# 关于本课程

本书的主要目标是为你提供有关 Java 应用和 Java applets 的面向对象的程序设计所必需的知识和技能；并讲授 Java 编程语言句法和面向对象的概念，以及 Java 运行环境的许多特性， 对图形用户界面（GUIs）、多线程和网络的支持等。本课程包括了参加 SUN Certified Java Programmer 和 SUN Certified Java Developer 考试所必备的知识。

## 一、课程概述

**课程概述**

本课程主要包括以下内容：

- Java 编程语言句法
- 应用于 Java 编程语言的面向对象的概念
- 图形用户界面（GUI）编程
- 创建 Applet
- 多线程
- 网络

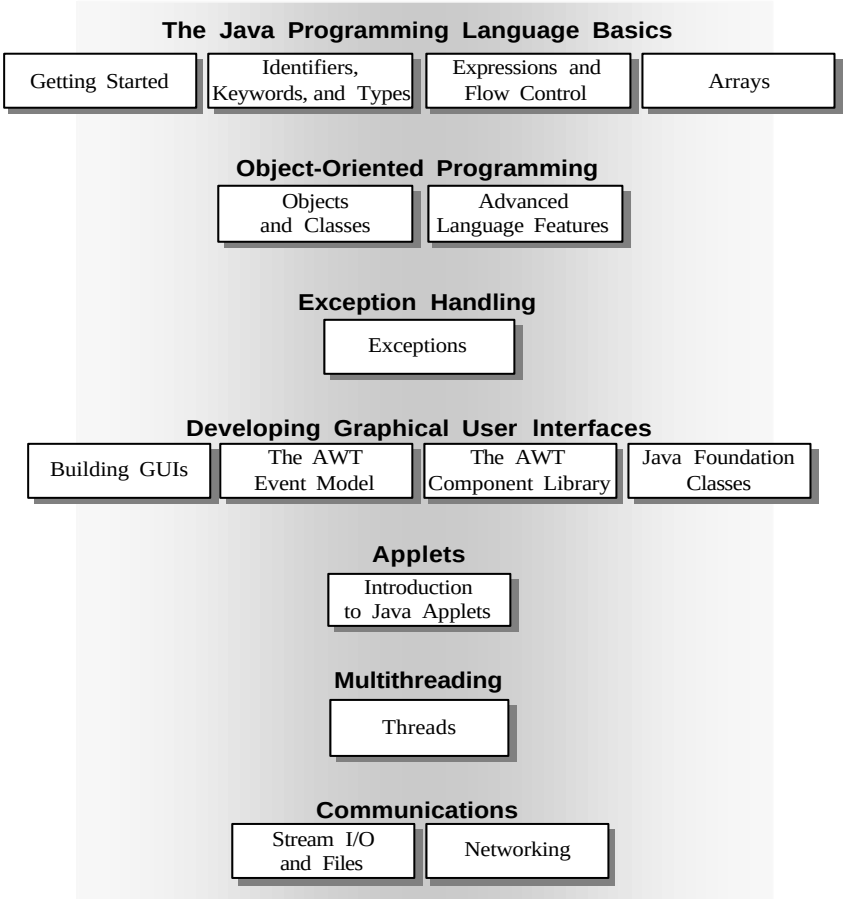
本课程首先讨论了 Java 运行环境和 Java 编程语言句法，然后阐述了应用于 Java 编程语言的面向对象的概念；随着课程的进展还将讨论有关 Java 平台的更先进的特性。

本课程授课对象应具备运用 Java 编程语言或其它语言进行基本语言设计的能力，它是“非程序员 Java 编程”（“Java Programming for Non—Programmers”，SL-110）课程的延续。

尽管 Java 编程语言独立于操作系统，但它所产生的图形用户界面（GUI）却可能依赖于执行其代码的操作系统。本课程中的例题所使用的代码运行于 Solaris ™ 操作系统，因而本手册中的图形具备 Motif GUI。如果将这些代码运行于 Windows 95 操作系统，则可产生 Microsoft Windows 95 GUI。本课程的内容适用于所有 Java 操作系统端口。

## 二、课程图

每一模块的课程都从一张课程图开始，使学员可掌握自己的学习进度。全课程设置流程图如下所示：



# 三、各模块概述

**各模块概述**

- 模块 1——起步
- 模块 2——标识符、关键字和类型
- 模块 3——表达式和流程控制
- 模块 4——数组
- 模块 5——对象和类
- 模块 6——高级语言特性
- 模块 7——异常
- 模块 8——建立 GUIs
- 模块 9——AWT 事件模型
- 模块 10——AWT 组件库
- 模块 11——Java 基础类
- 模块 12——Java Applets 介绍
- 模块 13——线程
- 模块 14——流式 I/O 和文件
- 模块 15——网络

- **模块 1——起步**

本模块概述了 Java 编程语言的主要特性及 Java 应用程序，阐述了类和包的概念，并介绍了一些常用 Java 包。

- **模块 2——标识符、关键字和类型**

Java 编程语言与 C 语言类似，具有许多编程构造。本模块对这些构造作了一般介绍并讲授了每一构造所要求的一般句法。此外，还介绍了采用集合数据类型进行数据关联的面向对象的方法。

- **模块 3——表达式和流程控制**

本模块介绍了包括运算符和 Java 程序控制句法在内的表达式。

- **模块 4——数组**

本模块讲解了如何声明、创建、初始化和拷贝 Java 数组。

- **模块 5——对象和类**

本模块是在模块 2 的基础上，对 Java 对象的概念作了进一步的阐述，包括重载、覆盖、子类和构造函数。

- **模块 6——高级语言特性**

本模块是模块 5 的继续，它进一步讲解了 Java 面向对象的编程模块，包括一些新的 JDK1.1 特性——降级和内部类。该模块还介绍了在 Java 开发包 JDK™ 1.2 中新增的收集的概念。

- **模块 7——异常**

本模块为 Java 程序员提供了一种在运行 Java 程序时俘获错误的机制，并阐述了预定义异常和用户定义异常。

- **模块 8——建立 GUIs**

在 Java 编程语言中，所有图形用户界面都是建立在框架和面板的概念之上。本模块介绍了布局管理和容器的概念。

- **模块 9——AWT 事件模型**

Java 编程语言 1.1 版的最重要的变化就是将事件送入和送出 Java 组件的方法的变化。本模块阐述了 JDK1.0 和 1.1 的事件模型的区别，并演示了如何建立小型事件柄（compact event handler）的方法。

- **模块 10——AWT 组件库**

本模块介绍了用于建立 Java GUIs 的抽象窗口工具包（AWT）组件，并演示了 Java AWT 组件和 1.1 事件模型共同工作的方法。

- **模块 11——Java 基础类介绍**

本模块重点介绍了 JDK1.2 的一个重要特性——Java 基础类（JFC），阐述了 Swing 组件和它们的可插入式外观及感觉的体系结构，并介绍了一例基本的 Swing 应用及 JFC 的其它特点，例如在二维图形、存取性及

拖放（Drag and drop）等方面的应用。

- 模块 12——Java Applets 介绍

本模块演示了 applet 和应用程序开发之间的区别，并介绍了 JDK 1.2 的声音增强功能.

- 模块 13——线程

线程是一个复杂的论题。本模块阐述了与 Java 编程语言相关的线程问题，并介绍了一个简单的线程通信和同步传输实例。

- 模块 14——流式 I/O 和文件

本模块阐述了既可读写数据又可读写文本文件的类，并介绍了对象流的概念。

- 模块 15——网络

本模块介绍了 Java 网络编程包并演示了传输控制协议/Internet 协议（TCP/IP）客户——服务器模型。

## 四、课程目标

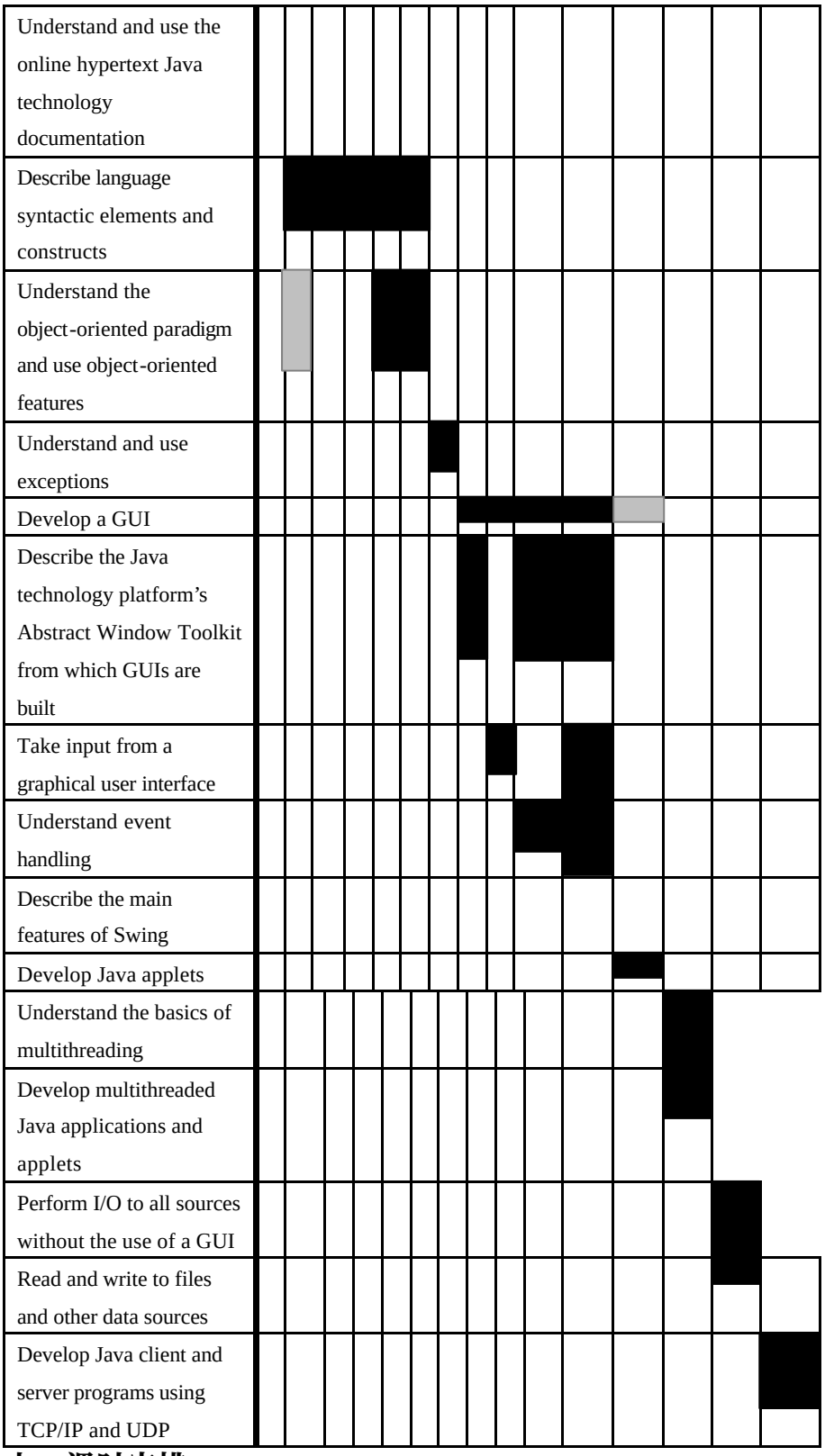
完成本课程的学习后，你应该能够：

- 描述语言的主要特性
- 编译并运行 Java 应用程序
- 理解和使用在线超文本 Java 技术文件
- 描述语言句法元素和构造
- 理解面向对象的例子并使用该语言的面向对象特性
- 理解并使用异常
- 开发图形用户界面
- 描述 Java 技术平台上的用于建立 GUIs 的抽象窗口工具包
- 从 GUI 输入
- 理解事件处理
- 描述 Swing 的主要特性
- 开发 Java applets
- 读写文件和其它数据源
- 在不使用 GUI 的情况下，对所有数据源执行输入输出
- 理解多线程基础
- 开发多线程 Java 应用程序和 applets
- 使用 TCP/IP 和用户数据报协议（UDP）开发客户和服务器程序

# 五、从各模块中所获得的技能

下表的左侧显示有关的 Java 应用程序编程技能，右侧的黑格表示出讲解其左侧相应技能的有关模块，而灰格则表示简述其左侧相应技能的有关模块。

| Skills Gained | Module | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Describe Key language features | ■ | | | | | | | | | | | | | | |
| Compile and run a Java application | ■ | | | | | | | | | | | | | | |




## 六、课时安排

**下表给出了大致的课程进度**

| Module | Day 1 | Day 2 | Day 3 | Day 4 | Day 5 |
|---|---|---|---|---|---|
| About This Course | A.M. | | | | |
| Module 1-Getting Started | A.M. | | | | |
| Module 2-Identifiers,Keywords,and Types | A.M. | | | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| Module 3-Expressions and Flow Control | P.M. | | | | | |
| Module 4-Arrays | P.M. | | | | | |
| Module 5-Objects and Classes | | A.M. | | | | |
| Module 6-Advanced Language Features | | P.M. | | | | |
| Module 7-Exceptions | | | A.M. | | | |
| Module 8-Building GUIs | | | A.M. | | | |
| Module 9-The AWT Event Model | | | P.M. | | | |
| Module 10-The AWT Component Library | | | | A.M. | | |
| Module 11-Introduction to JFC | | | | A.M. | | |
| Module 12-Introduction to Applets | | | | P.M. | | |
| Module 13-Threads | | | | | | A.M. |
| Module 14-Stream I/O and Files | | | | | P.M. | |
| Module 15-Networking | | | | | P.M. | |

## 七、未包括的论题

**未包括的论题**

- **一般编程概念。本课程不是为从未参与过编程的人员而设置。**
- **一般面向对象概念。**

**本课程未包括的论题见上表，由** SUN Educational Services（SES）**提供的其它课程包括了上述论题。**

- **面向对象的概念——见** OO-100：**面向对象的技术和概念**
- **面向对象的设计和分析——见** OO-120 **面向对象的设计和分析**
- **一般编程概念——见** SL-110：**非程序员** Java **编程**

# 八、如何做好准备？

**如何做好准备？**

**在学习本课程前，你应该已经完成了下列内容的学习：**

- SL-110 **非程序员** Java **编程**
- **用** C **或** C++**创建编译程序**
- **用文本编辑器创建并编辑文本文件**
- **使用** World Wide Web（WWW）**浏览器，如** Netscape Navigator™

**在学习本课程前，你应该已经完成了下列内容的学习：**

- SL-110 **非程序员** Java **编程**

**或能够：**

- **用** C **或** C++**创建编译程序**
- **用文本编辑器创建并编辑文本文件**
- **使用** World Wide Web（WWW）**浏览器，如** Netscape Navigator™

## 九、自我介绍：

| 自我介绍 |
| --- |
| - 姓名<br>- 公司<br>- 职务、职责<br>- 编程经历<br>- 参与本课程学习的原因<br>- 对本课程的预期 |

现在你已经进入本课程的学习，请互相介绍并填写上面的表格。

## 十、如何使用本教材

| 如何使用本教材 |
| --- |
| - 课程图<br>- 相关问题<br>- 页头图<br>- 教程<br>- 练习<br>- 检查你的进度<br>- 思考题 |

为使你学好本课程，本教材采用了包括下列主要项目的授课模式：

- 课程图——每一模块都从内容概述开始，这样你就可以了解该模块在实现整个课程目标中所起的作用。
- 相关问题——每一模块的相关问题部分提供给你一些有关该模块的问题供学习者之间进行讨论，并鼓励你思考在 Java 应用程序编程过程中你的兴趣与该模块内容的关系
- 页头图——页头图可帮助你及时而轻松的跟上教师的进度。它并不是在每一页都出现
- 教程——教师将为你提供有关模块的特有信息，这些信息将有助于你掌握做练习所需要的知识和技能
- 练习——练习将为你提供测试你的技能和应用所学概念的机会。本教程中所提供的例题代码亦有助于你完成练习。
- 检查你的进度——在这一部分，模块的目标被重新提及，有时是以问题形式出现。这样，在你进入下一模块之前，就可以检查你是否已完成了目前这一模块所要求达到的目标。
- 思考题——具有挑战性的问题有助于你应用所学模块内容或预测待学模块的内容。

## 十一、图表和印刷体例说明

下列图表及印刷体例代表着不同的含义：

辅助资源——表示可获取辅助参考资料。

讨论——表示此时建议以小组或班级形式进行讨论。

练习目标——表示练习的目标，这个练习适合于正在讨论的内容。

---

注意——附加重要信息、强调信息、有趣或特殊信息

---

---

警告——对数据或机器的潜在损坏

---

**印刷体例**

**字体** Courier **系用来表示命令名称、文件名及目录名，同时也表示计算机屏幕输出。例如：**

```
Use ls –al to all files.
System% You have mail
```

**黑体** courier bold **系用来表示你打印的字符和数字。**

**例如：**

```
system% su
Password:
```

**斜体** courier italic **系用来表示变量和命令行，它可以被实际名称或值所代替。**

**例如：**

**To delete a file, type** *rm filename.*

**斜体** Palatino **系用来表示书名、新词或术语或需强调的词。**

**例如：**

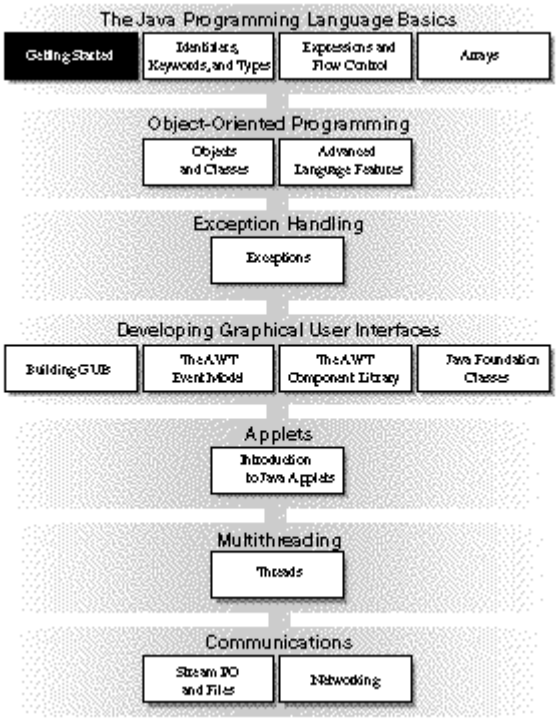
Read Chapter 6 in *User's Guide.*

These are called *class* options

You *must* be root to do this.

