你需要将 mail.jar 和 activation.jar 添加到你的 CLASSPATH 中。

发送一封简单的 E-mail

下面是一个发送简单 E-mail 的例子。假设你的 localhost 已经连接到网络。

```
1    // 文件名 SendEmail.java
2
3    import  java.util.*;
4    import  javax.mail.*;
5    import  javax.mail.internet.*;
6    import  javax.activation.*;
7
8    public  class  SendEmail
9    {
10         public  static  void  main(String [] args)
11         {
12              // 收件人电子邮箱
13              String to =  "abcd@gmail.com";
14
15              // 发件人电子邮箱
16              String from =  "web@gmail.com";
```

```
17
18              // 指定发送邮件的主机为 localhost
19              String host =  "localhost";
20
21              // 获取系统属性
22              Properties properties = System.getProperties();
23
24              // 设置邮件服务器
25              properties.setProperty("mail.smtp.host", host);
26
27              // 获取默认 session 对象
28              Session session = Session.getDefaultInstance(properties);
29
30              try{
31                    // 创建默认的 MimeMessage 对象
32                    MimeMessage message =  new  MimeMessage(session);
33
34                    // Set From: 头部头字段
35                    message.setFrom(new  InternetAddress(from));
36
37                    // Set To: 头部头字段
38                    message.addRecipient(Message.RecipientType.TO,
39                                                                  new  InternetAddress(to
40
41                    // Set Subject: 头部头字段
42                    message.setSubject("This is the Subject Line!");
43
44                    // 设置消息体
45                    message.setText("This is actual message");
```

```
46
47                      // 发送消息
48                      Transport.send(message);
49                      System.out.println("Sent message successfully....");
50              }catch  (MessagingException mex) {
51                      mex.printStackTrace();
52              }
53          }
54    }
```

编译并运行这个程序来发送一封简单的 E-mail：

```
1    $ java SendEmail
2    Sent message successfully....
```

如果你想发送一封 e-mail 给多个收件人，那么使用下面的方法来指定多个收件人 ID：

```
1    void  addRecipients(Message.RecipientType type,
2                                          Address[] addresses)
3    throws  MessagingException
```

下面是对于参数的描述：

- **type:**要被设置为 TO, CC 或者 BCC. 这里 CC 代表抄送、BCC 代表秘密抄送 y. 举例：*Message.RecipientType.TO*
- **addresses:** 这是 email ID 的数组。在指定电子邮件 ID 时，你将需要使用 InternetAddress()方法。

发送一封 HTML E-mail

下面是一个发送 HTML E-mail 的例子。假设你的 localhost 已经连接到网络。

和上一个例子很相似，除了我们要使用 setContent()方法来通过第二个参数为 "text/html"，来设置内容来指定要发送 HTML 内容。

```
1    // 文件名 SendHTMLEmail.java
2
3    import  java.util.*;
4    import  javax.mail.*;
5    import  javax.mail.internet.*;
6    import  javax.activation.*;
7
8    public  class  SendHTMLEmail
9    {
10         public  static  void  main(String [] args)
11         {
12
13              // 收件人电子邮箱
14              String to =  "abcd@gmail.com";
15
16              // 发件人电子邮箱
17              String from =  "web@gmail.com";
18
19              // 指定发送邮件的主机为 localhost
20              String host =  "localhost";
21
22              // 获取系统属性
23              Properties properties = System.getProperties();
24
25              // 设置邮件服务器
26              properties.setProperty("mail.smtp.host", host);
27
```

```
28              // 获取默认的 Session 对象。
29              Session session = Session.getDefaultInstance(properties);
30
31              try{
32                    // 创建默认的 MimeMessage 对象。
33                    MimeMessage message =  new  MimeMessage(session);
34
35                    // Set From: 头部头字段
36                    message.setFrom(new  InternetAddress(from));
37
38                    // Set To: 头部头字段
39                    message.addRecipient(Message.RecipientType.TO,
40                                                                  new  InternetAddress(to
41
42                    // Set Subject: 头字段
43                    message.setSubject("This is the Subject Line!");
44
45                    // 发送 HTML 消息, 可以插入 html 标签
46                    message.setContent("<h1>This is actual message</h1>",
47                                                          "text/html"  );
48
49                    // 发送消息
50                    Transport.send(message);
51                    System.out.println("Sent message successfully....");
52              }catch  (MessagingException mex) {
53                    mex.printStackTrace();
54              }
55        }
56  }
```

编译并运行此程序来发送 HTML e-mail：

```
$ java SendHTMLEmail
Sent message successfully....
```

