Java编程指南

极客学院出版

前言

Java 是一种高级的编程语言，它最初是由 Sun 公司开发并于 1995 年公开发布的。Java 可以在不同的平台上运行，例如：Windows，Mac OS 和不同版本的 Unix。本指南将让你对 Java 有一个彻底的认识与了解。

本指南将带你用简单并且实用的方法来学习 Java 编程语言。

适用人群

本指南是为那些初学者准备的，可以帮助他们理解编程语言从低级到高级的概念。

学习前提

在你准备应用本指南中所举的不同种类的例子进行练习之前，我们假设你已经知道什么是计算机程序和什么是计算机编程语言了。

编译、执行 Java 程序

在本指南所提供的大多数例子中你将会看到试一试选项，所以大胆的应用这个选项并且享受这个学习的过程。

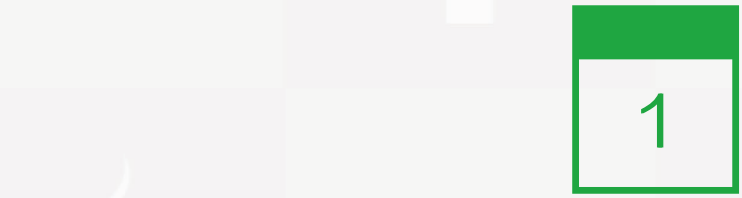
看下面这个例子:

``` public class MyFirstJavaProgram {

```
public static void main(String []args) {
   System.out.println("Hello World");
}
```

} ```
```

目录



1 Java 基础

Java 概述

Java 编程语言最初是由太阳计算机系统公司开发的，该公司由 James gosling 于 1995 年创立，它的主要组成部分就是 Java 平台。

截止到 2008 年 12 月，最新发布的 Java 标准版本是第六版（J2SE）。随着 Java 的发展进步和它的广泛流行， Java 做出了很多调整从而适应不同类型的平台。例如： J2EE 是为企业级应用程序设计的， J2ME 是为移动应用程序设计的。

Sun 计算机系统有限公司将新的 J2 版本分别命名为 Java SE、Java EE 和 Java ME。 Java 承诺 “**编写一次，随处运行**”。

Java 是：

- **面向对象的**：在 Java 中，所有的东西都是一个对象。 Java 可以很容易的扩展原因就是因为它是基于对象模型的。
- **平台独立的**： Java 不像包括 C 和 C++ 语言在内的其他语言，当 Java 被编译时，它并不是编译到特定的机器中，而是用具有平台独立性的字节码编译。这种字节码在网页上是分布式存储的，并且可以再不同的平台通用的虚拟机上运行。
- **简单的**：Java 是为了易于学习而设计的。如果你能够理解面向对象程序设计的基本概念，那么你就很容易掌握 Java 了。
- **安全的**：由于 Java 的安全特征它可以用来开发防病毒、防干扰的系统。它的身份验证技术是基于公开密钥加密技术的。
- **体系结构中立的**：Java 编译器可以生成一个结构中立的对象文件格式，它能够使被编译过的代码在 Java 运行系统存在的情况下在很多进程中运行。
- **便捷的**：由于 Java 的结构中立性以及它的运行不受限制的特征使得它十分便捷， Java 中的编译器是由 ANSI C 语言编写的，它具有很好的接口是因为它是可移植性系统操作接口的子接口。
- **稳健的**：Java 主要通过检查编译时间错误以及运行检查来努力消除有错误倾向的情况。
- **多线程的**：因为 Java 的多线程性的特征使得它可以用于编写同时执行众多任务的程序。这个特征可以使开发者平稳设计运行交互式应用程序。
- **易于理解的**：Java 的字节码可以很容易翻译成本地机器码并且很容易存储于任意位置。这个发展进程很快并且很有分析性，因为他们的联系就像增加和减轻体重的过程。
- **高性能**：由于使用了准时编译器，它可以表现出很高的性能。

- **分布式的**：Java 是为互联网分布式的结构而设计的。
- **动态的**：Java 被认为是比 C 和 C++ 更有活力的语言因为它是为适应动态环境而设计的。Java 程序可以携带大量的运行信息，这些信息可以用来区分对象间的入口问题。

Java 的历史

James Gosling 在 1991 年启动 Java 语言项目是为了在他的众多的电视机顶盒项目之一上应用，这种语言起初被称为 Oak ，是因为 Gosling 的办公室外有一棵像树，也被称为过 Green ，最终被以一些随机的字母命名为 Java。

Sun 计算机有限公司于 1995 年发行第一个公开版本 Java1.0。它承诺“**编写一次，随处运行**”（WORA），并且提供在任意主流平台上无成本运行。

2006 年 11 月 13 日，根据 GNU 的通用公共许可证（GPL），太阳计算机有限公司发布了许多免费且开源的 Java 应用程序。

2007 年 5 月 8 日，太阳计算机有限公司完成了一项进程，它将除去一小部分没有版权的代码外的核心代码全部免费且开源。

你所需要的工具

为了试验本指南中所给的例子，你需要一台处理器为奔腾 200MHz，最小内存 64MB（推荐128MB）的电脑。

你还需要下列软件：

- Linux 7.1 或者 Windows 95/98/2000/XP 操作系统
- Java JDK 5
- 微软记事本程序或其他文本编辑器

本指南将提供用 Java 创建图形用户界面，网络及网页应用程序的必要技巧。

接下来是什么呢?

下一章我们将告诉你去哪里可以获得 Java 以及它的参考资料。
最后，它将指导你如何安装 Java 并告诉你如何为开发 Java 应用程序做开发环境方面的准备。

Java 环境设置

本地环境设置

如果你依然想要为 Java 编程语言设置环境，那么本节将指导你如何在你的电脑上下载和设置 Java。请按照以下步骤进行环境设置。

Java SE 可以从下载 Java 这个链接免费下载。你可以根据你的系统类型下载相应版本的 Java。

按照上述指导下载 Java 然后运行 .exe 文件进行安装。你在电脑上安装完 Java 之后，你需要将环境变量设置到指定目录。

Windows XP/7/8 系统下的设置方法

假设你把 Java 安装在 `c:\Program Files\java\jdk` 路径下：

- 右键点击「我的电脑」选择「属性」选项
- 在高级标签下点击「环境变量」按钮
- 现在，改变变量的路径使其包含可执行的 Java 程序。
  例如：如果现在的路径设置的是
  `C:\WINDOWS\SYSTEM32`，
  那么就要将其改成 `C:\WINDOWS\SYSTEM32;c:\ProgramFiles\java\jdk\bin`。

Linux, UNIX, Solaris, FreeBSD 系统下的设置方法

环境变量路径必须指向 Java 文件的安装位置。如果进行该设置时有任何问题，请参考shell帮助文档。

例如，如果你用 bash 作为你的 shell ，那么在你的 shell 最后加入如下代码
`.bashrc: export PATH=/path/to/java:$PATH`

流行的 Java 编辑器

在编写 Java 程序时，你需要一个文本编辑器。市场中有很多精致的编辑器。但是就现在而言，你可以考虑下面几个：

- **记事本**：在 Windows 计算机中你可以使用像记事本（本指导推荐），日记本这样的简单的文本编辑器。
- **Netbeans**：这是一款开源且免费的 Java 编辑器。你可以从以下链接下载 http://www.netbeans.org/index.html
- **Eclipse**：这是一款由 eclipse 开源社区开发的 Java 编辑器。你可以从以下链接下载 http://www.eclipse.org/

接下来是什么呢?

下一章我们将教你如何编写并且运行你的第一个程序和一些开发应用程序所必备的重要语法。

Java 基本语法

Java 应用程序可以被定义为对象的集合，这些对象通过调用各自的方法来进行通信。下面让我们具体看一看类，对象，方法，实体变量是什么含义。

- **对象:**对象具有状态和行为。例如：狗有它的状态—颜色，名字，品种，同时也有行为—摇尾巴，汪汪叫，吃东西。对象时类的一个实例。
- **类:**类可以被定义为描述对象所支持的类型的行为和状态的模板或蓝图。
- **方法:**方法是一种基本的行为。类中可以包含很多方法。在方法中，可以编写逻辑，操纵数据，执行动作。
- **实体变量:**每个对象都有它的特殊的实体变量的集合，一个对象的状态是由那些实体变量所被赋的值所决定的。

第一个 Java 程序

让我们看一下下面可以输出 “Hello World” 的代码。

```
public class MyFirstJavaProgram {

  /* This is my first java program.
   * This will print 'Hello World' as the output
   */

  public static void main(String []args) {
    System.out.println("Hello World"); // prints Hello World
  }
}
```

让我们看一下如何保存这个文件，编译运行这个程序。请按照以下步骤操作：

- 打开记事本添加上述代码
- 以 `MyFirstJavaProgram.java` 为文件名保存文件
- 打开命令提示符窗口转到你保存类的位置。假设是 `C:\`
- 在窗口中输入 `javac MyFirstJavaProgram.java` 然后按回车来编译你的代码。如果你的代码没有错误，那么命令提示符将会转到下一行（假设：路径变量设置成功）。
- 现在输入 `java MyFirstJavaProgram` 来运行你的程序
- 你将会看到屏幕上显示 “Hello World”

```
C : > javac MyFirstJavaProgram.java
C : > java MyFirstJavaProgram
Hello World
```

基本语法

关于 Java 程序，记住一下几点很重要。

- **大小写敏感性**： Java 是一种大小写敏感的语言，这就意味着 Hello 和 hello 在 Java 中代表不同的意思。
- **类的命名**：所有类的名称首字母必须大写。
  如果类名称中包含几个单词，那么每个单词的首字母都要大写。
  例如类 `MyFirstJavaClass`
- **方法的命名**：所有方法名称必须以小写字母开头。
  如果方法名称中包含几个单词，那么其中的每个单词的首字母都要大写。
  例如 `public void myMethodName()`
- **程序文件名**：程序的文件名必须和类的名称准确匹配。
  但保存文件时，你应当以类的名称保存（注意区分大小写），并在文件名后加 .java 的后缀（如果文件名和类名不匹配那么将无法编译你的程序）。
  例如：假设类名是 `MyFirstJavaProgram`，那么文件名就应该是 `MyFirstJavaProgram.java` 。
- public static void main(String args[])：Java 程序都是从 `main（）` 方法开始处理的，这个方法是 Java 程序的强制性的部分。

Java 标识符

Java 的所有的组成部分都要有自己的名称。类、变量和方法的名称称为标识符。

在 Java 中，需要记住关于标识符的一下几点。如下：

- 所有标识符必须以字母（ A 到 Z 或者 a 到 z ）、货币字符（ $ ）或者下划线(_)开头。
- 在第一个标识符之后可以有任意字母组合。
- 关键字不能被用作标识符。
- 大多数标识符需要区分大小写。
- 合法标识符的例子： age, $salary, _value, __1_value
- 非法标识符的例子： 123abc, -salary

Java 修饰符

如其语言一样，方法和类等等是可以通过修饰符修饰的。Java 中有两种修饰符：

- **访问修饰符**：default, public , protected, private
- **非访问修饰符**：final, abstract, strictfp

我们将在下一节继续学习修饰符相关知识。

Java 变量

在 Java 中我们可以看到如下变量：

- 本地变量
- 类变量（静态变量）
- 实例变量（非静态变量）

Java 数组

数组时储存有多重相同变量类型的对象。然而，数字自身也是堆中的一个对象。我们将要学习如何声明，建立，初始化数组。

Java 枚举值

枚举是在 Java5.0 版本中被引进的。枚举限制了变量要有一些预先定义的值。枚举列表中的值称为枚举值。

运用枚举值可以大大减少你的代码中的漏洞。

举例来说，如果我们想为一家鲜榨果汁店编个程序，就可以将杯子的尺寸限制为小中和大。这样就可以确保人们不会定大中小尺寸之外的了。

例如：

```
class FreshJuice {

  enum FreshJuiceSize{ SMALL, MEDIUM, LARGE }
  FreshJuiceSize size;
```

```
}

public class FreshJuiceTest {

  public static void main(String args[]){
    FreshJuice juice = new FreshJuice();
    juice.size = FreshJuice. FreshJuiceSize.MEDIUM ;
    System.out.println("Size: " + juice.size);
  }
}
```

上述例子会输出如下结果：

```
Size: MEDIUM
```

> 注：枚举可以自己声明也可以在类中声明。方法变量和构造器也可以在枚举值中定义。

Java 关键字

下面列出的是 Java 中保留的关键字。这些关键字不能用作常量、变量和其他标识符的名字。

关键字	关键字	关键字	关键字
abstract	assert	boolean	break
byte	case	catch	char
class	const	continue	default
do	double	else	enum
extends	final	finally	float
for	goto	if	implements
import	instanceof	int	interface
long	native	new	package
private	protected	public	return
short	static	strictfp	super
switch	synchronized	this	throw
throws	transient	try	void
volatile	while		

Java 中的注释

Java 像 C 和 C++ 一样支持单行或多行注释。所有注释中的字母都会被 Java 编译器忽略。

```
public class MyFirstJavaProgram{

  /* This is my first java program.
   * This will print 'Hello World' as the output
   * This is an example of multi-line comments.
   */

   public static void main(String []args){
      // This is an example of single line comment
      /* This is also an example of single line comment. */
      System.out.println("Hello World");
   }
}
```

使用空行

一行只有空格的行可能是注释，这样的行叫做空行，Java 会完全忽略它。

继承

在 Java 中类可以从类中产生。简单来说，如果你想要创建一个新类并且现在已经存在一个包含你所需要代码的类，那么就有可能从这些存在的代码创建你的类。

这个概念可以使你在没有在新类中重写代码的情况下重复利用文件和方法。在这种情况下已经存在的类叫做超类，后来产生的类叫做子类。

接口

在 Java 语言中，接口可以定义为对象之间如何通信的合同。就继承性而言接口扮演了重要角色。

接口定义了子类所需要用的方法。但是方法的实施还是取决于子类。

接下来是什么呢?

下一节将讲解Java编程中的对象和类。在该章节末你就会清楚了解什么是Java中的对象和类。

Java 的对象和类

Java 是一种面向对象的语言。作为一种具有面向对象特征的语言
Java 包括以下几项基本概念：

- 多态性
- 继承性
- 封装性
- 抽象性
- 类
- 对象
- 实例
- 消息解析

在这一章，我们将深入学习类和对象。

- **对象**：对象具有状态和行为。如果我们考虑现实世界我们可以在我们身边找到很多对象，小汽车，狗，人类等等。所有这些对象都具有状态和行为。
- **类**：类可以被定义为描述对象支持类型的行为、状态的的模板、蓝图。

Java 中的对象

现在让我们深入学习对象。如果我们考虑一条狗，那么它的状态就是-名字，品种，颜色，它的行为就是汪汪，摇尾巴，跑。

如果我们将软件中的对象和现实世界中的对象对比，那么我们将会发现他们有很多相似的特征。软件中的对象也具有状态和行为。软件的状态被储存在文件中，它的行为通过方法来表现。

因此，在软件发展过程中方法控制对象的内在状态并且对象和对象间的交流通过方法来完成。

Java 中的类

类是有独立的对象创造出的蓝图。

下面给出了一个类的例子：

```
public class Dog{
  String breed;
  int age;
  String color;

  void barking(){
  }

  void hungry(){
  }

  void sleeping(){
  }
}
```

类可以包括以下的变量类型：

- **局部变量**：在方法，构造器或区域中定义的变量成为局部变量。变量将会在方法内产生和发展，然后当方法结束变量就会破坏。
- **实例变量**：实例变量是在类内但是在方法外的变量。这些变量是当类被装载时被实体化的。实例变量可以是从特定类的任何方法，构造器，区域中存取。
- **类变量**：类变量是在类中声明的变量，它处在任何方法之外，有静态关键字。

类可以有任意数量的方法来存取不同种类方法的值。在上面的例子中，`barking()`，`hungry()` 和 `sleeping()` 是方法。
下面提到的是一些深入了解 Java 语言所必须讨论的重要话题。

构造器

当我们讨论类时，其中一个重要的子话题就是构造器。每一个类都有一个构造器。如果我们不单独为一个类编写构造器那么 Java 的编译器将会给这个类建立一个默认的构造器。

每当一个新的对象被创建，至少一个构造器将会被调用。构造器的一个最主要的原则就是他们必须和类有同样的名字。一个类可以有不止一个构造器。

下面给出了一个构造器的例子：

```
public class Puppy{
  public Puppy(){
```

```
  }

  public Puppy(String name){
    // This constructor has one parameter, name.
  }
}
```

在需要只创建一个类的实例的时，Java 也支持单例。

创造一个对象

如前所述，类为对象提供了蓝图。因此基本来说一个对象是从一个类中创造出来的。在 Java 中，新的关键词被用来创造新的对象。当我们从类中创造对象时需要三步：

- **声明**：变量声明可以声明其所代表的对象类型。
- **实例化**：“新的”关键词用来创造对象。
- **初始化**：“新的”关键词伴随着一个构造器的启用，这个将新的对象初始化。

下面给出了一个创造对象的例子：

```
public class Puppy{

  public Puppy(String name){
    // This constructor has one parameter, name.
    System.out.println("Passed Name is :" + name );
  }
  public static void main(String []args){
    // Following statement would create an object myPuppy
    Puppy myPuppy = new Puppy( "tommy" );
  }
}
```

如果编译并运行上述程序，那么将输出下列结果：

```
Passed Name is :tommy
```

访问实体变量和方法

实体变量和方法是通过创造对象来访问的。为了访问一个实体变量完全有效的路径应该如下所示：

```
/* First create an object */
ObjectReference = new Constructor();
```

```
/* Now call a variable as follows */
ObjectReference.variableName;
/* Now you can call a class method as follows */
ObjectReference.MethodName();
```

例子

这个例子解释了如何存取类的实体变量和方法：

```
public class Puppy{

  int puppyAge;

  public Puppy(String name){
    // This constructor has one parameter, name.
    System.out.println("Passed Name is :" + name );
  }
  public void setAge( int age ){
     puppyAge = age;
  }

  public int getAge( ){
     System.out.println("Puppy's age is :" + puppyAge );
     return puppyAge;
  }
  public static void main(String []args){
    /* Object creation */
    Puppy myPuppy = new Puppy( "tommy" );

    /* Call class method to set puppy's age */
    myPuppy.setAge( 2 );

    /* Call another class method to get puppy's age */
    myPuppy.getAge( );

    /* You can access instance variable as follows as well */
    System.out.println("Variable Value :" + myPuppy.puppyAge );
  }
}
```

如果我们编译运行上述程序，那么将会产生如下结果：

```
Passed Name is :tommy
Puppy's age is :2
Variable Value :2
```

源文件声明规则

在本节的最后一部分让我们学习一下源文件声明规则。当在源文件中声明类，输入和打包语法时，这些规则是十分重要的。

- 每个源文件中只能有一个公共类。
- 一个源文件可以有很多非公共类。
- 公共类的名称必须是源文件的名称同时也要以 `.java` 为后缀。举例来说：类的名称是 `public class Employee{}` ，那么源文件就应该是 `Employee.java` 。
- 如果类是在一个程序包中定义的，那么程序包的声明必须是源文件的第一个声明。
- 如果输入声明出现那么他们必须被写在封装声明和类声明之间。如果没有封装声明那么输入声明必须在源文件的第一行。
- 输入和打包声明会暗示所有源文件中的存在的类。在源文件中很难为不同的类区分输入和封装声明。
  类有不同的访问级别并且有很多不同的类；抽象的类，最终的类等等。我将会在后面的访问控制修饰符章节解释这些。
  除了以上提到的类的类型之外， Java 也有像内部类和匿名类这样的特殊类。

Java 程序包

简而言之，就是一种分类类和接口的一种方法。当用 Java 开发程序时，数百个类和接口会被编写，因此分类这些类不但是必须的而且也是会使问题变得容易的。

Import 语法

在 Java 中，如果给出包括封装和类的名称的全限定名，那么编译器很容易定位到源类和源代码。Import 语法是给编译器寻找特定类的适当位置的一种方法。

举例来说，下面这行语句将会要求编译器去装载 `java_installation/java/io` 路径下的所有可用的类:

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

一个简单的案例学习

在我们的案例学习中，我们将创造两个类。他们是 Employee 和 EmployeeTest。

首先打开记事本输入下列代码。记得这个是 Employee 类，这个类是公共类。现在以 Employee.java 为文件名保存这个源文件。

这个 Employee 类包括四个实体变量姓名（name），年龄（age），职位（designation）和薪水（salary）。这个类有一个确定的需要参数的的构造器。

```
import java.io.*;
public class Employee{
   String name;
   int age;
   String designation;
   double salary;

   // This is the constructor of the class Employee
   public Employee(String name){
      this.name = name;
   }
   // Assign the age of the Employee  to the variable age.
   public void empAge(int empAge){
      age =  empAge;
   }
   /* Assign the designation to the variable designation.*/
   public void empDesignation(String empDesig){
      designation = empDesig;
   }
   /* Assign the salary to the variable    salary.*/
   public void empSalary(double empSalary){
      salary = empSalary;
   }
   /* Print the Employee details */
   public void printEmployee(){
      System.out.println("Name:"+ name );
      System.out.println("Age:" + age );
      System.out.println("Designation:" + designation );
      System.out.println("Salary:" + salary);
   }
}
```

正如本指导之前所述，编程从主函数开始。因此，我们为了运行这个 Employee 类就应当建立主函数和类。我们将分别为这些任务创建类。

下面所给的是 EmployeeTest 类，这个类创建了两个 Employee 类的实例，并且为每个对象调用方法从而给每个变量赋值。

在 EmployeeTest.java 文件中保存下列代码

```
import java.io.*;
public class EmployeeTest{

  public static void main(String args[]){
    /* Create two objects using constructor */
    Employee empOne = new Employee("James Smith");
    Employee empTwo = new Employee("Mary Anne");

    // Invoking methods for each object created
    empOne.empAge(26);
    empOne.empDesignation("Senior Software Engineer");
    empOne.empSalary(1000);
    empOne.printEmployee();

    empTwo.empAge(21);
    empTwo.empDesignation("Software Engineer");
    empTwo.empSalary(500);
    empTwo.printEmployee();
  }
}
```

现在编译两个类然后运行 EmployeeTest，你将会看到如下结果：

```
C :> javac Employee.java
C :> vi EmployeeTest.java
C :> javac EmployeeTest.java
C :> java EmployeeTest
Name:James Smith
Age:26
Designation:Senior Software Engineer
Salary:1000.0
Name:Mary Anne
Age:21
Designation:Software Engineer
Salary:500.0
```

接下来是什么呢?

下一章我们将讨论 Java 中的基本数据类型以及他们在开发应用程序时如何应用。

Java 基本数据类型

变量就是用来储存值而保留的内存位置。这就意味着当你创建一个变量时就会在内存中占用一定的空间。

基于变量的数据类型，操作系统会进行内存分配并且决定什么将被储存在保留内存中。因此，通过给变量分配不同的数据类型，你可以在这些变量中存储整数，小数或者字字母。

Java 中有两种有效地数据类型：

- 原始数据类型
- 引用数据类型

原始数据类型

Java 支持 8 种原始数据类型。原始数据类型是由该语言预先定义的并用关键词命名的。下面让我们深入学习一下这 8 种数据类型。

字节型（byte）

- 字节型是一种 8 位有正负的二进制整数
- 最小值是 −128(−2^7)
- 最大值是 127(2^7−1)
- 默认值为 0
- 字节型数据类型主要是为了在大型数组内节省空间，主要是替代整数由于字节型比整数小 4 倍。
- 例如：byte a = 100 , byte b = −50

短整数（short）

- 短整数是一种 16 位有正负的二进制整数
- 最小值是 −32768(−2^15)
- 最大值是 32767(2^15−1)
- 短整数类型的数据也可以像字节型一样用于节省空间。短整数比整数小两倍
- 默认值为 0

- 例如：short s = 10000, short r = −20000

整数型（int）

- 整数型是一种 32 位有正负的二进制整数
- 最小值是 − 2,147,483,648(−2^31)
- 最大值是 2,147,483,647(2^31 −1)
- 整数型一般默认被应用于整数值除非担心内存不够用。
- 默认值为 0
- 例如：int a = 100000, int b = −200000

长整型（long）

- 长整型是一种 64 位有正负的二进制整数
- 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)
- 这种数据类型一般是在需要比整数型范围更大时应用。
- 默认值为 0L
- 例如：long a = 100000L, int b = −200000L

浮点型（float）

- 浮点型数据是一种单精度的 32 位 IEEE 754 标准下的浮点数据。
- 浮点型数据主要是为了在大型浮点数字数组中节约内存。
- 默认值是 0.0f。
- 浮点型数据不能用于如货币这样的精确数据。
- 例如：float f1 = 234.5f

双精度型（double）

- 双精度型数据是一种双精度的 64 位 IEEE 754 标准下的浮点数据。

- 这种数据类型主要是默认被用于表示小数的值，一般是默认的选择。
- 双精度型数据不能用于如货币这样的精确数据。
- 默认值是 0.0d
- 例如：double d1 = 123.4

布尔型（boolean）

- 布尔型数据代表一个信息比特。
- 它只有两个可能的值：真（true）和假（false）
- 这种数据类型用于真假条件下的简单标记。
- 默认值是假（false）
- 例如：boolean one = true

字符型（char）

- 字符型数据是简单的 16 位 Unicode 标准下的字符。
- 最小值是：'\u0000' (或 0)。
- 最大值是：'\uffff' (或 65,535)。
- 字符型数据可以用来储存任意字母。
- 例如：char letter A（字符型的字母A）='A'

引用数据类型

- 引用数据类型是由类的编辑器定义的。他们是用于访问对象的。这些变量被定义为不可更改的特定类型。例如：Employee， Puppy 等等。
- 类对象和数组变量就是这种引用数据类型。
- 任何引用数据类型的默认值都为空。
- 一个引用数据类型可以被用于任何声明类型和兼容类型的对象。
- 例如：Animal animal = new Animal("giraffe");

Java 常量

常量是代表固定值的源代码。他们直接以代码的形式代表而没有任何估计。

常量可以被分配给任意的原始变量类型。例如：

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

字节型，整数型，长整型和短整型也可以由十进制，十六进制和八进制计数系统表示。

当用这些技术系统表示直接量时，前缀 0 是为了标明八进制，前缀 0x 是为了标明十六进制。例如：

```
int decimal = 100;
int octal = 0144;
int hexa = 0x64;
```

Java 中的字符串型常量的规定和其他大多数语言一样，也是要写在双引号中间。字符串型直接量的例子如下：

```
"Hello World"
"two\nlines"
"\"This is in quotes\""
```

字符和字符串型常量可以包含任意的 Unicode 字母。例如：

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

Java 语言也支持一些特殊的转义序列的字符和字符串直接量。他们是：

转义字符	含义
\n	换行 (0x0a)
\r	回车 (0x0d)
\f	换页 (0x0c)
\b	退格 (0x08)
\s	空格 (0x20)
\t	tab
\"	双引号
\'	单引号
\	反斜杠
\ddd	八进制字符 (ddd)
\uxxxx	十六进制 UNICODE 字符 (xxxx)

接下来是什么呢?