# 第一章 起 步

**本模块对** Java **编程语言做了一般性描述，包括** Java **虚拟机、垃圾收集和安全性等。**



## 第一节 相关问题

讨论——下列问题与本模块相关

- Java 编程语言是一种完整的语言吗？它是不是仅用于编写 Web 程序 ？
- 你为什么需要另一种编程语言 ？
- Java 平台是如何在其它语言平台上提高性能的 ？

## 第二节 目 标

完成本模块的学习后，你应该能够：

- 描述 Java 编程语言的主要特性
- 描述 Java 虚拟机的主要功能
- 描述垃圾收集是如何进行的
- 列举由处理代码安全性的 Java 平台所执行的三大任务
- 定义 Class，Package，Applets 和 Applications
- 编写、编译并运行简单 Java 应用程序
- 利用 Java 技术应用程序编程界面(API)的在线文档来标识 Java. lang 包的方法。

参考资料

辅助资源——下列参考资料可为本模块所讨论的问题提供更详细的补充

- Lindholm and Yellin. 1997. The Java Virtual Machine Specification. Addison-Wesley.
- Yellin, Frank. Low-Level Security in Java, white paper. [Online]. Available: http://www.javasoft.com/sfaq/verifier.html.

# 第三节 什么是 Java 编程语言 ？

| 什么是 Java 编程语言 |
| --- |
| Java 是： |
| - 一种编程语言 |
| - 一种开发环境 |
| - 一种应用环境 |
| - 一种部署环境 |
| - 句法与 C++相似，语义与 Small Talk 相似 |
| - 用来开发 applets，又用来开发 applications |

Java 是：

- 一种编程语言
- 一种开发环境
- 一种应用环境
- 一种部署环境

Java 编程语言的句法与 C++的句法相似，语义则与 Small Talk ™ 的语义相似。Java 编程语言可被用来创建任何常规编程语言所能创建的应用程序。

在 World Wide Web（WWW）和能够运行称为 applets 程序的浏览器的有关介绍中，人们经常提及 Java 编程语言。Applets 是一种贮存于 WWW 服务器的用 Java 编程语言编写的程序，它通常由浏览器下载到客户系统中，并通过浏览器运行。Applets 通常较小，以减少下载时间，它由超文本标识语言（HTML）的 Web 页来调用。

Java applications 是一种独立的程序，它不需要任何 Web 浏览器来执行。它们是一种典型的通用程序；可运行于任何具备 Java 运行环境的设备中。

### 1.3.1 Java 编程语言的主要目标

Java 编程语言的主要目标

提供一种解释环境为

- 提高开发速度
- 代码可移植性
- 使用户能运行不止一个活动线程
- 当程序运行时，能动态支持程序改变

提供更好的安全性

设计 Java 编程语言的主要目标是

提供一种易于编程的语言，从而

- 消除其它语言在诸如指针运算和存储器管理方面影响健壮性的缺陷。
- 利用面向对象的概念使程序真正地成为可视化程序
- 为使代码尽可能清晰合理、简明流畅提供了一种方法

Java 编程语言的主要目标

设计 Java 编程语言的主要目标是

提供一种易于编程的语言

- 消除其它语言在诸如指针运算和存储器管理方面影响健壮性的缺陷。
- 利用面向对象的概念使程序真正地成为可视化程序
- 为使代码尽可能清晰合理、简明流畅提供了一种方法

Java 编程语言的主要目标（续）

- 为获得如下两点益处提供一种解释环境
  - 提高开发速度——消除编译—链接—装载—测试周期。
  - 代码可移植性——使操作系统能为运行环境做系统级调用
- 为运行不止一个活动线程的程序提供了一种方式
- 通过允许下载代码模块，从而当程序运行时也能动态支持程序改变。
- 为那些保证安全性而装载的代码模块提供了一种检查方法。

Java 编程语言的主要目标

下列特性使这些目标付诸实现：

- Java 虚拟机（JVM）
- 垃圾收集
- 代码安全性

Java 编程语言的主要目标（续）

精心开发的 Java 技术体系结构为上述目标的实现提供了保证。Java 的如下特性使这些目标得以实现

- Java 虚拟机
- 垃圾收集
- 代码安全性

### 1.3.2 Java 虚拟机

Java 虚拟机

- 提供硬件平台规范
- 解读独立于平台的已编译的字节码
- 可当作软件或硬件来实现
- 可在 Java 技术开发工具或 Web 浏览器上实现

Java 虚拟机规范为 Java 虚拟机（JVM）作了如下定义：

在真实机器中用软件模拟实现的一种想象机器。Java 虚拟机代码被存储在 .class 文件中；每个文件都包含最多一个 public 类。

Java 虚拟机规范为不同的硬件平台提供了一种编译 Java 技术代码的规范，该规范使 Java 软件独立于平台，因为编译是针对作为虚拟机的“一般机器”而做，这个“一般机器”可用软件模拟并运行于各种现存的计算机系统，也可用硬件来实现。

| Java 虚拟机 |
| --- |
| JVM 为下列各项做出了定义<br>- 指令集（中央处理器［CPU］ ）<br>- 注册集<br>- 类文件格式<br>- 栈<br>- 垃圾收集堆<br>- 存储区 |

Java 虚拟机（续）

编译器在获取 Java 应用程序的源代码后，将其生成字节码，它是为 JVM 生成的一种机器码指令。每个 Java 解释器，不管它是 Java 技术开发工具，还是可运行 applets 的 Web 浏览器，都可执行 JVM。

**JVM 为下列各项做出了定义**

- 指令集（相当于中央处理器［CPU］ ）
- 注册集
- 类文件格式
- 栈
- 垃圾收集堆
- 存储区

| Java 虚拟机 |
| --- |
| - 由保持适当类型约束的字节码形成代码<br>- 大部分类型检查在编译代码时完成<br>- 每个由 SUN ™ 批准的 JVM 必须能够运行任何从属类文件 |

Java 虚拟机（续）

JVM 的代码格式由紧缩有效的字节码构成。由 JVM 字节码编写的程序必须保持适当的类型约束。大部分类型检查是在编译时完成。

任何从属的 Java 技术解释器必须能够运行任何含有类文件的程序，这些类文件应符合 Java 虚拟机规范中所指定的类文件格式。

### 1.3.3 垃圾收集

| 垃圾收集 |
| --- |
| - 不再需要的分配存储器应取消分配<br>- 在其它语言中，取消分配是程序员的责任<br>- Java 编程语言提供了一种系统级线程以跟踪存储区分配<br>- 垃圾收集<br>  - 可检查和释放不再需要的存储器<br>  - 可自动完成上述工作<br>  - 可在 JVM 实现周期中，产生意想不到的变化 |

许多编程语言都允许在程序运行时动态分配存储器，分配存储器的过程由于语言句法不同而有所变化，但总是要将指针返回到存储区的起始位置

当分配存储器不再需要时（存储器指针已溢出范围），程序或运行环境应取消分配存储器。

在 C，C++ 或其它语言中，程序员负责取消分配存储器。有时，这是一件很困难的事情。因为你并不总是事先知道存储器应在何时被释放。当在系统中没有能够被分配的存储器时，可导致程序瘫痪，这种程

序被称作具有存储器漏洞。

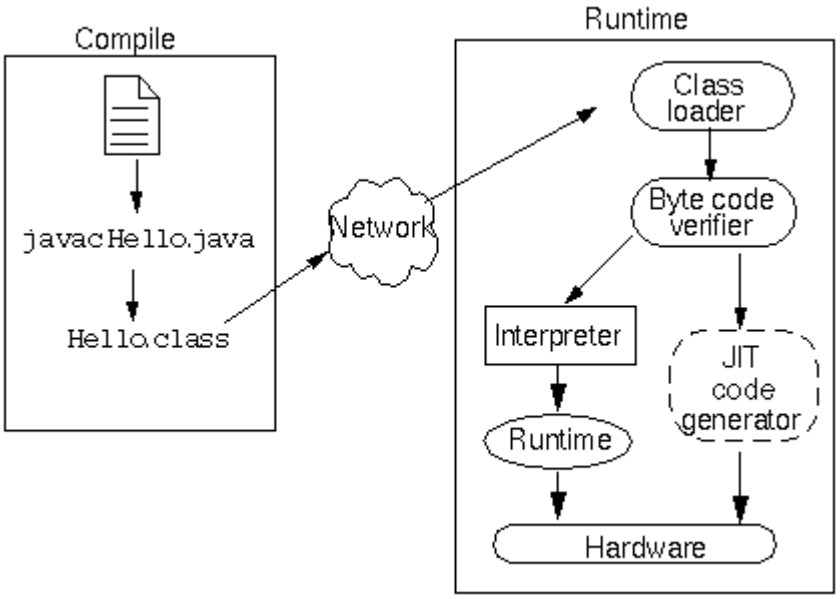
Java 编程语言解除了程序员取消分配存储器的责任，它可提供一种系统级线程以跟踪每一存储器的分配情况。在 Java 虚拟机的空闲周期，垃圾收集线程检查并释放那些可被释放的存储器。

垃圾收集在 Java 技术程序的生命周期中自动进行，它解除了取消分配存储器的要求，并避免了存储器漏洞。然而，垃圾收集可在 JVM 实现的周期中，产生意想不到的变化。

### 1.3.4 代码的安全性

概述

下图显示了 Java 技术的运行环境及其加强代码安全性的方法。




Java 源程序文件通过“编译”，在你的感觉中，就好象是将它们从程序员编写的文本文件格式转换成了一组字节码。字节码被存储在 .class 文件中。

构成 Java 软件程序的字节码在运行时被加载、校验并在解释器中运行。当运行 applets 时，字节码可被下载，然后由建于浏览器中的 JVM 进行解释。解释器具备两种功能，一是执行字节码，二是对底层硬件做适当调用。

在一些使用 Java 技术的运行环境中，部分校验过的字节码被编译成原始机器码并直接运行于硬件平台。这就使 Java 软件代码能够以 C 或 C++接近的速度运行，只是在加载时， 因为要编译成原始机器码而略有延迟。

---

注意－－SUN Microsystems™ 通过采用新的技术，使 Java 虚拟机的性能已经有了进一步的提高。这种新型虚拟机被称为 HotSpot™ 虚拟机，它具备了使 Java 编程语言能象编译 C++一样快速运行的潜力。HotSpot 虚拟机对操作系统具有本地多线程支持能力，而不是虚拟多线程。因而，HotSpot 虚拟机可保证对一个应用程序来说，并不一定有代码才能使用这一能力。HotSpot 技术解决了性能与移植性之间的难题。

---

Java 运行环境

执行三大任务

- 加载代码
- 校验代码
- 执行代码

代码安全性（续）

Java 运行环境

一个 Java 技术的运行环境可运行为 JVM 编译的代码并执行如下三大任务：

- 加载代码－由类加载器执行
- 校验代码－由字节码校验器执行
- 执行代码－由运行时的解释器执行

类加载器

类加载器为程序的执行加载所需要的全部类。类加载器将局部文件系统的类名空间与来自网络源的类名空间相分离，以增加安全性。由于局部类总是首先加载，因而可限制任何“特洛依木马”的应用。

当全部类被加载后，可执行文件的存储器格式被确定。这时，特定的存储器地址被分配给符号引用并创建检索表格。由于存储器格式在运行时出现，因而 Java 技术解释器增加了保护以防止对限制代码区的非法进入。

| 字节码校验 |
| --- |
| 可保证<br>- 代码符合 JVM 规范<br>- 代码不破坏系统完整性<br>- 代码不会引起操作数栈上溢或下溢<br>- 所有操作代码的参数类型的准确性<br>- 无非法数据转换（整数到指针的转换） |

## 代码安全性（续）

字节码校验器

Java 软件代码在实际运行之前要经过几次测试。JVM 将代码输入一个字节码校验器以测试代码段格式并进行规则检查－－检查伪造指针、违反对象访问权限或试图改变对象类型的非法代码。

---

注意----所有源于网络的类文件都要经过字节码校验器

---

代码安全性（续）

校验过程

字节码校验器对程序代码进行四遍校验，这可以保证代码符合 JVM 规范并且不破坏系统的完整性。如果校验器在完成四遍校验后未返回出错信息，则下列各点可被保证：

- 类符合 JVM 规范的类文件格式
- 无访问限制违例
- 代码未引起操作数栈上溢或下溢
- 所有操作代码的参数类型将总是正确的
- 无非法数据转换发生，如将整数转换为对象引用
- 对象域访问是合法的

# 第四节 一个基本的 Java 应用程序

象其它编程语言一样，Java 编程语言也被用来创建应用程序。一个共同的小应用程序范例是在屏幕上显示字串“Hello World!”。下列代码给出了这个 Java 应用程序。

### 1.4.1 HelloWorldApp

```
1.//
2.// Sample HelloWorld application
3.//
```

```
4.public class HelloWorldApp{
5.public static void main (String args[]) {
6.System.out.println ("Hello World!");
7.}
8.}
```

以上程序行是在你的屏幕上打印“Hello World!”所需的最少组件。

### 1.4.2 描述 HelloWorldApp

**第 1-3 行**

程序中的 1-3 行是注释行

```
1 //
2 // Sample HelloWorld application
3 //
```

**第 4 行**

第 4 行声明类名为 HelloWorldApp。类名（Classname）是在源文件中指明的，它可在与源代码相同的目录上创建一个 *classname* .class 文件。在本例题中，编译器创建了一个称为 HelloWorldApp.class 的文件，它包含了公共类 HelloWorldApp 的编译代码。

```
4 public class HelloWorldApp {
```

**第 5 行**

第 5 行是程序执行的起始点。Java 技术解释器必须发现这一严格定义的点，否则将拒绝运行程序。

其它程序语言（特别是 C 和 C++）也采用 main ()声明作为程序执行的起始点。此声明的不同部分将在本课程的后几部分介绍。

如果在程序的命令行中给出了任何自变量，它们将被传递给 main()方法中被称作 args 的 String 数组。在本例题中，未使用自变量。

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

- public－方法 main()可被任何程序访问，包括 Java 技术解释器。
- static－ 是一个告知编译器 main()是用于类 HelloWorldApp 中的函数的关键字。为使 main()在程序做其它事之前就开始运行，这一关键字是必要的。
- void－ 表明 main()不返回任何信息。这一点是重要的，因为 Java 编程语言要进行谨慎的类型检查，包括检查调用的方法确实返回了这些方法所声明的类型。
- String args [] － 是一个 String 数组的声明，它将包含位于类名之后的命令行中的自变量。

```
java HelloWorldApp args [0] args [1] ····
```

**第 6 行**

第 6 行声明如何使用类名、对象名和方法调用。它使用由 System 类的 out 成员引用的 PrintStreamout 对象的 println()方法，将字串“Hello World!”打印到标准输出上。

```
6    System.out.println (“Hello World!”);
```

在这个例子中，println()方法被输入了一个字串自变量并将其写在了标准输出流上。

**第 7-8 行**

本程序的 7-8 行分别是方法 main()和类 HelloWorldApp 的下括号。

```
7          }
8       }
```

### 1.4.3 编译并运行 HelloWorldApp

编译并运行 HelloWorldApp

- 编译 HelloWorldApp.java

```
javac HelloworldApp.java
```

- 运行应用程序

```
java HelloWorldApp
```

- 判定编译和运行的共同错误

**编译**

**当你创建了** HelloWorldApp.java **源文件后，用下列程序行进行编译：**

```
c:\student\javac HelloWorldApp.java
```

**如果编译器未返回任何提示信息，新文件** HelloWorldApp.class **则被存储在与源文件相同的目录中，除非另有指定。**

**如果在编译中遇到问题，请参阅本模块的查错提示信息部分。**

**运行**

**为运行你的** HelloWorldApp **应用程序，需使用** Java **解释器和位于** bin **目录下的** java:

```
c:\student\ java HelloWorldApp
Hello World!
```

**注意－必须设置 PATH 环境变量以发现 java 和 javac，确认它包括 c:\jdk1.2\bin**

### 1.4.4 编译查错

**编译时的错误**

**以下是编译时的常见错误**

- `javac:Command not found`

  PATH **变量未正确设置以包括** javac **编译器。**javac **编译器位于** JDK **目录下的** bin **目录。**

- `HelloWorldApp.java:6: Method printl(java.lang.String)`

```
not found in class java.io.PrintStream.System.
out.printl ("Hello World!");
```

  **方法名** println **出现打印错误。**

- `In class HelloWorldApp:main must be public or static`

  **该错误的出现是因为词** static **或** public **被放在了包含** main **方法的行之外。**

**运行时的错误**

- `can't find class HelloWorldApp` **（这个错误是在打印** java HelloWorldApp **时产生的）**

**通常，它表示在命令行中所指定的类名的拼写与** *filename*.class **文件的拼写不同。**Java **编程语言是一种大小写区别对待的语言。**

**例如：** `public class Helloworldapp {`

**创建了一个** HellloWorldapp.class**，它不是编译器所预期的类名**(HelloWorldApp.class)**。**

- **命名**

**如果 .java 文件包括一个公共类，那么它必须使用与那个公共类相同的文件名。例如在前例中的类的定义是**

```
public class HelloWorldapp
```

**源文件名则必须是** HelloWorldapp.java

- **类计数**

**在源文件中每次只能定义一个公共类。**

**源文件布局**

| **源文件布局** |
| --- |
| **包含三个“顶级”要素**<br>- **一个可选择的包声明**<br>- **任意数量的输入语句**<br>- **类和界面声明** |