发送带有附件的 E-mail

下面是一个发送带有附件的 E-mail 的例子。假设你的 localhost 已经连接到网络。

```
// 文件名 SendFileEmail.java

import  java.util.*;
import  javax.mail.*;
import  javax.mail.internet.*;
import  javax.activation.*;

public  class  SendFileEmail
{
      public  static  void  main(String [] args)
      {

            // 收件人电子邮箱
            String to =  "abcd@gmail.com";

            // 发件人电子邮箱
            String from =  "web@gmail.com";

            // 指定发送邮件的主机为 localhost
            String host =  "localhost";

            // 获取系统属性
```

```
23              Properties properties = System.getProperties();
24
25              // 设置邮件服务器
26              properties.setProperty("mail.smtp.host", host);
27
28              // 获取默认的 Session 对象。
29              Session session = Session.getDefaultInstance(properties);
30
31              try{
32                    // 创建默认的 MimeMessage 对象。
33                    MimeMessage message =  new  MimeMessage(session);
34
35                    // Set From: 头部头字段
36                    message.setFrom(new  InternetAddress(from));
37
38                    // Set To: 头部头字段
39                    message.addRecipient(Message.RecipientType.TO,
40                                                                  new  InternetAddress(to
41
42                    // Set Subject: 头字段
43                    message.setSubject("This is the Subject Line!");
44
45                    // 创建消息部分
46                    BodyPart messageBodyPart =  new  MimeBodyPart();
47
48                    // 消息
49                    messageBodyPart.setText("This is message body");
50
51                    // 创建多重消息
```

```
52                      Multipart multipart =  new  MimeMultipart();
53
54                      // 设置文本消息部分
55                      multipart.addBodyPart(messageBodyPart);
56
57                      // 附件部分
58                      messageBodyPart =  new  MimeBodyPart();
59                      String filename =  "file.txt";
60                      DataSource source =  new  FileDataSource(filename);
61                      messageBodyPart.setDataHandler(new  DataHandler(source));
62                      messageBodyPart.setFileName(filename);
63                      multipart.addBodyPart(messageBodyPart);
64
65                      // 发送完整消息
66                      message.setContent(multipart );
67
68                      //     发送消息
69                      Transport.send(message);
70                      System.out.println("Sent message successfully....");
71              }catch  (MessagingException mex) {
72                      mex.printStackTrace();
73              }
74          }
75   }
```

编译并运行你的程序来发送一封带有附件的邮件。

```
1   $ java SendFileEmail
2   Sent message successfully....
```

用户认证部分

如果需要提供用户名和密码给 e-mail 服务器来达到用户认证的目的，你可以通过如下设置来完成：

```
1    props.setProperty("mail.user",  "myuser");
2    props.setProperty("mail.password",  "mypwd");
```

e-mail 其他的发送机制和上述保持一致。

JAVA 培训教程—Java Applet 基础

applet 是一种 Java 程序。它一般运行在支持 Java 的 Web 浏览器内。因为它有完整的 Java API 支持,所以 applet 是一个全功能的 Java 应用程序。

如下所示是独立的 Java 应用程序和 applet 程序之间重要的不同：

- Java 中 applet 类继承了 java.applet.Applet 类
- Applet 类没有定义 main()，所以一个 Applet 程序不会调用 main()方法，
- Applets 被设计为嵌入在一个 HTML 页面。
- 当用户浏览包含 Applet 的 HTML 页面，Applet 的代码就被下载到用户的机器上。
- 要查看一个 applet 需要 JVM。 JVM 可以是 Web 浏览器的一个插件，或一个独立的运行时环境。
- 用户机器上的 JVM 创建一个 applet 类的实例，并调用 Applet 生命周期过程中的各种方法。
- Applets 有 Web 浏览器强制执行的严格的安全规则，applet 的安全机制被称为沙箱安全。
- applet 需要的其他类可以用 Java 归档（JAR）文件的形式下载下来。

Applet 的生命周期

Applet 类中的四个方法给你提供了一个框架，你可以再该框架上开发小程序：

- **init:** 该方法的目的是为你的 applet 提供所需的任何初始化。在 Applet 标记内的 param 标签被处理后调用该方法。
- **start:** 浏览器调用 init 方法后，该方法被自动调用。每当用户从其他页面返回到包含 Applet 的页面时，则调用该方法。
- **stop:**当用户从包含 applet 的页面移除的时候，该方法自动被调用。因此，可以在相同的 applet 中反复调用该方法。
- **destroy:** 此方法仅当浏览器正常关闭时调用。因为 applets 只有在 HTML 网页上有效，所以你不应该在用户离开包含 Applet 的页面后遗漏任何资源.