本章向你介绍不同种类的数据类型，下一章讲为你讲解不同的变量类型以及他们的用法。这将让你能够更好地理解他们如何在Java的类和实例等等中的应用。

Java 变量类型

变量可以给我们提供我们程序可以操纵的命名的存储。Java 中的每种变量都有特定的类型，这决定了变量的大小和它的设计占用内存空间；这一些列的值可以存储在那个内存空间中；变量可以应用的操作。

在使用前你必须现将所要使用的变量进行声明。声明变量的基本格式如下：

```
data type variable [ = value][, variable [= value] ...] ;
```

这里的 data type 是 Java 的一种数据类型，variable 是一种变量的名称。要声明一个以上的特定变量类型，你可以采用逗号分隔开。

下面是 Java 中有效的变量声明和赋值的例子：

```
int a, b, c; // Declares three ints, a, b, and c.
int a = 10, b = 10; // Example of initialization
byte B = 22; // initializes a byte type variable B.
double pi = 3.14159; // declares and assigns a value of PI.
char a = 'a'; // the char variable a iis initialized with value 'a'
```

本章将介绍 Java 中的各种可用的变量类型。Java 中共有三种变量：

- 本地变量
- 实例变量
- 类、静态变量

本地变量

- 本地变量在方法、构造器或者块中声明
- 本地变量在方法、构造器或者块进入时被创建，一旦退出该变量就会被摧毁
- 可访问描述符不能用于本地变量
- 本地变量仅在已经声明的方法、构造器或者块中可见
- 本地变量在栈深度内部实施
- 本地变量没有默认值，因此本地变量必须被声明并且在第一次使用前要给它赋值

例子

这里，age（年龄）是本地变量。这是在 pupAge() 方法下定义的，它的范围仅限于这个方法。

```
public class Test{
  public void pupAge(){
    int age = 0;
    age = age + 7;
    System.out.println("Puppy age is : " + age);
  }

  public static void main(String args[]){
    Test test = new Test();
    test.pupAge();
  }
}
```

上述代码会输出如下结果：

```
Puppy age is: 7
```

例子

下面的例子使用了本地变量 age 但是没有进行初始化，所以在编辑是就会显示错误。

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

  public static void main(String args[]){
    Test test = new Test();
    test.pupAge();
  }
}
```

编辑时会产生如下错误：

```
Test.java:4:variable number might not have been initialized
age = age + 7;
```

```
^
1 error
```

实例变量

- 实例变量在类中声明，但是它在方法、构造器或者块外。
- 当堆中的对象被分配了一个空间时，每个实例变量的位置就被创建了。
- 当对象采用关键字“ new ”创建时实例变量就被创建了，当对象被销毁时它也就被销毁了。
- 实例变量的值必须被一个以上的方法、构造器或者块，或者类中必须出现的对象的状态的重要部分所引用。
- 实例变量可以在类水平上在使用前或者使用后声明。
- 实例变量可以用可访问描述符。
- 实例变量对类中的所有方法、构造器或者块可见。一般来讲，推荐将这些变量私有（访问层面）。然而子类的可见性可用可访问描述符给予那些变量。
- 实例变量有默认值。数字的默认值为零，布尔型默认值为假，对象引用默认值为空。在声明或者构造器内可以进行赋值。
- 实例变量可以采用直接在类中叫名字方式访问。然而在静态方法和不同的类（实例变量可以被访问）中应当使用完全限定名称。ObjectReference.VariableName

例子

```
import java.io.*;

public class Employee{
  // this instance variable is visible for any child class.
  public String name;

  // salary  variable is visible in Employee class only.
  private double salary;

  // The name variable is assigned in the constructor.
  public Employee (String empName){
    name = empName;
  }

  // The salary variable is assigned a value.
  public void setSalary(double empSal){
    salary = empSal;
```

```
  }

  // This method prints the employee details.
  public void printEmp(){
    System.out.println("name  : " + name );
    System.out.println("salary :" + salary);
  }

  public static void main(String args[]){
    Employee empOne = new Employee("Ransika");
    empOne.setSalary(1000);
    empOne.printEmp();
  }
}
```

上述代码会输出如下结果：

```
name  : Ransika
salary :1000.0
```

类、静态变量

- 类变量也叫静态变量，它是在类中用 static 关键字声明的，但是它在方法、构造器或者块之外。
- 每个类中只有一个类变量，不管这个类有多少对象。
- 除了作为常量被声明之外，类变量很少被应用。常量是被作为 public、private, final 和 static 被声明的变量。实例变量的初始值不会被改变。
- 静态变量存储在静态内存中，很少采用静态变量而不是声明结束或者用常量public 或 private 之一。
- 静态变量随着程序的开始和结束而开始和结束。
- 可见性和实例变量相似。然而大多数静态变量被声明为public由于他们必须为类的使用者所用。
- 默认值和实例变量相似。对于数字的默认值为零，布尔型默认值为假，对象引用默认值为空。在声明或者构造器内可以进行赋值。除此之外，可以在特殊的静态初始化区赋值。
- 静态变量可以用类的名称访问。ClassName.VariableName
- 当静态变量被作为 public static final 声明时，变量（常量）名称都要用大写字母。如果静态变量不是 public 和 final，它的命名方法和实例变量和本地变量相同。

例子

```
import java.io.*;

public class Employee{
  // salary  variable is a private static variable
  private static double salary;

  // DEPARTMENT is a constant
  public static final String DEPARTMENT = "Development ";

  public static void main(String args[]){
    salary = 1000;
    System.out.println(DEPARTMENT+"average salary:"+salary);
  }
}
```

上述代码会输出如下结果：

```
Development average salary:1000
```

> 注：如果变量从类外访问，常量就必须以 Employee.DEPARTMENT 访问。

接下来是什么呢?

在本章你已经学会应用可访问描述符（ public & private ）了。下一张将为你详细讲解可访问描述符和不可访问描述符。

Java 描述符的类型

描述符是你添加到那些定义中来改变他们的意思的关键词。Java 语言有很多描述符，包括以下这些：

- 可访问描述符
- 不可访问描述符

应用描述符，你可以在类、方法、变量中加入相应关键字。描述符要先于声明，如下面的例子所示（斜体）：

```
public class className {
  // ...
}
private boolean myFlag;
static final double weeks = 9.5;
protected static final int BOXWIDTH = 42;
public static void main(String[] arguments) {
  // body of method
}
```

可访问描述符

Java 提供一系列可访问描述符来设定类，变量，方法和构造器的访问级别。四种访问级别如下：

- 默认的，对封装可见。不需要描述符。
- 仅对类可见（private）
- 全部可见（public）
- 对封装和子类可见（protected）

不可访问描述符

- Java 提供一些不可访问描述符来满足其他功能。
- Static 描述符是用来创造类方法和变量的。
- Final 描述符用来最终确定和实施类、方法和变量的。
- Abstract 描述符用来创造不允许实例化的类和方法。
- synchronized 和 volatile 描述符用来当做线的。

接下来是什么呢?

在下一节，我们将讨论基本的 Java 运算符。这一章将会给你一个在开发应用程序时如何运用这些运算符的概述。

Java 基本运算符

Java 针对操控变量提供了一组丰富的运算符。我们可以将所有的 Java 运算符分为如下几组：

- 算数运算符
- 关系运算符
- 位运算符
- 逻辑运算符
- 赋值运算符
- 其他运算符

算数运算符

算术运算符在数学表达中的使用和它们在代数中的使用是相同的。下面的表格列举了算数运算符：

假设整体变量 A 有 10 个，变量 B 有 20 个，那么：

示例

运算符	描述	例子
+	加法 - 在运算符的另一端增加	A+B 为 30
−	减法 - 将右侧的操作数从左侧的操作数中减去	A − B 为−10
*	乘法 - 将运算符两端的值相乘	A * B 为200
/	除法 - 用右侧操作数除左侧操作数	B / A 为2
%	系数 − 用右侧操作数除左侧操作数并返回馀数	B % A 为0
++	增量 - 给操作数的值增加1	B++ 为21
−−	减量 - 给操作数的值减去1	B—为19

关系运算符

以下是 Java 语言可支持的关系运算符。

假设变量 A 有 10，变量 B 有 20，那么：

示例

运算符	描述	例子
==	检查双方操作数的值是否相等，如果相等那么条件为真	(A == B) 不为真。
!=	检查双方操作数的值是否相等，如果不相等那么条件为真	(A != B) 为真。
>	检查左侧的操作数是否大于右侧的操作数，如果大于那么条件为真	(A > B) 不为真。
<	检查左侧的操作数是否小于右侧的操作数，如果小于那么条件为真	(A < B) 为真。
>=	检查左侧的操作数是够大于等于右侧的操作数，如果是那么条件为真	(A >= B) 不为真。
<=	检查左侧的操作数是否小于等于右侧的操作数，如果是那么条件为真	(A <= B) 为真。

位运算符

Java 定义了几种运算符，这类运算符可被运用于整数型式，long, int，short，字符型和字节型。

位运算符作用于二进制系统间传输标准，并执行按位操作。假设如果 a 等于 60；b 等于 13；现在在二进制型式下它们就如下所示：

```
a = 0011 1100

b = 0000 1101
##
a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a  = 1100 0011
```

以下表格列举了位运算符：

假设整数变量 A 有 60 个，B 有 13 个那么：

示例

运算符	描述	例子
&	二进制AND运算符在结果上复制一位如果在双方操作数同时存在	(A & B) 为12，即 0000 1100

运算符	描述	例子
|	二进制OR运算符在结果上复制一位如果在任何一个操作数上存在	(A | B) 为61，即0011 1101
^	二进制XOR 运算符复制位，如果它是设置在一个操作数上而不是两个。	(A ^ B) 为49， 即0011 0001
~	二进制补充运算符是一元的，b并有“翻转”位的影响	(~A) 为 -61，由于是带符号的二进制数，那2的补位形式为1100 0011
<<	二进制左移运算符。左侧操作数的值根据右侧操作数指定的位的数量移至左侧。	A << 2 为240 ，即1111 0000
>>	二进制右移运算符。左侧操作数的值根据右侧操作数指定的位的数量移至右侧。	A >> 2 为 15即1111
>>>	右移补零运算符。左侧操作数的值根据右侧操作数指定的位的数量移至右，并且转移的值用零补满。	A >>>2 为15 ，即0000 1111

逻辑运算符

下表格列举了逻辑运算符：

假设布尔数学体系变量 A 为真，B 为假，那么：

示例

运算符	描述	例子
&&	称为逻辑与运算符。如果双方操作数都不为零，那么条件为真。	(A && B) 为真.
||	称为逻辑或运算符. 如果双方操作数其中的任何一个都不为零，那么条件为真。	(A || B) 为真.
!	称为逻辑非运算符. 用作翻转操作数的逻辑状态。如果一个条件为真，那么逻辑非运算符为假。	!(A && B) 为真.

赋值运算符

以下是由 Java 语言所支持的赋值运算符：

示例

运算符	描述	例子
=	简单及运算符, 将右侧操作数的值赋给左侧操作数	C = A + B 会将 A + B 的值赋给 C
+=	增加及赋值运算符, 它将右侧的操作数增加到左侧的操作数并且结果赋给左侧操作数	C += A 同等于 C = C + A
−=	减去及赋值运算符，它将右侧操作数从左侧操作数中减去并将结果赋给左侧操作数	C −= A 同等于C = C − A
*=	乘以及赋值运算符，它将右侧操作数与左侧相乘并将结果赋给左侧操作数	C *= A 同等于 C = C * A
/=	除以及赋值运算符，它将右侧操作数除左侧操作数并将结果赋给左侧操作数	C /= A 同等于 C = C / A
%=	系数及赋值运算符 需要系数运用两个操作数并且将结果赋给左侧操作数	C %= A is 同等于 C = C % A
<<=	左移和赋值运算符	C <<= 2 同等于C = C << 2
>>=	右移和赋值运算符	C >>= 2 同等于 C = C >> 2
&=	按位和赋值运算符	C &= 2 同等于C = C & 2
^=	按位异或及赋值运算符	C ^= 2 同等于 C = C ^ 2
|=	按位可兼或及赋值运算符	C |= 2 同等于C = C | 2

其它运算符

以下是由 Java 语言所支持的一些其他的运算符：

条件运算符（？： ）

条件运算符同样也被称作为三元运算符。这种运算符由三个操作数组成，被用作评估布尔数学体系表达。这种运算符的目的是来决定哪些值应被赋予到变量上。这个运算符被写作如下：

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

以下是示例：

```
public class Test {

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

```
    b = (a == 1) ? 20: 30;
    System.out.println( "Value of b is : " +  b );

    b = (a == 10) ? 20: 30;
    System.out.println( "Value of b is : " + b );
  }
}
```

这样就会有如下结果：

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

Instanceof 符

这种操作符只用于对象引用变量。这种操作符检查对象是否是独特类型（类型或接口类型）。Instanceof 运算符写为：

```
( Object reference variable ) instanceof  (class/interface type)
```

如果在运算符左侧的由变量所指代的对象为右侧的类型或接口类型通过 IS-A 检查，那么结果为真。以下是示例：

```
public class Test {

  public static void main(String args[]){
    String name = "James";
    // following will return true since name is type of String
    boolean result = name instanceof String;
    System.out.println( result );
  }
}
```

这就会产出如下结果：

```
true
```

这种运算符仍会返回到真如果被比较的对象是与右边类型兼容的赋值。以下是另一个例子：

```
class Vehicle {}

public class Car extends Vehicle {
  public static void main(String args[]){
    Vehicle a = new Car();
    boolean result =  a instanceof Car;
```

```
    System.out.println( result );
  }
}
```

这样将会产生以下的结果：

```
true
```

Java 运算符的优先级

运算符优先级决定一个表达式里术语的分组。它影响着一个表达式如何求值。一定的运算符比其他运算符拥有更高的优先级；例如：乘法运算符比加法运算符有更高的优先级：

例如，x=7+3 * 2; 这里x 被赋值为13，不是20，是因为运算符 * 比运算符+由更高的优先级， 所以它首先运算乘法 3*2, 然后再加7。

这里，有着最高优先级的运算符在这个表格的最高一层，最低优先权的则出现在最底部。在一个表达式中，越高等级的优先权的运算符会最先被求值。

类	运算符	关联性
后缀	() [] . (dot operator)	从左到右
一元	++ − − ! ~	从右到左
乘法的	* / %	从左到右
加法的	+ −	从左到右
移位	>> >>> <<	从左到右
关系的	> >= < <=	从左到右
相等	== !=	从左到右
位与	&	从左到右
位异或	^	从左到右
位或	|	从左到右
逻辑与	&&	从左到右
逻辑或	||	从左到右
有条件的	?:	从右到左
赋值	= += −= *= /= %= >>= <<= &= ^==	从右到左
逗号	,	从左到右

接下来是?

接下来的一章将会解释 Java 程序设计中的循环控制。该章节将会描述不同种类的循环以及这些循环如何运用到 Java 程序发展和以什么样的目的来使用它们。

Java 循环控制

可能存在一种情况，当我们需要执行的代码块数次，通常被称为一个循环。
Java有非常灵活的三循环机制。可以使用以下三种循环之一：

- while 循环
- do...while 循环
- for 循环

截至Java5，对增强的for循环进行了介绍。这主要是用于数组。

while 循环

while循环是一个控制结构，可以重复的特定任务次数。

语法

while循环的语法是：

```
while(Boolean_expression)
{
  //Statements
}
```

在执行时，如果布尔表达式的结果为真，则循环中的动作将被执行。只要该表达式的结果为真，执行将继续下去。

在这里，while循环的关键点是循环可能不会永远运行。当表达式进行测试，结果为假，循环体将被跳过，在while循环之后的第一个语句将被执行。

示例

```
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");
    }
  }
}
```

这将产生以下结果:

```
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

do ... while循环类似于while循环，不同的是一个do ... while循环是保证至少执行一次。

语法

do...while循环的语法是：

```
do
{
  //Statements
} while (Boolean_expression);
```

请注意，布尔表达式出现在循环的结尾，所以在循环中的语句执行前一次布尔测试。

如果布尔表达式为真，控制流跳回，并且在循环中的语句再次执行。这个过程反复进行，直到布尔表达式为假。

示例

```
public class Test {
```

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

    do{
      System.out.print("value of x : " + x );
      x++;
      System.out.print("\n");
    }while( x < 20 );
  }
}
```

这将产生以下结果:

```
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
```

for 循环

for循环是一个循环控制结构，可以有效地编写需要执行的特定次数的循环。

知道一个任务要重复多少次的时候，for循环是有好处的。

语法

for循环的语法是:

```
for(initialization; Boolean_expression; update)
{
  //Statements
}
```

下面是一个for循环的控制流程:

- 初始化步骤首先被执行，并且仅一次。这个步骤可声明和初始化任何循环控制变量。不需要把一个声明放在这里，只需要一个分号出现。

- 接下来，布尔表达式求值。如果是 true，则执行循环体。如果是false，则循环体不执行, 并且流程控制的跳转到经过for循环的下一个语句。
- 之后循环体在for循环执行时，控制流程跳转备份到更新语句。该语句允许更新任何循环控制变量。这个语句可以留空，只要一个分号出现在布尔表达式之后。
- 布尔表达式现在再次评估计算。如果是true，循环执行，并重复这个过程（循环体，然后更新的步骤，然后布尔表达式）。之后，布尔表达式为 false，则循环终止。

示例

```
public class Test {

  public static void main(String args[]) {

    for(int x = 10; x < 20; x = x+1) {
      System.out.print("value of x : " + x );
      System.out.print("\n");
    }
  }
}
```

这将产生以下结果:

```
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
```

for 循环在 Java 中新特性

截至Java5，对增强的for循环进行了介绍。这主要是用于数组。

语法

增强的for循环的语法是：

```
for(declaration : expression)
{
  //Statements
}
```

- 声明: 新声明块变量，这是一种与你所正在访问数组中的元素兼容的变量。该变量在for块内可被利用并且它的值作为当前的数组元素将是相同的。
- 表达: 这个计算结果完成需要循环数组。表达式可以是一个数组变量或返回一个数组的方法调用。

示例

```
public class Test {

  public static void main(String args[]){
    int [] numbers = {10, 20, 30, 40, 50};

    for(int x : numbers ){
      System.out.print( x );
      System.out.print(",");
    }
    System.out.print("\n");
    String [] names ={"James", "Larry", "Tom", "Lacy"};
    for( String name : names ) {
      System.out.print( name );
      System.out.print(",");
    }
  }
}
```

这将产生以下结果:

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

break 关键字

关键字break是用来停止整个循环的。 break关键字必须使用于任何循环中或一个switch语句中。

关键字break将停止最内层循环的执行，并开始执行在块之后的下一行代码。

语法

break语法是任何循环中一个单独的语句：

```
break
```

示例

```
public class Test {

  public static void main(String args[]) {
    int [] numbers = {10, 20, 30, 40, 50};

    for(int x : numbers ) {
      if( x == 30 ) {
       break;
      }
      System.out.print( x );
      System.out.print("\n");
    }
  }
}
```

这将产生以下结果:

```
10
20
```

continue 关键字

continue关键字可以在任一环的控制结构使用。它使循环立即跳转到循环的下一次迭代.

- 在for循环中，continue关键字会导致控制流立即跳转到更新语句。

- 在一个while循环或do/while循环，控制流立即跳转到布尔表达式。

语法

continue 语法是任何循环中一个单独的语句：

```
continue
```

示例

```
  public static void main(String args[]) {
    int [] numbers = {10, 20, 30, 40, 50};

    for(int x : numbers ) {
      if( x == 30 ) {
       continue;
      }
      System.out.print( x );
      System.out.print("\n");
    }
  }
}
```

这将产生以下结果：

```
10
20
40
50
```

接下来是?

在接下来的章节，我们将会学习在Java程序设计中的决策语句。

Java 条件判断

在 Java 中有两种类型的条件判断语句，它们分别是：

- if 语句
- switch 语句

if 语句:

if 语句由一个布尔表达式后跟一个或多个语句组成。

语法

if 语句的语法是：

```
if(Boolean_expression)
{
   //Statements will execute if the Boolean expression is true
}
```

如果布尔表达式的值为 true，那么代码里面的块 if 语句将被执行。如果不是 true，在 if 语句（大括号后）结束后的第一套代码将被执行。

示例

```
public class Test {

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

      if( x < 20 ){
         System.out.print("This is if statement");
      }
   }
}
```

这将产生以下结果：

```
This is if statement
```

if...else 语句

任何 if 语句后面可以跟一个可选的 else 语句，当布尔表达式为 false，语句被执行。

语法

if...else 的语法是:

```
if(Boolean_expression){
  //Executes when the Boolean expression is true
}else{
  //Executes when the Boolean expression is false
}
```

示例

```
public class Test {

  public static void main(String args[]){
    int x = 30;

    if( x < 20 ){
      System.out.print("This is if statement");
    }else{
      System.out.print("This is else statement");
    }
  }
}
```

这将产生以下结果:

```
This is else statement
```

if...else if...else 语句

if 后面可以跟一个可选的 else if...else 语句，在测试不同条件下单一的 if 语句和 else if 语句是非常有用的。

当使用 if , else if , else 语句时有几点要牢记。

- 一个 if 语句可以有0个或一个 else 语句 且它必须在 else if 语句的之后。

- 一个 if 语句 可以有0个或多个 else if 语句且它们必须在 else 语句之前。
- 一旦 else if 语句成功, 余下 else if 语句或 else 语句都不会被测试执行。

语法

if...else 的语法是:

```
if(Boolean_expression 1){
   //Executes when the Boolean expression 1 is true
}else if(Boolean_expression 2){
   //Executes when the Boolean expression 2 is true
}else if(Boolean_expression 3){
   //Executes when the Boolean expression 3 is true
}else {
   //Executes when the none of the above condition is true.
}
```

示例

```
public class Test {

   public static void main(String args[]){
      int x = 30;

      if( x == 10 ){
         System.out.print("Value of X is 10");
      }else if( x == 20 ){
         System.out.print("Value of X is 20");
      }else if( x == 30 ){
         System.out.print("Value of X is 30");
      }else{
         System.out.print("This is else statement");
      }
   }
}
```

这将产生以下结果:

```
Value of X is 30
```

嵌套 if...else 语句

它始终是合法的嵌套 if-else 语句，这意味着你可以在另一个 if 或 else if 语句中使用一个 if 或 else if 语句。

语法

嵌套 if...else 的语法如下：

```
if(Boolean_expression 1){
   //Executes when the Boolean expression 1 is true
   if(Boolean_expression 2){
      //Executes when the Boolean expression 2 is true
   }
}
```

因为我们有嵌套的 if 语句，所以可以用类似的方式嵌套 else if...else。

示例

```
public class Test {

   public static void main(String args[]){
      int x = 30;
      int y = 10;

      if( x == 30 ){
         if( y == 10 ){
            System.out.print("X = 30 and Y = 10");
         }
      }
   }
}
```

这将产生以下结果：

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

switch 语句

switch 语句允许一个变量来对一系列值得相等性进行测试。每个值被称为一 case，并且被启动的变量会为每一个 case 检查。

语法

增强的 for 循环的语法是：

```
switch(expression){
   case value :
     //Statements
     break; //optional
   case value :
     //Statements
     break; //optional
   //You can have any number of case statements.
   default : //Optional
     //Statements
}
```

以下规则适用于 switch 语句：

- 在 switch 语句中使用的变量只能是一个字节，short，int 或 char。
- 在一个 switch 语句中可以有任何数量的 case 语句。每个 case 后跟着即将被比较的值和一个冒号。
- 对于 case 的值必须是相同的数据类型作为开关变量，它必须是一个常量或文字。
- 当被启动了的变量与 case 是相等的，那 case 后的语句将执行，一直到 break 为止。
- 当达到一个 break 语句，switch 终止，并且控制流跳转到跟着 switch 语句的下一行。
- 不是每一个 case 需要包含一个 break。如果没有出现 break，控制流将贯穿到后面的 case 直到 break 为止。
- switch 语句可以有一个可选默认 case ，它必须出现在 switch 的结束处。在执行一项任务时没有任何 case 是真，那默认 case 可被使用。在默认 case 中不需要 break。

示例

```
public class Test {
```

```
  public static void main(String args[]){
    //char grade = args[0].charAt(0);
    char grade = 'C';

    switch(grade)
    {
      case 'A' :
        System.out.println("Excellent!");
        break;
      case 'B' :
      case 'C' :
        System.out.println("Well done");
        break;
      case 'D' :
        System.out.println("You passed");
      case 'F' :
        System.out.println("Better try again");
        break;
      default :
        System.out.println("Invalid grade");
    }
    System.out.println("Your grade is " + grade);
  }
}
```

编译并运行上面使用各种命令行参数的程序。这将产生以下结果：

```
$ java Test
Well done
Your grade is a C
$
```

接下来是?

下一章讨论了有关 Number 类（在 java.lang 包中）以及 Java 语言及其子类。

我们将寻找到一些使用这些类的实例化，而不是原始数据类型，以及类如格式化，还有当你使用 Number 需要知道的数学函数。

Java 数字

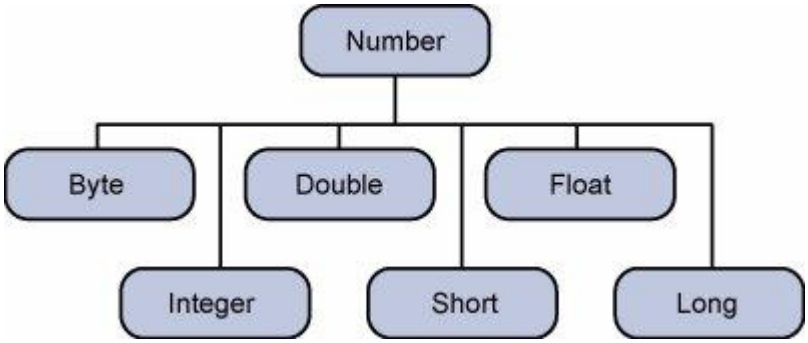
通常情况下，当我们处理数字时，使用原始数据类型，如 byte，int，long，double 等。

示例

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

然而，在开发中，我们会遇到需要使用对象而不是原始数据类型的情况。为了实现这个， Java 为每个原始数据类型提供包装类。

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



图片 1.1 image

这种包装是由编译器处理，这个过程称为装箱。因此，当一个原始数据类型被使用，当需要一个对象时，编译器将原始数据放入其包装类。同样地，编译器也能将对象取出返回到原始数据类型。Number 是 java.lang 包的一部分。

这里是装箱和拆箱的一个例子：

```
public class Test{

  public static void main(String args[]){
    Integer x = 5; // boxes int to an Integer object
    x =  x + 10;   // unboxes the Integer to a int
    System.out.println(x);
  }
}
```

这将产生以下结果:

```
15
```

当 x 被分配整数值，则编译器将整数放入箱中，因为 x 为整数对象。然后，x 被拆箱，以便它们可以被添加为整数。

Number 方法

以下是对 Number 类实现的所有子类中实现的实例方法的列表：

SN	方法描述
1	xxxValue() 这个Number对象的值转换为XXX的数据类型并返回
2	compareTo() 把这个Number对象与参数做比较
3	equals() 确定这个数字对象是否等于参数
4	valueOf() 返回一个Integer对象持有指定的原始值
5	toString() 返回表示指定的int或整数的值的String对象
6	parseInt() 此方法用于获取某个字符串的原始数据类型
7	abs() 返回参数的绝对值
8	ceil() 返回的最小整数大于或等于该参数。返回为double
9	floor() 返回的最大整数小于或等于该参数。返回为double
10	rint() 返回的整数，它是最接近值该参数值。返回为double
11	round() 返回最接近的long或者int，通过该方法的返回类型所指参数
12	min() 返回两个参数中较小的
13	max() 返回两个参数中较大的
14	exp() 返回自然对数的底数e，该参数的幂值
15	log() 返回参数的自然对数
16	pow() 返回第一个参数的提高至第二个参数的幂值

SN	方法描述
17	sqrt() 返回参数的平方根
18	sin() 返回指定的double值的正弦值
19	cos() 返回指定的double值的余弦值
20	tan() 返回指定的double值的正切值
21	asin() 返回指定的double值的反正弦
22	acos() 返回指定的double值的反余弦值
23	atan() 返回指定的double值的反正切值
24	atan2() 将直角坐标(x,y)转换为极坐标(r, θ)并返回 θ
25	toDegrees() 将参数转换为度
26	toRadians() 将参数转换为弧度
27	random() 返回一个随机数

接下来是?

在下一节中，我们将学习 Character 类。将学习如何在 Java 中使用 Character 对象和基本数据类型 char。

Java 字符

一般情况下，当我们处理字符时，我们用原始数据类型 char。

示例

```
char ch = 'a';

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

// an array of chars
char[] charArray ={ 'a', 'b', 'c', 'd', 'e' };
```

然而在开发中，我们会遇到需要使用对象而不是原始数据类型的情况。为了达到这个需求。Java 为原始数据类型 char 提供了包装类 Character。

Character 类为操控字符提供了一系列有用处的类（例如：静态类）。你可以借助 Character 构造函数创造一个 Character 对象。

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

Java 编译器也将能在某些情况下为你创造一个 Character 对象。例如：如果你将一个原始 char 传输到一个可预期对象的方法，编译器就会为你自动将 char 转化成 Character。 如果转换从反方向进行，这个特点被称之为自动装箱或拆箱。

示例

```
// 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

有反斜杠（\）在前的字符是一个转义序列并且对于编译器有特殊的意义。

换行符(\n)在 System.out.println() 语句中经常使用，在字符串打印出来后换行。

以下的表格展示了 Java 转义序列：

转义序列	描述
\t	在文本中插入一个标签。
\b	在文本中插入一个退格。
\n	在文本中插入一个换行符。
\r	在文本中插入一个回车。
\f	在文本中插入一个换页。
\'	在文本中插入一个单引号字符。
\\	在文本中插入一个反斜杠字符。

当一个转义序列遇到一个打印语句，编译器就会相应地解译。

示例

如果你想把引号放入引号内，必须使用转义序列, \” ,在内部引用:

```
public class Test {

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

这将产生以下结果：

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

Character 方法

以下列表是实现 Character 类所有子类的重要的实例方法：

SN	方法描述
1	isLetter() 确定具体的char值是一个字母
2	isDigit() 确定具体的char值是一个数字
3	isWhitespace() 确定具体的char值是一个空格
4	isUpperCase() 确定具体的char值是一个大写字母
5	isLowerCase() 确定具体的char值是一个小写字母
6	toUpperCase() 返回指定字符值的大写形式
7	toLowerCase() 返回指定字符值的小写写形式
8	toString() 返回代表指定的字符值的一个String对象,即一个字符的字符串

若想查看完整的方法，请参阅 java.lang.Character API 规范。

接下来是?