**一个** Java **源文件可包含三个“顶级”要素：**

- **一个包声明（可选）**
- **任意数量的输入语句**

- 类和界面声明

该三要素必须以上述顺序出现。即，任何输入语句出现在所有类定义之前；如果使用包声明，则包声明必须出现在类和输入语句之前。

# 第五节 类和包介绍

**类和包介绍**

1. 什么是类和包？
2. Java 类库中的几个重要包

java.lang
java.awt
java.applet
java.net
java.io
java.util

类是描述提供某种功能的模块的一般术语。Java 开发集（JDK）给出了一套标准的类（称作类库），这些类可执行大部分所需的基本行为—— 不仅为编程任务（例如，类可提供基本的数学函数、数组和字符串），而且为图形和网络。

类库被组织成许多包，每个包都包含几个类。如下所列为一些重要的包：

java.lang 包含一些形成语言核心的类，如 String、Math、Integer 和 Thread。

java.awt 包含了构成抽象窗口工具包（AWT）的类，这个包被用来构建和管理应用程序的图形用户界面。

java.applet 包含了可执行 applet 特殊行为的类。

java.net 包含执行与网络相关的操作的类和处理接口及统一资源定位器(URLs)的类。

java.io 包含处理 I/O 文件的类。

java.util 包含为任务设置的实用程序类，如随机数发生、定义系统特性和使用与日期日历相关的函数。

# 第六节 使用 Java API 文档

**使用 Java API 文档**

1. 一组超文本标识语言(HTML)文件提供了有关 API 的信息
2. 一个包包含了对所有类信息的超链接
3. 一个类文档包括类层次、一个类描述、一组成员变量和构造函数等

一组 HTML 文件以文档的形式提供了有关 API 的信息，这个文档的布局是等级制的，因而主页列出所有的包为超链接。如果选中了一个特殊包的热链接，作为那个包成员的类将被列出。从一个包页选中一个类的热链接将提交一页有关那个类的信息。

图 1-1 显示了这样一个类文档

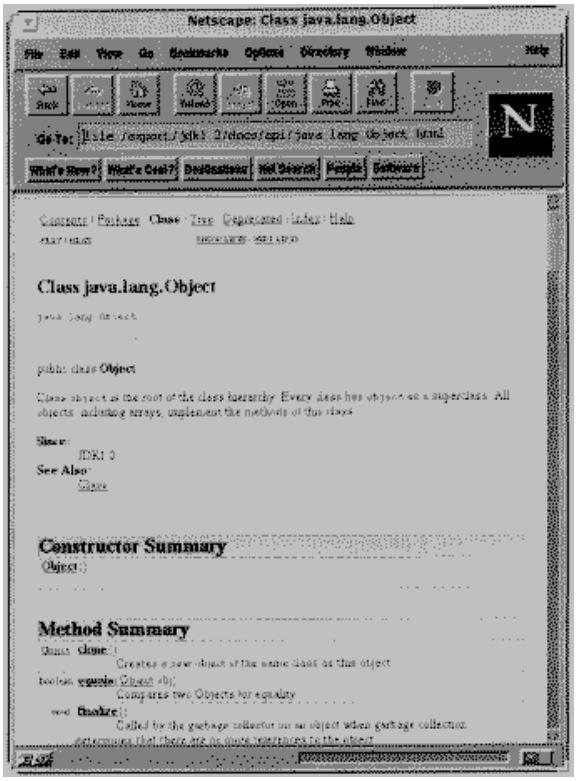



图 1-1

**一个类文档的主要部分包括：**

- 类层次
- 类和类的一般目的描述
- 成员变量列表
- 构造函数列表
- 方法列表
- 变量详细列表及目的和用途的描述
- 构造函数详细列表及描述
- 方法详细列表及描述

# 练习：基本任务

**练习目标－－在本练习里，你将利用** Java API **文档练习如何识别包、类和方法并练习标准输入输出方法。你还将编写、编译和运行两个简单的使用这些方法的应用程序。**

**一、准备**

**理解本模块中提出的概念和术语对理解文件和运用文件信息编写程序至关重要。**

**二、任务**

1 **级：阅读文件**

1.**你的老师将指导你如何启动** API **浏览器并打开** Java API **在线文档的索引页。**

2.**找到** java.lang **包**

3.**在此包中定义了那些类？在** System **类中有那些方法？**System.out.println **方法是在什么包中定义的？什么是标准输入方法调用？**

2 **级：创建一个** Java **应用程序**

1.**使用任意文本编辑器，创建一个可打印你选定的字串的与** HelloWorldApp **类似的应用程序。**

2.**编译程序并纠正错误。**

3.用解释器运行程序。

3 级：使用标准输入和标准输出

编写一个称为 MyCat 的应用程序，它将从 stdin 中读出一行并将这一行写回到 stdout. 无论 stdin 还是 stdout 都在 java.lang.System 类中。

## 三、检查你的进度

在进入下一模块之前，请确认你已经能够：

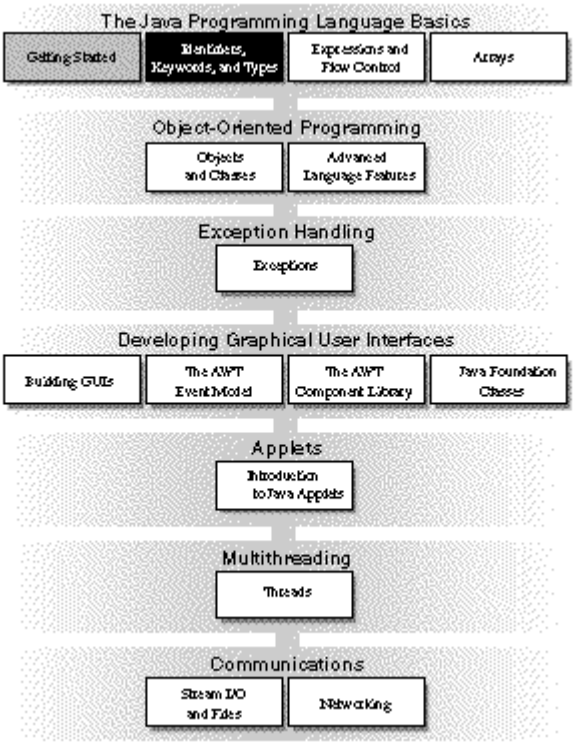
- 描述 Java 编程语言的主要特性
- 描述 JVM
- 描述垃圾收集是如何进行的
- 列出由处理代码安全性的 Java 平台所执行的三个任务
- 定义术语类、包、applets 和 application
- 编写、编译并运行一个简单的 Java 应用程序
- 使用 Java 技术应用程序编程界面(API)的在线文档识别 java.lang 包中的方法。

## 四、思考题

使用 Java 编程语言将为你的工作带来什么益处？

# 第二章 标识符、关键字和类型

本模块阐述了在 Java 技术程序中使用的基本元素，包括变量、关键字、原始类型和类类型。




## 第一节 相关问题

讨论——下列问题与本模块阐述的论题相关。

- 你是如何理解类的？
- 你是如何理解一个对象的？

## 第二节 目标

完成本模块的学习后，你应该能够：

- 在一个源程序中使用声明
- 区分有效和无效标识符
- 确认 Java 技术关键字
- 列出八个原始类型
- 为数字类型和文本类型定义文字值
- 定义术语 *class*、*object*、*member variable* 和 *reference variable*
- 为一个简单的包含原始成员变量的类创建一个类定义
- 声明类类型变量
- 使用 new 构造一个对象
- 描述缺省初始化
- 使用点符号访问一个对象的成员变量
- 描述一个参考变量的意义
- 规定分配类类型变量的结果

## 第三节 注 释

**注释**

三种允许的 Java 技术程序注释风格

```
//comment on one line
/* comment on one
or more line */
/** documenting comment */
```

### 2.3.1 概述

有三种插入注释的允许风格：

```
//comment on one line
/* comment on one or more line */
/** documenting comment */
```

紧放在声明(变量、方法或类的声明)之前的文档注释表明，注释应该被放在自动生成的文档中(由 javadoc 命令生成的 HTML 文件)以当作对声明项的描述。

注意------有关这些注释的格式和 javadoc 工具的使用，请见 JDK1.2 API 文件的 docs/tooldocs/win32 目录

### 2.3.2 分号、块和空白

**分号、块和空白**

- 一个语句是一行用分号(；) 终止的代码

```
totals=a+b+c+d+e+f；
```

- 一个块是以上括号和下括号为边界的语句集合

```
{
    x=y+1
    y=x+1
}
```

在 Java 编程语言中，语句是一行由分号(;)终止的代码。

例如

```
totals=a+b+c+d+e+f；
```

与下式相同

```
total=a+b+c+
 d+e+f；
```

一个块(block)或一个复合语句是以上括号和下括号（{}）为边界的语句集合；块语句也被用来组合属于某个类的语句。

分号、块和空白（续）

**分号、块和空白**

- 一个块可被用在一个类的定义中

```
public class Date {
   int day;
   int month;
   int year;
}
```

- 块语句可被嵌套
- Java 程序中允许任意多的空白

语句块可被嵌套。HelloWorldApp 类由 main 方法构成，这个方法就是一个语句块，它是一个独立单元，单元本身可作为在类 HelloWorldApp 块中的一组事务之一。

其它一些块语句或组的例子如下：

```
// a block statement
```

```
{
x = y + 1;
y = x + 1;
}

Semicolons, Blocks, and Whitespace
// a block used in a class definition
public class MyDate {
int day;
int month;
int year;
}

// an example of a block statement nested within
// another block statement
while ( i < large ) {
a = a + i;
if ( a == max ) {
b = b + a; // nested block is here
a = 0;
}
}
```

**在源代码元素之间允许空白，空白的数量不限。空白(包括空格、**tabs **和新行)可以改善你的对源代码的视觉感受。**

```
{
int x;
x = 23 * 54;
}

{
    int  x
    x  =  23  +  54;
}
```

## 第四节　标识符

| 标识符 |
| --- |
| - 是赋予变量、类和方法的名称 |
| - 可从一个字母、下划线(_)或美元符号($)开始 |
| - 是大小写区别对待的，且无最大长度 |

**在** Java **编程语言中，标识符是赋予变量、类或方法的名称。标识符可从一个字母、下划线(_)或美元符号($)开始，随后也可跟数字。标识符是大小写区别对待的并且未规定最大长度。**

**有效标识符如下：**

- identifier
- userName
- User_name
- _sys_varl
- $change

Java **技术源程序采用有效的** 16-bit **双字节字符编码标准**(Unicode) **，而不是** 8-bit ASCII **文本。因而，一个字母有着更广泛的定义，而不仅仅是** a **到** z **和** A **到** Z**。**

谨慎对待非 ASCII 字符，因为 Unicode 可支持看似相同而实际*不同* 的字符。

标识符不能是关键字，但是它可包含一个关键字作为它的名字的一部分。例如，thisone 是一个有效标识符，但 this 却不是，因为 this 是一个 Java 关键字。Java 关键字将在后面讨论。

---

注意——包含美元符号（$）的关键字通常用的较少， 尽管它在 BASIC 和 VAX/VMS 系统语言中有着广泛的应用。由于它们不被熟知， 因而最好避免在标识符中使用它们， 除非有本地使用上的习惯或其他不得已的原因。

---

# 第五节 Java 关键字

表 2-1 列出了使用在 Java 编程语言中的关键字。全小写

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

关键字对 Java 技术编译器有特殊的含义，它们可标识数据类型名或程序构造（construct）名。

以下是有关关键字的重要注意事项：

- true、false 和 null 为小写，而不是象在 C++语言中那样为大写。严格地讲，它们不是关键字，而是文字。然而，这种区别是理论上的。
- 无 sizeof 运算符；所有类型的长度和表示是固定的，不依赖执行。
- goto 和 const 不是 Java 编程语言中使用的关键字

# 第六节 基本 Java 类型

Java 编程语言定义了八个原始类型

- **逻辑类** boolean
- **文本类** char
- **整数类** byte, short, int, long
- **浮点类** double, float

## 2.6.1 原始类型

Java 编程语言为八个原始数据类型和一个特殊类型定义了文字值。原始类型可分为四种：

- **逻辑类** boolean
- **文本类** char
- **整数类** byte, short, int, long
- **浮点类** double, float

## 2.6.2 基本 Java 类型

### 1.逻辑类——boolean

**逻辑类——boolean**

boolean **数据类型有两种文字值：**true **和** false。

**例如：**`boolean truth = true;`

**上述语句声明变量** truth **为** boolean **类型，它被赋予的值为** true。

**逻辑值有两种状态，即人们经常使用的“** on **”和“** off **” 或“** true **”和“** false **”或“** yes **”和“** no **”，这样的值是用** boolean **类型来表示的。**boolean **有两个文字值，即** true **和** false。**以下是一个有关** boolean **类型变量的声明和初始化：**

```
boolean truth = true; //declares the variable truth
                      //as boolean type and assigns it
                      //the value true
```

---

**注意——在整数类型和 boolean 类型之间无转换计算。有些语言（特别值得强调的是 C 和 C++）允许将数字值转换成逻辑值， 这在 Java 编程语言中是不允许的；boolean 类型只允许使用 boolean 值。**

---

## 2.文本类——char 和 String

**文本类——**char **和** String

char

- **代表一个** 16-bit Unicode **字符**
- **必须包含用单引号（’’）引用的文字**
- **使用下列符号：**

  ‘a’

  ‘\t’ **一个制表符**

  ‘\u????’ **一个特殊的** Unicode **字符，**????**应严格使用四个** 16 **进制数进行替换**

**使用** char **类型可表示单个字符。一个** char **代表一个** 16-bit **无符号的（不分正负的）**Unicode **字符。一个** char **文字必须包含在单引号内**(‘’)。

‘a’

‘\t’ **一个制表符**

‘\u????’ **一个特殊的** Unicode **字符。**????**应严格按照四个** 16 **进制数字进行替换**

String **不是原始类型，而是一个类（**class**），它被用来表示字符序列。字符本身符合** Unicode **标准，且上述** char **类型的反斜线符号适用于** String**。与** C **和** C++**不同，**String **不能用** \0 **作为结束。**

**文本类——**char **和** String

String

- **不是一个原始数据类型，它是一个类**
- **具有用双引号引用的文字**

  `“The quick brown fox jumped over the lazy dog.”`
- **可按如下情形使用：**

  `String greeting = “Good Morning!! \n”;`

  `String err_meg = “ record not found !”;`

## 文本类——char 和 String (续)

String **的文字应用双引号封闭，如下所示：**

`“The quick brown fox jumped over the lazy dog.”`

Char **和** String **类型变量的声明和初始化如下所示：**

```
char ch = `A'; // declares and initializes a char variable
char ch1,ch2 ; // declares two char variables
// declare two String variables and initialize them
String greeting = "Good Morning !! \n" ;
String err_msg = "Record Not Found !" ;
String str1,str2 ; // declare two String variables
```

### 2.6.3 整数类——byte, short, int, long

**整数类**——byte, short, int, long

- **采用三种进制——十进制、八进制和** 16 **进制**

2 **十进制值是** 2

077 **首位的** 0 **表示这是一个八进制的数值**

0xBAAC **首位的** 0x **表示这是一个** 16 **进制的数值**

- **具有缺省** int
- **用字母“** L **”和“** l **”定义** long

**在** Java **编程语言中有四种整数类型，每种类型可使用关键字** byte, short, int **和** long **中的任意一个进行声明。整数类型的文字可使用十进制、八进制和** 16 **进制表示，如下所示：**

2 **十进制值是** 2

077 **首位的** 0 **表示这是一个八进制的数值**

0xBAAC **首位的** 0x **表示这是一个** 16 **进制的数值**

---

**注意——所有 Java 编程语言中的整数类型都是带符号的数字。**

---

### 整数类——byte, short, int, long（续）

**整数类文字属** int **类型，除非在其后直接跟着一个字母“** L **”。** L **表示一个** long **值。请注意，在** Java **编程语言中使用大写或小写** L **同样都是有效的，但由于小写** l **与数字** 1 **容易混淆，因而，使用小写不是一个明智的选择。上述文字的** long **的形式如下：**

2L **十进制值是** 2**，是一个** long

077L **首位的** 0 **表示这是一个八进制的数值**

0xBAACL **前缀** 0x **表示这是一个** 16 **进制的数值**

**整数类**——byte, short, int, long

**每个整数数据类型具有下列范围：**

| Integer Length | Name or Type | Range |
|---|---|---|
| 8 bits | byte | $-2^{7}$ ... $2^{7}-1$ |
| 16 bits | short | $-2^{15}$ ... $2^{15}-1$ |
| 32 bit | int | $-2^{31}$ ... $2^{31}-1$ |
| 64 bits | long | $-2^{63}$ ... $2^{63}-1$ |

### 整数类——byte, short, int, long（续）

**四个整数类型的长度和范围如表** 2-2 **所示。范围的表示是按** Java **编程语言规范定义的且不依赖于平台。**

**表** 2-2 **整数数据类型—范围**

Integer Length Name or Type Range

| | | |
|---|---|---|
| 8 bits | byte | $-2^7 \ldots 2^7-1$ |
| 16 bits | short | $-2^{15} \ldots 2^{15}-1$ |
| 32 bit | int | $-2^{31} \ldots 2^{31}-1$ |
| 64 bits | long | $-2^{63} \ldots 2^{63}-1$ |

### 2.6.4 浮点——float 和 double

浮点——float 和 double

- 缺省为 double
- 浮点文字包括小数点或下列各项之一
  - E 或 e (增加指数值)
  - F 或 f (float)
  - D 或 d （double）

3.14 一个简单的浮点值（a double）

4.02E23 一个大浮点值

2.718F 一个简单的 float 长度值

123.4E+306D 一个大的带冗余 D 的 double 值

浮点变量可用关键字 float 或 double 来声明，下面是浮点数的示例。如果一个数字文字包括小数点或指数部分，或者在数字后带有字母 F 或 f（float）、D 或 d（double），则该数字文字为浮点。