- **paint:** 该方法在 start()方法之后立即被调用，或者在 applet 需要重绘在浏览器的时候调用。paint()方法实际上继承于 java.awt。

"Hello, World" Applet:

下面是一个简单的 Applet 程序 HelloWorldApplet.java:

```
1    import  java.applet.*;
2    import  java.awt.*;
3
4    public  class  HelloWorldApplet  extends  Applet
5    {
6          public  void  paint (Graphics g)
7          {
8                g.drawString ("Hello World",  25,  50);
9          }
10   }
```

这些 import 语句将以下类导入到我们的 applet 类中：

```
1    java.applet.Applet.
2    java.awt.Graphics.
```

没有这些 import 语句，Java 编译器就识别不了 Applet 和 Graphics 类。

Applet 类

每一个 applet 都是 java.applet.Applet 类的子类，基础的 Applet 类提供了供衍生类调用的方法,以此来得到浏览器上下文的信息和服务。

这些方法做了如下事情：

- 得到 applet 的参数

- 得到包含 applet 的 HTML 文件的网络位置
- 得到 applet 类目录的网络位置
- 打印浏览器的状态信息
- 获取一张图片
- 获取一个音频片段
- 播放一个音频片段
- 调整此 applet 的大小

除此之外，Applet 类还提供了一个接口，该接口供 Viewer 或浏览器来获取 applet 的信息，并且来控制 applet 的执行。

Viewer 可能是：

- 请求 applet 作者、版本和版权的信息
- 请求 applet 识别的参数的描述
- 初始化 applet
- 销毁 applet
- 开始执行 applet
- 结束执行 applet

Applet 类提供了对这些方法的默认实现，这些方法可以在需要的时候重写。

"Hello，World"applet 都是按标准编写的。唯一被重写的方法是 paint 方法。

Applet 的调用

applet 是一种 Java 程序。它一般运行在支持 Java 的 Web 浏览器内。因为它有完整的 Java API 支持,所以 applet 是一个全功能的 Java 应用程序。

<applet>标签是在 HTML 文件中嵌入 applet 的基础。以下是一个调用"Hello World"applet 的例子；

```
1   <html>
```

```
2    <title>The Hello, World Applet</title>
3    <hr>
4    <applet code="HelloWorldApplet.class"  width="320"  height="120">
5    If your browser was Java-enabled, a  "Hello, World"
6    message would appear here.
7    </applet>
8    <hr>
9    </html>
```

注意： 你可以参照 HTML Applet 标签来更多的了解从 HTML 中调用 applet 的方法。

<applet>标签的属性指定了要运行的 Applet 类。Width 和 height 用来指定 applet 运行面板的初始大小。applet 必须使用</applet>标签来关闭。

如果 applet 接受参数，那么参数的值需要在标签里添加，该标签位于<applet>和</applet>之间。浏览器忽略了 applet 标签之间的文本和其他标签。

不支持 Java 的浏览器不能执行<applet>和</applet>。因此，在标签之间显示并且和 applet 没有关系的任何东西，在不支持的 Java 的浏览器里是可见的。

Viewer 或者浏览器在文档的位置寻找编译过的 Java 代码，要指定文档的路径，得使用<applet>标签的 codebase 属性指定。

如下所示：

```
1    <applet codebase="http://amrood.com/applets"
2    code="HelloWorldApplet.class"  width="320"  height="120">
```

如果 applet 所在一个包中而不是默认包，那么所在的包必须在 code 属性里指定，例如：

```
1    <applet code="mypackage.subpackage.TestApplet.class"
2                          width="320"  height="120">
```

获得 applet 参数

下面的例子演示了如何使用一个 applet 响应来设置文件中指定的参数。该 Applet 显示了一个黑色棋盘图案和第二种颜色。

第二种颜色和每一列的大小通过文档中的 applet 的参数指定。

CheckerApplet 在 init()方法里得到它的参数。也可以在 paint()方法里得到它的参数。然而，在 applet 开始得到值并保存了设置，而不是每一次刷新的时候都得到值，这样是很方便，并且高效的。

applet viewer 或者浏览器在 applet 每次运行的时候调用 init()方法。在加载 applet 之后，Viewer 立即调用 init()方法（Applet.init()什么也没做），重写该方法的默认实现，添加一些自定义的初始化代码。