在下一个部分，我们将会浏览 Java 的 String 类。你将会学习到如何有效地声明和使用 Strings 并且学习在 String 类中一些重要的方法。

Java 字符串

字符串，它被广泛应用于 Java 编程，是一个字符序列。在 Java 编程语言中，字符串是对象。

Java 平台提供了 String 类来创建和操作字符串。

创建字符串

最直接的方式来创建一个字符串是这样写的：

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

当你创建一个字符串时，编译器在这种情况下用它的值创建一个 String 对象，如："Hello world!'。

任何其他对象可以通过使用 new 关键字，并通过构造函数创建 String 对象。 String 类有11种构造函数提供使用不同类型的字符串的初始值，如一个字符数组。

```
public class StringDemo{

   public static void main(String args[]){
      char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.'};
      String helloString = new String(helloArray);
      System.out.println( helloString );
   }
}
```

这将产生以下结果：

```
hello.
```

注 String 类是不可变的，因此，一旦创建了 String 对象那么是不能改变的。如果需要大量修改字符的字符串，那么应该使用 StringBuffer & StringBuilder 类。

String 长度

用于获取有关对象的信息的方法称为存取方法。可以和字符串一起使用的一个存取方法是 length() ，它返回包含在字符串对象中的字符数。

下面的两行代码被执行之后，len 等于17：

```
public class StringDemo {

  public static void main(String args[]) {
    String palindrome = "Dot saw I was Tod";
    int len = palindrome.length();
    System.out.println( "String Length is : " + len );
  }
}
```

这将产生以下结果:

```
String Length is : 17
```

连接字符串

String类包括用于连接两个字符串的方法:

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

这返回一个新的字符串，即在 string1 结尾处添加 string2。还可以使用 concat（ ）方法连接字符串，如:

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

字符串更常使用 “ + ” 运算符连接在一起，如:

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

这将产生:

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

看看下面的例子:

```
public class StringDemo {

  public static void main(String args[]) {
    String string1 = "saw I was ";
    System.out.println("Dot " + string1 + "Tod");
  }
}
```

这将产生以下结果:

```
Dot saw I was Tod
```

创建格式化字符串

已经有 printf() 和 format() 方法来打印输出格式的数字。 String 类有一个等价的方法 format()，它返回一个 String 对象，而不是一个 PrintStream 对象。

使用字符串的静态 format() 方法允许创建可重复使用的格式化字符串，而不是一次性的 print 语句。例如，如果代替以下方法：

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

可以这样写：

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

String 方法

这里是由 String 类支持的方法列表：

SN	方法及描述
1	char charAt(int index) 返回指定索引处的字符。
2	int compareTo(Object o) 将这个字符串与另一个对象比较。
3	int compareTo(String anotherString) 比较两个字符串的字典顺序。
4	int compareToIgnoreCase(String str) 比较两个字符串按字典顺序，不区分大小写的差异。
5	String concat(String str) 将指定的字符串串连到这个字符串的结尾。
6	boolean contentEquals(StringBuffer sb) 返回true当且仅当该字符串代表相同的字符序列作为指定的StringBuffer。
7	static String copyValueOf(char[] data) 返回表示所指定的数组中的字符序列的字符串。

SN	方法及描述
8	static String copyValueOf(char[] data, int offset, int count) 返回表示所指定的数组中的字符序列的字符串。
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) 比较此字符串与指定的对象。
11	boolean equalsIgnoreCase(String anotherString) 比较这个字符串到另一个字符串，忽略大小写的考虑。
12	byte getBytes() 将此String解码使用平台的默认字符集，并将结果存储到一个新的字节数组中的字节序列。
13	byte[] getBytes(String charsetName 将此String解码使用指定的字符集的字节序列，并将结果存储到一个新的字节数组。
14	void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) 从这个字符串复制字符到目标字符数组。
15	int hashCode() 为这个字符串返回一个哈希代码。
16	int indexOf(int ch) 返回此字符串指定字符第一次出现处的索引。
17	int indexOf(int ch, int fromIndex) 返回索引这个字符串中指定字符第一次出现处，指定索引处开始搜索。
18	int indexOf(String str) 返回此字符串指定子字符串的第一次出现处的索引。
19	int indexOf(String str,int fromIndex) 返回这个字符串中指定子字符串的第一次出现处的索引，从指定的索引处开始。
20	String intern() 返回字符串对象的规范化表示。
21	int lastIndexOf(int ch) 返回此字符串指定字符最后一次出现处的索引。
22	int lastIndexOf(int ch, int fromIndex) 返回此字符串指定字符最后一次出现处的索引，从指定索引开始向后搜索。
23	int lastIndexOf(String str) 返回此字符串指定子字符串的最右边出现处的索引。
24	int lastIndexOf(String str, int fromIndex) 返回索引这个字符串中指定子字符串的最后出现处，从指定的索引开始处向后搜索。
25	int length() 返回此字符串的长度。
26	boolean matches(String regex) 判断此字符串是否与给定的正则表达式匹配。
27	boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len) 测试两个字符串的区域等于。
28	boolean regionMatches(int toffset, String other, int ooffset, int len) 测试两个字符串的区域都是相等的。

SN	方法及描述
29	String replace(char oldChar, char newChar) 返回从newChar更换oldChar所有出现在此字符串中产生一个新的字符串。
30	String replaceAll(String regex, String replacement 替换此字符串中给定的正则表达式与给定替换相匹配的每个子字符串。
31	String replaceFirst(String regex, String replacement) 替换此字符串匹配给定的正则表达式给定替换第一个子字符串。
32	String[] split(String regex) 分割围绕给定的正则表达式匹配的这个字符串。
33	String[] split(String regex, int limit) 分割围绕给定的正则表达式匹配的这个字符串。
34	boolean startsWith(String prefix) 测试此字符串是否以指定的前缀开头。
35	boolean startsWith(String prefix, int toffset) 测试此字符串是否以指定索引开始的指定前缀开始。
36	CharSequence subSequence(int beginIndex, int endIndex) 返回一个新的字符序列，这个序列的子序列。
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) 返回一个新的字符串，它是此字符串的一个子字符串。
39	char[] toCharArray() 这个字符串转换为一个新的字符数组。
40	String toLowerCase() 将所有在此字符串中的字符使用默认语言环境的规则小写。
41	String toLowerCase(Locale locale) 将所有在此字符串中的字符使用给定Locale的规则小写。
42	String toString() 这个对象（它已经是一个字符串！）返回字符串形式（这里是自己本身）。
43	String toUpperCase() 使用默认语言环境的规则将此String中所有的字符转换为大写。
44	String toUpperCase(Locale locale) 使用给定Locale的规则将此String中所有的字符转换为大写。
45	String trim() 返回字符串的一个副本，开头和结尾的空格去除。
46	static String valueOf(primitive data type x) 返回传递的数据类型参数的字符串表示形式。

Java 数组

Java 提供了一个数据结构：数组，用于存储相同类型的元素的一个固定大小的连续集合。数组是用于存储数据的集合，但往往将数组认为是相同类型的变量的集合。

跟声明单个变量相反，如 number0, number1, ... number99, 声明一个数组变量，如数字和使用 numbers[0], numbers[1] ..., numbers[99] 来表示各个变量。

此次教程将介绍如何使用索引变量声明数组变量，创建数组，并处理数组。

声明数组变量

要使用一个程序的数组，必须声明一个变量来引用数组，必须指定数组的变量可以引用的类型。下面是来声明一个数组变量的语法：

```
dataType[] arrayRefVar;   // preferred way.

or

dataType arrayRefVar[];  //  works but not preferred way.
```

注 风格 dataType[] arrayRefVar 是首选的。风格 dataType arrayRefVar[] 来自于 C/C++ 语言， 方便Java 继承 C/C++ 的编程风格。

示例

下面的代码片段是这种语法的例子：

```
double[] myList;        // preferred way.

or

double myList[];        //  works but not preferred way.
```

创建数组

可以通过使用 new 运算符使用以下语法创建一个数组：

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

上面的语句做了两件事：

* 它创建了一个使用 new dataType[arraySize] 的数组; * 它将新创建的数组引用分配给变量 arrayRefVar。

声明数组变量，建立一个数组，并分配给变量数组引用可以在一个语句中组合使用，如下所示：

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

另外，可以创建数组，如下所示：

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

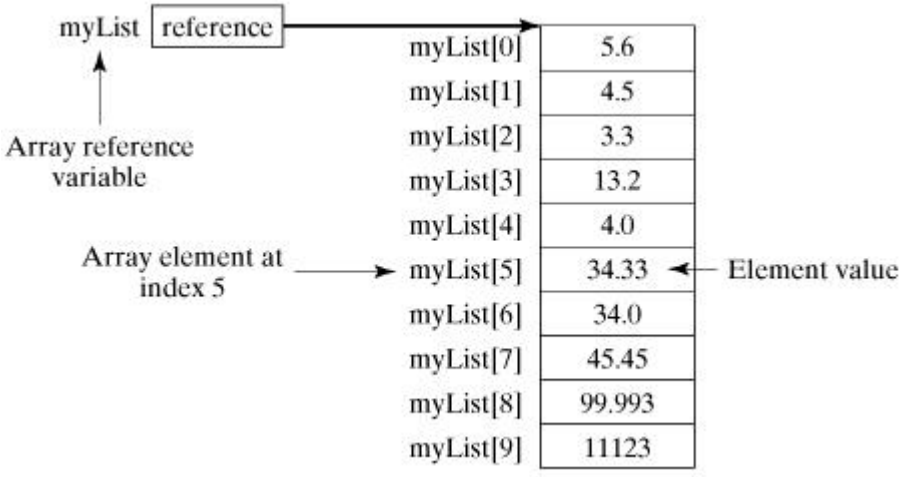
数组元素通过索引访问。数组的下标是从0开始的，也就是说，它们从0开始到 arrayRefVar.length-1。

示例

下面的语句声明一个数组变量 myList，创建 double 类型的10个元素的数组，并把它的引用分配到 myList：

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

以下图片代表数组 myList。在这里，myList 有10个 double 值，索引是从0到9。



图片 1.2 image

处理数组

当处理数组元素时，经常使用的是 loop 循环或 foreach 循环，因为一个数组中所有的元素是相同类型的并且数组的大小是已知的。

示例

下面是一个演示如何创建，初始化和处理数组的完整例子：

```
public class TestArray {

  public static void main(String[] args) {
    double[] myList = {1.9, 2.9, 3.4, 3.5};

    // Print all the array elements
    for (int i = 0; i < myList.length; i++) {
      System.out.println(myList[i] + " ");
    }
    // Summing all elements
    double total = 0;
    for (int i = 0; i < myList.length; i++) {
      total += myList[i];
    }
    System.out.println("Total is " + total);
    // Finding the largest element
    double max = myList[0];
    for (int i = 1; i < myList.length; i++) {
      if (myList[i] > max) max = myList[i];
    }
    System.out.println("Max is " + max);
  }
}
```

这将产生以下结果:

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

foreach 循环

JDK 1.5 引入了一个新的 for 循环被称为 foreach 循环或增强的 for 循环，它无需使用一个索引变量来完整的遍历数组。

示例

下面的代码显示数组 myList 中的所有元素：

```
public class TestArray {

  public static void main(String[] args) {
    double[] myList = {1.9, 2.9, 3.4, 3.5};

    // Print all the array elements
    for (double element: myList) {
      System.out.println(element);
    }
  }
}
```

这将产生以下结果：

```
1.9
2.9
3.4
3.5
```

将数组传递给方法

正如传递基本类型值的方法，也可以将数组传递给方法。例如，下面的方法显示在一个 int 数组中的元素：

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

你可以通过传递数组调用它。例如，下面的语句调用方法 PrintArray 显示3，1，2，6，4，2：

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

从一个方法返回一个数组

一个方法也可以返回一个数组。例如，下面所示的方法返回一个数组，它是另一个数组的反转：

```
public static int[] reverse(int[] list) {
  int[] result = new int[list.length];

  for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {
    result[j] = list[i];
  }
  return result;
}
```

Arrays 类

java.util.Arrays 中的类包含各种静态方法用于排序和搜索数组、数组的比较和填充数组元素。这些方法为所有基本类型所重载。

SN	方法和描述
1	public static int binarySearch (Object[] a, Object key) 使用二进制搜索算法搜索对象的指定数组（字节，整数，双精度等）来指定值。该数组必须在进行此调用之前进行分类。如果它被包含在列表 (-(insertion point + 1)， 将返回索引搜索关键字。
2	public static boolean equals (long[] a, long[] a2) 如果多头的两个指定数组彼此相等返回true。两个数组认为是相等判定方法：如果两个数组包含相同的元素数目，并在两个数组元素的所有相应对相等。如果两个数组相等，返回true。同样的方法可以用于所有其它的原始数据类型 (Byte, short, Int, etc.)
3	public static void fill(int[] a, int val) 将指定的int值到指定的int型数组中的每个元素。同样的方法可以用于所有其它的原始数据类型(Byte, short, Int etc.)
4	public static void sort(Object[] a) 将对象指定的数组升序排列，根据其元素的自然顺序。同样的方法可以用于所有其它的原始数据类型(Byte, short, Int, etc.)

Java 日期和时间

Java 在 java.util 包中提供了 Date 类，这个类封装了当前的日期和时间。

Date 类支持两种构造函数。第一个构造函数初始化对象的当前日期和时间。

```
Date( )
```

下面的构造函数接收一个参数等于自1970年1月1日午夜起已经过的毫秒数

```
Date(long millisec)
```

一旦有一个可用的日期对象，可以调用以下任何一种支持的方法使用时间：

SN	方法和描述
1	boolean after(Date date) 如果调用Date对象包含或晚于指定的日期则返回true，否则，返回false。
2	boolean before(Date date) 如果调用Date对象包含或早于日期指定的日期返回true，否则，返回false。
3	Object clone( ) 重复调用Date对象。
4	int compareTo(Date date) 调用对象的值与日期比较。如果这两个值相等返回0。如果调用对象是早于日期返回一个负值。如果调用对象迟于日期返回正值。
5	int compareTo(Object obj) 以相同的compareTo(Date)操作 如果obj是一个类日期。否则，它会抛出一个ClassCastException。
6	boolean equals(Object date) 如果调用Date对象包含相同的时间及日期指定日期则返回true，否则，返回false。
7	long getTime( ) 返回自1970年1月1日起已经过的毫秒数。
8	int hashCode( ) 返回调用对象的哈希代码。
9	void setTime(long time) 设置所指定的时间，这表示从1970年1月1日从午夜的时间和日期以毫秒为单位经过的时间。
10	String toString( ) 调用Date对象转换为字符串，并返回结果。

获取当前日期和时间

在 Java 中容易得到当前的日期和时间。可以使用一个简单的 Date 对象的 toString() 方法，如下所示打印当前日期和时间：

```
import java.util.Date;

public class DateDemo {
   public static void main(String args[]) {
      // Instantiate a Date object
      Date date = new Date();

      // display time and date using toString()
      System.out.println(date.toString());
   }
}
```

这将产生以下结果:

```
Mon May 04 09:51:52 CDT 2009
```

日期比较

有以下三种方式来比较两个日期：

- 可以使用 getTime() 来获得自1970年1月1日午夜十二时起已经过的毫秒数，然后比较两个对象的值。
- 可以使用 before( ), after( ), 和 equals( ) 方法，由于12日在18日前，例如， new Date(99, 2, 12).before(new Date (99, 2, 18)) 返回值为 true。
- 可以使用 compareTo() 方法，这是由 Comparable 接口定义，由 Date 实现。

使用 SimpleDateFormat 格式化日期

SimpleDateFormat 是一个具体的类，以本地方式用于格式化和转换日期。SimpleDateFormat 允许选择用户定义的模式为日期时间格式。例如：

```
import java.util.*;
import java.text.*;

public class DateDemo {
```

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

    Date dNow = new Date( );
    SimpleDateFormat ft =
    new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");

    System.out.println("Current Date: " + ft.format(dNow));
  }
}
```

这将产生以下结果:

```
Current Date: Sun 2004.07.18 at 04:14:09 PM PDT
````
```

### 简单的 DateFormat 格式代码

要指定时间格式，使用时间模式字符串。在这个模式中，所有的 ASCII 字母被保留为模式字母，其定义如下：

|字符|  描述|  例子|
|:-----|:-----|:------|
|G|  时代指示器|  AD|
|y|  四位数年份|  2001|
|M|  年中的月份|  July or 07|
|d|  月份中日期|  10|
|h|  时间 A.M./P.M.(1~12)  |12|
|H|  天中的小时 (0~23)|  22|
|m|  小时中的分钟|  30|
|s|  分钟中的秒钟|  55|
|S|  毫秒|  234|
|E|  星期中的天|  Tuesday|
|D|  年中的天  | 360|
|F|  月中星期中的天|  2 (second Wed. in July)|
|w|  年中的星期|  40|
|W|  月中的星期|  1|
|a|  A.M./P.M. 标记|  PM|
|k|  天中的小时(1~24)|  24|
|K|  小时A.M./P.M. (0~11)|  10|
|z|  时区  | 东部标准时间|
|'|  脱离文本|  分隔符|
|''|  单引号  | `

### 用 printf 格式化日期

日期和时间格式用 printf 方法可以非常轻松地做到。可以使用两个字母的格式，从 `t` 开始并在下面给出的表格中的其中一个字母结束。
```

```
import java.util.Date;

public class DateDemo {

public static void main(String args[]) { // Instantiate a Date object Date date = new Date();

 // display time and date using toString()
 String str = String.format("Current Date/Time : %tc", date );

 System.out.printf(str);

} }
```

这将产生以下结果:

```
Current Date/Time : Sat Dec 15 16:37:57 MST 2012
```

如果提供日期多次格式化是一种不好的做法。一个格式字符串可以指示要格式化的参数的索引。
索引必须紧跟在 % 之后，并必须由 $ 终止。例如:

```
import java.util.Date;

public class DateDemo {

public static void main(String args[]) { // Instantiate a Date object Date date = new Date();

   // display time and date using toString()
   System.out.printf("%1$s %2$tB %2$td, %2$tY",
                     "Due date:", date);

} }
```

这将产生以下结果:

```
Due date: February 09, 2004
```

或者，也可以使用 < 标志。则表示相同的参数，根据前述格式规范，应再次使用。例如:

```
import java.util.Date;

public class DateDemo {

public static void main(String args[]) { // Instantiate a Date object Date date = new Date();
```

```
   // display formatted date
   System.out.printf("%s %tB %<te, %<tY",
                     "Due date:", date);
```

} }

```
这将产生以下结果:
```

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 日期  |2004-02-09|
|D|  U.S. 格式时间 (月/日/年)|02/09/2004|
|T|  24-时制  |18:05:19|
|r|  12-时制  |06:05:19 pm|
|R|  24-时制，无秒|  18:05|
|Y|  四位数年份 (用前行零列)|  2004|
|y|  年份的后两位数(用前行零列)|  04|
|C|  年份的前两位(用前行零列)|  20|
|B|  完整月份名称  |February|
|b|  缩写月份名称  |Feb|
|m|  两位数月份 (用前行零列)|  02|
|d|  两位数日期 (用前行零列)|  03|
|e|  两位数日期(无前行零列)|  9|
|A|  完整星期名称  |Monday|
|a|  缩写星期名称  |Mon|
|j|  年中的三位数天数(用前行零列)|  069|
|H|  两位数小时(用前行零列), 00 和 23之间|  18|
|k|  两位数小时(无前行零列), 0 和 23 之间|  18|
|I|  两位数小时 (用前行零列), 01和12之间  |06|
|l|  两位数小时 (无前行零列), 1 和12之间|  6|
|M|  两位数分钟 (用前行零列)  |05|
|S|  两位数秒钟(用前行零列)  |19|
|L|  三位数毫秒 ( 用前行零列)  |047|
|N|  九位数纳秒 (用前行零列)|  047000000|
|P|  大写上下午标记  |PM|
|p|  小写上下午标记  |pm|
|z|  RFC 822 从GMT数字抵消|  -0800|
|Z|  时区  |PST|
|s|  从 1970-01-01 00:00:00 的秒数GMT|  1078884319|
```

|Q|　从 1970-01-01 00:00:00 的毫秒数GMT|　1078884319047|

有相关的日期和时间等有用的类。欲了解更多详细信息，可以参考 Java 标准文档。

字符串转换日期

SimpleDateFormat 类有一些额外的方法，如 parse()，它试图根据存储在给定 SimpleDateFormat 的对象的格式来转换字符串。例

import java.util.*; import java.text.*;

public class DateDemo {

public static void main(String args[]) { SimpleDateFormat ft = new SimpleDateFormat ("yyyy-MM-dd");

```
  String input = args.length == 0 ? "1818-11-11" : args[0];

  System.out.print(input + " Parses as ");

  Date t;

  try {
     t = ft.parse(input);
     System.out.println(t);
  } catch (ParseException e) {
     System.out.println("Unparseable using " + ft);
  }
```

} }

上述程序的运行示例将产生以下结果：

$ java DateDemo 1818-11-11 Parses as Wed Nov 11 00:00:00 GMT 1818 $ java DateDemo 2007-12-01 2007-12-01 Parses as Sat Dec 01 00:00:00 GMT 2007

休眠一段时间
你可以进行任何期间的时间休眠，从一毫秒直到你的电脑的整个生命周期。例如，下面的程序会休眠10秒：

import java.util.*;

public class SleepDemo { public static void main(String args[]) { try { System.out.println(new Date( ) + "\n"); Thread.sleep(5*60*10); System.out.println(new Date( ) + "\n"); } catch (Exception e) { System.out.println("Got an exception!"); } } }

这将产生以下结果:

Sun May 03 18:04:41 GMT 2009

Sun May 03 18:04:51 GMT 2009

测量执行时间

有时候，可能需要测量的时间点以毫秒为单位。因此，让我们再一次重新写上面的例子:

```
import java.util.*;

public class DiffDemo {

public static void main(String args[]) { try { long start = System.currentTimeMillis( ); System.out.printl
n(new Date( ) + "\n"); Thread.sleep(5*60*10); System.out.println(new Date( ) + "\n"); long end = Syste
m.currentTimeMillis( ); long diff = end - start; System.out.println("Difference is : " + diff); } catch (Exce
ption e) { System.out.println("Got an exception!"); } } }
```

这将产生以下结果:

Sun May 03 18:16:51 GMT 2009

Sun May 03 18:16:57 GMT 2009

Difference is : 5993

GregorianCalendar 类

GregorianCalendar 是一个 Calendar 类具体的实现，即你所熟悉的对正常 Gregorian 公历的实现。本教程中不讨论 Calendar 类，

Calendar 的 getInstance() 方法返回与当前日期和时间默认语言环境和时区初始化的一个 GregorianCalendar。 GregorianCalen

也有几个构造函数的 GregorianCalendar 对象:

SN	构造函数描述
1	GregorianCalendar() 在默认时区与默认语言环境使用当前时间构造默认的GregorianCalendar。
2	GregorianCalendar(int year, int month, int date) 在默认时区设置默认的语言环境用给定的日期构造一个GregorianCale
3	GregorianCalendar(int year, int month, int date, int hour, int minute) 用给定的日期和时间设置为与默认语言环境的默认
4	GregorianCalendar(int year, int month, int date, int hour, int minute, int second) 用给定的日期和时间设置为与默认语
5	GregorianCalendar(Locale aLocale) 基于当前时间与给定语言环境的默认时区构建一个GregorianCalendar。

|6| GregorianCalendar(TimeZone zone)
基于当前时间，使用默认的语言环境在给定的时区构建一个GregorianCalendar。
|7| GregorianCalendar(TimeZone zone, Locale aLocale)
基于当前时间与给定语言环境的给定时区构建一个GregorianCa

这里是由 GregorianCalendar 类提供的一些有用的方法的列表：

SN	方法和描述
1	void add(int field, int amount) 基于日历的规则，以给定的时间字段添加指定（有符号的）时间量。
2	protected void computeFields() 将UTC转换为毫秒时间字段值.
3	protected void computeTime() 覆盖日历转换时间域值为UTC的毫秒.
4	boolean equals(Object obj) 这个GregorianCalendar与一个对象引用比较.
5	int get(int field) 获取给定时间域的值.
6	int getActualMaximum(int field) 返回该字段可能的最大值，给定到当前的日期.
7	int getActualMinimum(int field) 返回该字段可能具有的最小值，给定当前的日期.
8	int getGreatestMinimum(int field) 对于给定的字段中返回高最低值（如果有变化）.
9	Date getGregorianChange() 获取公历更改日期.
10	int getLeastMaximum(int field) 对于给定的字段返回最低的最大值，如果有变化.
11	int getMaximum(int field) 返回给定字段中的最大值.
12	Date getTime() 获取日历的当前时间.
13	long getTimeInMillis() 获取日历的当前时间长.
14	TimeZone getTimeZone() 获取时区.
15	int getMinimum(int field) 返回给定字段中的最小值.
16	int hashCode() 重写hashCode.
17	boolean isLeapYear(int year) 确定给定年份是闰年.
18	void roll(int field, boolean up) 加上或减去（上/下）的时间在给定的时间字段一个单元，不更改更大的字段.
19	void set(int field, int value) 设置时间字段与给定值.
20	void set(int year, int month, int date) 设置为年，月，日的值.
21	void set(int year, int month, int date, int hour, int minute) 设置为年，月，日，小时和分钟值.
22	void set(int year, int month, int date, int hour, int minute, int second) 设置为字段的年，月，日，小时，分钟和秒的值
23	void setGregorianChange(Date date) 设置GregorianCalendar更改日期.
24	void setTime(Date date) 设置日历的当前时间与给定日期.
25	void setTimeInMillis(long millis) 从给定long值设置日历的当前时间.
26	void setTimeZone(TimeZone value) 将时区设置与给定的时区值.
27	String toString() 返回此日历的字符串表示形式.