3.14 一个简单的浮点值（a double）

4.02E23 一个大浮点值

2.718F 一个简单的 float 长度值

123.4E+306D 一个大的带冗余 D 的 double 值

浮点——float 和 double

浮点数据类型具有下列范围：

| 浮点长度 | 名称或类型 |
|---|---|
| 32 bits | float |
| 64 bits | double |

浮点——float 和 double（续）

表 2—3 浮点数据类型——范围

| 浮点长度 | 名称或类型 |
|---|---|
| 32 bits | float |
| 64 bits | double |

注意——浮点文字除非明确声明为 float，否则为 double

## 第七节 变量、声明和赋值

Java 技术规范的浮点数的格式是由电力电子工程师学会（IEEE）754 定义的，它使用表 2—3 的长度，并且是独立于平台的。

下列程序显示了如何为整数、浮点、boolean、字符和 string 类型变量声明和赋值。

```
1. public class Assign {
2. public static void main(String args []) {
3. int x, y; // declare int
  // variables
```

```
4.float z = 3.414f; // declare and assign
  // float
5.double w = 3.1415; // declare and assign
  // double
6.boolean truth = true; // declare and assign
  // boolean
7.char c; // declare character
  // variable
8.String str; // declare String
9.String str1 = "bye"; // declare and assign
  // String variable
10.c = 'A'; // assign value to char
  // variable
11.str = "Hi out there!"; // assign value to
  // String variable
12.x = 6;
13.y = 1000; // assign values to int variables
14....
15.}
16.}
```

**非法赋值举例**

```
y = 3.1415926; // 3.1415926 is not an int.
          // Requires casting and decimal will
          // be truncated.
w = 175,000; // the comma symbol ( , ) cannot appear
truth = 1; // a common mistake made by ex- C / C++
     // programmers.
z = 3.14156 ; //can't fit double into a
          //Float. Requires casting.
```

# 第八节 Java 编码约定

Java **编码约定**

- **类：**
  - class AccountBook
  - class ComplexVariable
- **界面：**
  - interface Account
- **方法：**
  - balanceAccount（）
  - addComplex （）

Java **编程语言的一些编码约定是：**

*classes*——**类名应该是名词，大小写可混用，但首字母应大写。例如：**

class AccountBook

class ComplexVariable

*interface*——**界面名大小写规则与类名相同。**

interface Account

*method*——方法名应该是动词，大小写可混用，但首字母应小写。在每个方法名内，大写字母将词分隔并限制使用下划线。例如：

```
balanceAccount ()
addComplex ()
```

Java 编码约定
- 变量

currentCustomer
- 常量

HEAD-COUNT
MAXIMUM-SIZE

**Java 编码约定**

***Variables***——所有变量都可大小写混用，但首字符应小写。词由大写字母分隔，限制用下划线，限制使用美元符号（$），因为这个字符对内部类有特殊的含义。

```
currentCustomer
```

变量应该代表一定的含义，通过它可传达给读者使用它的意图。尽量避免使用单个字符， 除非是临时“ 即用即扔 ”的变量（例如，用 i, j, k 作为循环控制变量）

***constant***——原始常量应该全部大写并用下划线将词分隔；对象常量可大小写混用。

```
HEAD-COUNT
MAXIMUM-SIZE
```

***control structures***——当语句是控制结构的一部分时，即使是单个语句也应使用括号（{} ）将语句封闭。例如：

```
if (condition) {

   do something
} else   {
   do something else
}
```

***spacing***—— 每行只写一个语句并使用四个缩进的空格使你的代码更易读。

***comments***——用注释来说明那些不明显的代码段落；对一般注释使用 // 分隔符， 而大段的代码可使用 /* ··· */分隔符。使用 /** ··· */将注释形成文档，并输入给 javadoc 以生成 HTML 代码文档。

```
// A comment that takes up only one line.
/* Comments that continue past one line and take up space on multiple lines...*/
/** A comment for documentation purposes.
@see Another class for more information
*/
```

---

注意——@see 是一个有关类或方法的特殊的 javadoc 标记符（”see also”）。有关 javadoc 的详细资料， 请参见“ The Design of Distributed Hyperlinked Programming Documentation”(Lisa 著)的有关文档系统的完整定义。该资料可从下列地址获得：http://www.javasoft.com/doc/api_documentation.html.

---

## 第九节 理解对象

**理解对象**

- **回顾对象的历史**
- **创建一个新的类型，如** date

```
public class date {
    int day;
    int month;
    int year;
}
```

- **声明一个变量**

```
Date myBirth, yourBirth
```

- **访问成员**

```
myBirth.day  = 26;
myBirth.month = 11;
yourBirth.year = 1960;
```

### 2.9.1 回顾对象的历史

早些时候的编程语言和初级程序员将每个变量看作相互无关的实体。例如，如果一个程序需处理某个日期，则要声明三个单独的整数：

```
int day, month, year;
```

上述语句作了两件事，一是当程序需要日、月或年的有关信息时，它将操作一个整数； 二是为那些整数分配存储器。

尽管这种作法很容易理解，但它存在两个重大缺陷。首先，如果程序需同时记录几个日期，则需要三个不同的声明。例如，要记录两个生日，你可能使用：

```
int  myBirthDay, myBirthMonth, myBirthYear;
int  yourBirthDay, yourBirthMonth, yourBirthYear;
```

这种方法很快就会引起混乱，因为需要的名称很多。

第二个缺陷是这种方法忽视了日、月和年之间的联系并把每个变量都作为一个独立的值，每个变量都是一个独立单元(在本例中为 date)的一部分并被相应地处理。

### 2.9.2 创建一个新类型

为克服上述两种缺陷，Java 编程语言使用类来创建新类型。请看下列原始类型声明：

```
int  day;
```

Java 编程语言被用来分配一定量的存储器并解释该存储器的内容。于是，要定义一个新的类型，你必须指出需要多大存储器和如何解释存储器内容。这不是根据字节数或位的顺序和含义来做，而是根据已经定义的其它类型来做。

例如，要定义一个表示日期的类型，你需要足够的存储器存储三个整数变量；进而，日、月和年的意义即由这些整数变量给出。如下所示：

```
class MyDate {
int day;
int month;
int year;
}
```

词 class 是 Java 编程语言的一个关键字，必须全部小写。名称 MyDate 按照大小写的有关约定处理，而不是由语意要求来定。

---

注意----- ***class*** 不仅仅是一个集合数据类型，这个问题以后还将进一步讨论。

---

一个变量可被声明为归属于类型 MyDate，从而日、月和年部分将被隐含声明。例如：

```
MyDate  myBirth, yourBirth;
```

使用这个声明，Java 编程语言允许变量的部分(day, month 和 year)通过调用成员和使用点(．)运算符而被访问。例如：

```
myBirth.day  =  26;
myBirth.month = 11;
yourBirth.year = 1960;
```

### 2.9.3 创建一个对象

**创建一个对象**

- 原始类型的声明可分配存储器空间
- 非原始类型的声明不分配存储器空间
- 声明的变量不是数据本身，而是数据的引用(或指针)

当任何原始类型(如 boolean, byte, short,char,int,long,float 或 double 类型) 的变量被声明时，作为上述操作的一部分，存储器空间也同时被分配。使用非原始类型(如 String 或 class)的变量的声明不为对象分配存储器空间。

事实上，使用 class 类型声明的变量不是数据本身，而是数据的引用(reference)。

---

注意----你也可以认为引用是一个指针(pointer)，这可能会有助于你的理解。实际上，在大多数实现中，也确实可以这样认为。值得注意的是，Java 编程语言实际上不支持指针数据类型。

---

在你可以使用变量之前，实际存储器必须被分配。这个工作是通过使用关键字 new 来实现的。如下所示：

```
MyDate myBirth;
myBirth = new MyDate ();
```

第一个语句(声明)仅为引用分配了足够的空间，而第二个语句则通过调用对象为构成 MyDate 的三个整数分配了空间。对象的赋值使变量 myBirth 重新正确地引用新的对象。这两个操作被完成后，MyDate 对象的内容则可通过 myBirth 进行访问。

假使定义任意一个 class XXXX, 你可以调用 new XXXX () 来创建任意多的对象，对象之间是分隔的。一个对象的引用可被存储在一个变量里，因而一个"变量点成员"(如 myBirth.day)可用来访问每个对象的单个成员。请注意在没有对象引用的情况下，仍有可能使用对象，这样的对象称作"匿名"对象。

### 2.9.4 创建一个对象----存储器分配和布局

**创建一个对象----存储器分配和布局**

一个声明仅为一个引用分配存储器

```
MyDate today
today = new MyDate();
```

today ????

在一个方法体中，声明

```
MyDate today
today = new MyDate();
```

仅为一个引用分配存储器

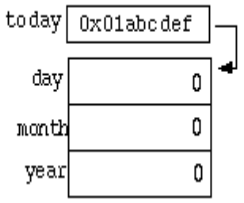
today ????

关键字 new 意味着存储器的分配和初始化

```
MyDate today;
today = new MyDate();
```

赋值则建立了引用变量，从而它可适当地引用新的创建对象

```
MyDate today;
 today = new MyDate();
```




使用一个语句同时为引用 today 和由引用 today 所指的对象分配空间也是可能的。

```
MyDate today = new MyDate ();
```

### 2.9.5 引用类型的赋值

**引用变量的赋值**

请考虑下列代码片段：

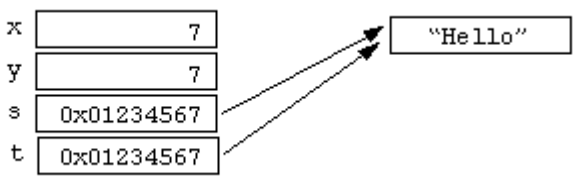
```
int x = 7;
int y = x;
String s = “Hello”;
String t = s;
```

在 Java 编程语言中，用类的一个类型声明的变量被指定为引用类型，这是因为它正在引用一个非原始类型，这对赋值具有重要的意义。请看下列代码片段：

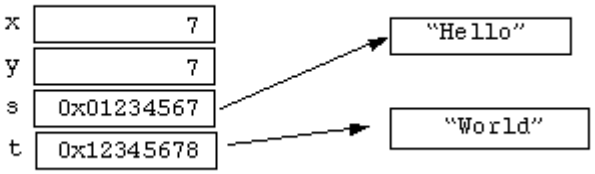
```
int x = 7;
int y = x;
String s = “Hello”;
String t = s;
```

四个变量被创建：两个原始类型 int 和两个引用类型 String。x 的值是 7，而这个值被复制到 y；x 和 y 是两个独立的变量且其中任何一个的进一步的变化都不对另外一个构成影响。

至于变量 s 和 t，只有一个 String 对象存在，它包含了文本”Hello”，s 和 t 均引用这个单一的对象。



将变量 t 重新定义，则新的对象 World 被创建，而 t 引用这个对象。上述过程被描述如下：




### 2.9.6 术语回顾

**术语回顾**

- Class
- Object
- Reference type
- member

本模块中介绍了几个术语，简单回顾如下：

- 类------在 Java 编程语言中定义新类型的一种途径。类声明可定义新类型并描述这些类型是如何实现的。有许多关于类的其它特性还未讨论。
- 对象------一个类的实例。类可被认为是一个模板------你正在描述一个对象模型。一个对象就是你每次使用 new 创建一个类的实例的结果。
- 引用类型------一个用户定义类型，它可引用类、界面和数组。
- 成员------构成一个对象的元素之一。这个词也被用作定义类的元素(elements)。成员变量(member variabl)、实例变量(instance variable)和域(field)也经常被互换使用。

## 练习：使用对象

练习目标——正确使用 Java 关键字，编写一个创建类的程序并从类创建一个对象。编译并运行程序，然后校验引用是否被赋值并检查引用是不是按照本模块所描述的那样操作的。

**一、准备**

为了成功地完成本练习，你必须能够编译并运行 Java 程序，并且需要熟悉有关类和对象的面向对象的概念和引用的概念。

**二、任务**

1 级：创建一个类和相应的对象

1. 一个点可用 x 和 y 坐标描述。定义一个称为 MyPoint 的类来表达上述想法。你应该称这个文件为什么?
2. 在你的类中编写一个类方法，然后为类型 MyPoint 声明两个变量，将变量称为 start 和 end；用 new MyPoint()创建对象并分别将引用值赋予变量 start 和 end；
3. 将值 10 赋予对象 start 的成员 x 和 y；
4. 将值 20 赋予对象 end 的 x 值，将值 20 赋予对象 end 的 y 值。
5. 分别打印 MyPoint 对象(start 和 end)的成员值(x 和 y)。

---

注意----为完成第 5 步，你需要更多地了解 system 类。带自变量 String 的 System.out.println()可输出 string 并开始新的一行，而 System.out.print()不开始新的一行。如果你使用 System.out.print()，你应该在这个应用程序结束之前，调用 System.out.println()或 System.out.flush()，否则你会发现，输出

的最后一行未被显示出来。

---

为显示数字，你可以使用下列形式(稍后将在本课程中讨论)

```
System.out.println("Start MyPoint = x："+start.x + " y " + start.y);
```

---

注意--如果一个操作数已经是一个 String，则加号(+)将另一个操作数转换成 String。

---

6. 编译并运行程序。

2 级：检验引用赋值

使用你在前一个练习中 MyPoint 类，增加代码到 main()方法，以完成下列事项：

1. 为类型 MyPoint 声明一个新的变量，称之为 stray。将现存变量 end 的引用值赋予 stray；
2. 打印 end 和 stray 变量的成员 x 和 y 的值；
3. 赋予变量 stray 的成员 x 和 y 新的值；
4. 打印 end 和 stray 的成员的值；编译并运行 MyPoint 类。end 的值反映了 stray 内的变化，表明两个变量都引用了同一个 MyPoint 对象；
5. 将 start 变量的成员 x 和 y 赋予新的值；
6. 打印 start 和 end 的成员值; 再次编译并运行 MyPoint 类，start 的值仍然独立于 stray 和 end 的值，表明 start 变量仍然在引用一个 MyPoint 对象，而这个对象与 stray 和 end 引用的对象是不同的。

## 三、练习小结

讨论——用几分钟的时间讨论一下在以上练习中你所获得的经验、感想和发现。

经验 解释 总结 应用

## 四、检查你的进度

在进入下一模块之前，请确认你已经能够：

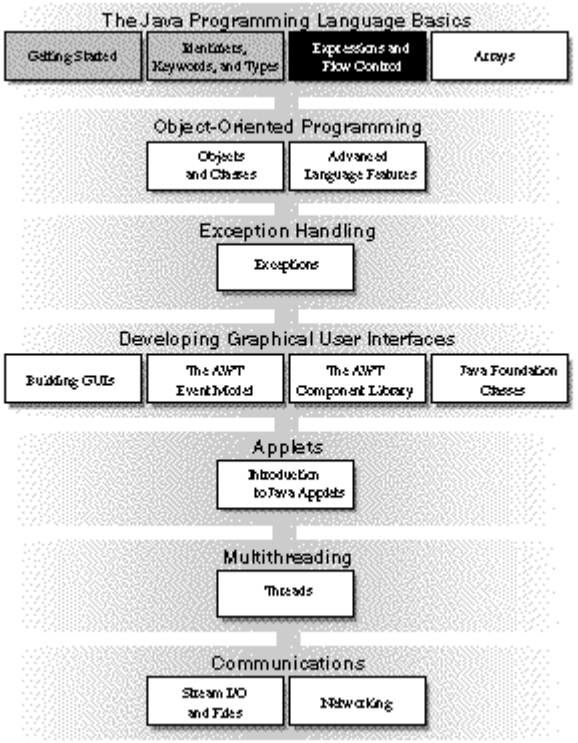
- 在源程序中使用注释
- 区分有效和无效标识符
- 识别 Java 技术关键字
- 列出八个原始类型
- 为数字和文本类型定义文字值
- 定义术语 *class, object,member,variable,referrence variable*
- 为一个简单的包含原始成员变量的类创建一个类定义
- 声明类类型的变量
- 使用 new 构造一个对象
- 描述缺省初始化
- 使用点符号访问一个对象的成员变量
- 描述引用变量的意义
- 陈述分配类类型变量的结果

## 五、思考题

在你现有应用程序中，你可以构思一个使用类和对象的例子吗

# 第三章 表达式和流程控制

本模块讨论变量、运算符和算术表达式并列出不同的管理运行程序路径的控制结构。




## 第一节 相关问题

讨论——下列问题与本模块阐述的论题相关。

- 什么类型的变量对程序员有用(例如,其它语言的程序员想了解 Java 编程语言是如何定义和处理全局变量和局部变量的) ？
- 复合类可以有同名的变量吗？如果可以，它们的作用域有多大？
- 列出用于其它语言的控制结构；一般语言都采用什么方法进行流程控制和中断流程(如在循环或开关语句中) ？

## 第二节 目 标

完成本模块的学习后，你应该能够：

- 区分实例变量和局部变量；
- 描述实例变量是如何被初始化的；
- 确认并更正编译器错误；
- 辨认、描述并使用 Java 软件运算符；
- 区分合法和非法原始类型赋值；
- 确认 boolean 表达式和它们在控制构造中的要求；
- 辨认赋值兼容性和在基本类型中的必要计算；
- 使用 if, switch,for,while 和 do 句型结构和 break 和 continue 的标号形式作为程序中的流程控制结构。

## 第三节 表达式

### 3.3.1 变量和作用域

**变量和作用域**

- 局部(local)变量是在一个方法内定义的变量，也被称作自动（*automatic*）、临时（*temporary*）或栈（*stack*）变量
- 当一个方法被执行时，局部变量被创建；当一个方法被终止时，局部变量被清除
- 局部变量必须使用之前初始化，否则编译时将出错

你已经看到了两种变量的描述方法：原始类型变量和引用类型变量。你也看到了变量被声明的位置：在方法内(方法是面向对象的术语，它可引用函数或子程序，例如：main())或在方法外但在类定义之内。变量也可被定义为方法参数或构造函数参数。