Applet.getParameter()方法通过给出参数名称得到参数值。如果得到的值是数字或者其他非字符数据，那么必须解析为字符串类型。

下例是 CheckerApplet.java 的梗概：

```
1    import  java.applet.*;
2    import  java.awt.*;
3    public  class  CheckerApplet  extends  Applet
4    {
5          int  squareSize =  50;// 初始化默认大小
6          public  void  init () {}
7          private  void  parseSquareSize (String param) {}
8          private  Color parseColor (String param) {}
9          public  void  paint (Graphics g) {}
10   }
```

下面是 CheckerApplet 类的 init()方法和私有的 parseSquareSize()方法：

```
1    public  void  init ()
2    {
3          String squareSizeParam = getParameter ("squareSize");
4          parseSquareSize (squareSizeParam);
5          String colorParam = getParameter ("color");
6          Color fg = parseColor (colorParam);
7          setBackground (Color.black);
8          setForeground (fg);
9    }
10   private  void  parseSquareSize (String param)
11   {
12         if  (param ==  null)  return;
13         try  {
14               squareSize = Integer.parseInt (param);
15         }
16         catch  (Exception e) {
17             // 保留默认值
18         }
19   }
```

该 applet 调用 parseSquareSize()，来解析 squareSize 参数。parseSquareSize()调用了库方法 Integer. parseInt()，该方法将一个字符串解析为一个整数，当参数无效的时候，Integer.parseInt()抛出异常。

因此，parseSquareSize()方法也是捕获异常的，并不允许 applet 接受无效的输入。

Applet 调用 parseColor()方法将颜色参数解析为一个 Color 值。parseColor()方法做了一系列字符串的比较，来匹配参数的值和预定义颜色的名字。你需要实现这些方法来使 applet 工作。

指定 applet 参数

如下的例子是一个 HTML 文件，其中嵌入了 CheckerApplet 类。HTML 文件通过使用标签的方法给 applet 指定了两个参数。

```
1   <html>
2   <title>Checkerboard Applet</title>
3   <hr>
4   <applet code="CheckerApplet.class"  width="480"  height="320">
5   <param name="color"  value="blue">
6   <param name="squaresize"  value="30">
7   </applet>
8   <hr>
9   </html>
```

注意： 参数名字大小写不敏感。

应用程序转换成 Applet

将图形化的 Java 应用程序（是指，使用 AWT 的应用程序和使用 java 程序启动器启动的程序）转换成嵌入在 web 页面里的 applet 是很简单的。

下面是将应用程序转换成 applet 的几个步骤：

- 编写一个 HTML 页面，该页面带有能加载 applet 代码的标签。
- 编写一个 JApplet 类的子类，将该类设置为 public。否则，applet 不能被加载。
- 消除应用程序的 main()方法。不要为应用程序构造框架窗口，因为你的应用程序要显示在浏览器中。
- 将应用程序中框架窗口的构造方法里的初始化代码移到 applet 的 init()方法中，你不必显示的构造 applet 对象，浏览器将通过调用 init()方法来实例化一个对象。
- 移除对 setSize()方法的调用，对于 applet 来讲，大小已经通过 HTML 文件里的 width 和 height 参数设定好了。
- 移除对 setDefaultCloseOperation()方法的调用。Applet 不能被关闭，它随着浏览器的退出而终止。

- 如果应用程序调用了 setTitle()方法，消除对该方法的调用。applet 不能有标题栏。（当然你可以给通过 html 的 title 标签给网页自身命名）
- 不要调用 setVisible(true),applet 是自动显示的。

事件处理

Applet 类从 Container 类继承了许多事件处理方法。Container 类定义了几个方法，例如：processKeyEvent()和 processMouseEvent()，用来处理特别类型的事件，还有一个捕获所有事件的方法叫做 processEvent。

为了响应一个事件，applet 必须重写合适的事件处理方法。

```
1    import  java.awt.event.MouseListener;
2    import  java.awt.event.MouseEvent;
3    import  java.applet.Applet;
4    import  java.awt.Graphics;
5
6    public  class  ExampleEventHandling  extends  Applet
7                                                    implements  MouseListener {
8
9            StringBuffer strBuffer;
10
11           public  void  init() {
12                     addMouseListener(this);
13                     strBuffer =  new  StringBuffer();
14                   addItem("initializing the apple ");
15           }
16
17           public  void  start() {
18                   addItem("starting the applet ");
19           }
```

```
20
21          public  void  stop() {
22                  addItem("stopping the applet ");
23          }
24
25          public  void  destroy() {
26                  addItem("unloading the applet");
27          }
28
29          void  addItem(String word) {
30                  System.out.println(word);
31                  strBuffer.append(word);
32                  repaint();
33          }
34
35          public  void  paint(Graphics g) {
36                    //Draw a Rectangle around the applet's display area.
37                  g.drawRect(0,  0,
38                                                  getWidth() -  1,
39                                                  getHeight() -  1);
40
41                    //display the string inside the rectangle.
42                  g.drawString(strBuffer.toString(),  10,  20);
43          }
44
45
46          public  void  mouseEntered(MouseEvent event) {
47          }
48          public  void  mouseExited(MouseEvent event) {
```

```
49          }
50          public  void  mousePressed(MouseEvent event) {
51          }
52          public  void  mouseReleased(MouseEvent event) {
53          }
54
55          public  void  mouseClicked(MouseEvent event) {
56                    addItem("mouse clicked! ");
57          }
58   }
```