示例

```
import java.util.*;

public class GregorianCalendarDemo {

public static void main(String args[]) { String months[] = { "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
"Aug", "Sep", "Oct", "Nov", "Dec"};

  int year;
  // Create a Gregorian calendar initialized
```

```
  // with the current date and time in the
  // default locale and timezone.
  GregorianCalendar gcalendar = new GregorianCalendar();
  // Display current time and date information.
  System.out.print("Date: ");
  System.out.print(months[gcalendar.get(Calendar.MONTH)]);
  System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
  System.out.println(year = gcalendar.get(Calendar.YEAR));
  System.out.print("Time: ");
  System.out.print(gcalendar.get(Calendar.HOUR) + ":");
  System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
  System.out.println(gcalendar.get(Calendar.SECOND));

  // Test if the current year is a leap year
  if(gcalendar.isLeapYear(year)) {
    System.out.println("The current year is a leap year");
  }
  else {
    System.out.println("The current year is not a leap year");
  }
```

} }

这将产生以下结果:

Date: Apr 22 2009 Time: 11:25:27 The current year is not a leap year

在 Calendar 类中的可用常量的完整列表，可以参考标准的 Java 文档。

Java 正则表达式

Java 提供了 java.util.regex 包来与正则表达式进行模式匹配。Java 正则表达式和 Perl 编程语言非常相似，也容易学习。

正则表达式是一个特殊的字符序列，有助于你用一种专门的语法模式来匹配或找到其他字符串或字符串集。他们可以用来搜索编辑或是

java.util.regex 包主要包含了下面的三个类:

- __Pattern 类__:一个 Pattern 对象是正则表达式编译表示。 Pattern 类没有提供公共的构造函数。要创建一个 Pattern 对象，你必

- __Matcher 类__:一个 Matcher 对象是用来解释模式和执行与输入字符串相匹配的操作。和 Pattern 类一样 Matcher 类也是没有构

- __PatternSyntaxException__: 一个 PatternSyntaxException 对象是一个不被检查的异常，来指示正则表达式中的语法错误。

捕获组

捕获组是一种将多个字符抽象为一个处理单元的方法。他们通过用括号将字符分组来创建。举个例子，正则表达式（dog）创建一个组包

捕获组通过从左向右计算括号的个数来进行计数。在正则表达式((A)(B(C)))中，这里有四个组：

- ((A)(B(C)))

- (A)

- (B(C))

- (C)

为了在表达式中计算有多少个组，可以调用 matcher 对象中的 groupCount 方法。 groupCount 方法返回一个 int 类型来显示正则表

这里也有特殊的组，组0总是代表了整个表达式。这个组不包含在 groupCount 负责的所有组内。

示例

下面的例子展现了如何从给定的字符数字串中找出数字串

```
import java.util.regex.Matcher; import java.util.regex.Pattern;

public class RegexMatches { public static void main( String args[] ){

  // String to be scanned to find the pattern.
  String line = "This order was placed for QT3000! OK?";
  String pattern = "(.*)(\\d+)(.*)";

  // Create a Pattern object
  Pattern r = Pattern.compile(pattern);

  // Now create matcher object.
  Matcher m = r.matcher(line);
  if (m.find( )) {
    System.out.println("Found value: " + m.group(0) );
    System.out.println("Found value: " + m.group(1) );
    System.out.println("Found value: " + m.group(2) );
  } else {
    System.out.println("NO MATCH");
  }

} }
```

这将会得到下面的结果

Found value: This order was placed for QT3000! OK? Found value: This order was placed for QT300
Found value: 0

```
### 正则表达式语法

这里的表格记录了 java 中可用的所有正则表达式的元字符语法：

|子表达式|匹配对应|
|:----------|:----------|
|^|  匹配一行的开头|
| $  |匹配一行的结尾  |
|.|匹配除了换行符的任何单个字符，也可以利用 m 选项允许它匹配换行符|
|[...]|  匹配括号内的任意单个字符。|
|[^...]|匹配不在括号内的任意单个字符。  |
|\A|  整个字符串的开始|
|\z|  整个字符串的结束|
|\Z|  整个字符串的结束，除了最后一行的结束符|
|re*|  匹配0或者更多的前表达事件|
|re+|  匹配1个或更多的之前的事件|
|re?|  匹配0或者1件前表达事件|
|re{ n}|  匹配特定的n个前表达事件|
|re{ n,}|匹配n或者更多的前表达事件|
|re{ n, m}|匹配至少n最多m件前表达事件  |
|a\| b|匹配a或者b|
|(re)|正则表达式组匹配文本记忆|
|(?: re)|没有匹配文本记忆的正则表达式组  |
|(?> re)|  匹配无回溯的独立的模式|
|\w|  匹配单词字符|
|\W|  匹配非单词字符|
|\s|  匹配空格。等价于 [\t\n\r\f] |
|\S|  匹配非空格|
|\d|  匹配数字. 等价于 [0-9] |
|\D|  匹配非数字|
|\A|匹配字符串的开始|
|\Z|  匹配字符串的末尾，如果存在新的一行，则匹配新的一行之前|
|\z|  匹配字符串的末尾|
|\G|  匹配上一次匹配结束的地方|
|\n|  返回参考捕获组号“N”|
|\b|  不在括号里时匹配单词边界。在括号里时匹配退格键|
|\B|  匹配非词边界|
|\n, \t, etc.|匹配换行符，回车符，制表符，等|
|\Q|  引用字符的初始，结束于\E|
|\E|  结束由\Q开始的引用|
```

Matcher 类的方法

这里列出了有用的实例方法

index 方法

index方法提供有用的指标值，精确地显示输入字符串中相匹配的位置：

SN	方法描述
1	public int start() 返回之前匹配开始索引
2	public int start(int group) 返回被之前匹配操作得出的组捕获的子序列
3	public int end() 返回在最后一个字符匹配之后的偏移量
4	public int end(int group) 返回在之前匹配操作得出的组捕获的子序列之后的偏移量

Study 方法

Study 方法根据输入字符串返回一个布尔类型数据来指示该模式是否被找到。

SN	方法描述
1	public boolean lookingAt() 　试图匹配输入序列，从模式的起始位置开始
2	public boolean find() 试图找到下一个输入序列的子序列来进行模式匹配
3	public boolean find(int start) 重置匹配，并且试图找到下一个从某个特定位置开始的输入序列的子序列来进行模式匹配
4	public boolean matches() 试图去匹配模式的整个区域

Replacement 方法

Replacement 方法是在一个输入字符串中替换文本的有效方法。

SN	方法描述
1	public Matcher appendReplacement(StringBuffer sb, String replacement) 　实现一个无目的的添加和代替步骤
2	public StringBuffer appendTail(StringBuffer sb) 实现一个有目的的添加和代替步骤
3	public String replaceAll(String replacement) 代替每一个输入序列的子序列，与给出的代替字符串的模式匹配
4	public String replaceFirst(String replacement) 代替第一个输入序列的子序列，与给出的代替字符串的模式匹配
5	public static String quoteReplacement(String s) 返回一个特定字符串逐字替换的字符串。这个方法产生了一个字符串将作为

start 和 end 方法

下面是一个例子，计算 "cats" 在输入字符串中出现的次数：

import java.util.regex.Matcher; import java.util.regex.Pattern;

public class RegexMatches { private static final String REGEX = "\bcat\b"; private static final String IN PUT = "cat cat cat cattie cat";

```
public static void main( String args[] ){
  Pattern p = Pattern.compile(REGEX);
  Matcher m = p.matcher(INPUT); // get a matcher object
  int count = 0;

  while(m.find()) {
    count++;
    System.out.println("Match number "+count);
    System.out.println("start(): "+m.start());
    System.out.println("end(): "+m.end());
 }
```

} }

这是产生的结果：

Match number 1 start(): 0 end(): 3 Match number 2 start(): 4 end(): 7 Match number 3 start(): 8 end(): 11 Match number 4 start(): 19 end(): 22

你可以看到这个例子用次边界来确保字母 "c""a""t" 不仅仅是一个长单词子串。它也给出了一些关于在输入字符串中匹配位置的有用的信

start方法返回值是之前end方法返回值加1。

matches 和 lookingAt 方法

matches 和 lookingAt 方法都是按照一定的模式匹配输入序列，两种方法不同的是 matches 方法需要匹配整个输入序列，找出其中不

两种方法都总是开始于输入字符串的起始位置。这里是一个例子：

import java.util.regex.Matcher; import java.util.regex.Pattern;

public class RegexMatches { private static final String REGEX = "foo"; private static final String INPUT = "fooooooooooooooooo"; private static Pattern pattern; private static Matcher matcher;

```
public static void main( String args[] ){
  pattern = Pattern.compile(REGEX);
  matcher = pattern.matcher(INPUT);

  System.out.println("Current REGEX is: "+REGEX);
  System.out.println("Current INPUT is: "+INPUT);
```

```
  System.out.println("lookingAt(): "+matcher.lookingAt());
  System.out.println("matches(): "+matcher.matches());
```

} }

这是产生的结果:

Current REGEX is: foo Current INPUT is: fooooooooooooooooo lookingAt(): true matches(): false

replaceFirst 方法和 replaceAll 方法

replaceFirst 和 replaceAll 方法替换了匹配给定正则表达式的文本。正如它们的名字所表明的，replaceFirst 替换第一个情况，而 rep

这里是解释功能的例子:

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);
```

} }

以上将会产生如下结果:

The cat says meow. All cats say meow.

appendReplacement 和 appendTail 方法

Matcher 类还提供了 appendReplacement 和 appendTail 两种方法来替换文本。

这里是解释功能的例子:

import java.util.regex.Matcher; import java.util.regex.Pattern;

public class RegexMatches { private static String REGEX = "a*b"; private static String INPUT = "aabfo oaabfooabfoob"; private static String REPLACE = "-"; public static void main(String[] args) { Pattern p = Pattern.compile(REGEX); // get a matcher object Matcher m = p.matcher(INPUT); StringBuffer sb = new StringBuffer(); while(m.find()){ m.appendReplacement(sb,REPLACE); } m.appendTail(sb); Syste m.out.println(sb.toString()); } }

以上将会产生如下结果：

-foo-foo-foo-

PatternSyntaxException Class 方法

PatternSyntaxException 是一个未检查的、在正则表达式模式指示语法错误的特例。PatternSyntaxException 类提供了以下的方法

SN	方法描述
1	public String getDescription() 检索错误的描述
2	public int getIndex() 检索误差指标
3	public String getPattern() 检索错误的正则表达式模式
4	public String getMessage() 返回一个包含语法错误的描述及其指标的多行字符串、错误的正则表达式模式以及显示模式里

Java 方法

一个 Java 方法是为了执行某个操作的一些语句的组合。举个例子来说，当你调用 System.out.println 方法时，系统实际上会执行很多

现在你将学习怎么创建你自己的方法，他们可以有返回值也可以没有返回值，可以有参数，也可以没有参数，重载方法要使用相同的方

创建方法

我们用下面的例子来解释方法的语法：

public static int funcName(int a, int b) { // body }

在这里

- public static：修饰符
- int：返回值类型
- funcName：函数名称
- a，b：形式参数
- int a,int b：参数列

方法也包含过程或函数。

- 过程：他们不返回值
- 函数：他们返回值

方法的定义包含方法头和方法体。如下所示：

modifier returnType nameOfMethod (Parameter List) { // method body }

以上的语法包括

- modifier：他定义了方法的访问类型，它是可选的。
- returnType：方法是可能返回一个值的。
- nameOfMethod：这是方法的名称。方法签名包括方法名称和参数列表。
- Parameter List：参数列表，它是参数的次序，类型，以及参数个数的集合。这些都是可选的，当然方法也可以没有参数。
- 方法体：方法体定义了这个方法是用来做什么的。

示例

这是上面定义的方法max()，该方法接受两个参数num1和num2返回两者之间的最大值。

/** the snippet returns the minimum between two numbers */ public static int minFunction(int n1, int n 2) { int min; if (n1 > n2) min = n2; else min = n1;

return min; }

方法调用

要想使用一个方法，该方法必须要被调用。方法调用有两种方式，一种是有返回值的，一种是没有返回值的。

调用方法很简单，当程序需要调用一个方法时，控制程序转移到被调用的方法，方法将会返回两个条件给调用者：

- 返回一条执行语句
- 执行到方法结束

将返回void的方法作为一个调用语句，让我看下面的例子：

System.out.println("wiki.jikexueyuan.com!");

该方法的返回值可以通过下面的例子被理解：

int result = sum(6, 9);

示例

下面的例子表明了怎么定义方法和怎么调用它：

```
public class ExampleMinNumber{

public static void main(String[] args) { int a = 11; int b = 6; int c = minFunction(a, b); System.out.println("Minimum Value = " + c); }

/** returns the minimum of two numbers */ public static int minFunction(int n1, int n2) { int min; if (n1 > n2) min = n2; else min = n1;

  return min;

} }
```

将会产生如下的结果

```
Minimum value = 6
```

关键字 void

关键字 void 允许我们创建一个没有返回值的方法。这里我们在下一个例子中创建一个 void 方法 methodRankPoints。这个方法是没有

```
public class ExampleVoid {

public static void main(String[] args) { methodRankPoints(255.7); }

public static void methodRankPoints(double points) { if (points >= 202.5) { System.out.println("Rank:A1"); } else if (points >= 122.4) { System.out.println("Rank:A2"); } else { System.out.println("Rank:A3"); } } }
```

这将产生如下的结果：

```
Rank:A1
```

通过值来传递参数

在调用函数时参数是必须被传递的。并且他们的次序必须和他们创建时的参数次序是一样的。参数可以通过值或引用来传递。

通过值传递参数意味着调用方法的参数，通过参数值来传递给参数。

示例

下面的程序给出了一个例子来显示通过值来传递参数。在调用方法后参数值是不会发生变化的。

```
public class swappingExample {

public static void main(String[] args) { int a = 30; int b = 45;

  System.out.println("Before swapping, a = " +
                     a + " and b = " + b);

  // Invoke the swap method
  swapFunction(a, b);
  System.out.println("\n**Now, Before and After swapping values will be same here**:");
  System.out.println("After swapping, a = " +
                    a + " and b is " + b);

}

public static void swapFunction(int a, int b) {

  System.out.println("Before swapping(Inside), a = " + a
                      + " b = " + b);
  // Swap n1 with n2
  int c = a;
  a = b;
  b = c;

  System.out.println("After swapping(Inside), a = " + a
                     + " b = " + b);

} }
```

这将产生如下的结果：

Before swapping, a = 30 and b = 45 Before swapping(Inside), a = 30 b = 45 After swapping(Inside), a = 45 b = 30

Now, Before and After swapping values will be same here: After swapping, a = 30 and b is 45

方法的重载

当一个方法有两个或者更多的方法，他们的名字一样但是参数不同时，就叫做方法的重载。它与覆盖是不同的。覆盖是指方法具有相同

让我们来考虑之前的找最小整型数的例子。如果我们要求寻找浮点型中最小的数时，我们就需要利用方法的重载来去创建函数名相同，

下面的例子给予解释：

```
public class ExampleOverloading{

public static void main(String[] args) { int a = 11; int b = 6; double c = 7.3; double d = 9.4; int result1 = mi
nFunction(a, b); // same function name with different parameters double result2 = minFunction(c, d);
System.out.println("Minimum Value = " + result1); System.out.println("Minimum Value = " + result2); }

// for integer public static int minFunction(int n1, int n2) { int min; if (n1 > n2) min = n2; else min = n1;

  return min;

} // for double public static double minFunction(double n1, double n2) { double min; if (n1 > n2) min = n
2; else min = n1;

  return min;

} }
```

这将产生如下结果：

```
Minimum Value = 6 Minimum Value = 7.3
```

重载方法使程序易读。在这里,两种方法名称相同但参数不同。产生整型和浮点类型的最小数作为程序运行结果。

使用命令行参数

有时你想要在程序运行之前传递参数。这可以通过给 main 函数传递命令行参数来实现。

在命令行中，当要执行程序文件时，一个命令行参数是紧接着文件名字后面的出现的。要接受命令行参数在 Java 程序中是十分容易的。

示例

下面的例子展示了将所有命令行参数输出的程序：

```
public class CommandLine {

public static void main(String args[]){ for(int i=0; i<args.length; i++){ System.out.println("args[" + i + "]: "
+ args[i]); } } }
```

通过以下方法来执行该程序：

java CommandLine this is a command line 200 -100

这将产生如下的结果：

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

构造函数

这是一个简单的使用构造函数的例子:

// A simple constructor. class MyClass { int x;

// Following is the constructor MyClass() { x = 10; } }

你可以通过以下方法来调用构造函数来实例化一个对象：

public class ConsDemo {

public static void main(String args[]) { MyClass t1 = new MyClass(); MyClass t2 = new MyClass(); System.out.println(t1.x + " " + t2.x); } }

通常，你将需要用构造函数来接受一个或多个参数。参数的传递和以上介绍的普通方法的参数传递是一样的，就是在构造函数的名字后面

示例

这是一个简单的使用构造函数的例子:

// A simple constructor. class MyClass { int x;

// Following is the constructor MyClass(int i) { x = i; } }

你可以通过以下方法来调用构造函数来实例化一个对象：

public class ConsDemo {

public static void main(String args[]) { MyClass t1 = new MyClass(10); MyClass t2 = new MyClass(20); System.out.println(t1.x + " " + t2.x); } }

这将产生如下的结果：

10 20

可变长参数

JDK1.5 能够允许你传递可变长的同一类型的参数。用如下方法进行声明：

typeName... parameterName

方法声明时，你要在省略号前明确参数类型，并且只能有一个可变长参数，并且可变长参数必须是所有参数的最后一个。

示例

public class VarargsDemo {

public static void main(String args[]) { // Call method with variable args
printMax(34, 3, 3, 2, 56.5); printMax(new double[]{1, 2, 3}); }

public static void printMax(double... numbers) { if (numbers.length == 0) { System.out.println("No argument passed"); return; }

double result = numbers[0];

for (int i = 1; i < numbers.length; i++) if (numbers[i] > result) result = numbers[i]; System.out.println("The max value is " + result); } }

这将产生如下的结果：

The max value is 56.5 The max value is 3.0

finalize() 方法

你可以定义一个方法，仅在被垃圾收集器销毁之前才会被调用。这个方法叫做 finalize() 方法，它也可以用来确保一个对象被干净清除了

举个例子，你也许用 finalize() 来确保被一个对象打开的文件已经关闭了。

为了给类添加一个终结器，你只需定义 finalize() 方法。Java要回收该类的一个对象时，会调用该方法。

在 finalize() 方法中，你将指定一些必须在对象销毁之前要做的行为。

finalize()方法一般是如下形似：

protected void finalize( ) { // finalization code here }

这里，关键字 protected 是为了保证在类外的代码不能访问 finalize() 方法。

这意味着你不能知道 finalize() 什么时候执行。举个例子，如果你的程序在垃圾收集器发生之前就结束了，finalize() 方法将不会被执行

Java 文件和 I/O

在 Java 中 java.io 包含的每一个类几乎都要进行输入和输出操作。所有的这些流代表一个输入源和输出目的地。在 java.io 包中支持许

流可以被定义为一个序列的数据。输入流用来从一个源中读数据，输出流用来向一个目的地写数据。

Java 提供了强大且灵活的有关文件和网络的 I/O 功能，但本教程涵盖的是非常基础的流和 I/O 操作。我们可以看到一个接一个的最基本

字节流

Java 字节流是用来处理8比特字节的输入和输出。尽管有许多的有关字节流的类，但是最常用的是 FileInputStream 类和 FileOutputS

import java.io.*;

public class CopyFile { public static void main(String args[]) throws IOException { FileInputStream in = null; FileOutputStream out = null;

```
  try {
    in = new FileInputStream("input.txt");
    out = new FileOutputStream("output.txt");

    int c;
    while ((c = in.read()) != -1) {
      out.write(c);
    }
  }finally {
    if (in != null) {
      in.close();
    }
    if (out != null) {
      out.close();
    }
  }
```

} }

接下来我们需要有一个文件 input.txt ，内容如下：

This is test for copy file.

下一步，编译上面的程序并执行它，结果会创建一个文件叫做 output.txt，它的内容和上面的 input.txt 内容是一样的。所以让我们将上

$javac CopyFile.java $java CopyFile

字符流

Java 字节流是用来处理8比特字节的输入和输出，Java 字符流用于处理16位 unicode 的输入和输出。尽管这里有许多关于字符流的类

我们能够重写上面的例子来完成把一个输入文件（包含 unicode 字符）的内容复制到输出文件:

```
import java.io.*;

public class CopyFile { public static void main(String args[]) throws IOException { FileReader in = null;
FileWriter out = null;

  try {
    in = new FileReader("input.txt");
    out = new FileWriter("output.txt");

    int c;
    while ((c = in.read()) != -1) {
      out.write(c);
    }
  }finally {
    if (in != null) {
      in.close();
    }
    if (out != null) {
      out.close();
    }
  }

} }
```

接下来我们需要有一个文件 input.txt ，内容如下：

This is test for copy file.

下一步，编译上面的程序并执行它，结果会创建一个文件叫做 output.txt，它的内容和上面的 input.txt 内容是一样的。所以让我们将上

$javac CopyFile.java $java CopyFile

标准流

所有的编程语言都提供了对标准 I/O 流的支持，即用户可以从自己的键盘上进行输入，并且从屏幕上输出。如果你了解 C 或 C++ 编程

- Standard Input：这是用来将数据反馈给用户的程序，通常键盘作为标准输入流并且表示为 System.in。
- Standard Output：这是用于输出用户程序产生的数据，通常电脑屏幕作为标准输出流，并且表示为 System.out。
- Standard Error: 这是用来输出用户产生的错误数据，通常电脑屏幕作为标准错误流，并且表示为 System.err。

下面是一个示例程序用来创建一个 InputStreamReader 来读标准输入流直到用户输入字符 "q"：

import java.io.*;

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

ader cin = null;

```
  try {
    cin = new InputStreamReader(System.in);
    System.out.println("Enter characters, 'q' to quit.");
    char c;
    do {
      c = (char) cin.read();
      System.out.print(c);
    } while(c != 'q');
  }finally {
    if (cin != null) {
      cin.close();
    }
  }
```

} }

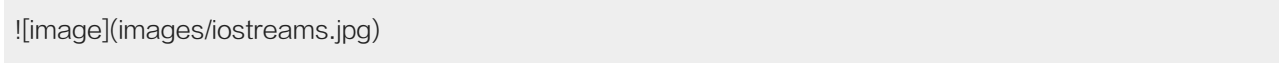
让我们把上面的代码放到 ReadConsole.java 中并且用如下方法编译执行它。这个程序将不断地读写相同的字符直到我们按下 "q"。

$javac ReadConsole.java $java ReadConsole Enter characters, 'q' to quit. 1 1 e e q q

读写文件

像我们之前描述的，一个流可以定义为一个序列的数据。 输入流用来从一个源中读数据，输出流用来向一个目的地写数据。

这里是相关类的层次来表示输入和输出流：



两个重要的流是 FileInputStream 和 FileOutputStream，我们将会在本教程中讨论。

文件输入流

这个流是用来从文件中读数据的。对象能够通过用关键字 new 来创建，并且这里有很多可用的不同类型的构造函数。

下面的构造函数以文件名的字符串为参数来创建一个输入流对象去读文件。

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

下面的构造函数需要以一个文件对象作为参数来去创建一个输入流对象去读文件，首先我们用 File() 方法来去创建一个文件对象：

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

一旦你有了一个输入流对象，这里将有很多的提供帮助的方法来进行读操作或是其他在流中的操作。

SN	方法描述
1	public void close() throws IOException{} 这个方法关闭文件输出流。释放有关文件的所有系统资源。抛出IO异常。
2	protected void finalize()throws IOException {} 这个方法会切断和文件的连接。确保这个文件输出流的关闭方法在这个流
3	public int read(int r)throws IOException{} 这个方法从InputStream 中读到特定字节数的数据。返回一个int类型。返回下
4	public int read(byte[] r) throws IOException{} 这个方法是从输入流中读 r 个长度字节到数组中。返回所有读到的字节数，
5	public int available() throws IOException{} 给出能从输入流中读到的字节数，返回一个int类型数据。

文件输出流

FileOutputStream 是用来创建一个文件，并向其中写入数据。如果之前没有该文件，该流会在打开流之前创建一个文件。

这里是两个能够产生 FileOutputStream 对象的构造函数。

下面的构造函数以文件名的字符串作为参数来创建一个输出流对象去写文件：

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

下面的构造函数需要以一个文件对象作为参数来去创建一个输出流对象去写文件，首先我们用 File() 方法来去创建一个文件对象：

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

一旦你有了一个输出流对象，这里将有很多的提供帮助的方法来进行写操作或是其他在流中的操作。

SN	方法描述
1	public void close() throws IOException{} 这个方法关闭文件输出流。释放有关文件的所有系统资源。抛出IO异常。
2	protected void finalize()throws IOException {} 这个方法会切断和文件的连接。确保这个文件输出流的关闭方法在这个流
3	public void write(int w)throws IOException{} 这个方法用来写特定的字节到输出流。
4	public void write(byte[] w) 从字节数组中写 w 长度字节到输出流。

示例

下面是一个例子来演示 InputStream 和 OutputStream：

```
import java.io.*;

public class fileStreamTest{

public static void main(String args[]){

try{ byte bWrite [] = {11,21,3,40,5}; OutputStream os = new FileOutputStream("test.txt"); for(int x=0; x
< bWrite.length ; x++){ os.write( bWrite[x] ); // writes the bytes } os.close();

  InputStream is = new FileInputStream("test.txt");
  int size = is.available();

  for(int i=0; i< size; i++){
    System.out.print((char)is.read() + "  ");
  }
  is.close();

}catch(IOException e){ System.out.print("Exception"); }
} }
```

上面的代码将产生 test.txt 文件并且以二进制格式写入给出的数字。同样将在 stdout 屏幕输出。

文件导航和 I/O

这里我们需要查看很多类来去了解文件导航和I/O。

- [File Class](http://www.tutorialspoint.com/java/java_file_class.htm)
- [FileReader Class](http://www.tutorialspoint.com/java/java_filereader_class.htm)
- [FileWriter Class](http://www.tutorialspoint.com/java/java_filewriter_class.htm)

Java目录

目录是一个文件可以包含其他文件和目录的列表。你想要在目录中列出可用文件列表，可以通过使用 File 对象创建目录，获得完整详细

创建目录

这里有两个有用的文件方法，能够创建目录：

- mkdir( ) 方法创建了一个目录，成功返回 true ，创建失败返回 false。失败情况是指文件对象的路径已经存在了,或者无法创建目录,因
- mkdirs( ) 方法创建一个目录和它的上级目录。

以下示例创建 “/ tmp / user / java / bin” 目录:

import java.io.File;

public class CreateDir { public static void main(String args[]) { String dirname = "/tmp/user/java/bin"; Fil e d = new File(dirname); // Create directory now. d.mkdirs(); } }

编译并执行以上代码创建 “/ tmp /user/ java / bin”。

提示：Java 自动按 UNIX 和 Windows 约定来处理路径分隔符。如果在 Windows 版本的 Java 中使用正斜杠(/),仍然可以得到正确的

目录列表

如下，你能够用 File 对象提供的 list() 方法来列出目录中所有可用的文件和目录

import java.io.File;

public class ReadDir { public static void main(String[] args) {

```
  File file = null;
  String[] paths;

  try{
    // create new file object
    file = new File("/tmp");

    // array of files and directory
    paths = file.list();

    // for each name in the path array
    for(String path:paths)
    {
      // prints filename and directory name
      System.out.println(path);
    }
```

```
  }catch(Exception e){
     // if any error occurs
     e.printStackTrace();
  }
```

} }

```
基于你/ tmp目录下可用的目录和文件，将产生以下结果：
```

test1.txt test2.txt ReadDir.java ReadDir.class

```
## Java 异常处理

异常是一个程序执行过程中出现的问题。引起异常的原因包括以下几点：

- 用户输入无效的数据
- 用户打开一个不能被找到的文件
- 网络连接已经丢失或 JVM 已经耗尽内存

一些异常是由于用户的错误，也有是因为程序员的错误，还有是因为一些物理资源在某些形式上的错误。

在 Java 中了解异常处理,您需要了解异常的三个类别:

- 检测异常：一个已检测异常通常是用户错误或是一个程序员不能预见的错误，如果一个文件将要被打开，但系统找不到这个文件，异
- 运行时异常：运行时异常是可以被程序语法避免的。不同于检测异常，运行时异常在编译时可以被忽略。
- 错误：这并不是异常，但这不是用户或程序员可以控制的。错误经常在你的代码中被忽略因为你针对一个错误几乎做不了任何事。比如

### 异常的层次结构

所有的异常类都是 java.lang.Exception 类的子类型。异常类都是 Throwable 类的子类。除了异常类 Error 类也是由 Throwable 类产

错误一般不会由 Java 程序解决。这些条件通常是在 Java 程序解决不了的错误出现时才会发生。 Errors 用来去指示那些运行时环境生

Exception 类含有两个子类：IOException 类和 RuntimeException 类。

![image](images/exceptions.jpg)

### 异常方法

下面是 Throwable 类的重要方法列表。

|SN|    方法描述|
|:---------- |:----------|
|1|    public String getMessage() <br> 返回关于发生异常的细节信息，这些信息在Throwable的构造函数中被初始化|
```

|2| public Throwable getCause()
返回发生异常的原因，由 Throwable 对象来表示|
|3| public String toString()
 返回与getMessage()的结果相联系的类的名称|
|4| public void printStackTrace()
打印 toString()跟踪错误输出流的栈地址的结果。|
|5| public StackTraceElement [] getStackTrace()
 返回一个数组,其中包含每个元素在栈的地址。元素的索引0代表调用栈的
|6| public Throwable fillInStackTrace()
 用当前栈地址来填充 Throwable 对象的栈地址,添加到任何先前的栈地址信息。|

捕获异常

一个方法用关键字 try 和 catch 来捕获异常。一个 try/catch 块放置在可能产生异常的代码外。在 try/catch 块内的代码是被保护的，并

try { //Protected code }catch(ExceptionName e1) { //Catch block }

关于 catch 的声明，你必须要指明你要捕获的异常的类型。如果受保护的代码中发生异常，跟在 try 后面的 catch 块会被检测。如果异

示例

下面的例子声明了一个含有两个元素的数组。然后代码试图访问数组的第三个元素，这将抛出一个异常。

// File Name : ExcepTest.java import java.io.*; public class ExcepTest{

public static void main(String args[]){ try{ int a[] = new int[2]; System.out.println("Access element three :" + a[3]); }catch(ArrayIndexOutOfBoundsException e){ System.out.println("Exception thrown :" + e); } System.out.println("Out of the block"); } }

这将产生如下的结果

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

多个catch块

一个 try 块可以有多个 catch 块。多个 catch 块的语法如下:

try { //Protected code }catch(ExceptionType1 e1) { //Catch block }catch(ExceptionType2 e2) { //Catch block }catch(ExceptionType3 e3) { //Catch block }

前面的语句演示三个 catch 块,但是你可以在一个 try 后面跟任意数量的 catch 块。当被保护代码中出现异常时，异常首先会被第一个 c

示例

下面是代码段显示如何使用多个 try/catch 语句。

try { file = new FileInputStream(fileName); x = (byte) file.read(); }catch(IOException i) { i.printStackTra ce(); return -1; }catch(FileNotFoundException f) //Not valid! { f.printStackTrace(); return -1; }

throws/throw 关键词

如果一个方法不能处理检测异常，那么该方法必须用关键字 throws 来声明。throws 关键字在方法签名的最后出现。

你可以通过关键字 throw 来抛出一个异常或进行你刚捕获到的异常的实例化。试着去理解关键字 throws 和 throw 的不同。

以下方法声明它抛出 RemoteException 异常:

import java.io.*; public class className { public void deposit(double amount) throws RemoteExceptio n { // Method implementation throw new RemoteException(); } //Remainder of class definition }

一个方法可以声明抛出一个以上的异常,这些异常用逗号分隔。例如,下面的方法声明抛出 RemoteException 和 InsufficientFundsExc

import java.io.*; public class className { public void withdraw(double amount) throws RemoteExcept ion, InsufficientFundsException { // Method implementation } //Remainder of class definition }

关键字 finally

finally 关键字用于创建跟在 try 后面的代码块,finally 代码块总是会被执行的，不管是怎样的异常发生。

使用 finally 块允许你运行您想要执行的任何 cleanup-type 语句,无论在受保护的代码中发生什么。

finally 代码块出现在最后一个 catch 块之后并且语法如下:

try { //Protected code }catch(ExceptionType1 e1) { //Catch block }catch(ExceptionType2 e2) { //Catch block }catch(ExceptionType3 e3) { //Catch block }finally { //The finally block always executes. }

示例

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 statem ent is executed"); } } }

这将产生如下结果:

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

以下几点需要注意:
- catch 在没有 try 关键字的情况下是不能够出现的。
- try/catch 语句块中并不是强制出现 finally 块。
- try 语句块不能独立存在（即没有任何 catch 和 finally 块）。
- 在 try catch 和 finally 块之间是不能出现任何代码的。

声明自己的异常

你可以在 Java 中创建自己的异常。编写自己的异常类，要记住以下几点:
- 所有的异常必须 Throwable 的子类。
- 如果你想写一个自动遵守正确处理或声明规则的检测异常,需要继承 Exception 类。
- 如果你想编写一个运行时异常,需要继承 RuntimeException 类。

我们如下可以定义自己的异常类:

class MyException extends Exception{ }

你只需要继承 Exception 类来创建你自己的异常类。这些被认为是检测异常。以下 InsufficientFundsException 类是一个用户定义的

例子:

// File Name InsufficientFundsException.java import java.io.*;

public class InsufficientFundsException extends Exception { private double amount; public InsufficientFundsException(double amount) { this.amount = amount; } public double getAmount() { return amount; } }

为了演示使用用户定义的异常,下面的 CheckingAccount 类包含一个 withdraw() 方法抛出了一个 InsufficientFundsException 异常

// File Name CheckingAccount.java import java.io.*;

public class CheckingAccount { private double balance; private int number; public CheckingAccount(int number) { this.number = number; } public void deposit(double amount) { balance += amount; } public void withdraw(double amount) throws InsufficientFundsException { if(amount <= balance) { balance

-= amount; } else { double needs = amount - balance; throw new InsufficientFundsException(needs); } } public double getBalance() { return balance; } public int getNumber() { return number; } }

下面的 BankDemo 程序演示了调用 CheckingAccount 的 deposit() 和 withdraw() 方法。

// File Name BankDemo.java public class BankDemo { public static void main(String [] args) { CheckingAccount c = new CheckingAccount(101); System.out.println("Depositing $500..."); c.deposit(500.00); try { System.out.println("\nWithdrawing $100..."); c.withdraw(100.00); System.out.println("\nWithdrawing $600..."); c.withdraw(600.00); }catch(InsufficientFundsException e) { System.out.println("Sorry, but you are short $" + e.getAmount()); e.printStackTrace(); } } }

编译上述三个文件并运行 BankDemo,将产生以下结果:

Depositing $500...

Withdrawing $100...

Withdrawing $600... Sorry, but you are short $200.0 InsufficientFundsException at CheckingAccount.withdraw(CheckingAccount.java:25) at BankDemo.main(BankDemo.java:13) ```

常见的异常

在 Java 中,可以定义两种异常和错误 - JVM 异常：这些异常/错误是由 JVM 在逻辑层上或专门抛出的。例子：NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException。 - 程序性异常：这些异常是由应用程序或是编写 API 的程序员明确抛出的，如：IllegalArgumentException, IllegalStateException。

2

Java 面向对象

Java 继承

继承可以被定义为一个对象获取另一个对象属性的过程。使用继承可以使信息以继承顺序有序管理。

当我们谈论起继承，最常用的关键字应该为 extends 和 implements。 这些关键字将决定一个对象是否是 A 类型或是其他类型。通过使用这些关键字,我们可以使一个对象获得另一个对象的属性。

IS-A 关系

IS-A 是说：这个对象所属于另一个对象。让我们来看怎么用关键字 extends 来实现继承。