在方法内定义的参数被称为局部(local)变量，有时也被用为自动（*automatic*）、临时（*temporary*）或栈（*stack*）变量。

在方法外定义的变量是在使用 new Xxxx ()调用构造一个对象时被创建。在构造一个对象时，可创建两种变量。一是类变量，它是用 static 关键字来声明的；只要需要对象，类变量就将存在。二是实例变量，它不需用 static 关键字来声明；只要对象被当作引用，实例变量就将存在。实例变量有时也被用作成员变量，因为它们是类的成员。

方法参数变量定义在一个方法调用中传送的自变量，每次当方法被调用时，一个新的变量就被创建并且一直存在到程序的运行跳离了该方法。

当执行进入一个方法时，局部变量被创建，当执行离开该方法时，局部变量被取消。因而，局部变量有时也被引用为“临时或自动”变量。在成员函数内定义的变量对该成员变量是“局部的”，因而，你可以在几个成员函数中使用相同的变量名而代表不同的变量。 该方法的应用如下所示：

```
class OurClass {
   int i; // instance variable of class OurClass
   int firstMethod() {
       int j=0; // local variable
             // both i and j are accessible from
             // this point
             ...
             return 1;
       } // end of firstMethod()
   int secondMethod(float f) { //method parameter
        int j=0; //local variable. Different from the
              // j defined in firstMethod().
              // Scope is limited to the body of
             // secondMethod().
              // Both i(instance variable of the
              // class OurClass) and j (local
              // variable of this method) are
              // accessible from this point.
             ....
             return 2;
       } // end of secondMethod()
    } // end of class OurClass
```

### 3.3.2 变量初始化

在 Java 程序中，任何变量都必须经初始化后才能被使用。当一个对象被创建时，实例变量在分配存储器的同时被下列值初始化：

```
byte
            0
 short
            0
 int
            0
 long
            0L
 float
            0.0f
 double
            0.0d
 char
            '\u0000' (NULL)
 boolean
            false
 All reference types
            Null
```

---

注意—— 一个具有空值的引用不引用任何对象。试图使用它引用的对象将会引起一个异常。异常是出现在运行时的错误，这将在模块 7“异常”中讨论。

---

在方法外定义的变量被自动初始化。局部变量必须在使用之前做“手工”初始化。如果编译器能够确认一个变量在初始化之前可被使用的情形，编译器将报错。

```
public void doComputation() {
int x = (int)(Math.random() * 100);
int y;
int z;
if (x > 50) {
y = 9;
}
z = y + x; // Possible use before initialization
}
```

### 3.3.3 运算符

Java 软件运算符在风格和功能上都与 C 和 C++极为相似。表 3-1 按优先顺序列出了各种运算符（“L to R”表示左到右结合，“R to L”表示右到左结合）

```
 Separator  []   ()   ;   ,

 R to L  ++  -- + - ~ ! (data type)
 L to R  *  /  %
 L to R  +  -
 L to R  <<  >>  >>>
 L to R  <  >  <=  >= instanceof
 L to R  ==  !=
```

```
L to R  &
L to R  ^
L to R  |
L to R  &&
L to R  ||
R to L  ?:
R to L   =   *=   /=   %=   +=   -=    <<=   >>=   >>>=   &=   ^=   |=
```

---

**注意——instanceof 是 Java 编程语言特有的运算符，将在模块 5“对象和类”中讨论**

---

### 3.3.4 逻辑表达式

**逻辑表达式**

**支持的位运算符如下：**

**!**——NOT　　&——AND

|——XOR　　^——OR

**布尔运算符如下：**

~ ——Complement(**取反**)　　&——AND

^——XOR　　|——OR

**多数 Java 运算符是从其它语言借取得并且和人们一般期待的功能一样。**

**关系和逻辑运算符返回布尔结果，int 到 boolean 不能自动转换。**

```
int  i = 1;
if ( i ) //generates a compile error
if (i !=0) // Correct
```

**支持的位运算符是！、&、|、^， 支持的布尔逻辑运算符是 NOT、AND、XOR 和 OR。每个运算都返回布尔结果。运算符&&和||是运算符&和|的布尔等价物。布尔逻辑运算符将在下一页中讨论。**

### 3.3.5 位运算

**Java 编程语言支持整数数据类型的位运算，它们的运算符~、&、^和|分别表示位运算的 NOT（为求反）、位 AND、位 XOR 和位 OR。移位运算符将稍后在本课程中讨论。**

### 3.3.6 布尔逻辑运算符

**布尔逻辑运算符**

- **运算符是&&(AND)和||（OR）**
- **运算符使用举例：**

  **见 3-11 页程序**

**运算符&& (定义为 AND)和||（定义为 OR）执行布尔逻辑表达式。请看下面的例子：**

```
MyDate d = null;
if ((d! = null) && (d,day() > 31))  {
// do something with d
}
```

**形成 if ()语句自变量的布尔表达式是合法且安全的。这是因为当第一个子表达式是假时，第二个子表**

达式被跳过，而且当第一个子表达式是假时，整个表达式将总是假，所以不必考虑第二个子表达式的值。类似的，如果使用||运算符，而且第一个表达式返回真， 则第二个表达式不必求值，因为整个表达式已经被认为是真。

### 3.3.7 用加号（+）进行串链接

**用加号（+）进行串链接**

运算符 + 能够：

- 进行 String 链接
- 生成一个新的 String

例如：

```
String salutation = "Dr. ";
String name = "Jack " + "Arthur";
String title = salutation + name;
```

最后一行的结果是：

```
Dr. Jack Arthur
```

一个自变量必须是 String 对象

非串被自动转换为 String 对象

运算符 + 能够进行 String 对象的链接并生成一个新的 String：

```
String salutation = "Dr. ";
String name = "Jack " + "Arthur";
String title = salutation + name;
```

最后一行的结果是：

```
Dr. Jack Arthur
```

如果+运算符中有一个自变量为 String 对象，则其它自变量将被转换成 String。所有对象都可被自动转换成 String，尽管这样做的结果可能是意义含糊的。不是串的对象是通过使用 toString() 成员函数而转换成串的等价物的。

### 3.3.8 右移位运算符>>和>>>

**右移位运算符>>和>>>**

算术或符号右移位如下所示：

见 3-13 页

- 在移位的过程中， 符号位被拷贝

逻辑或非符号右移位运算符(>>>)

- 作用于位图
- 在移位的过程中， 符号位不被拷贝

Java 编程语言提供了两种右移位运算符

运算符>>进行算术或符号右移位。移位的结果是第一个操作数被 2 的幂来除，而指数的值是由第二个数给出的。例如：

```
128 >> 1 gives 128/2 1 = 64
256 >> 4 gives 256/2 4 = 16
-256 >> 4 gives -256/2 4 = -16
```

逻辑或非符号右移位运算符>>>主要作用于位图，而不是一个值的算术意义；它总是将零置于最重要的位上。例如：

```
1010 ... >> 2 gives 111010 ...
1010 ... >>> 2 gives 001010 ...
```

在移位的过程中， >>运算符使符号位被拷贝。

---

注意——移位运算符将它们右侧的操作数模 32 简化为 int 类型左侧操作数，模 64 简化为 long 类型右侧操作数。因而，任何 int x，x >>> 32 都会导致不变的 x 值，而不是你可能预计的零。

---

---

注意------值得称赞的重要一点是：>>>运算符仅被允许用在整数类型，并且仅对 int 和 long 值有效。如果用在 short 或 byte 值上，则在应用>>>之前，该值将通过带符号的向上类型转换被升级为一个 int。有鉴于此，无符号移位通常已成为符号移位。

---

### 3.3.9 左移位运算符(<<)

| 左移位运算符(<<) |
| --- |
| 128 << 1 等于 $128*2^1 = 256$ |
| 16 <<2 等于 $16*2^2 =64$ |

运算符<<执行一个左移位。移位的结果是：第一个操作数乘以 2 的幂，指数的值是由第二个数给出的。例如：

```
128 << 1 gives 128*2^1 = 256
 16  <<2 gives 16*2^2 =64
```

### 3.3.10 类型转换

**类型转换**

- 如果赋值的信息丢失，程序员必须采用类型转换的方法确认赋值
- 在 short 和 char 之间的赋值需要一个显式转型

见 3-16 程序

在赋值的信息可能丢失的地方，编译器需要程序员用类型转换（typecast）的方法确认赋值。例如，它可以“挤压”一个 long 值到一个 int 变量中。显式转型做法如下：

```
long  bigValue =99L;
int  squashed = (int)  (bigValue) ;
```

在上述程序中，期待的目标类型被放置在圆括号中，并被当作表达式的前缀，该表达式必须被更改。一般来讲，建议用圆括号将需要转型的全部表达式封闭。否则，转型操作的优先级可能引起问题。

---

注意——重温 short 的范围是：$-2^{15}$ 至 $2^{15}-1$；char 的范围是：0 至 $2^{15}-1$。
因而在 short 和 char 之间的赋值总需要一个显式转型

---

### 3.3.11 升级和表达式的类型转换

**升级和表达式的类型转换**

- 变量被自动升级为一个较长的形式（如：int 至 long 的升级）
- 如果变量类型至少和表达式类型一样大（位数相同），则表达式是赋值兼容的。

当没有信息丢失时，变量可被自动升级为一个较长的形式（如：int 至 long 的升级）

```
long bigval = 6; // 6 is an int type, OK
int smallval = 99L; // 99L is a long, illegal
double z = 12.414F; // 12.414F is float, OK
float z1 = 12.414; // 12.414 is double, illegal
```

一般来讲，如果变量类型至少和表达式类型一样大（位数相同），则你可认为表达式是赋值兼容的。

对 + 运算符来说，当两个操作数是原始数据类型时，其结果至少有一个 int，并且有一个通过提升操作数到结果类型、或通过提升结果至一个较宽类型操作数而计算的值，这可能会导致溢出或精度丢失。例如：

```
short  a，b，c
    a=1；
    b=2；
    c= a+b；
```

上述程序会因为在操作 short 之前提升每个 short 至 int 而出错。然而，如果 c 被声明为一个 int，或按如下操作进行类型转换：

```
    c = (short)(a+b);
```

则上述代码将会成功通过。

# 第四节　分支语句

**分支语句**

if, else 语句

```
    if（布尔表达式）{
       语句或块；
      }
    if（条件为真）  {
       语句或块；
      }   else   {
       语句或块；
      }
```

条件语句使部分程序可根据某些表达式的值被有选择地执行。Java 编程语言支持双路 if 和多路 switch 分支语句。

### 3.4.1　if, else 语句

if, else 语句的基本句法是：

```
 if（布尔表达式）{
    语句或块；
  }   else   {
    语句或块；
  }
```

例：

```
int count;
```

```
1.count = getCount(); // a method defined in the program
2.if (count < 0) {
3.System.out.println("Error: count value is negative.");
4.}else {
5.System.out.println("There will be " + count +
6." people for lunch today.");
7.}
```

在 Java 编程语言中，if ()用的是一个布尔表达式，而不是数字值，这一点与 C/C++不同。前面已经讲过，布尔类型和数字类型不能相互转换。因而，如果出现下列情况：

```
if (x) // x is int
```

你应该使用下列语句替代：

```
if (x ! = 0)
```

全部 else 部分是选择性的，并且当测试条件为假时如不需做任何事，else 部分可被省略。

### 3.4.2 switch 语句

**分支语句**

switch 语句

switch 语句的句法是：

```
switch (expr1) {
    case expr2:
    statements;
    break;
    case expr3:
    statements;
    break;
    default:
    statements;
    break;
}
```

switch 语句的句法是：

```
switch (expr1) {
    case expr2:
    statements;
    break;
    case expr3:
    statements;
    break;
    default:
    statements;
    break;
}
```

---

注意——在 switch (expr1) 语句中，expr1 必须与 int 类型是赋值兼容的；byte，short 或 char 类型可被升级；不允许使用浮点或 long 表达式。

---

当变量或表达式的值不能与任何 case 值相匹配时，可选缺省符（default）指出了应该执行的程序代码。如果没有 break 语句作为某一个 case 代码段的结束句，则程序的执行将继续到下一个 case，而不检查 case 表达式的值。

例 1：

```
1.switch (colorNum) {
2.case 0:
3.setBackground(Color.red);
4.break;
5.case 1:
6.setBackground(Color.green);
7.break;
8.default:
9.setBackground(Color.black);
10.break;
11.}
```

例 2：

```
1.switch (colorNum) {
2.case 0:
3.setBackground(Color.red);
4.case 1:
5.setBackground(Color.green);
6.default:
7.setBackground(Color.black);
8.break;
9.}
```

例 2 设定背景颜色为黑色，而不考虑 case 变量 colorNum 的值。如果 colorNum 的值为 0，背景颜色将首先被设定为红色，然后为绿色，再为黑色。

# 第五节　循环语句

**循环语句**

for 语句

```
for (init_expr; boolean testexpr; alter_expr) {
     statement or block
  }
```

循环语句使语句或块的执行得以重复进行。Java 编程语言支持三种循环构造类型：for, while 和 do loops。for 和 while 循环是在执行循环体之前测试循环条件，而 do loops 是在执行完循环体之后测试循环条件。这就意味着 for 和 while 循环可能连一次循环体都未执行， 而 do loops 将至少执行一次循环体。

### 3.5.1　for 循环

for 循环的句法是：

```
for (init_expr; boolean testexpr; alter_expr) {
statement or block
}
```

例如：

```
for (int i = 0; i < 10; i++) {
    System.out.println("Are you finished yet?");
}
System.out.println("Finally!");
```

---

注意——Java 编程语言允许在 for () 循环结构中使用逗号作为分隔符。 例如，for (i= 0, j = 0; j<10; i++, j++)是合法的；它将 i 值初始化为零，并在每执行完一次循环体后，增加一次它们的值。

---

### 3.5.2 while 循环

| 循环语句 |
| --- |
| while 循环<br>while （布尔表达式） {<br>语句或块<br>} |

while 循环的句法是：

```
while （布尔表达式） {
语句或块
}
```

例如：

```
int i = 0;
while (i < 10) {
    System.out.println("Are you finished yet?");
    i++;
}
System.out.println("Finally!");
```

请确认循环控制变量在循环体被开始执行之前已被正确初始化，并确认循环控制变量是真时，循环体才开始执行。控制变量必须被正确更新以防止死循环。

### 3.5.3 do 循环

| 循环语句 |
| --- |
| do/while 语句<br>do {<br>语句或块；<br>}<br>while (布尔测试) |

do 循环的句法是：

```
do {
语句或块；
}
while (布尔测试)
```

例如：

```
int i = 0;
do {
```

```
    System.out.println("Are you finished yet?");
    i++;
} while (i < 10);
System.out.println("Finally!");
```

象 while 循环一样，请确认循环控制变量在循环体中被正确初始化和测试并被适时更新。

作为一种编程惯例，for 循环一般用在那种循环次数事先可确定的情况，而 while 和 do 用在那种循环次数事先不可确定的情况。

# 第六节　特殊循环流程控制

**特殊循环流程控制**

- break ［标注］；
- continue［标注］；
- label: 语句；// where statement must be any
  // legal statement.

下列语句可被用在更深层次的控制循环语句中：


- break ［标注］；
- continue［标注］；
- label: 语句；// where statement must be any
  // legal statement.


break 语句被用来从 switch 语句、loop 语句和预先给定了 label 的块中退出。

continue 语句被用来略过并跳到循环体的结尾。

label 可标识控制需要转换到的任何有效语句，它被用来标识循环构造的复合语句。

例如

```
loop: while (true) {
    for (int i=0; i < 100; i++) {
      switch (c = System.in.read()) {
         case -1:
         case ` \n ` :
            // jumps out of while-loop to line #12
            break loop;
         ....
      }
      } // end for
} // end while

test: for (...) {
       ....
          while (...) {
              if (j > 10) {
                  // jumps to the increment portion of
```

```
                // for-loop at line #13
                continue test;
              }
            } // end while
          } // end for
```

## 练习：使用表达式

**练习目标——你将编写、编译并运行两个使用标识符、表达式和控制结构的算法程序。**

**一、准备**

**为成功地完成本练习，你必须具备编译和运行** Java **程序的能力，并且熟悉流程控制构造。**

**二、任务**

1 **级：创建一个阶乘应用程序**

**一个数** X **的阶乘（通常记作** X!**）等于** X*(X-1)*(X-2)……*1**。例如** 4!**等于** 4×3×2×1=24**。**

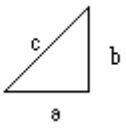
**创建一个称作** Factor **的应用程序，利用该应用程序可打印** 2**，**4**，**6 **和** 10 **的阶乘。**