如下调用该 applet：

```
1   <html>
2   <title>Event Handling</title>
3   <hr>
4   <applet code="ExampleEventHandling.class"
5   width="300"  height="300">
6   </applet>
7   <hr>
8   </html>
```

最开始运行，applet 显示 "initializing the applet. Starting the applet."，然后你一点击矩形框，就会显示"mouse clicked" 。

显示图片

applet 能显示 GIF,JPEG,BMP 等其他格式的图片。为了在 applet 中显示图片，你需要使用 java.awt.Graphics 类的 drawImage()方法。

如下实例演示了显示图片的所有步骤：

```
1    import  java.applet.*;
2    import  java.awt.*;
3    import  java.net.*;
4    public  class  ImageDemo  extends  Applet
5    {
6        private  Image image;
7        private  AppletContext context;
8        public  void  init()
9        {
10               context =  this.getAppletContext();
11               String imageURL =  this.getParameter("image");
12               if(imageURL ==  null)
13               {
14                     imageURL =  "java.jpg";
15               }
16               try
17               {
18                     URL url =  new  URL(this.getDocumentBase(), imageURL);
19                     image = context.getImage(url);
20               }catch(MalformedURLException e)
21               {
22                     e.printStackTrace();
23                     // Display in browser status bar
24                     context.showStatus("Could not load image!");
25               }
26         }
27         public  void  paint(Graphics g)
28         {
29               context.showStatus("Displaying image");
```

```
30                g.drawImage(image,  0,  0,  200,  84,  null);
31                g.drawString("www.javalicense.com",  35,  100);
32          }
33    }
```

如下调用该 applet：

```
1    <html>
2    <title>The ImageDemo applet</title>
3    <hr>
4    <applet code="ImageDemo.class"  width="300"  height="200">
5    <param name="image"  value="java.jpg">
6    </applet>
7    <hr>
8    </html>
```

播放音频

Applet 能通过使用 java.applet 包中的 AudioClip 接口播放音频。AudioClip 接口定义了三个方法：

- **public void play()：** 从一开始播放音频片段一次。
- **public void loop()：** 循环播放音频片段
- **public void stop()：** 停止播放音频片段

为了得到 AudioClip 对象，你必须调用 Applet 类的 getAudioClip()方法。无论 URL 指向的是否是一个真实的音频文件，该方法都会立即返回结果。

直到要播放音频文件时，该文件才会下载下来。

如下实例演示了播放音频的所有步骤：

```
1    import  java.applet.*;
```

```
2    import  java.awt.*;
3    import  java.net.*;
4    public  class  AudioDemo  extends  Applet
5    {
6          private  AudioClip clip;
7          private  AppletContext context;
8          public  void  init()
9          {
10               context =  this.getAppletContext();
11               String audioURL =  this.getParameter("audio");
12               if(audioURL ==  null)
13               {
14                     audioURL =  "default.au";
15               }
16               try
17               {
18                     URL url =  new  URL(this.getDocumentBase(), audioURL);
19                     clip = context.getAudioClip(url);
20               }catch(MalformedURLException e)
21               {
22                     e.printStackTrace();
23                     context.showStatus("Could not load audio file!");
24               }
25         }
26         public  void  start()
27         {
28               if(clip !=  null)
29               {
30                     clip.loop();
```

```
31              }
32          }
33          public  void  stop()
34          {
35                  if(clip !=  null)
36                  {
37                          clip.stop();
38                  }
39          }
40    }
```

如下调用 applet：

```
1   <html>
2   <title>The ImageDemo applet</title>
3   <hr>
4   <applet code="ImageDemo.class"  width="0"  height="0">
5   <param name="audio"  value="test.wav">
6   </applet>
7   <hr>
```