```
public class Animal{
}

public class Mammal extends Animal{
}

public class Reptile extends Animal{
}

public class Dog extends Mammal{
}
```

现在,基于上面的例子,在面向对象编程中,请遵循以下几点: - Animal 是 Mammal 类的父类。 - Animal 是 Reptile 类的父类。 - Reptile 和 Mammal 是 Animal 类的子类。 - Dog 是 Mammal 类和 Animal 类的子类。

现在，针对 IS-A 关系，我们可以说: - Mammal 是一个 Animal - Reptile 是一个 Animal - Dog 是一个 Mammal - Dog 也是一个 Animal

使用关键字 extends，子类可以继承父类除了私有属性的所有属性。

使用 instance 操作我们可以保证 Mammal 实际上是一个 Animal。

示例

```
public class Dog extends Mammal{

  public static void main(String args[]){

    Animal a = new Animal();
```

```
    Mammal m = new Mammal();
    Dog d = new Dog();

    System.out.println(m instanceof Animal);
    System.out.println(d instanceof Mammal);
    System.out.println(d instanceof Animal);
  }
}
```

这将得到如下结果：

```
true
true
true
```

我们已经很好理解了 extends 关键字，让我们来看 implements 关键字是如何确立 IS-A 关系的。

implements 关键字，是在类继承接口的时候使用的。接口是不能被类使用 extends 继承的。

示例

```
public interface Animal {}

public class Mammal implements Animal{
}

public class Dog extends Mammal{
}
```

关键字 instanceof

让我们使用 instanceof 操作符来检查确定是否 Mammal 实际上是 Animal, dog 实际上是一种Animal。

```
interface Animal{}

class Mammal implements Animal{}

public class Dog extends Mammal{
  public static void main(String args[]){

    Mammal m = new Mammal();
    Dog d = new Dog();
```

```
    System.out.println(m instanceof Animal);
    System.out.println(d instanceof Mammal);
    System.out.println(d instanceof Animal);
  }
}
```

这将产生如下结果：

```
true
true
true
```

HAS-A 关系：

该关系是基于使用方便的。这决定了一个类是否含有 A 的一些属性。该关系帮助减少代码重复和漏洞的出现。

让我们看下面的例子:

```
public class Vehicle{}
public class Speed{}
public class Van extends Vehicle{
   private Speed sp;
}
```

这个例子表明 Van 含有 Speed.因为我们单独定义了 Speed 类，我们不必将整个 Speed 类的代码加入 Van 类，使其在多个应用程序中重用 Speed 类。

在面向对象中，用户不用去考虑哪一个对象在做实际的工作。为了实现这个功能， Van 类向用户隐藏了实现具体细节的类。当用户让 Van 类去做一项工作时，Van 类或者自己来做，或者求助其他类来做这项工作。

请记住一个非常重要的事实，Java 只支持单继承，这意味着一个类只能继承一个类，所以以下的是非法的：

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

然而，一个类可以实现一个或多个接口，这使得 Java 可以摆脱不能多继承的问题。

Java 重写

在上一章节中，我们讨论了父类和子类。如果一个类从它的父类继承了一个方法，如果这个方法没有被标记为 final ，就可以对这个方法进行重写。

重写的好处是：能够定义特定于子类类型的行为，这意味着子类能够基于要求来实现父类的方法。

在面向对象编程中， overriding 意味着去重写已经存在的方法。

示例

让我们来看以下的例子：

```
class Animal{

  public void move(){
    System.out.println("Animals can move");
  }
}

class Dog extends Animal{

  public void move(){
    System.out.println("Dogs can walk and run");
  }
}

public class TestDog{

  public static void main(String args[]){
    Animal a = new Animal(); // Animal reference and object
    Animal b = new Dog(); // Animal reference but Dog object

    a.move();// runs the method in Animal class

    b.move();//Runs the method in Dog class
  }
}
```

这将产生如下结果：

```
Animals can move
Dogs can walk and run
```

在上面的例子中，你可以看到尽管 b 是 Animal 类型，但它运行了 dog 类的方法。原因是：在编译时会检查引用类型。然而，在运行时，JVM 会判定对象类型到底属于哪一个对象。

因此，在上面的例子中，虽然 Animal 有 move 方法，程序会正常编译。在运行时，会运行特定对象的方法。

考虑下面的例子：

```
class Animal{

  public void move(){
    System.out.println("Animals can move");
  }
}

class Dog extends Animal{

  public void move(){
    System.out.println("Dogs can walk and run");
  }
  public void bark(){
    System.out.println("Dogs can bark");
  }
}

public class TestDog{

  public static void main(String args[]){
    Animal a = new Animal(); // Animal reference and object
    Animal b = new Dog(); // Animal reference but Dog object

    a.move();// runs the method in Animal class
    b.move();//Runs the method in Dog class
    b.bark();
  }
}
```

这将产生如下结果：

```
TestDog.java:30: cannot find symbol
symbol  : method bark()
location: class Animal
            b.bark();
             ^
```

这个程序在编译时将抛出一个错误，因为 b 的引用类型 Animal 没有一个名字叫 bark 的方法。

方法重写规则

- 重写方法的参数列表应该与原方法完全相同。
- 返回值类型应该和原方法的返回值类型一样或者是它在父类定义时的子类型。
- 重写函数访问级别限制不能比原函数高。举个例子：如果父类方法声明为公有的，那么子类中的重写方法不能是私有的或是保护的。
- 只有被子类继承时，方法才能被重写。
- 方法定义为 final，将导致不能被重写。
- 一个方法被定义为 static，将使其不能被重写，但是可以重新声明。
- 一个方法不能被继承，那么也不能被重写。
- 和父类在一个包中的子类能够重写任何没有被声明为 private 和 final 的父类方法。
- 和父类不在同一个包中的子类只能重写 non-final 方法或被声明为 public 或 protected 的方法。
- 一个重写方法能够抛出任何运行时异常，不管被重写方法是否抛出异常。然而重写方法不应该抛出比被重写方法声明的更新更广泛的已检查异常。重写方法能够抛出比被重写方法更窄或更少的异常。
- 构造函数不能重写。

使用 super 关键字

当调用父类的被重写的方法时，要用关键字 super。

```
class Animal{

  public void move(){
    System.out.println("Animals can move");
  }
}

class Dog extends Animal{

  public void move(){
    super.move(); // invokes the super class method
    System.out.println("Dogs can walk and run");
  }
```

```
}

public class TestDog{

  public static void main(String args[]){

    Animal b = new Dog(); // Animal reference but Dog object
    b.move(); //Runs the method in Dog class

  }
}
```

这将产生如下结果:

```
Animals can move
Dogs can walk and run
```

Java 多态

多态性是指对象能够有多种形态。在 OOP 中最常用的多态性发生在当父类引用指向孩子类对象时。

任何能够通过一个以上的 IS-A 测试的 Java 对象被认为是多态的。在 Java 中所有对象都是多态的，因为任何一个对象都会有一个他们自己类型的和 Object 类的 IS-A 关系。

重要的是知道，通过引用变量是唯一可以用来访问一个对象的方法。引用变量可以只有一个类型。引用变量一旦被声明是不能被改变的。

引用变量能够重新分配到其他提供的没有被声明为 final 的对象。引用变量的类型将决定它可以调用的对象的方法。

一个引用变量能够引用任何一个对象的声明类型或任何声明类型的子类型。一个引用变量可以声明为一个类或接口类型。

示例

让我们看下面的例子：

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

现在 Deer 类是多态的，因为他有多个继承机制。针对上面的例子有以下说法：

- Deer 就是 Animal
- Deer 就是 Vegetarian
- Deer 就是 Deer
- Deer 就是 Object

当我们提供引用变量来引用 Deer 对象，下面的声明是合法的：

```
Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
```

所有的引用变量 d, a, v, o 在堆中引用同一个对象 Deer。

虚方法

在这一节中，将展示在 Java 中被覆盖方法的行为在设计类时是如何体现多态性的好处的。

我们已经讨论了覆盖方法，一个子类可以覆盖它父类的方法。一个被覆盖的方法实际上隐藏在父类当中，并且不会被调用，除非子类在覆盖方法中用 super 关键字。

```
/* File name : Employee.java */
public class Employee
{
  private String name;
  private String address;
  private int number;
  public Employee(String name, String address, int number)
  {
    System.out.println("Constructing an Employee");
    this.name = name;
    this.address = address;
    this.number = number;
  }
  public void mailCheck()
  {
    System.out.println("Mailing a check to " + this.name
     + " " + this.address);
  }
  public String toString()
  {
    return name + " " + address + " " + number;
  }
  public String getName()
  {
    return name;
  }
  public String getAddress()
  {
    return address;
  }
  public void setAddress(String newAddress)
  {
    address = newAddress;
  }
  public int getNumber()
  {
```

```
    return number;
  }
}
```

现在我们如下继承 Employee 类：

```
/* File name : Salary.java */
public class Salary extends Employee
{
  private double salary; //Annual salary
  public Salary(String name, String address, int number, double
    salary)
  {
    super(name, address, number);
    setSalary(salary);
  }
  public void mailCheck()
  {
    System.out.println("Within mailCheck of Salary class ");
    System.out.println("Mailing check to " + getName()
    + " with salary " + salary);
  }
  public double getSalary()
  {
    return salary;
  }
  public void setSalary(double newSalary)
  {
    if(newSalary >= 0.0)
    {
      salary = newSalary;
    }
  }
  public double computePay()
  {
    System.out.println("Computing salary pay for " + getName());
    return salary/52;
  }
}
```

现在,你仔细研究下面的程序,试图确定它的输出:

```
/* File name : VirtualDemo.java */
public class VirtualDemo
{
  public static void main(String [] args)
```

```
  {
    Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
    Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
    System.out.println("Call mailCheck using Salary reference --");
    s.mailCheck();
    System.out.println("\n Call mailCheck using Employee reference--");
    e.mailCheck();
  }
}
```

这将产生如下的结果:

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0
```

这里我们实例化两个 Salary 对象。一个用 Salary 引用 s，另一个用 Employee 引用 e。

当调用 s.mailCheck() 方法时，编译器在编译时发现 mailCheck() 在 Salary 类中，并且 JVM 在运行时调用 Salary类的 mailCheck() 方法。

调用 e 的 mailCheck() 是略有不同的因为 e 是一个 Employee 的引用。当编译器发现 e.mailCheck() 时,编译器在 Employee 类中发现 mail.Check() 方法。

这里在编译时,编译器使用 Employee 的 mailCheck() 方法来验证。在运行时，JVM 调用 Salary 类的 mailCheck() 类。

这种行为被称为虚方法调用，该方法也被称为虚方法。Java 中所有此规则的方法行为，无论是什么数据类型的引用，运行时会调用被覆盖方法，在编译时都会遵循于源码。

Java 抽象

Abstraction 是指在 OOP 中让一个类抽象的能力。一个抽象类是不能被实例化的。类的功能仍然存在，它的字段，方法和构造函数都以相同的方式进行访问。你只是不能创建一个抽象类的实例。

如果一个类是抽象的，即不能被实例化，这个类如果不是子类它将没有什么作用。这体现了在设计过程中抽象类是如何被提出的。一个父类包含子类的基本功能集合，但是父类是抽象的，不能自己去使用功能。

抽象类

使用关键字 abstract 来声明一个抽象类。它出现在关键字 class 的前面。

```
/* File name : Employee.java */
public abstract class Employee
{
  private String name;
  private String address;
  private int number;
  public Employee(String name, String address, int number)
  {
    System.out.println("Constructing an Employee");
    this.name = name;
    this.address = address;
    this.number = number;
  }
  public double computePay()
  {
   System.out.println("Inside Employee computePay");
   return 0.0;
  }
  public void mailCheck()
  {
    System.out.println("Mailing a check to " + this.name
     + " " + this.address);
  }
  public String toString()
  {
    return name + " " + address + " " + number;
  }
  public String getName()
  {
```

```
    return name;
  }
  public String getAddress()
  {
    return address;
  }
  public void setAddress(String newAddress)
  {
    address = newAddress;
  }
  public int getNumber()
  {
   return number;
  }
}
```

注意, Employee 类没什么不同。类现在是抽象的,但它仍然有三个字段,七个方法,一个构造函数。

现在如果你尝试一下代码：

```
/* File name : AbstractDemo.java */
public class AbstractDemo
{
  public static void main(String [] args)
  {
    /* Following is not allowed and would raise error */
    Employee e = new Employee("George W.", "Houston, TX", 43);

    System.out.println("\n Call mailCheck using Employee reference--");
    e.mailCheck();
   }
}
```

当你进行编译时，你将得到下面的错误：

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

继承抽象类

我们可以如下继承 Employee 类：

```
/* File name : Salary.java */
public class Salary extends Employee
{
   private double salary; //Annual salary
   public Salary(String name, String address, int number, double
      salary)
   {
      super(name, address, number);
      setSalary(salary);
   }
   public void mailCheck()
   {
      System.out.println("Within mailCheck of Salary class ");
      System.out.println("Mailing check to " + getName()
      + " with salary " + salary);
   }
   public double getSalary()
   {
      return salary;
   }
   public void setSalary(double newSalary)
   {
      if(newSalary >= 0.0)
      {
         salary = newSalary;
      }
   }
   public double computePay()
   {
     System.out.println("Computing salary pay for " + getName());
     return salary/52;
   }
}
```

这里，我们不能实例化一个新的 Employee，但是如果我们实例化一个新的 Salary 对象，Salary 对象将继承 Employee 的三个字段和七个方法。

```
/* File name : AbstractDemo.java */
public class AbstractDemo
{
   public static void main(String [] args)
   {
     Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
     Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
```

```
    System.out.println("Call mailCheck using Salary reference --");
    s.mailCheck();

    System.out.println("\n Call mailCheck using Employee reference--");
    e.mailCheck();
  }
}
```

这将产生如下结果：

```
Constructing an Employee
Constructing an Employee
Call mailCheck using  Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.
```

抽象方法

如果你想一个提供特定方法的类，但是你想要在他的子类中实际实现这个方法，你可以在父类中声明这个方法为抽象的。

abstract 关键字也被用来定义抽象方法。一个抽象方法是有方法签名的但没有方法实体。

抽象方法无需定义，并且它的签名以分号结束，不需要花括号。

```
public abstract class Employee
{
  private String name;
  private String address;
  private int number;

  public abstract double computePay();

  //Remainder of class definition
}
```

声明一个抽象方法有两个结果:

- 如果一个类中含有一个抽象方法，类必须也是抽象的。
- 任何一个子类必须覆盖这个抽象方法，或者继续将它声明为抽象方法。

子类继承一个抽象方法，必须要去覆盖他。如果不这样做的话，它们必须将其继续声明为抽象，或在它们的子类中去覆盖它们。

最终，后代类不得不去实现抽象方法；否则你会一直有一个不能被实例化的抽象类。

如果 Salary 继承 Employee 类，则他必须如下要去实现 computerPay() 方法：

```
/* File name : Salary.java */
public class Salary extends Employee
{
   private double salary; // Annual salary

   public double computePay()
   {
      System.out.println("Computing salary pay for " + getName());
      return salary/52;
   }

   //Remainder of class definition
}
```

Java 封装

封装是面向对象程序设计四大基本概念之一，其余三个分别是继承，多态和抽象。

封装是一种可以使类中的字段私有并能通过公有方法来访问私有字段的技术。如果一个字段被声明为私有，它就不能在类的外部被访问，从而隐藏了类内部的字段。基于这个原因，封装有时也被称为数据隐藏。

封装可以被认为是一种能够保护代码和数据被定义在类外的其它代码任意访问的屏障。访问数据和代码由一个接口严格控制。

封装的主要好处是修改我们实现的代码而又不会破坏其他人使用我们的代码。封装的这个特性使我们的代码具有可维护性、灵活性以及扩展性。

示例

如下是一个使用了封装的例子：

```
/* File name : EncapTest.java */
public class EncapTest{

  private String name;
  private String idNum;
  private int age;

  public int getAge(){
    return age;
  }

  public String getName(){
    return name;
  }

  public String getIdNum(){
    return idNum;
  }

  public void setAge( int newAge){
    age = newAge;
  }

  public void setName(String newName){
```

```
    name = newName;
  }

  public void setIdNum( String newId){
    idNum = newId;
  }
}
```

公有方法是从类外访问到类内字段的入口。通常情况下，这些方法被定义为 getters 和 setters 。因此想要访问类内变量的任何其他类要使用 getters 和 setters 方法。

EncapTest 类的变量可以像如下的方式访问：

```
/* File name : RunEncap.java */
public class RunEncap{

  public static void main(String args[]){
    EncapTest encap = new EncapTest();
    encap.setName("James");
    encap.setAge(20);
    encap.setIdNum("12343ms");

    System.out.print("Name : " + encap.getName()+
                     " Age : "+ encap.getAge());
   }
}
```

这将产生下述结果：

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

封装的优点

- 类中的字段可以被设置为只读或只写。
- 类可以完全控制它字段里面所存储的东西。
- 类的使用者不用知道类是如何存储数据的。类可以改变字段的数据类型而类的使用者不需要改变任何之前的代码。

Java 接口

接口是抽象方法的集合。如果一个类实现了一个接口，那么就需要继承这个接口中的所有抽象方法。

接口不是类。写一个接口和写一个类很相似，但是它们是两个不同的概念。类是描述一个对象的成员属性和行为。接口只包含一个类所实现的行为。

除非实现了接口的类是抽象的,否则接口中的所有方法都需要在类中实现。

在以下方面，接口和类非常相似： - 一个接口可以包含任意数量的方法。 - 一个接口以 `.java` 的扩展名写入文件中，并且接口的名字与文件名相同。 - 接口的字节码位于一个 `.class` 文件中。 - 接口位于包中，并且相应的字节码文件必须在和该包名匹配的文件夹结构下。

然而,在以下方面,接口和类是不同的: - 不能实例化一个接口。 - 接口不能包含构造方法。 - 接口中的所有方法都是抽象的。 - 接口不能包含实例变量。接口中唯一能出现的变量必须被同时声明为 static 和 final 。 - 接口不能被类继承；它应该被类实现。 - 一个接口可以继承多个接口。

声明接口

`interface` 关键字用来声明一个接口。下面是一个声明接口的简单例子：

示例

如下是描述了接口的例子：

```
/* File name : NameOfInterface.java */
import java.lang.*;
//Any number of import statements

public interface NameOfInterface
{
   //Any number of final, static fields
   //Any number of abstract method declarations\
}
```

接口有下述属性： - 接口默认就是抽象的。当需要声明一个接口的时候不需要使用 `abstract` 关键字。 - 接口中的每个方法默认也是抽象的，所以 `abstract` 关键字也不需要。 - 接口中的方法默认是 public 的。

示例：

```
/* File name : Animal.java */
interface Animal {

   public void eat();
   public void travel();
}
```

接口的实现

当一个类实现一个接口的时候，你可以认为类就是签订一个条约，同意去执行接口中的各种行为。如果一个类没有实现接口中的所有行为，这个类就必须声明为 abstract 。

类使用 implements 关键字来实现一个接口。这个 implements 关键字写在类的声明部分中 extends（如果有）部分的后面。

```
/* File name : 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();
   }
}
```

这将产生下面的结果：

```
Mammal eats
Mammal travels
```

当覆写定义在接口中的方法时，如下是需要遵守的几条规则： – 异常不应该声明在实现的方法中，而是应该在声明的接口方法中或者那些声明方法的接口的子类。 – 当覆写方法的时候应该包含接口方法的签名和相同类型或子类型的返回值。 – 接口实现类本身可以是抽象的，如果是抽象的，则接口中的方法没必要全部实现。

当实现接口时有如下几条规则： – 类可以一次性实现多个接口。 – 类只可以继承一个父类，但是可以实现多个接口。 – 一个接口可以继承自另一个接口，和一个类继承自另一个类的方法相同。

接口的继承

一个接口可以继承另一个接口，和一个类继承自另一个类的方法相同。 `extends` 关键字用来继承一个接口，并且子接口要继承父接口的所有方法。

下述的 Sports 接口 被 Hockey 和 Football 接口继承。

```
//Filename: Sports.java
public interface Sports
{
   public void setHomeTeam(String name);
   public void setVisitingTeam(String name);
}

//Filename: Football.java
public interface Football extends Sports
{
   public void homeTeamScored(int points);
   public void visitingTeamScored(int points);
   public void endOfQuarter(int quarter);
}

//Filename: Hockey.java
public interface Hockey extends Sports
{
   public void homeGoalScored();
   public void visitingGoalScored();
   public void endOfPeriod(int period);
   public void overtimePeriod(int ot);
}
```

Hockey 接口有四个方法，但是它从 Sports 接口中继承了两个；因此，一个实现了 Hockey 接口的类需要实现全部的六个方法。类似的，实现了 Football 的类需要定义 Football 接口中三个方法和 Sports 接口中的两个方法。

多个接口的继承

一个 Java 类只可以继承一个父类，不可以多继承。 然而，接口不是类，一个接口可以继承多个父接口。

一旦使用了 `extends` 关键字，所有父接口声明时需要以逗号分隔。

例如，如果 Hockey 接口同时继承了 Sports 和 Event 接口，它需要像如下方式声明：

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

标识接口

继承接口的最普通用法是父接口不包含任何的方法。例如， 在 java.awt.event 包中的 MouseListener 接口继承了 java.util.EventListener 接口，像如下定义：

```
package java.util;
public interface EventListener
{}
```

一个内部没有任何方法的接口被称为 `tagging` interface。 tagging interface 有两个基本的用途：

创建一个共同的父类： 像 EventListener 接口， 它继承了很多 Java API 中的其它接口，你可以使用 tagging interface 在一组接口中创建一个共同的父类。例如，当一个接口继承了 EventListener 接口的时候， Java 虚拟机（JVM）就知道这个特殊的接口被用在事件代理上。

向类添加数据类型： 一个实现了 tagging interface 的类是不需要定义任何方法的（因为这个接口中本来就没有任何方法），但是这个类通过多态的特性变成了一个接口类型。

Java 包

在 Java 中使用包是为了防止命名冲突，来控制访问，使得搜索/定位和类、接口、枚举和注释等的使用更为简单。

包可以被定义为一组相关的类型（类、接口、枚举和注释），提供访问保护和命名空间管理。

在 Java 中一些已经存在的包有： - `java.lang` - 包含了基本类 - `java.io` - 包含有输入，输出功能的类

程序员可以定义自己的包来包含各种类和接口等。把你实现的相关类组织在一起是一种很好的做法，这样一个程序员可以很简单的知道哪些类、接口、枚举，注释是相关的。

由于包创建了一个新的命名空间，因此将不会和其他包中的名字有任何名字冲突。使用包，可以很简单的提供访问控制，并且也可以很简单的定位到相关类。

创建包

当创建一个包的时候，应该为包起一个名字，并且把 `package` 名字的声明语句放在每个包含类、接口、枚举和注释类型的源文件顶部。

`package` 声明语句应该放在源文件的第一行。在每个源文件中只能有一个包声明语句，并且它适用于所有类型的文件。

如果没有使用包声明，那么类、接口、枚举和注释类型的将会被放进一个无名的包中。

示例

让我们来看创建了一个叫做 `Animal` 的包的例子。习惯上使用小写名字的包以避免和类、接口名字的冲突。

在包 *animals* 中声明一个接口：

```
/* File name : Animal.java */
package animals;

interface Animal {
   public void eat();
   public void travel();
}
```

现在，在同一个包 *animals* 中实现一个类：

```
package animals;

/* File name : 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();
   }
}
```

现在，你可以编译这两个文件并将它们放置在一个叫做 `animals` 的子文件夹中，然后像下述来运行：

```
$ mkdir animals
$ cp Animal.class  MammalInt.class animals
$ java animals/MammalInt
Mammal eats
Mammal travels
```

import 关键字

如果一个类想使用同一个包中的另一个类，就没必要使用包名。同一个包中的类不用任何特殊的语法就可以彼此识别。

示例

一个叫做 Boss 的类添加进了已经包含了 Employee 的 payroll 包中。 在没有使用 payroll 前缀的情况下，Boss 类引用了 Employee 类，像下面演示的 Boss 类：

```
package payroll;

public class Boss
{
  public void payEmployee(Employee e)
  {
    e.mailCheck();
  }
}
```

如果 Boss 类 没有在 payroll 包中会发生什么？那么 Boss 类就必须使用下述的其中一种方法来引用位于其它包中的类。 – 可以使用全名。例如：

```
payroll.Employee
```

- 可以使用 import 关键字和通配符（*）来导入。例如：

```
import payroll.*;
```

- 可以使用 import 关键字来导入类本身。例如：

```
import payroll.Employee;
```

注： 一个类的文件可以包含任意数量的导入语句。这些导入语句必须位于包声明语句之后和类的定义语句之前。

包的目录结构

当一个类放置在一个包中时主要会发生以下两个结果： – 包的名字变成了类名字的一部分，就像我们在前一个章节刚刚讨论过的。 – 包的名字必须和相关字节码文件的目录结构匹配。

在 Java 里有以下一种管理文件的简单方式：

将类、接口、枚举、注释类型的源代码存在一个扩展名为 `.java` 的文本文件中。例如：

```
// File Name :  Car.java

package vehicle;
```

```
public class Car {
   // Class implementation.
}
```

现在，将源文件放在一个其名字能够反映出包的名字的文件夹中：

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

现在，正确的类名和路径名应该像下面这样： – 类名 -> vehicle.Car – 路径名 -> vehicle\Car.java (in windo ws)

一般情况下，一个公司使用它的逆序的互联网域名当做包的名称。例如：某公司的互联网域名是 apple.com ，那么它的包名应该是以 com.apple 开头。包名的每一个部分反映了一个子目录。

例如：一个公司有一个包含了 Dell.java 源文件的 com.apple.computers 的包，它将被包含在像下面的一系列子文件夹中：

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

在编译的时候，编译器为其中定义的每个类、接口和枚举创建了一个不同的输出文件。输出文件的基本名字是类型名和 `.class` 的扩展名。

例如:

```
// File Name: Dell.java

package com.apple.computers;
public class Dell{

}
class Ups{

}
```

现在，像下面使用 -d 选项来编译这个文件：

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

这将像下面存放编译后的文件：

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

你可以像下面这样导入所有定义在 *\com\apple\computers* 中的类和接口：

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

像 .java 这样的源文件， .class 这些编译后的文件应该在能够反映出包名的一系列文件夹中。然而， .class 文件的路径不用和 .java 源文件的路径相同。你可以独立的管理源文件和编译后文件的目录，如下：

```
<path-one>\sources\com\apple\computers\Dell.java

<path-two>\classes\com\apple\computers\Dell.class
```