2 **级：求解一个几何题程序**

**已知一个直角三角形，其弦（最长边）的长度由下列公式给出：**

**编写一个称作** hypotenuse **的** Java **软件程序，从已知直角三角形的直角边计算最长边。**

$$c = \sqrt{a^2 + b^2}$$




**提示：从** mod3/templates **目录中提供的模板解决方案入手，从命令行输入；同时注意** java.lang.Math **类。**

**三、练习小结**

**讨论——用几分钟的时间讨论一下在以上练习中你所获得的经验、感想和发现。**

**经验 解释 总结 应用**

**四、检查你的进度**

**在进入下一模块之前，请确认你已经能够：**

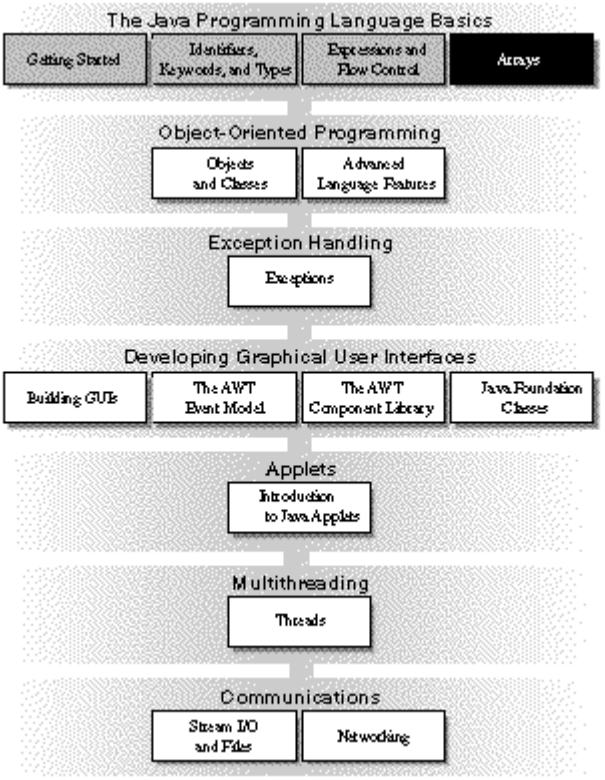
- **区分实例变量和局部变量；**
- **描述实例变量是如何被初始化的；**
- **确认并更正“可能的赋值前的引用”的编译器错误；**
- **辨认、描述并使用** Java **软件运算符；**
- **区分合法和非法原始类型赋值；**
- **确认** boolean **表达式和它们在控制构造中的要求；**
- **辨认赋值兼容性和在基本类型中的必要转型；**
- **使用** if, switch,for,while **和** do **句型结构和** break **和** continue **的标注形式作为程序的流程控制结构。**

**五、思考题**

- **多数编程语言都使用什么数据类型来集合相似的数据元素？**
- **你怎样对一组元素进行相同的操作（如一个数组）？**
- Java **编程语言使用什么数据类型？**

# 第四章 数 组

本模块将描述 Java 编程语言中如何定义、初始化和使用数组。




## 第一节 相关问题

讨论——下列问题与本模块阐述的论题相关：

- 一个数组的用途是什么？

## 第二节 目 标

完成本模块的学习后，你应该能够：

- 声明并创建原始数组、类数组或数组类型
- 解释为什么数组的元素需初始化
- 给出数组定义并初始化数组元素
- 确定一个数组中元素的数量
- 创建多维数组
- 编写从一个数组类型到另一个数组类型数组值的拷贝代码

## 第三节 数组的声明

**声明数组**

- 相同类型的成组数据对象
- 原始类型或类类型数组声明
- 为一个引用创建空间
- 数组是一个对象，而不是为原始类型储备的存储器

典型的数组是用来集合相同类型的对象并通过一个名称来引用这个集合。

你可以声明任何类型的数组——原始类型或类类型：

```
char s [ ] ;
Point p [ ]; // where point is a class
```

在 Java 编程语言中，即使数组是由原始类型构成，甚或带有其它类类型，数组也是一个对象。声明不能创建对象本身，而创建的是一个引用，该引用可被用来引用数组。数组元素使用的实际存储器可由 new 语句或数组初始化软件动态分配。

在以下部分，你将看到如何创建和初始化实际数组。

上述这种将方括号置于变量名之后的声明数组的格式，是用于 C、C++和 Java 编程语言的标准格式。这种格式会使声明的格式复杂难懂，因而，Java 编程语言允许一种替代的格式，该格式中的方括号位于变量名的左边：

```
char [ ] s;
Point [ ] p;
```

这样的结果是，你可以认为类型部分在左，而变量名在右。上述两种格式并存，你可选择一种你习惯的方式。声明不指出数组的实际大小。

---

注意----当数组声明的方括号在左边时，该方括号可应用于所有位于其右的变量

---

# 第四节 创建数组

**创建数组**

使用关键字 new 创建一个数组对象

```
s = new char [20];
p = new Point [100];
p[0] = new Point();
p[1] = new Point();
  .
  .
  .
```

你可以象创建对象一样，使用关键字 new 创建一个数组。

```
s = new char [20];
p = new Point [100];
```

第一行创建了一个 20 个 char 值的数组，第二行创建了一个 100 个类型 Point 的变量。r 然而，它并不创建 100 个 Point 对象；创建 100 个对象的工作必须分别完成如下：

```
p[0] = new Point();
p[1] = new Point();
  .
  .
  .
```

用来指示单个数组元素的下标必须总是从 0 开始,并保持在合法范围之内－－大于 0 或等于 0 并小于数组长度。任何访问在上述界限之外的数组元素的企图都会引起运行时出错。下面还要谈到一些更好的数组初始化方法。

# 第五节 初始化数组

**初始化数组**

- 初始化一个数组元素
- 用初始化值创建一个数组

```
String names [] = {
  “Georgianna”,
  “Jen”,
  “Simon”,
 };
```

当创建一个数组时，每个元素都被初始化。在上述 char 数组 s 的例子中，每个值都被初始化为 0 (\u0000-null)字符；在数组 p 的例子中，每个值都被初始化为 null，表明它还未引用一个 Point 对象。在经过赋值 p[0] = new Point()之后，数组的第一个元素引用为实际 Point 对象。

---

注意－－所有变量的初始化(包括数组元素)是保证系统安全的基础，变量绝不能在未初始化状态使用。

---

Java 编程语言允许使用下列形式快速创建数组：

```
String names [] = {
 “Georgianna”,
 “Jen”,
 “Simon”,
};
```

其结果与下列代码等同：

```
String names [] ;
names = new String [3];
names [0] = “Georgianna”;
names [1] = “Jen”;
names [2] = “Simon”;
```

这种”速记”法可用在任何元素类型。例如：

```
Myclass array [] = {
  new Myclass (),
  new Myclass (),
  new Myclass ()
 };
```

适当的类类型的常数值也可被使用：

```
Color palette  [] = {
   color.blue,
   color.red,
   color.white
 };
```

# 第六节　多维数组

**多维数组**

**数组的数组**

```
int twoDim [][] = new int [4][];
twoDim[0] = new int[5];
twoDim[1] = new int[5];

int twoDim [][] = new int [][4]; 非法
```

每个数组有 5 个整数类型的 4 个数组的数组

```
int twoDim [][] = new int [4][5];
twoDim[0] = new int[5];
twoDim[1] = new int[5];
```

Java 编程语言没有象其它语言那样提供多维数组。因为一个数组可被声明为具有任何基础类型，所以你可以创建数组的数组(和数组的数组的数组，等等)。一个二维数组如下例所示：

```
int twoDim [][] = new int [4][];
twoDim[0] = new int[5];
twoDim[1] = new int[5];
```

首次调用 new 而创建的对象是一个数组，它包含 4 个元素，每个元素对类型 array of int 的元素都是一个 null 引用并且必须将数组的每个点分别初始化。

---

注意－尽管声明的格式允许方括号在变量名左边或者右边，但此种灵活性不适用于数组句法的其它方面。例如：new int []**[4]**是非法的。

---

**多维数组**

- 非矩形数组的数组

```
twoDim[0] = new int [2];
twoDim[1] = new int [4];
twoDim[2] = new int [6];
twoDim[3] = new int [8];
```

- 每个数组有 5 个整数类型的 4 个数组的数组

```
int twoDim [][]  =  new int [4][5];
```

**多维数组**

因为这种对每个元素的分别初始化，所以有可能创建非矩形数组的数组。也就是说，twoDim 的元素可按如下方式初始化：

```
twoDim[0] = new int [2]
twoDim[1] = new int [4];
twoDim[2] = new int [6];
twoDim[3] = new int [8];
```

由于此种初始化的方法烦琐乏味，而且矩形数组的数组是最通用的形式，因而产生了一种"速记"方法来创建二维数组。例如：

```
int twoDim [][]  =  new int [4][5];
```

可被用来创建一个每个数组有 5 个整数类型的 4 个数组的数组。

# 第七节 数组界限

**数组界限**

**所有数组的下标都从 0 开始**

```
int list [] = new int [10];
for (int i= 0; i< list.length; i++)
 System.out.println(list[i]);
```

在 Java 编程语言中，所有数组的下标都从 0 开始。 一个数组中元素的数量被作为具有 length 属性的部分数组对象而存储；这个值被用来检查所有运行时访问的界限。如果发生了一个越出界限的访问，那么运行时的报错也就出现了。

使用 length 属性的例子如下：

```
int list [] = new int [10];
for (int i= 0; i< list.length; i++)
 System.out.println(list[i]);
```

使用 length 属性使得程序的维护变得更简单。

# 第八节 拷贝数组

**拷贝数组**

- 不能调整数组的大小
- 可使用相同的引用变量来引用一个全新的数组

```
int elements [] = new int [6];
elements = new int [10];
```

数组一旦创建后，其大小不可调整。然而，你可使用相同的引用变量来引用一个全新的数组：

```
int myArray [] = new int [6];
myArray = new int [10];
```

在这种情况下，第一个数组被有效地丢失，除非对它的其它引用保留在其它地方。

**拷贝数组**

**拷贝数组**

System.arraycopy()**方法**

```
// original array
1.int myArray[] = { 1, 2, 3, 4, 5, 6 };
2.
3.// new larger array
4.int hold[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
5.// copy all of the myArray array to the hold
6.// array, starting with the 0th index
7.System.arraycopy(myArray, 0, hold, 0,
8.myArray.length);
```

Java 编程语言在 System 类中提供了一种特殊方法拷贝数组，该方法被称

作 arraycopy()。例如，araycopy 可作如下使用：

```
// original array
1.int myArray[] = { 1, 2, 3, 4, 5, 6 };
2.
3.// new larger array
4.int hold[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
5.// copy all of the myArray array to the hold
6.// array, starting with the 0th index
7.System.arraycopy(myArray, 0, hold, 0,
8.myArray.length);
```

在这一点，数组 hold 有如下内容：1,2,3,4,5,6,4,3,2,1。

---

注意－在处理对象数组时，System.arraycopy()拷贝的是引用，而不是对象。对象本身不改变。

---

# 练习： 使用数组

练习目标－－在定义并初始化数组后，你将在程序中使用数组。

## 一、准备

为成功地完成本练习，请务必理解基本的矩阵概念并了解如何定位一个数组以获取它的值。

## 二、任务

### 1 级：基本数组的使用

1. 创建一个称作 BasicArray 的类，在...main()方法中声明两个变量，一个是 thisArray，另一个是 thatArray，它们应属类型 array of int。
2. 创建一个数组，它有 10 个 int 值，范围从 1 至 10。分配这个第三数组的引用给变量 thisArray。
3. 使用 for()循环打印 thisArray 的所有值。如何控制循环的次数？
4. 编译并运行程序。多少值被打印？这些值是什么？
5. 对每个 thisArray 的元素，建立它的值为索引值的阶乘。打印数组的值。
6. 编译并运行程序。
7. 分配 thisArray 的引用给变量 thatArray。打印 thatArray 的所有元素。
8. 编译并运行程序。tyatArray 的多少值被显示？这些值是什么？它们来自何处。
9. 修改 thisArray 的某些元素，打印 thatArray 的值。
10. 编译并运行程序；在 thatArray 的值中，你注意到了什么？
11. 创建一个有 20 个 int 值的数组。分配新数组的引用给变量 thatArray，打印 thatArray 的值。
12. 编译并运行程序。每个数组有多少值被显示? 这些值是什么？
13. 拷贝 thisArray 的值给 thatArray。你将使用什么方法调用? 你将如何限制拷贝元素的数量? thatArray 的元素 10 至 19 有什么变化？
14. 打印 thatArray 的值。
15. 编译并运行程序。你所显示的值都是正确的吗？如果不是，你知道有那些内容理解得不对吗？
16. 改变 thatArray 的某些值；打印 thisArray 和 thatArray。
17. 编译并运行程序。这些值是你所期待的吗？

### 2 级：数组的数组

1. 创建一个称作 Array2D 的类，在 main()方法中声明一个称作 twoD 的变量，它应属类型 array of array of int。

2. 创建一个元素类型为 int 的数组，该数组应包括 4 个元素并被赋值到变量 twoD 的 elements[0]。

3. 编写两个嵌套 for()循环语句以打印 twoD 的全部值。以矩阵的格式安排输出（可采用 System.out.print()方法)。

4. 编译并运行程序。 你应该能发现此时出现了运行错误(空指针异常），这是因为 twoD 的 elements[1] 至 [3]未被初始化。

5. 分别创建包括 5 个、6 个和 7 个元素的 int 数组，将这些数组的引用分别赋予 twoD 的 elements [1]，[2]和[3]；确认完成上述操作的代码是在第 3 步所描述的嵌套 for()循环之前插入的。

6. 编译并运行程序。这次你应该看到一个零值的非矩形布局。

7. 赋予 twoD 数组的每个元素一个明显的非零值(提示：使用 Math.random() 以获得随机值)。

8. 声明一个属类型 array of int 的称作 oneD 的变量。然后, 创建一个包括 4 个元素的 int 数组。将该数组的引用分别赋予数组 array twoD 和 oneD 的第一个元素。赋值后，打印 oneD 和 twoD 数组。

9.编译并运行程序。请注意通过打印 oneD 的值而显示的单个数组与 twoD 数组的元素是相同的。

3 级：字谜游戏

1. 创建一个称作 WordScrambler 的应用程序，它具有一个词数组(长度最大为 8 个字符），用它可将一个词的字母拼凑(重排顺序），然后组成一个新词。

2.允许使用者看到词的拼凑过程，并分解 5 个游戏的运行过程。

## 三、练习小结

讨论——用几分钟的时间讨论一下在以上练习中你所获得的经验、感想和发现。

- 经验 解释 总结 应用

## 四、检查你的进度

在进入下一模块之前，请确认你已经能够：

- 声明并创建原始数组、类数组或数组类型
- 解释为什么数组的元素需初始化
- 给出数组定义并初始化数组元素
- 确定一个数组中元素的数量
- 创建多维数组
- 编写从一个数组类型到另一个数组类型数组值的拷贝代码

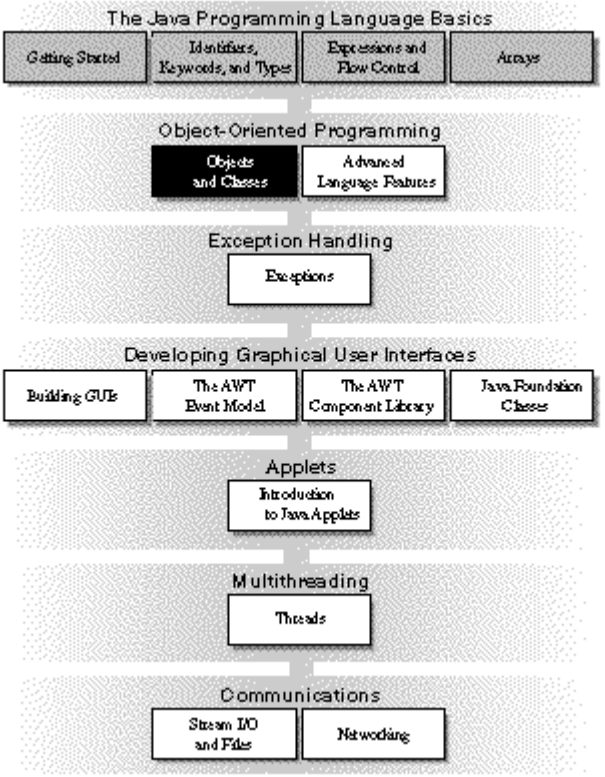
## 五、思考题

你怎样创建一个三维数组？

使用数组的缺点是什么？

# 第五章 对象和类

本模块是 Java 编程语言中讨论面向对象语句及面向对象特征两部分中的第一部分。




## 第一节 相关问题

讨论—下面的问题与本模块中出现的材料相关：

- 到目前为止学习的 Java 编程语言的元素存在于大部分语言中，不管它们是否是面向对象语言。
- Java 编程语言拥有什么特征使它成为一个面向对象语言？
- “面向对象”这个术语真正的含义是什么？

## 第二节 目 标

学完本模块，你便能：

- 定义封装、多态性以及继承
- 使用 private 及 public 访问修饰符
- 开发程序段创建并初始化一个对象
- 对一个特殊对象调用一个方法
- 描述构造函数及方法重载
- 描述 this 引用的用途
- 讨论为什么 Java 应用程序代码是可重复使用的
- 在一个 Java 程序中，确认：
  - package 语句
  - import 语句
  - 类、成员函数以及变量
  - 构造函数
  - 重载方法
  - 覆盖方法
  - 父类构造函数

## 第三节 对象基础

面向对象程序（OOP）语句能使现实世界中的概念在计算机程序中变成模块。它包括构造程序的特征以及组织数据和算法的机制。OOP 语言有三个特征：封装、多态性及继承。所有这些特征与类的概念是息

息相关的。

### 5.3.1 抽象数据类型

当数据类型是由数据项组成时，可以定义许多程序段或方法在该类型数据上专门运行。当程序语言定义一个基本类型如整数时，它同时也定义了许多运算方法（如加法、减法、乘法和除法），因而它可以在该类型的实例中运行。

在许多程序语言中，一旦一个集合数据类型已经定义，程序员定义应用函数在该类型的变量上运行，该变量在代码和集合类型（除非可能在命名规则中）之间无任何联系。

有些程序语言，包括 Java，允许在数据类型的声明和操作该类型变量的代码的声明之间有紧密的联系。这种联系通常被称为抽象数据类型。

### 5.3.2 类和对象

Java 编程语言中的抽象数据类型概念被认为是 class。类给对象的特殊类型提供定义。它规定对象内部的数据，创建该对象的特性，以及对象在其自己的数据上运行的功能。因此类就是一块模板。Objects 是在其类模块上建立起来的，很象根据建筑图纸来建楼。同样的图纸可用来建许多楼房，而每栋楼房是它自己的一个对象。

应该注意，类定义了对象是什么，但它本身不是一个对象。在程序中只能有类定义的一个副本，但可以有几个对象作为该类的实例。在 Java 编程语言中使用 new 运算符实例化一个对象。

在类中定义的数据类型用途不大，除非有目的地使用它们。方法定义了可以在对象上进行的操作，换言之，方法定义类来干什么。因此 Java 编程语言中的所有方法都属于一类。不象 C++程序，Java 软件程序不可能在类之外的全局区域有方法。