通过这样，可以把编译过后的文件夹给其它的程序员而不用暴露你的源文件。同时你也需要使用这种方式来管理源文件和编译过后的 class 文件以便编译器和 Java 虚拟机（JVM）能够找到程序中使用的所有类型。

编译后文件夹的全路径， \\classes， 叫做 class path， 在系统环境变量 CLASSPATH 中设置。编译器和 JVM 通过添加包名到 class path 中来构造 .class 文件的路径。

我们说 \\classes 是 class path， 包名是 com.apple.computers ，然后编译器和 JVM 将在 \\classes\com\apple\compters 中寻找 .class 文件。

一个 class path 可能包含几个路径。多个路径以分号（Windows）或者冒号（Unix）隔开。默认情况下，编译器和 JVM 查找当前目录和包含 Java 平台字节码的 JAR 文件以便于这些文件夹自动在 class path 中。

设置系统中的 CLASSPATH 环境变量

为了显示当前的 CLASSPATH 环境变量，使用以下在 Windows 和 UNIX（Bourne shell）中的命令： - 在 Windows 中 -> C:> set CLASSPATH - 在 UNIX 中 -> % echo $CLASSPATH

使用下述命令删除当前 CLASSPATH 变量中的内容： - 在 Windows 中 -> C:> set CLASSPATH= - 在 UNIX 中 -> % unset CLASSPATH; export CLASSPATH

使用下述命令设置 CLASSPATH 变量： - 在 Windows 中 -> set CLASSPATH=C:\users\jack\java\classes - 在 UNIX 中 -> % CLASSPATH=/home/jack/java/classes; export CLASSPATH

3

Java 进阶

Java 数据结构

Java 工具包中所提供的数据结构非常强大并且有很多的功能。这些数据结构包含以下的接口和类：

- Enumeration
- BitSet
- Vector
- Stack
- Dictionary
- Hashtable
- Properties

现在这些类都是旧的了， Java-2 已经引入了一种新的框架叫做 Collections Framework，在下一个章节中来讲解它。

Enumeration

枚举接口本身并不是一个数据结构，但是它在其他数据结构中非常重要。 The Enumeration 接口定义了一种方法可以从一个数据结构中连续获取元素。

例如，Enumeration 定义了一个叫做 nextElement 的方法，它可以用来在一个包含很多元素的数据结构中获得下一个元素。

BitSet

BitSet 类实现了一组 bits 或 flags，可以被独立的设置和清除。

在你需要保存一组布尔值的时候这个类非常有用；你只需要给每个值或集合赋值一个 bit 并且可以视情况设置或清除它。

Vector

Vector 类和传统的 Java 数组类似，但是它可以增添新的元素。

和数组一样，Vector 对象的元素可以通过下标序号来访问。

对于使用 Vector 类很好的是,在创建的时候不用担心去设置一个特定的集合大小;当必要的时候,它可以自动收缩和扩张。

Stack

Stack 类中的元素后进先出（LIFO）。

你可以简单认为 stack 就是一个对象的垂直堆栈；当你添加一个新元素的时候，它就位于其它元素的顶部。

当你将一个元素移出 stack，从栈的顶部开始移出。换句话说，加进栈的最后一个元素会被最先移除。

Dictionary

Dictionary 类是一个定义了键值对映射这种数据结构的抽象类。

当你想通过一个特殊的键来访问数据而不是通过一个整数下标时最好使用 Dictionary 这个类。

因为 Dictionary 类是抽象的，它只提供了键值对映射的数据结构框架而不是一个具体的实现。

Hashtable

Hashtable 类提供了一种组织数据的方式，依赖于一些用户自定义的键。

例如，在一个地址列表的 hash table 中，你可以依赖于像 ZIP code 这种键来存储和排序数据而不是依赖于一个人的名字。

在 hash table 中键的特殊含义完全依赖于 hash table 的用途和它所包含的数据。

Properties

Properties 是 Hashtable 的一个子类。它用来包含值的列表，在这个列表中键是一个字符串并且值也是一个字符串。

Properties 类被很多其它 Java 类使用。例如，当获取环境变量值时，它是 System.getProperties( ) 这个方法返回值的对象类型。

Java 集合

Java 2 之前，Java 为对象组的存储和操作提供了特别的类比如 字典，向量，堆栈和属性。尽管这些类确实有用，它们缺少一个中心的，统一的主题。因此，你使用向量的方法和你使用属性的方法是不同的。

集合框架被设计来满足几个目标

- 框架需要是高性能的。基础集合(动态数组，链表，数和哈希表)是高效的。
- 框架需要允许不同的集合类型以类似的方式和高度的互操作性工作。
- 扩展或者调整集合必须是简单的。

为此，整个集合框架被设计围绕一系列的标准接口。几个接口的标准实现例如 LinkedList, HashSet 和 TreeSet 被提供，如果你选择的话，你可以使用，你也可以实现你自己的集合。

一个集合框架是一个统一的体系结构表示和操作集合。所有的集合框架包含以下:

- **接口：** 这些是代表集合的抽象数据类型。接口允许集合独立操作它们表示的细节。在面向对象的语言中，接口通常形成一个层次结构。
- **实现，即类：** 这些是集合接口的具体实施。从本质上说，它们是可重用的数据结构。
- **算法：** 这些是在实现集合接口的对象上进行有用计算的方法，比如搜索和排序。算法被称为多态的，那就是说，同一个方法能被用在许多不同的合适的集合接口的实现上。

除了集合，框架定义了几个 map 接口和类 Maps 存储键值对。尽管 maps 不是正确使用集合的术语，但是他们完全由集合整合起来。

Collection 接口

集合框架定义了几个接口。如下提供了每个接口的概览:

SN	接口描述
1	Collection 接口 这让你可以使用对象组；它是集合层次阶段的顶端
2	List 接口 它继承了 Collection 并且 List 的一个实例存储了元素的一个有序集合
3	Set 它继承了 Collection 来处理集，它必须含有特殊的元素

SN	接口描述
4	SortedSet 它继承了 Set 来处理 排序的 set
5	Map 它将独特的键和值匹配
6	Map Entry 这描述了映射中的一个元素(一个键值对)。它是 Map 的一个内部类。
7	SortedMap 它继承了 Map 因此键按升序保持
8	Enumeration 它是旧有的接口并定义了你可以在对象的集合中列举(一次获得一个)元素的方法。这个旧有的接口被迭代器取代了。

Collection 类

Java 提供了一系列的实现集合接口的标准集合类。一些类提供了完全的能被直接使用的实现，其他就是抽象类，提供的被用来作为创建具体集合的实现。

标准的 collection 类在下面的表格中被概括:

SN	类描述
1	AbstractCollection 实现大部分的 Collection 接口
2	AbstractList 继承 AbstractCollection 并且实现大部分 List 接口
3	AbstractSequentialList 通过一个使用有序的而不是随机访问它的元素的集合继承 AbstractList
4	LinkedList 通过继承 AbstractSequentialList 实现一个链表
5	ArrayList 通过继承 AbstractList 实现一个动态数组
6	AbstractSet 继承 AbstractCollection 并实现大部分的 Set 接口
7	HashSet 用一个哈希表继承 AbstractSet
8	LinkedHashSet 继承 HashSet 来允许插入顺序迭代
9	TreeSet 实现在树中存储的一个集。继承 AbstractSet
10	AbstractMap 实现大部分的 Map 接口

SN	类描述
11	HashMap 用一个哈希表继承 AbstractMap
12	TreeMap 用一棵树继承 AbstractMap
13	WeakHashMap 用一个使用弱键的哈希表来继承 AbstractMap
14	LinkedHashMap 继承 AbstractMap 来允许插入顺序迭代
15	IdentityHashMap 继承 AbstractMap 类并且当比较文档时平等使用参考

AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList 和 AbstractMap 类提供了核心集合接口的实现，尽量减少努力来实现它们。

以下的由 java.util 定义的旧有的类在前面的指南中已经被讨论过:

SN	类描述
1	Vector 这实现一个动态数组。它和 ArrayList 类似，但也有一些不同。
2	Stack Stack 是 Vector 的实现标准的后进先出栈的子类
3	Dictionary Dictionary 是一个抽象的代表一个键值对存储库的类并且操作起来非常像 Map
4	Hashtable Hashtable 是初始的 java.util 的一部分并且是 Dictionary 的具体实现
5	Properties Properties 是 Hashtable 的一个子类。它被用来保持键是一个字符串并且值也是一个字符串的值的列表
6	BitSet 一个 BitSet 类创建一个特殊的保持 bit 数值的数组类型。这个数组的大小能根据需要增长

Collection 算法

集合框架定义了几个能被应用到 collections 和 maps 的算法。这些算法在 Collection 类的内部被定义为静态方法。

几个方法能抛出异常 ClassCastException，它发生在想要比较不兼容的类型时；或者异常UnsupportedOperationException，它发生在想要修改一个不能修改的集合时。

集合定义了三个静态变量：EMPTY_SET, EMPTY_LIST, 和 EMPTY_MAP。所有都是不变的。

SN	算法描述
1	The Collection Algorithms (http://www.tutorialspoint.com/java/java_collection_algorithms.htm) 这是所有算法实现的列表

如何使用 Iterator

通常，你想要在集合中循环元素。比如，你可能想要显示每个元素。

这么做最简单的方法是使用 Iterator，它是一个实现或者是 Iterator 或者 ListIterator 接口的对象。

Iterator 让你可以通过一个集合循环，获得或者除去元素。ListIterator 继承了 Iterator 来允许一个列表的双向遍历和元素的修改。

SN	Iterator 方法描述
1	使用 Java Iterator (http://www.tutorialspoint.com/java/java_using_iterator.htm) 这是所有由 Iterator 和 ListIterator 接口提供的有例子的方法的列表。

如何使用 Comparator

TreeSet 和 TreeMap 都以顺序保存元素。然而，是 Comparator 精确定义排序意味着什么。

这个接口让我们将一个给定的集合用不同数量的方法排序。这个接口也能被用来排列任何类的任何实例(甚至是我们不能修改的类)。

SN	Iterator 方法描述
1	使用 Java Comparator (http://www.tutorialspoint.com/java/java_using_comparator.htm) 这是所有由 Comparator 接口提供的有例子的方法的列表。

总结

Java 集合框架给了程序员打包数据结构和操作它们的算法的入口。

一个集合是一个能对其他对象引用的对象。collection 接口声明了能在每一个集合类型上操作的操作。

集合框架的类和接口都在 java.util 包内。

Java 泛型

如果我们可以写一个单独的能在一个整型数组，一个字符串数组或者一个任何类型支持排序的数组内排列元素的排序方法将会是很好的。

Java Generic 方法和 generic 类让程序员可以用一种单独的方法声明，一系列有关的方法或者用一个单独的类声明来各自指定一系列有关的类型。

Generic 也提供了编译时类型安全来允许程序员在编译时捕获无效的类型。

使用 Java Generic 概念，我们可以写一个泛型方法来给对象数组排序，然后调用带有整型数组，double 型数组，字符串数组或其他的泛型方法来为数组元素排序。

泛型方法

你可以写一个单独的可以被不同类型的参数调用的泛型方法声明。基于传给泛型方法的参数类型，编译器正确处理每个调用的方法。以下是定义泛型方法的规则:

- 所有的泛型方法声明在方法的返回值之前(下一个例子中的)有一个由尖括号分隔开的类型参数区域。()
- 类型参数能被用来声明返回类型和作为传给泛型方法的参数类型的占位符，就是为人所知的真正的类型参数。
- 一个泛型方法的主体像其他方法声明一样。注意到类型参数仅能代表引用类型，而不是原始类型(就像 int,double 和 char)。

示例

以下的例子说明了我们可以通过一个通用的方法来打印不同类型的数组:

```
public class GenericMethodTest
{
  // generic method printArray
  public static < E > void printArray( E[] inputArray )
  {
     // Display array elements
       for ( E element : inputArray ){
          System.out.printf( "%s ", element );
       }
       System.out.println();
   }
```

```
   public static void main( String args[] )
   {
      // Create arrays of Integer, Double and Character
      Integer[] intArray = { 1, 2, 3, 4, 5 };
      Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
      Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

      System.out.println( "Array integerArray contains:" );
      printArray( intArray  ); // pass an Integer array

      System.out.println( "\nArray doubleArray contains:" );
      printArray( doubleArray ); // pass a Double array

      System.out.println( "\nArray characterArray contains:" );
      printArray( charArray ); // pass a Character array
   }
}
```

这将产生以下结果:

```
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
```

受限的类型参数

有时候你想要限制允许传给一个类型参数的类型种类。例如，操作数的方法可能仅仅想要接受 Number 的实例或者它的子类。这就是有限的类型参数。

声明一个有限的类型参数，列举类型参数的名字，后面跟着扩展关键字和它的上界。

示例

下面的例子说明了如何使用扩展在一定意义上意味着“继承”(在类中)或是在“实现”(在接口中)。这个例子是返回三个 Comparable 对象中最大的泛型方法:

```
public class MaximumTest
{
```

```
  // determines the largest of three Comparable objects
  public static <T extends Comparable<T>> T maximum(T x, T y, T z)
  {
    T max = x; // assume x is initially the largest
    if ( y.compareTo( max ) > 0 ){
      max = y; // y is the largest so far
    }
    if ( z.compareTo( max ) > 0 ){
      max = z; // z is the largest now
    }
    return max; // returns the largest object
  }
  public static void main( String args[] )
  {
    System.out.printf( "Max of %d, %d and %d is %d\n\n",
              3, 4, 5, maximum( 3, 4, 5 ) );

    System.out.printf( "Maxm of %.1f,%.1f and %.1f is %.1f\n\n",
              6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );

    System.out.printf( "Max of %s, %s and %s is %s\n","pear",
      "apple", "orange", maximum( "pear", "apple", "orange" ) );
  }
}
```

这将会产生以下结果:

```
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

泛型类的声明看起来像一个非泛型类的声明，除了类名后跟着一个类型参数区域。

与泛型方法一样，一个泛型类的类型参数区域可以拥有一个或者更多的由逗号分隔的类型参数。这些类被称为参数化的类或是参数化的类型因为他们接受一个或更多的参数。

示例

下面的例子说明了我们怎样定义一个泛型类:

```
public class Box<T> {

  private T t;

  public void add(T t) {
    this.t = t;
  }

  public T get() {
    return t;
  }

  public static void main(String[] args) {
     Box<Integer> integerBox = new Box<Integer>();
     Box<String> stringBox = new Box<String>();

     integerBox.add(new Integer(10));
     stringBox.add(new String("Hello World"));

     System.out.printf("Integer Value :%d\n\n", integerBox.get());
     System.out.printf("String Value :%s\n", stringBox.get());
  }
}
```

这将产生以下结果:

```
Integer Value :10

String Value :Hello World
```

Java 序列化

Java 提供了一种机制，叫做对象序列化，这里对象被描述成一系列包括对象的数据以及有关对象的类型和在对象中存储的数据的类型的字节。

在一个序列化的对象被写进文件之后，它能在文件中被读出并被反序列化为类型信息和表示对象的字节，并且它的数据可以被用来重新创建在内存中的对象。

最让人印象深刻的是整个过程是 JVM 独立的，意味着一个对象能在一个平台上序列化，并能在在一个完全不同的平台上被反序列化。

`ObjectInputStream` 和 `ObjectOutputStream` 类是包含序列化和反序列化对象的方法的流。

ObjectOutputStream 类含有许多写各种各样数据类型的写方法，但是其中一个方法尤其突出：

```
public final void writeObject(Object x) throws IOException
```

上述的方法序列化了一个对象并将它送入输出流。相同的，ObjectInputStream 类包含以下反序列化对象的方法：

```
public final Object readObject() throws IOException,
                        ClassNotFoundException
```

这个方法检索流之外的下一个对象并且反序列化之。返回值是对象，所以你需要将它转换成正确的数据类型。

为了论证序列化在 java 中是如何工作的，我将使用我们之前在书中讨论过的 Employee 类。假设我们有以下的实现 Serializable 的 Employee 类：

```
public class Employee implements java.io.Serializable
{
   public String name;
   public String address;
   public transient int SSN;
   public int number;
   public void mailCheck()
   {
      System.out.println("Mailing a check to " + name
                        + " " + address);
   }
}
```

注意到为使一个类被成功序列化，两个条件必须被满足：

– 类必须实现 java.io.Serializable 类。

– 类中所有的字段必须被序列化。如果一个字段没有被序列化，它必须被标记为瞬态的。

如果你好奇地想知道 Java 标准类是否是可序列化的，可以查看下类的文档。测试是简单的：如果类实现了 java.io.Serializable，那它就是可序列化的；否则，它就不是。

序列化一个对象

ObjectOutputStream 类被用来序列化一个对象。下面的 SerializeDemo 程序实例化了一个 Employee 对象并且将它在一个文件中序列化。

当程序被执行完毕后，一个名为 employee.ser 的文件就被创建了。程序不生成任何输出，但是研究代码并试图确定程序是在做什么。

注释： 当序列化一个对象到一个文件，在 java 中标准的规定是给予文件一个 .ser 的扩展名。

```
import java.io.*;

public class SerializeDemo
{
  public static void main(String [] args)
  {
    Employee e = new Employee();
    e.name = "Reyan Ali";
    e.address = "Phokka Kuan, Ambehta Peer";
    e.SSN = 11122333;
    e.number = 101;
    try
    {
      FileOutputStream fileOut =
      new FileOutputStream("/tmp/employee.ser");
      ObjectOutputStream out = new ObjectOutputStream(fileOut);
      out.writeObject(e);
      out.close();
      fileOut.close();
      System.out.printf("Serialized data is saved in /tmp/employee.ser");
    }catch(IOException i)
    {
       i.printStackTrace();
    }
```

```
  }
}
```

反序列化一个对象

下面的 DeserializeDemo 程序反序列化了一个在 SerializeDemo 对象中被创建的 Employee 对象。研究这个程序并且试图确定它的输出：

```
import java.io.*;
public class DeserializeDemo
{
  public static void main(String [] args)
  {
    Employee e = null;
    try
    {
      FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");
      ObjectInputStream in = new ObjectInputStream(fileIn);
      e = (Employee) in.readObject();
      in.close();
      fileIn.close();
    }catch(IOException i)
    {
      i.printStackTrace();
      return;
    }catch(ClassNotFoundException c)
    {
      System.out.println("Employee class not found");
      c.printStackTrace();
      return;
    }
    System.out.println("Deserialized Employee...");
    System.out.println("Name: " + e.name);
    System.out.println("Address: " + e.address);
    System.out.println("SSN: " + e.SSN);
    System.out.println("Number: " + e.number);
  }
}
```

这将产生以下结果：

```
Deserialized Employee...
Name: Reyan Ali
Address:Phokka Kuan, Ambehta Peer
```

```
SSN: 0
Number:101
```

下面是一些需要注意的要点：

- try/catch 块试图抓住 readObject() 方法声明的 ClassNotFoundException。为了使一个 JVM 能反序列化一个对象，它必须能找到类的字节代码。如果 JVM 在一个对象的反序列化过程中不能找到一个类，它将抛出 ClassNotFoundException。
- 注意 readObject() 的返回值将被强制类型转换为 Employee 的引用。
- 当对象被序列化时 SSN 字段的值是 11122333，但是因为字段是短暂的，值没有送入输出流。反序列化的 Employee 对象的 SSN 字段值是 0。

Java 网络编程

术语网络编程指编写跨多种设备（电脑）执行的，设备使用一个网络互相连接的程序。

J2SE API 的 java.net 包包含了一个类的集合和提供底层通信细节的接口，允许你编写专注解决即将到来的问题的程序。

java.net 包为两种常用的网络协议提供支持：

- **TCP：** TCP 代表传输控制协议，允许两个应用程序间的可靠通信。TCP 通常在因特网协议上被使用，这被称为 TCP/IP。
- **UDP：** UDP 代表用户数据报协议，一个无连接的允许应用程序间数据包传输的协议。

本教程给以下两个主题提供更好的理解：

- **套接字编程：** 这是在网络中最广泛使用的概念并且被很详细地解释。
- **URL 处理：** 这将被个别地解释。点击这里来学习 java 语言中的 URL 处理 (http://www.tutorialspoint.com/java/java_url_processing.htm) 。

套接字编程

套接字利用TCP在两台电脑间提供通信机制。一个客户端程序在通信最后创建一个套接字并努力连接服务器套接字。

当连接建立时，服务器在通信结束时创建一个套接字对象。客户端和服务器现在可以通过从套接字读或者写来交流。

java.net.Socket 类代表一个套接字，而且 java.net.ServerSocket 类为服务器程序提供了一种机制来监听客户端并和它们建立连接。

以下步骤发生在两台电脑使用套接字建立TCP连接时：

- 服务器实例化一个 ServerSocket 对象，指示通信将产生在哪个端口号；
- 服务器调用 ServerSocket 类的 accept() 方法。这个方法等待直到一个客户端在给定的端口上连接到服务器；
- 在服务器等待后，一个客户端实例化一个 Socket 对象，指定服务器名称和连接的端口号；

- Socket 类的构造函数努力将客户端连接到指定的服务器和端口号。如果通信建立，客户端现在就拥有了一个能和服务器通信的 Socket 对象；
- 在服务器端，accept() 方法在服务器上返回一个连接到客户端套接字的新的套接字。

在连接建立后，通信可以使用 I/O 流产生。每个套接字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，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，所以数据可以在两个流同时发送。由以下提供完整的方法的类来实现套接字。

ServerSocket 类方法

java.net.ServerSocket 类被服务器应用程序使用来获得一个端口和监听客户端请求。

ServerSocket 类有四个构造函数：

SN	方法描述
1	public ServerSocket(int port) throws IOException 尝试创建一个连接到指定端口的服务器套接字。如果端口已经连接到另一个应用程序那么将产生一个异常。
2	public ServerSocket(int port, int backlog) throws IOException 和前一个构造函数相同，backlog 参数指定了在等待队列中有多少传入的客户端要存储
3	public ServerSocket(int port, int backlog, InetAddress address) throws IOException 和前一个构造函数相同，InetAddress 参数指定了本地捆绑的IP地址。InetAddress 用于有多个 IP 地址的服务器，允许服务器指定它的哪个 IP 地址 来接收客户端请求。
4	public ServerSocket() throws IOException 创建一个为绑定服务器的套接字。当使用这个构造函数时，当你准备好绑定服务器套接字时使用 bind()方法。

如果 ServerSocket 的构造函数不抛出一个异常，这意味着你的应用程序已经成功地绑定到特定的端口并且准备好客户端的请求了。

这里是一些 ServerSocket 类的常见方法：

SN	方法描述
1	public int getLocalPort() 返回服务器套接字正在监听的端口。如果你在构造函数中传入 0 作为端口号这个方法会是有用的，它会让服务器为你找一个端口。
2	public Socket accept() throws IOException 等待一个即将到来的客户端。这个方法直到或者一个客户端连接到特定端口服务器，或者套接字到时为止时阻塞，假设超时的值已经使用 setSoTimeout() 方法设置了。否则，这个方法将无限期阻塞。
3	public void setSoTimeout(int timeout) 把超时的值设为服务器套接字在 accept() 内等待客户端的时间。

SN	方法描述
4	public void bind(SocketAddress host, int backlog) 将套接字绑定在特定的服务器和 SocketAddress 对象的端口上。如果你使用无参数的构造函数实例化一个 ServerSocket 对象，使用这个方法。

当 ServerSocket 调用 accept() 方法直到一个客户端连接才返回。在一个客户端确实连接后，ServerSocket 在一个未指定的端口上创建一个新的套接字，并返回一个新套接字的引用。一个 TCP 连接现在就存在于客户端和服务器间了，通信就可以开始了。

Socket 类方法

java.net.Socket 类方法代表客户端和服务器都用来互相通信的套接字。客户端通过实例化而拥有一个 Socket 对象，然而服务器从 accept() 方法的返回值获得一个 Socket 对象。

Socket 类有5个客户端用来连接到服务器的构造函数：

SN	方法描述
1	public Socket(String host, int port) throws UnknownHostException, IOException. 这个方法努力连接到特定端口指定的服务器。如果这个构造函数不抛出一个异常，连接就是成功的并且客户端将会连接到服务器。
2	public Socket(InetAddress host, int port) throws IOException 这个方法和之前的构造函数相同，除了主机由一个 InetAddress 对象表示。
3	public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException. 连接到指定的主机和端口，在指定地址和端口上的本地主机创建一个套接字。
4	public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException. 这个方法和前一个构造函数相同，除了主机由一个 InetAddress 对象而不是一个 String 表示。
5	public Socket() 创建一个不连接的套接字。使用 connect() 方法来连接这个套接字到服务器。

当套接字构造函数返回时，它并不简单地实例化一个 Socket 对象，它实际上试图连接到指定的服务器和端口。

在 Socket 类中一些有趣的方法列举在此。注意客户端和服务器都有一个 Socket 对象，所以这些方法都能被客户端或者服务器调用。

SN	方法描述
1	public void connect(SocketAddress host, int timeout) throws IOException 这个方法将套接字连接到特定的主机。这个方法仅当你使用无参数的构造函数实例化 Socket 时才需要。
2	public InetAddress getInetAddress() 这个方法返回其套接字连接的其他电脑的地址。
3	public int getPort() 返回远端的机器上套接字绑定的端口。

SN	方法描述
4	public int getLocalPort() 返回本地机器上套接字绑定的端口。
5	public SocketAddress getRemoteSocketAddress() 返回远程套接字的地址
6	public InputStream getInputStream() throws IOException 返回套接字的输入流。输入流连接到远程套接字的输出流。
7	public OutputStream getOutputStream() throws IOException 返回套接字的输出流。输出流连接到远程套接字的输入流。
8	public void close() throws IOException 关闭套接字，这使得这个 Socket 对象不再能够连接到任何服务器。

InetAddress 类方法

这个类表示一个网络协议（IP）的地址。这些是在做套接字编程时将会用到的有用的方法：

SN	方法描述
1	static InetAddress getByAddress(byte[] addr) 考虑到原始的 IP 地址，返回一个 InetAddress 对象。
2	static InetAddress getByAddress(String host, byte[] addr) 基于提供的主机名和 IP 地址创建一个 InetAddress。
3	static InetAddress getByName(String host) 考虑到主机名，决定一个主机的 IP 地址。
4	String getHostAddress() 用文本表示返回的 IP 地址字符串。
5	String getHostName() 获得这个 IP 地址的主机名。
6	static InetAddress InetAddress getLocalHost() 返回本地主机。
7	String toString() 将 IP 地址转换为字符串。

套接字客户端示例

下列的 GreetingClient 是一个利用一个套接字连接到服务器的客户端程序，它发送一个问候，并等待响应。

```
// File Name GreetingClient.java

import java.net.*;
import java.io.*;

public class GreetingClient
```

```
{
  public static void main(String [] args)
  {
    String serverName = args[0];
    int port = Integer.parseInt(args[1]);
    try
    {
      System.out.println("Connecting to " + serverName
                         + " on port " + port);
      Socket client = new Socket(serverName, port);
      System.out.println("Just connected to "
                  + client.getRemoteSocketAddress());
      OutputStream outToServer = client.getOutputStream();
      DataOutputStream out =
                   new DataOutputStream(outToServer);

      out.writeUTF("Hello from "
                  + client.getLocalSocketAddress());
      InputStream inFromServer = client.getInputStream();
      DataInputStream in =
                    new DataInputStream(inFromServer);
      System.out.println("Server says " + in.readUTF());
      client.close();
    }catch(IOException e)
    {
      e.printStackTrace();
    }
  }
}
```

套接字服务器示例

下列的 GreetingServer 程序是一个使用 Socket 类来监听由命令行参数指定的端口号上的客户端的服务器应用程序的一个例子：

```
// File Name GreetingServer.java

import java.net.*;
import java.io.*;

public class GreetingServer extends Thread
{
  private ServerSocket serverSocket;
```

```
public GreetingServer(int port) throws IOException
{
  serverSocket = new ServerSocket(port);
  serverSocket.setSoTimeout(10000);
}

public void run()
{
  while(true)
  {
    try
    {
      System.out.println("Waiting for client on port " +
      serverSocket.getLocalPort() + "...");
      Socket server = serverSocket.accept();
      System.out.println("Just connected to "
           + server.getRemoteSocketAddress());
      DataInputStream in =
           new DataInputStream(server.getInputStream());
      System.out.println(in.readUTF());
      DataOutputStream out =
          new DataOutputStream(server.getOutputStream());
      out.writeUTF("Thank you for connecting to "
        + server.getLocalSocketAddress() + "\nGoodbye!");
      server.close();
    }catch(SocketTimeoutException s)
    {
      System.out.println("Socket timed out!");
      break;
    }catch(IOException e)
    {
      e.printStackTrace();
      break;
    }
  }
}
public static void main(String [] args)
{
  int port = Integer.parseInt(args[0]);
  try
  {
    Thread t = new GreetingServer(port);
    t.start();
  }catch(IOException e)
  {
```

```
        e.printStackTrace();
      }
    }
}
```

编译客户端和服务器然后像这样开始服务器：

```
$ java GreetingServer 6066
Waiting for client on port 6066...
```

像这样检查客户端程序：