看一个类的例子：

```
class EmpInfo {
   String name;
   String designation;
   String department;
 }
```

这些变量(name, designation 和 department)被称为类 EmpInfo 的成员。

实例化一个对象，创建它，然后如下所述对其成员赋值：

```
EmpInfo employee = new EmpInfo();  //creates instance
employee.name = Robert Javaman ";   // initializes
employee.designation = " Manager " ;  // the three
employee.department = " Coffee Shop " ;  // members
```

EmpInfo 类中的 employee 对象现在就可以用了。例如：

```
System.out.println(employee.name + " is " +
   employee.designation + " at " +
   employee.department);
```

打印结果如下：

```
Robert Javaman is Manager at Coffee Shop
```

如下所述，现在可以在类中放入方法 print( )来打印数据了。数据和代码可以封装在一个单个实体中，这是面向对象语言的一个基本特征。定名为 print( )的代码段可以被作为一个方法而调用，它是术语“函数”的面向对象的称法。

```
class EmpInfo {
   String name;
   String designation;
   String department;

   void print() {
        System.out.println(name + " is " + designation + " at " + department);
```

```
   }
}
```

一旦对象被创建并被实例化，该方法就打印出类成员的数据。按下述步骤实现：

```
EmpInfo employee = new EmpInfo(); // creates instance

employee.name = " Robert Javaman " ; // initializes
employee.designation = " Manager " ; // the three
employee.department = " Coffee Shop " ; // members
employee.print();// prints the details
```

看看集合数据类型 MyDate 和对下一个日期赋值的函数 tomorrow( )。

按如下所述在 MyDate 类型和 tomorrow( )方法之间创建一种联系：

```
public class MyDate {
   private int day, month, year;
   public void tomorrow() {
     // code to increment day
  }
 }
```

注—本声明中的“ private ”一词将在后文描述。

方法不是（作为分离的实体）在数据上运行，而数据（作为对象的一部分）对它本身进行操作。

```
MyDate d = new MyDate();
d.tomorrow();
```

这个注释表明行为是由对象而不是在对象上完成的。记住，可以用点记号来指向 MyDate 类中的字段。

这就意味着“ MyDate 对象的 day 字段由变量 d.调用。于是，前面的例子“ MyDate 对象的 tomorrow 行为由变量 d.调用 ” ，换言之，就是 d 对象对它本身进行 tomorrow （）运算。

方法是一个对象的属性并且能作为单个单元的一部分与它所在对象的数据发生密切的相互作用，这个是一个关键的面向对象的概念。（如果与这个概念不同，即，方法是分离的实体，从外部引入，作用在数据上。）message passing（消息传递）这个术语通常用来表达这样一个概念，即：指示一个对象在它本身数据上做某项工作，一个对象的方法定义了该对象能在它本身数据上做什么。

### 5.3.3 定义方法

**定义方法**

方法声明采取这样的格式：

```
<modifiers> <return_type> <name> ([<argument_list>]) <block> [throws <exception>] {<block>}
public void addDays(int days) {

}
```

Java 编程语言使用一种与其它语言，尤其是 C 和 C++，非常相似的办法来定义方法。其声明采用以下格式：

```
<modifiers> <return_type> <name> ([<argument_list>]) <block> [throws <exception>]
{<block>}
```

<name>可以是任何合法标识符，并带有用已经使用的名称为基础的某些限制条件。

<return_type>表示方法返回值的类型。如果方法不返回任何值，它必须声明为 void(空)。Java 技术对返回值是很严格的，例如，如果声明某方法返回一个 int 值，那么方法必须从所有可能的返回路径中返回一个 int（只能在等待返回该 int 值的上下文中被调用。）

<modifiers>段能承载许多不同的修饰符，包括公共的、受保护的，以及私有的。公共访问修饰符表示方法可以从任何其它代码调用。私有表示方法只可以由该类中的其它方法来调用。受保护将在以后的课

程中讨论。

<argument_list>允许将参数值传递到方法中。列举的元素由逗号分开，而每一个元素包含一个类型和一个标识符。

throws <exception>子句导致一个运行时错误（异常）被报告到调用的方法中，以便以合适的方式处理它。非正常的情况在<exception>中有规定。

例如：

```
 public void addDays(int days) {
}
```

告诉方法的本体，用<block>，来接受表示将天数增加到当前日期中的那个参数。 在这种方法中，值是以标识符 days 来引用的。

### 5.3.4 值传递

**值传递**

Java 编程语言只由值传递参数

当一个对象实例作为一个参数被传递到方法中时，参数的值就是对该对象的引用。

对象的内容可以在被调用的方法中改变，但对象的引用是永远不会改变的。

Java 编程语言只由值传递参数，也就是说，参数不能由被调用的方法来改变。当一个对象实例作为一个参数传递到方法中时，参数的值就是对象的引用。对象的内容可以在被调用的方法中改变，但对象引用是永远不会改变的。

下面的代码例子可以阐明这一点：

```
1 public class PassTest {
2
3       float ptValue;
4
5       // Methods to change the current values
6       public void changeInt (int value) {
7       value = 55;
8       }
9
10      public void changeStr (String value) {
11      value = new String ( " different " );
12      }
13
14      public void changeObjValue (PassTest ref) {
15      ref.ptValue = 99.0f;
16      }
17
18      public static void main (String args[]) {
19
20      String str;
21      int val;
22
23      // Create an instance of the class
24
25      PassTest pt = new PassTest ();
26      // Assign the int
27      val = 11;
28
```

```
29      // Try to change it
30      pt.changeInt (val);
31
32      // What is the current value?
33      System.out.println ( " Int value is: " + val);
34
35      // Assign the string
36      str = new String ( " hello " );
37
38      // Try to change it
39      pt.changeStr (str);
40
41      // What is the current value?
42      System.out.println ( " Str value is: " + str);
43
44      // Now set the ptValue
45      pt.ptValue = 101.0f;
46
47
48      // Now change the value of the float
49      // through the object reference
50      pt.changeObjValue (pt);
51
52      // What is the current value?
53      System.out.println ( " Current ptValue is: " +
54        pt.ptValue);
55        }
56      }
```

**这个代码输出如下内容：**

```
c:\student\source\> java PassTest
Int value is: 11
Str value is: hello
Current ptValue is: 99.0
```

**字符串对象是不会被** changeStr()**改变的，但是，**PassTest **对象的内容被改变了。**

### 5.3.5 this 引用

this **引用**

```
public class MyDate {
  private int day, month, year;
  public void tomorrow() {
    this.day = this.day + 1;
    // wrap around code...
  }
}
```

**关键字** this **是用来指向当前对象或类实例的。这里，**this.day **指的是当前对象的** day **字段。**

```
public class MyDate {
  private int day, month, year;
  public void tomorrow() {
    this.day = this.day + 1;
```

```
    // wrap around code...
  }
}
```

Java 编程语言自动将所有实例变量和方法引用与 this 关键字联系在一起，因此，使用关键字在某些情况下是多余的。下面的代码与前面的代码是等同的。

```
public class MyDate {
  private int day, month, year;
  public void tomorrow() {
    day = day + 1; // no ` this. ' before ` day '
    // wrap around code...
  }
}
```

也有关键字 this 使用不多余的情况。如，需要在某些完全分离的类中调用一个方法并将当前对象的一个引用作为参数传递时。例如：

```
Birthday bDay = new Birthday (this);
```

### 5.3.6 数据隐藏

**数据隐藏**

```
public class Date {
  private int day, month, year;
  pubic void tomorrow(){
    this.day = this.day +1;
    // wrap around code…
  }
}

public class DateUser {
  public static void main(String args[]) {
    Date mydate = new MyDate ();
    Mydate.day = 21; //illegal!
  }
}
```

在 MyDate 类的 day，month，year 声明中使用关键字 private，使从除了在 MyDate 类本身的方法以外的任何代码中访问这些成员成为不可能。因此，给 MyDate 类指定一个声明，下述代码是非法的：

```
public class DateUser {
  public static void main(String args[]) {
    MyDate d = new MyDate ();
    d.day = 21; // illegal!
  }
}
```

防止直接访问数据变量看起来奇怪，但它实际上对使用 MyDate 类的程序的质量有极大的好处。既然数据的单个项是不可访问的，那么唯一的办法就是通过方法来读或写。因此，如果要求类成员的内部一致性，就可以通过类本身的方法来处理。

思考一下允许从外部对其成员进行自由访问的 MyDate 类。代码做如下工作将是非常容易的：

```
MyDate d = new MyDate ();
d.day = 32; // invalid day
d.month = 2; d.day = 30; // plausible but wrong
d.month = d.month + 1; // omit check for wrap round
```

警告—这些和其它类似的赋值导致了在 MyDate 对象字段中无效的或不一致的值。这种情形是不可能马上作为问题暴露出来的，但肯定会在某个阶段终止程序。

如果类的数据成员没有暴露（被封装在类里），那么，类的用户会被迫使用方法来修改成员变量。这些

方法能进行有效性检查。考虑将下述方法作为 MyDate 类的一个部分。

```
public void setDay(int targetDay) {
  if (targetDay > this.daysInMonth()) {
    System.err.println( " invalid day " + targetDay);
  }
  else {
    this.day = targetDay;
  }
}
```

方法检查就是要看所要求设定的日期是否有效。如果日期无效，方法忽略了请求并打印出了信息。后面将会看到 Java 编程语言提供一个更有效的机制来处理溢出的方法参数。 但现在，你可以看到 Date 类对无效日期是受到有效保护的。

注—在上例中，daysInMonth()被指定为 Date 类中的一个方法。因此，它有一个 this 值，从这个值上，它可以提取回答询问所要求的 month 和 year。由于与成员变量一起使用，daysInMonth()方法的 this 的使用就多余了。

关于如何正确地调用方法，诸如“参数“月份”的值在对象中必须在有效范围内”的规则被叫做不变量（或前置和后置条件）。谨慎使用前置条件测试，可使类更容易再次使用，而且在重用中更可靠，因为如果任何方法被误用，使用该类的程序员能马上发现。

### 5.3.7 封装

封装

隐藏类的实现细节

迫使用户去使用一个界面去访问数据

使代码更好维护

除了保护对象的数据不被误修改外，在确保所要求的副作用被正确处理的情况下，迫使用户通过方法访问数据能使类的重用变得更简单。比如，在 MyDate 类的情形之下，考虑如何构造 tomorrow()方法。

如果数据完全可以访问，那么类的每一个用户都需要增加 day 的值，测试当前月份的天数，并处理月终（而且可能是年终）情形。这就毫无意义而且容易出错。仅管这些要求对日期来说是很好理解的，其它数据类型可能具有鲜为人知的相似的限制。通过迫使类的用户去使用所提供的 tomorrow()方法，保证每个人每次都能连续地并且正确地处理必要的副作用。

数据隐藏通常指的就是封装。它将类的外部界面与类的实现区分开来，隐藏实现细节。迫使用户去使用外部界面，即使实现细节改变，还可通过界面承担其功能而保留原样，确保调用它的代码还继续工作。这使代码维护更简单。

## 第四节 重载方法名称

重载方法名称

它可如下所示使用：

```
public void println(int i)
public void println(float f)
public void println()
```

参数表必须不同

返回类型可以不同

在某些情况下，可能要在同一个类中写几种做同样工作但带有不同参数的方法。考虑一个简单方法，它试图输出参数的文本表示法。这种方法被称做 println()。

现在假设打印每个 int，float，String 类型需要不同的打印方法。这是合情合理的，因为各种数据类型

要求不同的格式,而且可能要求不同的处理。可以分别创建三种方法,即 :printInt(),printfloat()和 printString()。但这是很乏味的。

Java 与其它几种编程语言一样，允许对不止一种方法重用方法名称。清楚地说，只有当某种东西能区分实际上需要哪种方法并去调用它时，它才能起作用。在三种打印方法的情况下，可能在参数的数量和类型的基础上对此进行区分。

通过重用方法名称，可用下述方法结束：

```
public void println(int i)
public void println(float f)
public void println()
```

当写代码来调用这些方法中的一种方法时，根据提供的参数的类型选择合适的一种方法。

有两个规则适用于重载方法：

调用语句的参数表必须有足够的不同，以至于允许区分出正确的方法被调用。正常的拓展晋升（如，单精度类型 float 到双精度类型 double）可能被应用，但是这样会导致在某些条件下的混淆。

方法的返回类型可以各不相同，但它不足以使返回类型变成唯一的差异。重载方法的参数表必须不同。

# 第五节 构造并初始化对象

**构造并初始化对象**

调用 new XXX()来为新对象分配空间会产生如下结果：

新对象的空间被分配并被初始化为 0 或空值

进行了显式的初始化

构造函数被执行

已经看到，为了给新对象分配空间，如何必须执行调用到 new XXX()。有时还会看到参数被放在括号中，如：new Button (“ press me”)。使用关键字 new 会导致下述情况：

首先，新对象的空间被分配并初始化到 0 或空值。在 Java 编程语言中，该阶段是不可分的，以确保在分配空间的过程中不会有大小为随机值的对象。

其次，进行显式的初始化。

第三，构造函数被执行，它是一种特殊方法。括号中传递给 new 的参数被传递给构造函数。

本节讨论这最后两个步骤。

## 5.5.1 显式成员初始化

**显式成员初始化**

```
public class Initialized {
  private int x = 5;
  private String name = " Fred " ;
  private Date created = new Date();
  // Methods go here
   ...
}
```

如果将简单的赋值表达式放在成员声明中，在对象构造过程中会进行显式成员初始化。

```
public class Initialized {
  private int x = 5;
  private String name = " Fred " ;
  private Date created = new Date();

 // Methods go here
...
}
```

## 5.5.2 构造函数

**构造函数**

方法名称必须完全与类名称相匹配

对于方法，不要声明返回类型

刚刚描述过的显式初始化机制为在一个对象中设定字段的初始值提供了一个简单的办法。但有时确实需要执行一个方法来进行初始化。也许还需要处理可能发生的错误。或者想用循环或条件表达式来进行初始化。或者想把参数传递到构造过程中以便要求构造新对象的代码能控制它创建的对象。

一个新对象的初始化的最终步骤是去调用一个叫做构造函数的方法。

构造函数是由下面两个规则所确认的：

方法名称必须与类名称完全相匹配。

对于方法，不要声明返回类型

**构造函数**

```
public class Xyz {
  // member variables go there

    public Xyz() {
      // set up the object.
   }

    public Xyz(int x) {
     // set up the object with a parameter
   }
}
```

```
public class Xyz {
  // member variables
  public Xyz() {             // No-arg constructor
  // set up the object.
  }
  public Xyz(int x) {          //int-arg constructor
    // set up the object using the parameter x.
  }
}
```

注—由于采用了方法，因此可以通过为几个构造函数提供不同的参数表的办法来重载构造函数。当发出 new Xyz(argument_list)调用的时候，传递到 new 语句中的参数表决定采用哪个构造函数。

### 5.5.3 调用重载构造函数

**调用重载构造函数**

```
public class Employee {
  private String name;
  private int salary;

public Employee(String n, int s) {
  name = n;
  salary = s;
}

public Employee(String n) {
  this(n, 0);
}

  public Employee() {
    this( " Unknown " );
  }
}
```

如果有一个类带有几个构造函数，那么也许会想复制其中一个构造函数的某方面效果到另一个构造函数中。可以通过使用关键字 this 作为一个方法调用来达到这个目的。

```
public class Employee {
    private String name;
```

```
    private int salary;
    public Employee(String n, int s) {
      name = n;
      salary = s;
    }
    public Employee(String n) {
      this(n, 0);
    }

    public Employee() {
      this( " Unknown " );
    }
}
```

在第二个构造函数中，有一个字符串参数，调用 this(n,0)将控制权传递到构造函数的另一个版本，即采用了一个 String 参数和一个 int 参数的构造函数中。

在第三个构造函数中，它没有参数，调用 this(“ Unknownn”)将控制权传递到构造函数的另一个版本，即采用了一个 String 参数的构造函数中。

注—对于 this 的任何调用，如果出现，在任何构造函数中必须是第一个语句。

### 5.5.4 缺省构造函数

**缺省构造函数**

每个类中都有

能够用 new Xxx()创建对象实例

如果增加一个带参数的构造函数声明，将会使缺省无效

每个类至少有一个构造函数。如果不写一个构造函数，Java 编程语言将提供一个。该构造函数没有参数，而且函数体为空。

缺省构造函数能用 new Xxx()创建对象实例，反之，你会被要求为每个类提供一个构造函数。

注—如果增加一个带参数的构造函数声明到一个类中，该类以前没有显式构造函数，那么将失去该缺省构造函数。基于这一点，对 new Xxx()的调用将会引起编译错误。认识到这一点很重要。

## 第六节 子 类

### 5.6.1 is a 关系

**Is a 关系**

Employee 类

```
public class Employee {
   String name;
   Date hireDate;
   Date dateOfBirth;
}
```

在编程中，常常要创建某件事的模型（如：一个职员），然后需要一个该基本模型的更专业化的版本。比如，可能需要一个经理的模型。显然经理实际上是一个职员，只是一个带有附加特征的职员。

看看下面的例子中类声明所阐述的：

```
public class Employee {
    String name;
    Date hireDate;
    Date dateOfBirth;
    String jobTitle;
    int grade;
```

```
    ...
}
```

这个例子阐述了在 Manager 和 Employee 类之间的数据复制。此外，还可能有许多适用于 Employee 和 Manager 两者的方法。因此，需要有一种办法从现有类来创建一个新类。这就叫做子类。

### 5.6.2 Extends 关键字

Extends 关键字

```
public class Employee {
  String name;
  Date hireDate;
  Date dateOfBirth;
  String jobTitle;
  int grade;
  ...
}

public class Manager extends Employee {
  String department;
  Employee [] subordinates;
}
```