```
$ java GreetingClient localhost 6066
Connecting to localhost on port 6066
Just connected to localhost/127.0.0.1:6066
Server says Thank you for connecting to /127.0.0.1:6066
Goodbye!
```

Java 发送邮件

用你的 Java 应用程序来发送一封电子邮件是足够简单的但是开始时你应该在你的机器上安装有 JavaMail API 和 Java Activation Framework (JAF)。

- 你可以从 Java 的标准企业网站上下载最新的 JavaMail (1.2 版本) (http://www.oracle.com/technetwork/java/index.html) 版本。
- 你可以从 Java 的标准企业网站上下载最新的 JAF (1.1.1 版本) (http://www.oracle.com/technetwork/java/index.html) 版本。

下载并解压这些文件，在新创建的顶级目录中你将找到许多应用程序的 jar 文件。你需要在你的 CLASSPATH 中添加 mail.jar 和 activation.jar 文件。

发送一封简单的电子邮件

这是从你的机器中发送一封简单的电子邮件的例子。这里假设你的本地主机连接到了因特网并且能够发送一封电子邮件。

```
// File Name SendEmail.java

import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendEmail
{
  public static void main(String [] args)
  {
    // Recipient's email ID needs to be mentioned.
    String to = "abcd@gmail.com";

    // Sender's email ID needs to be mentioned
    String from = "web@gmail.com";

    // Assuming you are sending email from localhost
    String host = "localhost";

    // Get system properties
```

```
    Properties properties = System.getProperties();

    // Setup mail server
    properties.setProperty("mail.smtp.host", host);

    // Get the default Session object.
    Session session = Session.getDefaultInstance(properties);

    try{
      // Create a default MimeMessage object.
      MimeMessage message = new MimeMessage(session);

      // Set From: header field of the header.
      message.setFrom(new InternetAddress(from));

      // Set To: header field of the header.
      message.addRecipient(Message.RecipientType.TO,
                           new InternetAddress(to));

      // Set Subject: header field
      message.setSubject("This is the Subject Line!");

      // Now set the actual message
      message.setText("This is actual message");

      // Send message
      Transport.send(message);
      System.out.println("Sent message successfully....");
    }catch (MessagingException mex) {
      mex.printStackTrace();
    }
  }
}
```

编译并运行这个程序来发送一封简单的电子邮件：

```
$ java SendEmail
Sent message successfully....
```

如果你想要给多个收信者发送一封电子邮件，那么以下的方法将被用来发送给指定的多个电子邮件 ID：

```
void addRecipients(Message.RecipientType type,
                   Address[] addresses)
throws MessagingException
```

这是参数的描述：

- type： 这将被设置为 TO，CC 或者 BCC。这里 CC 表示副本，BCC 表示 Black Carbon Copy。例如 Message.RecipientType.TO。
- addresses: 这是电子邮件 ID 的数组。当指定电子邮件 ID 时，你需要使用 InternetAddress()。

发送一封 HTML 电子邮件

这是从你的机器发送一封 HTML 电子邮件的例子。这里假设你的本地主机连接到了因特网并且能够发送一封电子邮件。

这个例子和前一个非常相似，除了在这我们用 setContent（ ）方法来设置第二个参数为 "text/html" 以指定 HTML 内容被包括在消息中的内容。

使用这个例子，你可以你喜欢的任何大的 HTML 内容。

```
// File Name SendHTMLEmail.java

import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendHTMLEmail
{
  public static void main(String [] args)
  {

    // Recipient's email ID needs to be mentioned.
    String to = "abcd@gmail.com";

    // Sender's email ID needs to be mentioned
    String from = "web@gmail.com";

    // Assuming you are sending email from localhost
    String host = "localhost";

    // Get system properties
    Properties properties = System.getProperties();

    // Setup mail server
    properties.setProperty("mail.smtp.host", host);
```

```
    // Get the default Session object.
    Session session = Session.getDefaultInstance(properties);

    try{
      // Create a default MimeMessage object.
      MimeMessage message = new MimeMessage(session);

      // Set From: header field of the header.
      message.setFrom(new InternetAddress(from));

      // Set To: header field of the header.
      message.addRecipient(Message.RecipientType.TO,
                           new InternetAddress(to));

      // Set Subject: header field
      message.setSubject("This is the Subject Line!");

      // Send the actual HTML message, as big as you like
      message.setContent("<h1>This is actual message</h1>",
                         "text/html" );

      // Send message
      Transport.send(message);
      System.out.println("Sent message successfully....");
    }catch (MessagingException mex) {
      mex.printStackTrace();
    }
  }
}
```

编译并运行这个程序来发送一封 HTML 的电子邮件：

```
$ java SendHTMLEmail
Sent message successfully....
```

发送电子邮件中的附件

这是一个从你的机器中发送一封带有附件的电子邮件的例子。这里假设你的本地主机连接到了因特网并且能够发送一封电子邮件。

```
// File Name SendFileEmail.java

import java.util.*;
import javax.mail.*;
```

```
import javax.mail.internet.*;
import javax.activation.*;

public class SendFileEmail
{
  public static void main(String [] args)
  {

    // Recipient's email ID needs to be mentioned.
    String to = "abcd@gmail.com";

    // Sender's email ID needs to be mentioned
    String from = "web@gmail.com";

    // Assuming you are sending email from localhost
    String host = "localhost";

    // Get system properties
    Properties properties = System.getProperties();

    // Setup mail server
    properties.setProperty("mail.smtp.host", host);

    // Get the default Session object.
    Session session = Session.getDefaultInstance(properties);

    try{
      // Create a default MimeMessage object.
      MimeMessage message = new MimeMessage(session);

      // Set From: header field of the header.
      message.setFrom(new InternetAddress(from));

      // Set To: header field of the header.
      message.addRecipient(Message.RecipientType.TO,
                           new InternetAddress(to));

      // Set Subject: header field
      message.setSubject("This is the Subject Line!");

      // Create the message part
      BodyPart messageBodyPart = new MimeBodyPart();

      // Fill the message
      messageBodyPart.setText("This is message body");
```

```
      // Create a multipar message
      Multipart multipart = new MimeMultipart();

      // Set text message part
      multipart.addBodyPart(messageBodyPart);

      // Part two is attachment
      messageBodyPart = new MimeBodyPart();
      String filename = "file.txt";
      DataSource source = new FileDataSource(filename);
      messageBodyPart.setDataHandler(new DataHandler(source));
      messageBodyPart.setFileName(filename);
      multipart.addBodyPart(messageBodyPart);

      // Send the complete message parts
      message.setContent(multipart );

      // Send message
      Transport.send(message);
      System.out.println("Sent message successfully....");
    }catch (MessagingException mex) {
      mex.printStackTrace();
    }
  }
}
```

编译并运行这个程序来发送一封 HTML 电子邮件：

```
$ java SendFileEmail
Sent message successfully....
```

用户身份认证部分

如果为了身份认证的目的需要给电子邮件服务器提供用户 ID 和密码，你可以像这样设置这些属性：

```
props.setProperty("mail.user", "myuser");
props.setProperty("mail.password", "mypwd");
```

电子邮件发送机制的剩余部分和上述解释的一样。

Java 多线程

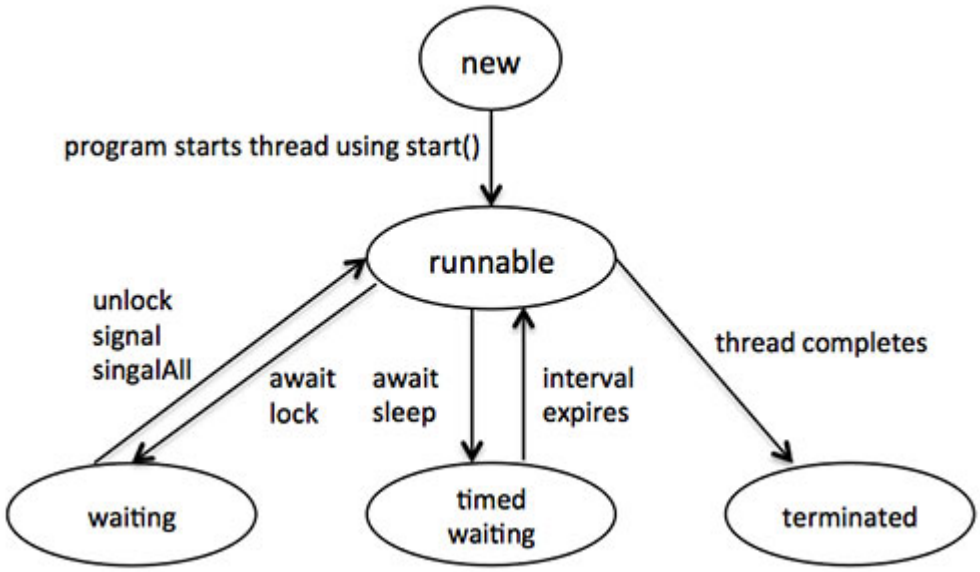
Java 是一种多线程编程语言，这意味着我们能用 java 开发多线程程序。一个多线程程序包含两个或更多的能并行运行的部分，并且每一部分能最优利用可用资源，尤其是当你的计算机有多个 CPU 时，同时解决不同的任务。

多任务处理表示多个过程共同处理像CPU这样共享的资源时。多线程将多任务处理的思想扩展到应用程序，你可以将一个单独的应用中的特定的操作细分为单独的线程。每一个线程能并行地运行。操作系统不仅在不同的应用程序中划分处理时间，而且在一个应用程序中的每一个线程中也是如此。

多线程让你可以用一种多个活动能并行地在同一个程序中进行的方法编写程序。

线程的生命周期

一个线程在它的生命周期中通过不同的阶段。例如，一个线程生成，开始，运行，然后死亡。下列的图表展示了一个线程的完整的生命周期。



图片 3.11

上述提到的阶段在这解释：

- new： 一个新的线程以一个新的状态开始了它的生命周期。它始终留在这个状态中直到程序开始线程。它也被称为一个生成的进程。

- Runnable： 在一个新生成的线程开始后，这个线程变得可运行。在这个状态下的线程被认为正在执行任务。
- Waiting： 有时候，一个线程当它等待另一个线程工作时跃迁到等待状态。一个线程仅当另一个线程发信号给等待的线程继续执行才跃迁回可运行转态。
- Timed waiting： 一个可运行的线程能进入定时等待状态等待指定的时间间隔。在这种转态下的线程当时间间隔死亡或者当它所等待的活动发生时跃迁回可运行状态。
- Terminated： 一个可运行的线程，当它完成了它的任务后进入 terminated 状态，否则它就结束。

线程优先级

每一个 Java 线程有一个帮助操作系统决定线程调度顺序的优先级。

Java 线程的优先级在 MIN_PRIORITY（一个为 1 的常数）和 MAX_PRIORITY（一个为 10 的常数)的范围内。缺省状况下，每一个线程给予优先级 NORM_PRIORITY（一个为 5 的常数）。

拥有更高优先级的线程对于一个程序来说更加重要，应当在低优先级的线程前被分配处理时间。然而，线程优先级不能保证线程执行和所依赖的平台。

通过实现 Runnable 接口创建线程

如果你的类想要作为一个线程被执行，那么你可以通过实现 Runnable 接口来到达这个目的。你将需要遵从三个基本步骤：

步骤一：

作为第一步你需要实现由 Runnable 接口提供的 run() 方法。这个方法为线程提供了进入点并且你将把你完全的业务逻辑放入方法中。下列是简单的 run() 方法语法：

```
public void run( )
```

步骤二：

在第二步你将使用以下的构造函数实例化一个 Thread 对象:

```
Thread(Runnable threadObj, String threadName);
```

threadObj 是实现 Runnable 接口的类的一个实例，threadName 是给新线程的名字。

步骤三：

一旦 Thread 对象被创建，你可以通过调用 run() 方法的 start() 方法来开始它。以下是 start() 方法的简单语法：

```
void start( );
```

示例

这里是一个创建一个新线程并使之运行的例子：

```
class RunnableDemo implements Runnable {
  private Thread t;
  private String threadName;

  RunnableDemo( String name){
     threadName = name;
     System.out.println("Creating " +  threadName );
  }
  public void run() {
    System.out.println("Running " +  threadName );
    try {
      for(int i = 4; i > 0; i--) {
        System.out.println("Thread: " + threadName + ", " + i);
        // Let the thread sleep for a while.
        Thread.sleep(50);
      }
   } catch (InterruptedException e) {
      System.out.println("Thread " +  threadName + " interrupted.");
   }
   System.out.println("Thread " +  threadName + " exiting.");
  }

  public void start ()
  {
    System.out.println("Starting " +  threadName );
    if (t == null)
    {
      t = new Thread (this, threadName);
      t.start ();
    }
  }

}

public class TestThread {
```

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

    RunnableDemo R1 = new RunnableDemo( "Thread-1");
    R1.start();

    RunnableDemo R2 = new RunnableDemo( "Thread-2");
    R2.start();
  }
}
```

这将产生以下结果：

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

通过继承 Thread 类来创建线程

第二个创建线程的方法是创建一个通过使用以下两个简单步骤继承 Thread 类的新的类。这个方法在解决 Thread 类中使用可行方法创建的多线程的问题上提供了更多的灵活性。

步骤一：

你将需要覆写 Thread 类中可用的 run()方法。这个方法为线程提供入口并且你将把你完全的业务逻辑放入方法中。以下是 run()方法的简单的语法。

```
public void run( )
```

步骤二：

一旦 Thread 对象被创建，你可以通过调用 run()方法的 start() 方法来开始它。以下是 start() 方法的简单语法：

```
void start( );
```

示例

这是前面的重写继承 Thread 的程序：

```
class ThreadDemo extends Thread {
   private Thread t;
   private String threadName;

   ThreadDemo( String name){
      threadName = name;
      System.out.println("Creating " +  threadName );
   }
   public void run() {
      System.out.println("Running " +  threadName );
      try {
         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
      } catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
      }
      System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start ()
   {
      System.out.println("Starting " +  threadName );
      if (t == null)
      {
         t = new Thread (this, threadName);
         t.start ();
      }
   }

}

public class TestThread {
```

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

    ThreadDemo T1 = new ThreadDemo( "Thread-1");
    T1.start();

    ThreadDemo T2 = new ThreadDemo( "Thread-2");
    T2.start();
  }
}
```

这将会产生以下结果：

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Thread 方法

以下是在 Thread 类中可以获得的重要方法的列表。

SN	接口描述
1	Methods with Description 在一个独立的执行路径中开始一个线程，然后在这个 Thread 对象上调用 run() 方法。
2	public void run() 如果这个 Thread 对象是使用一个单独的 Runnable 目标实例化的,run()方法被 Runnable 对象调用。
3	public final void setName(String name) 改变 Thread 对象的名字。也有一个 getName()方法来检索名字。
4	public final void setPriority(int priority) 设置 Thread 对象的优先级。可能的值在 1 到 10 之间。
5	public final void setDaemon(boolean on) 一个真值将这个线程标志为守护进程。

SN	接口描述
6	public final void join(long millisec) 当前进程在第二个线程上调用这个方法，使得当前进程阻塞直到第二个线程终结或者指定的毫秒数过去。
7	public void interrupt() 中断这个进程，如果由于任何原因而阻塞，使得它也继续执行。
8	public final boolean isAlive() 如果线程是活的，返回真值，可在线程已经开始但在运行到完成之前的任何时间。

以前的方法是被一个特殊的 Thread 对象调用的。以下在 Thread 类中的方法是静态的。调用静态方法会在当前运行的线程上执行操作。

SN	接口描述
1	public static void yield() 使得当前正在运行的线程让步于任何其他相同优先级的正在等待调度的线程。
2	public static void sleep(long millisec) 使当前运行的线程阻塞至少指定的毫秒数。
3	public static boolean holdsLock(Object x) 如果当前线程持有给定对象的锁，返回真值。
4	public static Thread currentThread() 返回对当前运行的线程的引用，也就是调用这个方法的线程。
5	public static void dumpStack() 为当前运行的线程打印堆栈跟踪，这在当调试多线程应用程序时是有用的。

示例

以下 ThreadClassDemo 程序展示了一些 Thread 类的方法。考虑实现 Runnable 的类 DisplayMessage:

```
// File Name : DisplayMessage.java
// Create a thread to implement Runnable
public class DisplayMessage implements Runnable
{
   private String message;
   public DisplayMessage(String message)
   {
      this.message = message;
   }
   public void run()
   {
      while(true)
      {
         System.out.println(message);
      }
   }
}
```

以下是继承 Thread 类的另一个类：

```
// File Name : GuessANumber.java
// Create a thread to extentd Thread
public class GuessANumber extends Thread
{
   private int number;
   public GuessANumber(int number)
   {
      this.number = number;
   }
   public void run()
   {
      int counter = 0;
      int guess = 0;
      do
      {
         guess = (int) (Math.random() * 100 + 1);
         System.out.println(this.getName()
                  + " guesses " + guess);
         counter++;
      }while(guess != number);
      System.out.println("** Correct! " + this.getName()
                  + " in " + counter + " guesses.**");
   }
}
```

以下是使用上面定义的类的主程序:

```
// File Name : ThreadClassDemo.java
public class ThreadClassDemo
{
   public static void main(String [] args)
   {
      Runnable hello = new DisplayMessage("Hello");
      Thread thread1 = new Thread(hello);
      thread1.setDaemon(true);
      thread1.setName("hello");
      System.out.println("Starting hello thread...");
      thread1.start();

      Runnable bye = new DisplayMessage("Goodbye");
      Thread thread2 = new Thread(bye);
      thread2.setPriority(Thread.MIN_PRIORITY);
      thread2.setDaemon(true);
```

```
    System.out.println("Starting goodbye thread...");
    thread2.start();

    System.out.println("Starting thread3...");
    Thread thread3 = new GuessANumber(27);
    thread3.start();
    try
    {
      thread3.join();
    }catch(InterruptedException e)
    {
      System.out.println("Thread interrupted.");
    }
    System.out.println("Starting thread4...");
    Thread thread4 = new GuessANumber(75);

    thread4.start();
    System.out.println("main() is ending...");
  }
}
```

这将产生以下结果。你可以一次一次地尝试这个例子并且你将每次得到不同的结果。

```
Starting hello thread...
Starting goodbye thread...
Hello
Hello
Hello
Hello
Hello
Hello
Goodbye
Goodbye
Goodbye
Goodbye
Goodbye
.......
```

Java Applet 基础

一个 Applet 是一个运行在网页浏览器上的 Java 程序。一个 Applet 可以使一个全功能的 Java 应用程序因为它在处理上拥有整个 Java API。

在一个 Applet 和一个独立的 Java 应用程序间有一些重要的不同，包括以下方面：

- 一个 Applet 是继承 java.applet.Applet 类的一个 Java 类。
- main() 方法不在 Applet 上被调用，并且一个 applet 类将不定义 main()。
- Applet 程序被设计嵌入到 HTML 页面中。
- 当一个用户查看一个带有 applet 的 HTML 页面，applet 的代码将被下载到用户的机器中。
- 需要 JVM 来查看一个 applet。JVM 可以是一个网页浏览器的一个插件，也可以是单独的运行环境。
- 用户机器上的 JVM 创建了 applet 类的实例并且在 applet 的生命周期中调用不同的方法。
- Applets 有网页浏览器实施的严格的安全规则。一个 applet 程序的安全性常常被称作沙箱安全，这是将 applet 比作一个正在沙箱中的，必须遵守许多规则的孩子。
- 其他 applet 需要的类可以在一个单独的 Java Archive(JAR) 文件中下载。

一个 Applet 的生命周期

Applet 类中的四个方法给了你构建 applet 程序时的框架：
- init： 这个方法适用于你的 applet 程序所需要的任何初始化。它在 applet 标记中的参数标签被处理后被调用。
- start： 这个方法在浏览器调用 init 方法后被自动调用。它也在无论何时使用者在去其他页面后返回到包含 applet 的页面时被调用。
- stop： 这个方法在使用者离开有 applet 所在的页面时被自动调用。因此，它在同一个 applet 中能被重复调用。 - destroy： 这个方法仅当浏览器正常关闭时被自动调用。因为 applet 程序是生存在 HTML 页面上的，你不应该在使用者离开有 applet 的网页后留下资源。
- paint： 在 start() 方法之后被立即调用，或是在 applet 需要在浏览器上重现它自身的任何时候。paint() 方法实际上是继承自 java.awt。

一个 “Hello,World” Applet

以下是一个简单的叫做 HelloWorldApplet.java 的 applet 程序：

```
import java.applet.*;
import java.awt.*;

public class HelloWorldApplet extends Applet
{
  public void paint (Graphics g)
  {
    g.drawString ("Hello World", 25, 50);
  }
}
```

这些引入的语句把类带入我们的 applet 类的范围内：

– java.applet.Applet. – java.awt.Graphics.

没有那些引入的语句，Java 编译器将不会认出 applet 类所指的 Applet 和 Graphics 类。

Applet 类

每一个 applet 类都是 java.applet.Applet 类的延伸。基本的 Applet 类提供了一个派生的 Applet 类用来调用从浏览器获取信息和服务的方法。

这还包括了做以下事情的方法：

– 获得 applet 参数 – 获得包含 applet 的 HTML 文件的网络地址 – 获得 applet 类目录的网络地址 – 在浏览器中打印状态信息 – 获取一个图像 – 获取一段音频剪辑 – 播放一段音频剪辑 – 调整 applet 的大小

此外，Applet 类通过观察器和浏览器获得有关 applet 的信息和控制 applet 的执行来提供一个接口。观察者可能： – 需要作者的信息，版本和 applet 的版权 – 需要 applet 识别的参数的描述 – 初始化 applet – 销毁 applet – 开始 applet 的执行 – 停止 applet 的执行

调用一个 Applet 程序

一个 applet 可能被一个 HTML 文件中嵌入的指令文件调用并通过一个 applet 观察器或者支持 Java 的浏览器查看文件。

标签是在一个 HTML 文件中嵌入一个 applet 的基础。以下是调用“Hello,World” applet 的例子：

```
<html>
<title>The Hello, World Applet</title>
<hr>
<applet code="HelloWorldApplet.class" width="320" height="120">
If your browser was Java-enabled, a "Hello, World"
```

```
message would appear here.
</applet>
<hr>
</html>
```

标签的 code 属性是需要的。它指定 Applet 类运行。宽度和高度也是需要来规定 applet 运行的面板的初始大小。applet 指令必须用 标签结束。

如果一个 applet 需要参数，值可能通过在 和 间添加 标签来传给参数。

Java 不支持的浏览器不处理 和 。因此，任何出现在标签之间的，和 applet 无关的，在 Java 不支持的浏览器内都是可见的。

观察器和浏览器在文件的地址寻找编译好的 Java 代码。否则指定，像下面这样使用 标签的 codebase 属性：

```
<applet codebase="http://amrood.com/applets"
code="HelloWorldApplet.class" width="320" height="120">
```

如果一个 applet 常驻在一个包内而不是默认，保存的包必须规定在 code 属性内，使用句点字符（.）来分离包/类成分：

```
<applet code="mypackage.subpackage.TestApplet.class"
        width="320" height="120">
```

获得 Applet 参数

下面的例子展示了如何进行 applet 响应设置文档中指定的参数。这个 applet 展示了黑色和第二种颜色的棋盘式样式。

第二种颜色和每个方格的大小会由文件中的 applet 的参数所指定。

CheckerApplet 在 init() 方法中获得它的参数。它也能在 paint() 方法中获得它的参数。然而，在 applet 一开始就获得值并保存设置，而不是在每次刷新时，是方便有效的。

applet 观察器或者浏览器在每一个 applet 运行时调用 init() 方法。观察器在加载 applet 程序后立即调用 init() 方法。(Applet.init() 实现后不做任何事)覆写默认的实现来插入自定义初始化代码。

Applet.getParameter() 方法获得一个给定参数名称的参数(参数的值总是一个字符串)。如果值是数值的或者其他非字符数据，字符串必须被解析。

以下是 CheckerApplet.java 的构架：

```
import java.applet.*;
import java.awt.*;
public class CheckerApplet extends Applet
{
  int squareSize = 50;// initialized to default size
  public void init () {}
  private void parseSquareSize (String param) {}
  private Color parseColor (String param) {}
  public void paint (Graphics g) {}
}
```

这里是 CheckerApplet 的 init() 和私有的 parseSquareSize() 方法：

```
public void init ()
{
  String squareSizeParam = getParameter ("squareSize");
  parseSquareSize (squareSizeParam);
  String colorParam = getParameter ("color");
  Color fg = parseColor (colorParam);
  setBackground (Color.black);
  setForeground (fg);
}
private void parseSquareSize (String param)
{
  if (param == null) return;
  try {
    squareSize = Integer.parseInt (param);
  }
  catch (Exception e) {
   // Let default value remain
  }
}
```

applet 调用 parseSquareSize() 来解析 squareSize 参数。parseSquareSize() 调用 library 方法 Integer.parseInt(),它解析了一个字符串并返回一个整型。Integer.parseInt() 每当它的参数无效时就抛出一个异常。

因此，parseSquareSize() 捕获一个异常，而不是允许 applet 因为错误的输入失效。

applet 调用 parseColor() 来解析 color 参数成一个 Color 值。parseColor() 进行了一系列的字符串比较来将参数值和一个预定义的颜色进行匹配。你需要实现这些方法来进行 applet 工作。

指定 Applet 参数

以下是一个有 CheckerApplet 嵌入的 HTML 文件的例子。HTML 文件通过 标签的方法指定两个 applet 的参数。

```
<html>
<title>Checkerboard Applet</title>
<hr>
<applet code="CheckerApplet.class" width="480" height="320">
<param name="color" value="blue">
<param name="squaresize" value="30">
</applet>
<hr>
</html>
```

注意： 参数名字不区分大小写

应用程序转换为 Applet

将一个图形化的 Java 应用程序(那就是，一个使用 AWT 并且你能用 Java 程序启动器启动的应用程序)转换为一个可以嵌入到一个网页中的 applet 程序是简单的。

这是将应用程序转换为 applet 的指定步骤。 – 制作一个有正确的标签的 HTML 网页来载入 applet 代码。 – 提供一个 JApplet 类的子类。使这个类公开。否则，applet 不能不载入。 – 消除应用程序中的 main 方法。不要为应用程序构建一个框架窗口。你的应用程序将在浏览器内被展示。 – 将任何初始化代码从框架窗口构造函数移动到 applet 的 init 方法。你不需要精确地构造 applet 对象。浏览器为你初始化了并且调用 init 方法。
– 移除对 setSize 的调用；对 applet 来说，调整宽度和高度的参数在 HTML 中已经做好了。
– 移除对 setDefaultCloseOperation 的调用。一个 applet 不能关闭；它在浏览器退出时消失。
– 如果应用程序调用 setTitle,消除了对方法的调用。Applet 不能拥有标题栏。(当然你可以用 HTML 标题标签来给网页定义自身标签) – 不要调用 setVisible(true)。applet 自动设置了。

事件处理

applet 从 Container 类中继承了一组事件处理的方法。Container 类为解决特殊的时间类型，定义了几种方法，比如 processKeyEvent 和 processMouseEvent，和一个万能的叫做 processEvent 的方法。

为了使一个事件生效，applet 必须覆写正确的事件方法。

```
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import java.applet.Applet;
import java.awt.Graphics;

public class ExampleEventHandling extends Applet
            implements MouseListener {

   StringBuffer strBuffer;

   public void init() {
   addMouseListener(this);
   strBuffer = new StringBuffer();
      addItem("initializing the apple ");
   }

   public void start() {
      addItem("starting the applet ");
   }

   public void stop() {
      addItem("stopping the applet ");
   }

   public void destroy() {
      addItem("unloading the applet");
   }

   void addItem(String word) {
      System.out.println(word);
      strBuffer.append(word);
      repaint();
   }

   public void paint(Graphics g) {
   //Draw a Rectangle around the applet's display area.
      g.drawRect(0, 0,
        getWidth() - 1,
        getHeight() - 1);

   //display the string inside the rectangle.
      g.drawString(strBuffer.toString(), 10, 20);
   }
```

```
    public void mouseEntered(MouseEvent event) {
    }
    public void mouseExited(MouseEvent event) {
    }
    public void mousePressed(MouseEvent event) {
    }
    public void mouseReleased(MouseEvent event) {
    }

    public void mouseClicked(MouseEvent event) {
    addItem("mouse clicked! ");
    }
}
```

现在，让我们像这样调用下面的 applet:

```
<html>
<title>Event Handling</title>
<hr>
<applet code="ExampleEventHandling.class"
width="300" height="300">
</applet>
<hr>
</html>
```

最初，applet 将会展示 “初始化 applet,开始 applet”。然后一旦你点击长方形内的 “鼠标点击” 将会显示。

显示图像

一个 applet 能显示 GIF，JPEG，BMP 和其它格式的图像。为了显示 applet 中的图像，你可以使用在 java.awt.Graphics 类中的 drawImage() 方法。

以下是展示所有步骤来显示图像的例子：

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class ImageDemo extends Applet
{
  private Image image;
  private AppletContext context;
  public void init()
  {
     context = this.getAppletContext();
```

```
    String imageURL = this.getParameter("image");
    if(imageURL == null)
    {
      imageURL = "java.jpg";
    }
    try
    {
      URL url = new URL(this.getDocumentBase(), imageURL);
      image = context.getImage(url);
    }catch(MalformedURLException e)
    {
      e.printStackTrace();
      // Display in browser status bar
      context.showStatus("Could not load image!");
    }
  }
  public void paint(Graphics g)
  {
    context.showStatus("Displaying image");
    g.drawImage(image, 0, 0, 200, 84, null);
    g.drawString("www.javalicense.com", 35, 100);
  }
}
```

现在，让我们像这样调用这个 applet:

```
<html>
<title>The ImageDemo applet</title>
<hr>
<applet code="ImageDemo.class" width="300" height="200">
<param name="image" value="java.jpg">
</applet>
<hr>
</html>
```

播放音频

一个 applet 能播放由 java.applet 包内的 AudioClip 接口表示的音频。AudioClip 接口有三个方法，包括： - public void play()： 从开始时，播放一次音频片段。
- public void loop()： 使音频片段持续重复播放。 - public void stop()： 停止播放音频片段。

为了获得一个 AudioClip 对象，你必须调用 Applet 类的 getAudioClip() 方法。getAudioClip() 方法立刻返回，无论 URL 是否解决一个真正的音频片段。当播放音频片段时它才被下载。

以下是播放音频的所有步骤的一个例子：

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class AudioDemo extends Applet
{
  private AudioClip clip;
  private AppletContext context;
  public void init()
  {
    context = this.getAppletContext();
    String audioURL = this.getParameter("audio");
    if(audioURL == null)
    {
      audioURL = "default.au";
    }
    try
    {
      URL url = new URL(this.getDocumentBase(), audioURL);
      clip = context.getAudioClip(url);
    }catch(MalformedURLException e)
    {
      e.printStackTrace();
      context.showStatus("Could not load audio file!");
    }
  }
  public void start()
  {
    if(clip != null)
    {
      clip.loop();
    }
  }
  public void stop()
  {
    if(clip != null)
    {
      clip.stop();
    }
  }
}
```

现在，让我们调用这个 applet:

```
<html>
<title>The ImageDemo applet</title>
<hr>
<applet code="ImageDemo.class" width="0" height="0">
<param name="audio" value="test.wav">
</applet>
<hr>
</html>
```

你可以使用你电脑的 test.wav 来测试上面的例子。

Java 文件注释

Java 语言支持三种注释形式：

注释	描述
/*text*/	编译器忽略 /* 到 */ 的所有东西
//text	编译器忽略从 // 到一行末尾的所有东西
/** documentation */	这是文档注释并且通常而言它被叫做 doc comment。JDK javadoc 工具当准备自动准备生成文件时使用 doc comment

这个指导是关于解释 Javadoc 的。我们将看到我们怎样能利用 Javadoc 来为我们的 Java 代码生成有用的文件。

什么是 Javadoc?

Javadoc 是 JDK 附带的一个工具，它被用来生成从需要预定义格式的文档的 Java 源代码至 HTML 格式的 Java 代码文件。

以下是一个简单的例子，其中红色部分代表 Java 注释：

```
/**
* The HelloWorld program implements an application that
* simply displays "Hello World!" to the standard output.
*
* @author  Zara Ali
* @version 1.0
* @since   2014-03-31
*/
public class HelloWorld {
   public static void main(String[] args) {
      // Prints Hello, World! on standard output.
      System.out.println("Hello World!");
   }
}
```

你可以将需要的 HTML 标签包括在描述部分内，比如，下面的例子利用 \<h1>...\</h1> 来定义头部和 \<p> 被用来创建段落间隔：

```
/**
* <h1>Hello, World!</h1>
* The HelloWorld program implements an application that
* simply displays "Hello World!" to the standard output.
* <p>
* Giving proper comments in your program makes it more
* user friendly and it is assumed as a high quality code.
*
*
* @author  Zara Ali
* @version 1.0
* @since   2014-03-31
*/
public class HelloWorld {
   public static void main(String[] args) {
      // Prints Hello, World! on standard output.
      System.out.println("Hello World!");
   }
}
```

Javadoc 标签

Javadoc 标签是 Javadoc 认可的关键字，它定义了下面信息的类型。

Javadoc 工具认可下面的标签：

标签	描述	语法
@author	添加类的作者	@author name-text
{@code}	不把文本转换成 HTML 标记和嵌套的 Java 标签而用代码字体展示它	{@code text}
{@docRoot}	表示从任何生成页面到生成文档的根目录的相对路径	{@docRoot}
@deprecated	添加一个注释暗示 API 应该不再被使用	@deprecated deprecated-text
@exception	用类名和描述文本给生成的文档添加一个副标题	@exception class-name description
{@inheritDoc}	从最近的可继承的类或可实现的接口继承注释	Inherits a comment from the immediate surperclass.
{@link}	用指向特定的包，类或者一个引用类的成员名的文档的可见文本标签插入在线链接	{@link package.class#member label}
{@linkplain}	和{@link}相同，除了链接的标签用纯文本标示而不是代码字体	{@linkplain package.class#member label}

标签	描述	语法
@param	给“参数”区域添加一个有特定参数名且后跟着特定描述的参数	@param parameter-name description
@return	添加一个有描述文本的“Returns”区域	@return description
@see	添加带有链接或者指向引用的文本入口的标题“See Also”	@see reference
@serial	在默认的序列化字段的文本注释中使用	@serial field-description | include | exclude
@serialData	记录由 writeObject( ) 或 writeExternal( )方法所写的数据	@serialData data-description
@serialField	记录一个 ObjectStreamField 成分	@serialField field-name field-type field-description
@since	给生成的文档添加一个带有特定 since 文本的"Since"标题	@since release
@throws	@throw 和 @exception 标签是同义词	@throws class-name description
{@value}	当{@value}被用在一个静态字段的文本注释中，它展示了那个常量的值	{@value package.class#field}
@version	当 -version 选项被使用时用特定的 version w文本给生成的文本添加一个“Version”副标题	@version version-text

示例

下面的程序使用一些重要的标签来做文档注释。你可以基于你的需求利用其它的标签。

关于 AddNum 类的文档将由 HTML 文件 AddNum.html 创建，但是同时一个名为 index.html 的主文件也将被创建。

```
import java.io.*;

/**
* <h1>Add Two Numbers!</h1>
* The AddNum program implements an application that
* simply adds two given integer numbers and Prints
* the output on the screen.
* <p>
* <b>Note:</b> Giving proper comments in your program makes it more
* user friendly and it is assumed as a high quality code.
*
* @author  Zara Ali
* @version 1.0
* @since   2014-03-31
*/
public class AddNum {
```

```
  /**
  * This method is used to add two integers. This is
  * a the simplest form of a class method, just to
  * show the usage of various javadoc Tags.
  * @param numA This is the first paramter to addNum method
  * @param numB  This is the second parameter to addNum method
  * @return int This returns sum of numA and numB.
  */
  public int addNum(int numA, int numB) {
    return numA + numB;
  }

  /**
  * This is the main method which makes use of addNum method.
  * @param args Unused.
  * @return Nothing.
  * @exception IOException On input error.
  * @see IOException
  */
  public static void main(String args[]) throws IOException
  {

    AddNum obj = new AddNum();
    int sum = obj.addNum(10, 20);

    System.out.println("Sum of 10 and 20 is :" + sum);
  }
}
```

现在，处理使用 Javadoc 的 AddNum.java 文件：

```
$ javadoc AddNum.java
Loading source file AddNum.java...
Constructing Javadoc information...
Standard Doclet version 1.7.0_51
Building tree for all the packages and classes...
Generating /AddNum.html...
AddNum.java:36: warning - @return tag cannot be used in method with void return type.
Generating /package-frame.html...
Generating /package-summary.html...
Generating /package-tree.html...
Generating /constant-values.html...
Building index for all the packages and classes...
Generating /overview-tree.html...
Generating /index-all.html...
```

```
Generating /deprecated-list.html...
Building index for all classes...
Generating /allclasses-frame.html...
Generating /allclasses-noframe.html...
Generating /index.html...
Generating /help-doc.html...
1 warning
$
```

你可以在这检查所有的生成的文档：AddNum (http://www.tutorialspoint.com/java/index.html) 。如果你正在使用 JDK 1.7 那么 Javadoc 不生成 stysheet.css，所以我建议从 http://docs.oracle.com/javase/7/docs/api/stylesheet.css 下载并使用标准的 stylesheet。

4

有用的资源

Java 快速参考指南

什么是 Java?

- 面向对象
- 平台独立
- 简单
- 安全
- 中立
- 可移植的
- 强健
- 多线程
- 可翻译的
- 高效
- 分布式的
- 动态的

Java 基本语法

- **对象** – 对象有状态和行为。例子：一条狗有状态：颜色，名字，品种以及行为–摇尾巴，吠，吃。一个对象是一个类的实例。
- **类** – 一个类能被定义成对象的模版/蓝图，用于描述对象类型的行为/状态。
- **方法** – 一个方法基本是一个行为。一个类能包含许多方法。逻辑是写在方法中的，数据被操作并且所有的行为被执行。
- **即时变量** – 每一个对象有它独有的即时变量的设置。一个对象的状态由赋予这些即时变量的值所创建。

第一个 Java 程序

让我们看看一个简单的能打印出单词 Hello World 的代码。

```
public class MyFirstJavaProgram{

  /* This is my first java program.
   * This will print 'Hello World' as the output
   */

  public static void main(String []args){
     System.out.println("Hello World"); // prints Hello World
  }
}
```

关于 Java 程序，记住下面几点很重要。

- **大小写敏感性** - Java 是区分大小写，这意味着标识符 Hello 和 hello 在 Java 中将有不同的含义。
- **类名** - 对于所有的类名来说第一个字母应该是大写的。
  如果几个单词被用来组成类的名字则每一个内部的单词的第一个应该是大写的。
  例子： 类 MyFirstJavaClass
- **方法名字** - 所有的方法名应该以一个小写字母开始。
  如果几个单词用来组成方法的名字，则每个内部单词的第一个字母应该是大写的。
  例子：public void myMethodName()
- **程序文件名** - 程序文件的名字应该准确匹配类名。
  当保存文件时你应该使用类名(记住 Java 是大小写区分的)并且在文件名字末尾附加 ".java"。(如果文件名和类名不匹配，你的程序将不会编译)
  例子：假设 'MyFirstJavaProgram' 是类名，那么文件应该被保存为 'MyFirstJavaProgram.java'。
- **public static void main(String args[])** - Java 程序处理从 main() 方法开始，这是每个 Java 程序的强制部分。

Java 标识符

所有的 Java 成分需要名字。类，变量和方法的名字叫做标识符。

在 Java 中有关标识符有几点需要记住。它们是以下几点：

- 所有的标识符应该以字母(A 到 Z 或者 a 到 z)，货币符号($)或者一个下划线(_)。
- 在第一个字符后，标识符可以有任意字符的组合。
- 关键词不能被用作标识符。
- 最重要的是标识符是区分大小写的。
- 合法标识符的例子:age, $salary, _value, _1_value
- 非法标识符的例子:123abc, -salary

Java 修饰符

像其他语言，使用修饰符修改类、方法等是有可能的。有两种修饰符。

- **访问修饰符:** default, public, protected, private
- **非访问修饰符:** final, abstract, strictfp

我们将在下一节寻找更多有关修饰符的细节。

Java 变量

我们将看到在 Java 中变量的以下类型:

- 本地变量
- 类变量(静态变量)
- 实例变量(非静态变量)

Java 数组

数组是存储同一类型的多个变量的对象。然而一个数组自身是堆上的一个对象。我们将在下一章调查如何声明，构建和初始化。

Java 枚举

枚举是在 Java 5.0 中被引进的。枚举限制了一个变量拥有几个预先定义的值中的一个。在枚举列表中的值被称为枚举。

使用枚举可能减少你代码中的 bug。

例如如果我们为一个新鲜果汁商店考虑一个应用程序，限制杯子的形状小，中，大是有可能的。这将确保没有任何人可以点出了小，中，大之外的型号。

示例

```
class FreshJuice{

  enum FreshJuiceSize{ SMALL, MEDIUM, LARGE }
  FreshJuiceSize size;
}

public class FreshJuiceTest{

  public static void main(String args[]){
    FreshJuice juice = new FreshJuice();
    juice.size = FreshJuice. FreshJuiceSize.MEDIUM ;
    System.out.println("Size :" + juice.size);
  }
}
```

注意： 枚举能被自身声明或者在一个类中。方法，变量，构造函数也能在枚举中被定义。

Java 关键词

以下的列表展示了 Java 中的保留字。这些保留字可能不会被作为常量或者变量或者任何其他标识符的名字而使用。

关键字	关键字	关键字	关键字
abstract	assert	boolean	break
byte	case	catch	char
class	const	continue	default
do	double	else	enum
extends	final	finally	float
for	goto	if	implements
import	instanceof	int	interface
long	native	new	package
private	protected	public	return
short	static	strictfp	super

关键字	关键字	关键字	关键字
switch	synchronized	this	throw
throws	transient	try	void
volatile	while		

Java 中的注释

Java 像 C 和 C++ 一样支持单行和多行注释。所有的在注释中的字符都会被 Java 编译器忽略。

```
public class MyFirstJavaProgram{

  /* This is my first java program.
   * This will print 'Hello World' as the output
   * This is an example of multi-line comments.
   */

   public static void main(String []args){
      // This is an example of single line comment
      /* This is also an example of single line comment. */
      System.out.println("Hello World");
   }
}
```

Java 中的数据类型

在 Java 中有两种数据类型:

- 原始数据类型
- 引用/对象数据类型

原始数据类型

Java 支持的总共有 8 种原始数据类型。原始数据类型是由语言预定义的并且由关键字命名。让我们现在仔细看看有关 8 种原始数据类型的细节。

- byte
- short
- int

- long
- float
- double
- boolean
- char

引用数据类型

- 引用变量是使用定义的类的构造函数创建的。它们被用来访问对象。这些变量被声明为一个特定的不能改变的类型。举例来说，Employee, Puppy 等等。
- Class 类，和不同的数组变量类型归入引用数据类型。
- 任何引用变量的默认值是 null。
- 一个引用变量能被用来指向任何声明类型的对象或者任何兼容的类型。
- 例子: Animal animal = new Animal("giraffe");

Java 字面量

一个字面量是由固定值表示的源代码。它们用代码直接表示而没有任何计算。

字面值可以被赋给任何原始数据类型。例如:

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

Java 中的 String 字面值是指定的，就像它们在大部分其他的语言中通过在一对双引号间包含一系列的字符。String 字面值的例子如下:

```
"Hello World"
"two\nlines"
"\"This is in quotes\""
```

Java 语言给 String 和 char 字面值支持很少的特殊的换码序列。它们是:

记号	符号表示
\n	新的一行(0x0a)
\r	回车(0x0d)

记号	符号表示
\f	进纸(0x0c)
\b	退格(0x08)
\s	空格(0x20)
\t	制表符
\"	双引号
\'	点引号
\	反斜线
\ddd	八进制字符
\uxxxx	十六进制 UNICODE 字符

Java 访问修饰符

Java 提供许多访问修饰符来为类，变量，方法和构造函数设置访问等级。四个访问等级如下:
– 对包可见。默认。不需要修饰符。 – 仅对类(private)可见 – 对所有可见(public) – 对包和所有的子类可见(protected)

Java 基本运算符号

Java 给操作变量提供了一组丰富的操作符。我们可以将所有的 Java 操作符区分成以下几组:

算数操作符

运算符	描述	例子
+	加–在操作符的每一侧加值	A + B 将得出30
–	减–从左边的操作数中减去右边的操作数	A – B 将得出 –10
*	乘–在操作符的每一侧倍增值	A * B 将得出 200
/	除–用右边的操作数除左边的操作数	B \ A 将得出2
%	模–用右边的操作数除左边的操作数并返回余数	B % A 将得出0
++	增值–操作数的值增加 1	B++ 得出21
––	减量–操作数的值减少 1	B–– 得出19

关系操作符

运算符	描述	例子
==	查看两个操作数的值是否相等，如果是的话，情况就是真。	(A == B)不为真

运算符	描述	例子
!=	查看两个操作数的值是否相等，如果值不相等，情况就是真。	(A != B)为真
>	查看左边操作数的值是否大于右边的操作数，如果是的话，情况就是真。	(A > B)不为真
<	查看左边操作数的值是否小于右边的操作数，如果是的话，情况就是真。	(A < B)为真
>=	查看左边操作数的值是否大于或等于右边的操作数，如果是的话，情况就是真。	(A >= B)不为true.
<=	查看左边操作数的值是否小于或等于右边的操作数，如果是的话，情况就是真。	(A <= B)为true.

位运算操作符

运算符	描述	例子
&	二进制的与操作符将一个位复制到结果中如果它存在于两个操作数中	(A & B)将得出 12 即 0000 1100
|	二进制的或操作符复制一个位如果它存在在两个操作数中的一个	(A | B)将得出 61 即 0011 1101
^	二进制的异或操作符复制一个位如果它在一个而不是两个操作数中被设置。	(A ^ B)将得出 49 即 0011 0001
~	二进制补运算符是一元的，并有”烙位”的影响	(~A)将得出 -61 即 2 的补码形式 1100 0011，因为一个有符号的二进制数。
<<	二进制左移运算符。左边操作数的值按右边操作数指定的位数向左移动。	A << 2 将得出 240 即 1111 0000
>>	二进制右移运算符。左边操作数的值按右边操作数指定的位数向右移动。	A >> 2 将得出 15 即 1111
>>>	右移填零运算符。左边操作数的值按右边操作数指定的位数向右移动并且值用 0 填充	A >>>2 将得出 15 即 0000 1111

逻辑操作符

运算符	描述	例子
&&	叫做逻辑与运算符。如果两个操作数都不为 0，那么情况为真	(A && B) 为假
||	叫做逻辑或运算符。如果两个操作数中一个任何一个不为0那么情况为真。	(A || B) 为真
!	叫做逻辑非运算符。用来颠倒操作数的逻辑状态。如果情况为真那么逻辑非运算符将是假。	!(A && B) 为真

赋值操作符

运算符	描述	例子
=	简单的赋值运算符，将值从右边的操作数赋给左边的操作数。	C = A + B 将把 A + B 的值赋给 C
+=	加后赋值运算符，它将右边操作数加上左边的操作数并将结果赋给左边的操作数	C += A 和 C = C + A 相等
−=	减后赋值运算符，它将左边操作数减去右边的操作数并将结果赋给左边的操作数	C −= A 和 C = C − A 相等
*=	乘后赋值运算符，它用左边操作数倍增右边的操作数并将结果赋给左边的操作数	C *= A 和 C = C * A 相等
/=	除后赋值运算符，它用右边操作数去除左边的操作数并将结果赋给左边的操作数	C /= A 和 C = C / A 相等
%=	取模后赋值运算符，它使用两个操作数取模并将结果赋给左边的操作数	C %= A 和 C = C % A 相等
<<=	左移后赋值运算符	C <<= 2 和 C = C << 2 相同
>>=	右移后赋值运算符	C >>= 2 和 C = C >> 2 相同
&=	按位与后赋值运算符	C &= 2 和 C = C & 2 相同
^=	按位异或后赋值运算符	C ^= 2 和 C = C ^ 2 相同
|=	按位或后赋值运算符	C |= 2 和 C = C | 2 相同

混杂操作符

很少有其它的操作符被 Java 语言支持。

条件操作符(?:)

条件操作符也以三元操作符为人所知。这个操作符由三个操作数组成并且被用来估计布尔表达式。操作符的目的是决定哪个值应该被赋给变量。操作符这样写:

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

instanceOf 操作符

这个操作符仅用于对象引用变量。操作符检查对象是否是一种特别类型的(class 类型或者 interface 类型)。instanceOf 操作符这样写:

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

Java 操作符的优先级

种类	运算符	关联性
后缀	() [] . (点操作符)	左到右
一元	++ − − ! ~	右到左
乘法	* / %	左到右
加法	+ −	左到右
移动	>> >>> <<	左到右
关系	> >= < <=	左到右
等式	== !=	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	?:	右到左
赋值	= += −= *= /= %= >>= <<= &= ^= |=	右到左
逗号	,	左到右

while 循环

一个 while 循环是一个允许你重复一个任务一定次数的控制结构。

语法

一个 while 循环的语法像这样:

```
while(Boolean_expression)
{
  //Statements
}
```

do...while 循环

do...while 循环和 while 循环是类似的，除了 do...while 循环保证执行至少一次。

语法

do...while 的语法像这样:

```
do
{
  //Statements
}while(Boolean_expression);
```

for 循环

for 循环是一个重复控制结构，允许你有效地写一个需要执行指定次数的循环。

当你知道一个任务需要被重复多少次时，for 循环是有用的。

语法

for 循环的语法像这样:

```
for(initialization; Boolean_expression; update)
{
  //Statements
}
```

Java 中增强的 for 循环

Java 5 中增强的 for 循环被引进。这主要是为数组而用的。

语法

增强的 for 循环像这样:

```
for(declaration : expression)
{
  //Statements
}
```

break 关键字

break 关键字被用来停止整个循环。break 关键字必须在任何循环中或者一个 switch 语句中使用。

break 关键字将停止最内层的循环执行和开始执行在块后代码的下一行。

continue 关键字

continue 关键字可以在任何控制结构的循环中被使用。它使循环立即跳转到循环的下一个迭代。
- 在一个 for 循环中，continue 关键字使控制流立即跳转到 update 语句。
- 在一个 while 循环或者 do/while 循环中，控制流立即跳转到布尔表达式。

语法

continue 的语法在任何循环中是一条单独的语句:

```
continue;
```

if 语句

if 语句由一个有一条或多条语句伴随的布尔表达式组成。

语法

if 语句的语法像这样:

```
if(Boolean_expression)
{
   //Statements will execute if the Boolean expression is true
}
```

if...else 语句

if 语句可以后接一个可选的 else 语句，它当布尔表达式是错误时执行。

语法

if...else 的语法像这样:

```
if(Boolean_expression){
   //Executes when the Boolean expression is true
}else{
   //Executes when the Boolean expression is false
}
```

if...else if...else 语句

if 语句后可以跟一个可选的 else if...else 语句，它对于测试使用单独的 if...else if 语句的不同状况是很有用的。

语法

if...else 的语法像这样：

```
if(Boolean_expression 1){
   //Executes when the Boolean expression 1 is true
}else if(Boolean_expression 2){
   //Executes when the Boolean expression 2 is true
}else if(Boolean_expression 3){
   //Executes when the Boolean expression 3 is true
}else {
   //Executes when the one of the above condition is true.
}
```

嵌入的 if...else 语句

嵌入 if...else 语句总是合法的。当使用 if，else if， else 语句时需要记住这几点:

- if 可以有 0 个或者 1 个 else 并且它必须在任何 else if 之后。
- if 可以有 0 到许多 else if 并且它们必须在 else 之前。
- 一旦一个 else if 成功，剩余的 else if 和 else 将不会执行。

语法

一个嵌入的 if...else 语法像以下这样:

```
if(Boolean_expression 1){
   //Executes when the Boolean expression 1 is true
   if(Boolean_expression 2){
      //Executes when the Boolean expression 2 is true
   }
}
```

switch 语句

switch 语句允许一个变量平等地用一系列的值进行测试。每一个值成为一个 case,进行 switch 的变量和每个 case 值比较是否相等。

语法

增强的 for 循环的语法像这样:

```
switch(expression){
    case value :
       //Statements
       break; //optional
    case value :
       //Statements
       break; //optional
    //You can have any number of case statements.
    default : //Optional
       //Statements
}
```

Java 方法

一个 Java 方法是组合到一起来执行一个操作的语句的集合。比如，当你调用 System.out.println 方法,系统确实为在控制台显示一条信息而执行了几条语句。

总体来说，方法由以下语法:

```
modifier returnValueType methodName(list of parameters) {
  // Method body;
}
```

- **修饰符：** 可选择的修饰符，告诉编译器如何调用方法。这定义了方法的入口类型。

- **返回值：** 方法会返回一个值。returnValueType 是方法返回的值的数据类型。一些方法执行所需的操作而不返回一个值。在这种情况下，returnValueType 就是关键字 void。
- **方法名：** 这是方法的实际名字。方法名和参数列一起构成了方法签名。
- **参数：** 参数就像占位符。当一个方法被调用，你将一个值传给参数。这个值被称为实际参数或参数。参数列表指的是方法参数的类型，顺序和数量。参数是可选的，那就是说，方法可能不含参数。
- **方法体：** 方法体包含定义方法做什么的语句集合。

Java 类&对象

- **对象** - 对象有状态和行为。例子:一条狗有状态：颜色，名字，品种以及行为-摇尾巴，吠，吃。一个对象是一个类的实例。
- **类** - 一个类能被定义成对象的模版/蓝图，用于描述对象类型的行为/状态。

一个类像以下这样:

```
public class Dog{
  String breed;
  int age;
  String color;

  void barking(){
  }

  void hungry(){
  }

  void sleeping(){
  }
}
```

一个类能含有任何以下的变量类型。

- **本地变量** - 在方法，构造函数或者块内定义的变量被称为本地变量。变量将在方法内被声明和初始化并且变量将在当方法完成时被销毁。
- **实例变量** - 实例变量是类内但是在方法外的变量。这些变量当类被载入时被实例化。实例变量能在任何那个特定类的方法，构造函数或块内被访问。
- **类变量** - 类变量是在一个类中，在任何方法之外被声明的变量，伴随一个 static 关键字。

异常处理

方法使用 try 和 catch 关键字的组合捕获异常。try/catch 块被放置在可能会生成异常的代码周围。有 try/catch 块的代码被称作受保护的代码，使用 try/catch 的语法像以下这样:

```
try
{
  //Protected code
}catch(ExceptionName e1)
{
  //Catch block
}
```

多个 catch 块

一个 try 块后能跟多个 catch 块。多个 catch 块的语法像这样:

```
try
{
  //Protected code
}catch(ExceptionType1 e1)
{
  //Catch block
}catch(ExceptionType2 e2)
{
  //Catch block
}catch(ExceptionType3 e3)
{
  //Catch block
}
```

throws/throw 关键字

如果方法不处理检查异常，方法必须用 throws 关键字声明它。throws 关键字在方法签名的最后出现。

你可以通过使用 throw 关键字抛出一个异常，可以是新实例化的或者刚刚捕获到的。尽量理解 throws 和 throw 关键字间的不同。

finally 关键字

finally 关键字被用来创建跟在 try 块后的代码。finally 代码块总是执行，无论一个异常是否发生。

使用 finally 块允许你运行你想要执行的任何 clean-type 语句，无论在受保护的代码中发生了什么。

finally 块在 catch 块的末尾出现并有以下语法:

```
try
{
  //Protected code
}catch(ExceptionType1 e1)
{
  //Catch block
}catch(ExceptionType2 e2)
{
  //Catch block
}catch(ExceptionType3 e3)
{
  //Catch block
}finally
{
  //The finally block always executes.
}
```

Java 库类

这个指导将包含 java.lang 包，它提供了对 Java 编程语言的设计非常重要的类。最重要的类是 Object，它是类层次的根。还有 Class,它是代表运行时类的实例。

这是 java.lang 包的类列表。这些类对一个 Java 编程者来说是非常重要的。点击一个类的链接来知道更多有关类的细节。更一步钻研的话，你可以参考标准 Java 文档。

SN	方法描述
1	Boolean 布尔值
2	Byte Byte 类包装在对象中一个原始类型 byte 的值。
3	Character Character 类包装在对象中一个原始类型 char 的值。
4	Class class 类的实例代表在一个运行 Java 应用程序中的类和接口。
5	ClassLoader class loader 是一个负责加载类的对象。
6	Compiler Compiler 类被提供来支持 Java-to-native-code 编译器和相关的服务。
7	Double Double　类包装在对象中一个原始类型 double 的值。
8	Float Float 类包装在对象中一个原始类型 float 的值。
9	Integer Integer 类包装在对象中一个原始类型 int 的值。
10	Long Long 类包装在对象中一个原始类型 long 的值。
11	Math Math 类包含执行基本的数字操作比如基本指数，对数，平方根和三角函数的方法。
12	Number 抽象的 Number 类是 BigDecimal,BigInteger,Byte,Double,Float, Integer,Long 和 Short 类的超类。
13	Object Object 类是类层次结构的根。
14	Package Package对象包含关于实现和　Java 包的版本信息。
15	Process Runtime.exec 方法创建了一个本地的 process 并且返回一个 Process 超类的能被用来控制 process 并获得有关它信息的实例。

SN	方法描述
16	Runtime 每一个 Java 应用程序有一个单独的允许应用程序接口它运行环境的 Runtime 类的实例。
17	RuntimePermission 这个类是运行时权限有关。
18	SecurityManager security manager 是一个运行应用程序实现安全策略的类。
19	Short short 类包装在对象中一个原始类型 short 的值。
20	StackTraceElement 堆栈跟踪中的一个元素，由 Throwable.getStackTrace()范围。
21	StrictMath StrictMath 类包含执行基本数字操作比如基本指数，对数，平方根和三角函数的方法。
22	String String 类代表字符串。
23	StringBuffer 一个 string buffer 实现一个可变的字符序列。
24	System System 类报班几个有用的类字段和方法。
25	Thread thread 指在程序中的执行的线程。
26	ThreadGroup 线程组代表一系列的线程。
27	ThreadLocal 这个类提供 thread-local 变量。
28	Throwable Throwable 类是所有 Java 语言中错误和异常的超类。
29	Void Void 类是一个不可实例化的占位符类，来保持对代表 Java 关键字 void 的 Class 对象的引用。

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/java/