在面向对象的语言中，提供了特殊的机制，允许程序员用以前定义的类来定义一个类。如下所示，可用关键字 extends 来实现：

```
public class Employee {
  String name;
  Date hireDate;
  Date dateOfBirth;
  String jobTitle;
  int grade;
  ...
}
public class Manager extends Employee {
  String department;
  Employee [] subordinates;
  ...
}
```

在这样的安排中，Manager 类被定义，具有 Employee 所拥有的所有变量及方法。所有这些变量和方法都是从父类的定义中继承来的。所有的程序员需要做的是定义额外特征或规定将适用的变化。

注--这种方法是在维护和可靠性方面的一个伟大进步。如果在 Employee 类中进行修改，那么， Manager 类就会自动修改，而不需要程序员做任何工作，除了对它进行编译。

注—对一个继承的方法或变量的描述只存在于对该成员进行定义的类的 API 文档中。当浏览探索一个（子）类的时候，一定要检查父类和其它祖先类中的继承成员。

### 5.6.3 参数和异类收集

参数和异类收集

具有共同点的类的收集被称做同类收集，如：数组

具有不同对象的收集叫异类收集，如：从各种其它类继承的类的收集

可以创建具有共同类的对象的收集（如数组）。这种收集被称作同类收集。

Java 编程语言有一个对象类，因此，由于多态性，它能收集所有种类的元素，正如所有类都扩展类对象一样。这种收集被称作异类收集。

创建一个 Manager 并慎重地将其引用赋到类型 Employee 的变量中似乎是不现实的。但这是可能的，而

且有很多为什么要取得这种效果的理由。

用这种方法，可以写出一个能接受通用对象的方法，在这种情况下，就是类 Employee，并在它的任何子类的对象上正确地运作。然后可以在应用类中产生一个方法，该应用类抽取一个职员，并将它的薪水与某个阈值进行比较来决定该职员的纳税责任。利用多态性，可以做到这些：

```
// In the Employee class
public TaxRate findTaxRate(Employee e) {
  // do calculations and return a tax rate for e
}
// Meanwhile, elsewhere in the application class
Manager m = new Manager();
:
TaxRate t = findTaxRate(m);
```

这是合法的，因为一个经理就是一个职员。

异类收集就是不相似的东西的收集。在面向对象语言中，可以创建许多东西的收集。所有的都有一个共同的祖先类-Object 类。如：

```
Employee [] staff = new Employee[1024];
staff[0] = new Manager();
staff[1] = new Employee();
```

甚至可以写出一个排序的方法，它将职员按年龄或薪水排序，而忽略其中一些人可能是经理。

注-每个类都是 Object 的一个子类，因此，可以用 Object 数组作为任何对象的容器。唯一不能被增加到 Object 数组中的唯一的东西就是基本变量（以及包装类，将在模块 6 中讨论，请注意这一点）。比 Object 数组更好的是向量类，它是设计来贮存异类收集对象的。

### 5.6.4 单继承性

**单继承性**

当一个类从一个唯一的类继承时，被称做单继承性。单继承性使代码更可靠。界面提供多继承性的好处，而且没有（多继承的）缺点。

Java 编程语言允许一个类仅能扩展成一个其它类。这个限制被称做单继承性。单继承性与多继承性的优点是面向对象程序员之间广泛讨论的话题。Java 编程语言加强了单继承性限制而使代码更为可靠，尽管这样有时会增加程序员的工作。模块 6 讨论一个被叫做界面（Interface）的语言特征，它允许多继承性的大部分好处，而不受其缺点的影响。

使用继承性的子类的一个例子如图 5-1 所示：

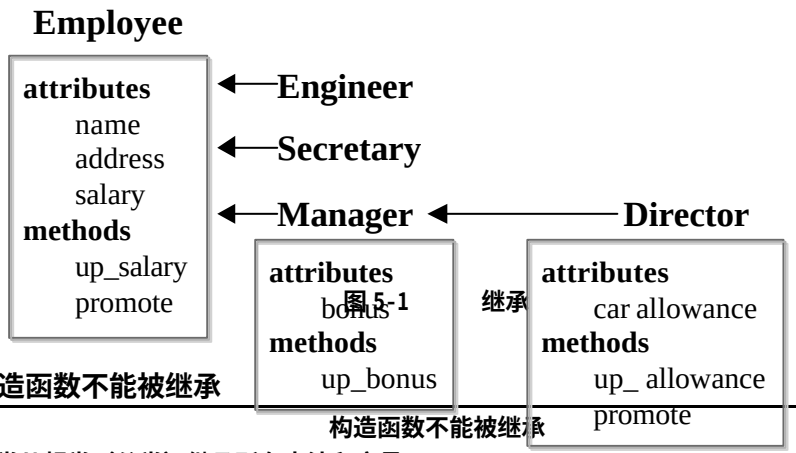



图 5-1

### 5.6.5 构造函数不能被继承

**构造函数不能被继承**

子类从超类（父类）继承所有方法和变量
子类不从超类继承构造函数
包含构造函数的两个办法是
　使用缺省构造函数
写一个或多个显式构造函数

尽管一个子类从父类继承所有的方法和变量，但它不继承构造函数，掌握这一点很重要。

一个类能得到构造函数，只有两个办法。或者写构造函数，或者根本没有写构造函数，类有一个缺省构造函数。

注-除了子构造函数外，父类构造函数也总是被访问。这将在后面的模块中详细讨论。

### 5.6.6 多态性

**多态性**

有能力拥有有许多不同的格式，就叫做多态性。比如，经理类能访问职员类方法。

一个对象只有一个格式

一个变量有许多格式，它能指向不同格式的对象

将经理描述成职员不只是描述这两个类之间的关系的一个简便方法。回想一下，经理类具有父类职员类的所有属性、成员和方法。这就是说，任何在 Employee 上的合法操作在 Manager 上也合法。如果 Employee 有 raiseSalary()和 fire()两个方法，那么 Manager 类也有。

一个对象只有一个格式（是在构造时给它的）。但是，既然变量能指向不同格式的对象，那么变量就是多态性的。在 Java 编程语言中，有一个类，它是其它所有类的父类。这就是 java.lang.object 类。因此，实际上，以前的定义被简略为：

```
public class Employee extends Object and
public class Manager extends Employee
```

Object 类定义许多有用的方法，包括 toString(),它就是为什么 Java 软件中每样东西都能转换成字符串表示法的原因。（即使这仅具有有限的用途）。

象大多数面向对象语言一样，Java 实际上允许引用一个带有变量的对象，这个变量是父类类型中的一个。因此，可以说：

```
Employee e = new Manager()
```

使用变量 e 是因为，你能访问的对象部分只是 Employee 的一个部分；Manager 的特殊部分是隐藏的。这是因为编译者应意识到，e 是一个 Employee，而不是一个 Manager。因而，下述情况是不允许的：

```
e.department = " Finance " ; // illegal
```

注-多态性是个运行时问题，与重载相反，重载是一个编译时问题。

### 5.6.7 关键字 super

**关键字 super**

super 被用在类中引用其超类。

super 被用来调用超类的成员变量。

超类行为就被调用，就好象对象是超类的组件。

调用行为不必发生在超类中，它能自动向上层类追溯。

关键字 super 可被用来引用该类中的超类。它被用来引用超类的成员变量或方法。

通常当覆盖一个方法时，实际目的不是要更换现有的行为，而是要在某种程度上扩展该行为。

用关键字 super 可获得：

```
public class Employee {
  private String name;
  private int salary;
public String getDetails() {
  return Name: " + name + "\nSalary: " + salary;
  }
}

public class Manager extends Employee {
  private String department;

  public String getDetails() {
```

```
    return super.getDetails() + // call parents'
                        // method
      "\nDepartment: " + department;
  }
}
```

请注意，super.method()格式的调用，如果对象已经具有父类类型，那么它的方法的整个行为都将被调用，也包括其所有副面效果。该方法不必在父类中定义。它也可以从某些祖先类中继承。

### 5.6.8 Instanceof 运算符

Instanceof 运算符

```
public class Employee extends Object
public class Manager extends Employee
public class Contractor extends Employee

public void method(Employee e) {
  if (e instanceof Manager) {
    // Get benefits and options along with salary
  }else if (e instanceof Contractor) {
    // Get hourly rates

  }else {
    // temporary employee
  }
}
```

假如能使用引用将对象传递到它们的父类中，那么有时你想知道实际有什么，这就是 instanceof 运算符的目的。假设类层次被扩展，那么你就能得到：

```
public class Employee extends Object
public class Manager extends Employee
public class Contractor extends Employee
```

注—记住，在可接受时， extends Object 实际上是多余的。在这里它仅作为一个提示项。

如果通过 Employee 类型的引用接受一个对象，它变不变成 Manager 或 Contractor 都可以。可以象这样用 instanceof 来测试：

```
public void method(Employee e) {
  if (e instanceof Manager) {
    // Get benefits and options along with salary
  }else if (e instanceof Contractor) {
    // Get hourly rates

  }else {
    // regular employee
  }
}
```

注—在 C++中，可以用 RTTI（运行时类型信息）来做相似的事，但在 Java 编程语言中的 instanceof 功能更强大。

### 5.6.9 对象的类型转换

**对象的类型转换**

使用 instanceof 来测试一个对象的类型。

用类型转换来恢复一个对象的全部功能。

用下述提示来检查类型转换的正确性：

向上的类型转换是隐含地实现的。

向下的类型转换必须针对子类并由编译器检查。

当运行时错误发生时，运行时检查引用类型。

在你接收父类的一个引用时，你可以通过使用 Instanceof 运算符判定该对象实际上是你所要的子类，并可以用类型转换该引用的办法来恢复对象的全部功能。

```
public void method(Employee e) {
  if (e instanceof Manager) {
    Manager m = (Manager)e;
    System.out.println( " This is the manager of " + m.department);
  }
  // rest of operation
}
```

如果不用强制类型转换，那么引用 e.department 的尝试就会失败，因为编译器不能将被称做 department 的成员定位在 Employee 类中。

如果不用 instanceof 做测试，就会有类型转换失败的危险。通常情况下，类型转换一个对象引用的尝试是要经过几种检查的：

向上强制类型转换类层次总是允许的，而且事实上不需要强制类型转换运算符。可由简单的赋值实现。

对于向下类型转换，编译器必须满足类型转换至少是可能的这样的条件。比如，任何将 Manager 引用类型转换成 Contractor 引用的尝试是肯定不允许的，因为 Contractor 不是一个 Manager。类型转换发生的类必须是当前引用类型的子类。

如果编译器允许类型转换，那么，该引用类型就会在运行时被检查。比如，如果 instanceof 检查从源程序中被省略，而被类型转换的对象实际上不是它应被类型转换进去的类型，那么，就会发生一个运行时错误(exception)。异常是运行时错误的一种形式，而且是后面模块中的主题。

# 第七节 覆盖方法

**覆盖方法**

子类可以修改从父类继承来的行为

子类能创建一个与父类方法有不同功能的方法，但具有相同的

名称

返回类型

参数表

除了能用附加额外特征的办法在旧类基础上产生一个新类，它还可以修改父类的当前行为。

如果在新类中定义一个方法，其名称、返回类型及参数表正好与父类中方法的名称、返回类型及参数相匹配，那么，新方法被称做覆盖旧方法。

注—记住，在同类中具有相同名称不同参数表的方法是被简单覆盖。这导致编译器使用所提供的参数来决定调用哪个方法。

考虑一下在 Employee 和 Manager 类中的这些方法的例子：

```
public class Employee {
  String name;
  int salary;
```

```
  public String getDetails() {
    return " Name: " + name + " \n " +
       "Salary: " + salary;
  }
}

public class Manager extends Employee {
  String department;

  public String getDetails() {
    return " Name: " + name + " \n " +
        " Manager of " + department;
  }
}
```

Manager 类有一个定义的 getDetails()方法，因为它是从 Employee 类中继承的。基本的方法被子类的版本所代替或覆盖了。

**覆盖方法**

**虚拟方法调用**

```
Employee e = new Manager();
e.getDetails();
```

**编译时类型与运行时类型**

假设第 121 页上的例子及下述方案是正确的：

```
Employee e = new Employee();
Manager m = new Manager();
```

如果请求 e.getDetails()和 m.getDetails，就会调用不同的行为。Employee 对象将执行与 Employee 有关的 getDetails 版本，Manager 对象将执行与 Manager 有关的 getDetails()版本。

不明显的是如下所示：

```
Employee e = new Manager();
e.getDetails();
```

或某些相似效果，比如一个通用方法参数或一个来自异类集合的项。

事实上，你得到与变量的运行时类型（即，变量所引用的对象的类型）相关的行为，而不是与变量的编译时类型相关的行为。这是面向对象语言的一个重要特征。它也是多态性的一个特征，并通常被称作虚拟方法调用。

在前例中，被执行的 e.getDetails()方法来自对象的真实类型，Manager。

注—如果你是 C++程序员，就会在 Java 编程语言和 C++之间得出一个重要的区别。在 C++中，你要想得到该行为，只能在源程序中将方法标记成 virtual。然而，在纯面向对象语言中，这是不正常的。当然，C++这么做是要提高执行速度。

# 第八节 调用覆盖方法

**覆盖方法的规则**

必须有一个与它所覆盖的方法相同的返回类型

不能比它所覆盖的方法访问性差

不能比它所覆盖的方法抛出更多的异常。

覆盖方法的规则

记住，子方法的名称以及子方法参数的顺序必须与父类中的方法的名称以及参数的顺序相同以便该方

法覆盖父类版本。下述规则适用于覆盖方法：

覆盖方法的返回类型必须与它所覆盖的方法相同。

覆盖方法不能比它所覆盖的方法访问性差

覆盖方法不能比它所覆盖的方法抛出更多的异常。（异常将在下一个模块中讨论。）

这些规则源自多态性的属性和 Java 编程语言必须保证“类型安全”的需要。考虑一下这个无效方案：

```
public class Parent {
  public void method() {
  }
}

public class Child extends Parent {
  private void method() {
  }
}

public class UseBoth {
  public void otherMethod() {
    Parent p1 = new Parent();
    Parent p2 = new Child();
    p1.method();
    p2.method();
  }
}
```

Java 编程语言语义规定，p2.method()导致方法的 Child 版本被执行，但因为方法被声明为 private，p2（声明为 Parent）不能访问它。于是，语言语义冲突。

# 第九节　调用父类构造函数

**调用父类构造函数**

对象的初始化是非常结构化的。

当一个对象被初始化时，下述行为按顺序发生：

存储空间被分配并初始化到 0 值

层次中的每个类都进行显式初始化

层次中的每个类都调用构造函数

在 Java 编程语言中，对象的初始化是非常结构化的。这样做是要保证安全。在前面的模块中，看到了当一个特定对象被创建的实例时发生了什么。由于继承性，图象被完成，而且下述行为按顺序发生：

存储空间被分配并初始化到 0 值

进行显式初始化

调用构造函数

层次中的每个类都会发生最后两个步骤，从最上层开始。

Java 技术安全模式要求在子类执行任何东西之前，描述父类的一个对象的各个方面都必须初始化。因此，Java 编程语言总是在执行子构造函数前调用父类构造函数的版本。

**调用父类构造函数**

**在许多情况下，使用缺省构造函数来对父类对象进行初始化。**

```
public class Employee {
  String name;
  public Employee(String n) {
    name = n;
  }
}

public class Manager extends Employee {
  String department;
  public Manager(String s, String d) {
    super(s);
    department = d;
  }
}
```

**无论是** super **还是** this**，都必须放在构造函数的第一行**

**调用父类构造函数**

**通常要定义一个带参数的构造函数，并要使用这些参数来控制一个对象的父类部分的构造。可能通过从子类构造函数的第一行调用关键字** super **的手段调用一个特殊的父类构造函数作为子类初始化的一部分。要控制具体的构造函数的调用，必须给** super()**提供合适的参数。当不调用带参数的** super **时，缺省的父类构造函数（即，带** 0 **个参数的构造函数）被隐含地调用。在这种情况下，如果没有缺省的父类构造函数，将导致编译错误。**

```
public class Employee {
  String name;
  public Employee(String n) {
    name = n;
  }
}
public class Manager extends Employee {
  String department;
  public Manager(String s, String d) {
    super(s); // Call parent constructor
          // with String argument
  department = d;
  }
}
```

**注—调用** super()**能将任意数量的合适的参数赋到父类中的不同构造函数中，但它必须是构造函数中的第一个语句。**

**当被使用时，**super **或** this **必须被放在构造函数的第一行。显然，两者不能被放在一个单独行中，但这种情况事实上不是一个问题。如果写一个构造函数，它既没有调用** super(…)**也没有调用** this(…)**，编译器自动插入一个调用到父类构造函数中，而不带参数。其它构造函数也能调用** super(…)**或** this(…)**，调用一个** Static **方法和构造函数的数据链。最终发生的是父类构造函数（可能几个）将在链中的任何子类构造函数前执行。**

# 第十节 编组类

### 5.10.1 包

包

包声明必须在源程序文件的开始被声明

根据源程序文件，只允许有一个包声明

```
// Class Employee of the Finance department for the
// ABC company
package abc.financedept;

public class Employee {
...
}
```

包名称是分层的，由圆点隔开

Java 编程语言提供 package 机制作为把相关类组成组的途径。迄今为止，所有的这些例子都属于缺省或未命名包。

可以用 package 语句表明在源程序文件中类属于一个特殊的包。

```
// Class Employee of the Finance department for the
// ABC company
package abc.financedept;


public class Employee {
...
}
```

包声明，如果有的话，必须在源程序文件的开始处。可以以空白和注解开始，而没有其它方式。只允许有一个包声明并且它控制整个源程序文件。

包名称是分层的，由圆点分隔。通常情况下，包名称的元素被整个地小写。然而，类名称通常以一个大写字母开始，而且每个附加单词的首字母可以被大写以区分类名称中的单词。

### 5.10.2 import 语句

Import 语句

告诉编译器到哪儿寻找类来使用，必须先于所有类声明

```
import abc.financeDept.*;
public class Manager extends Employee {
  String department;
  Employee [] subordinates;
}
```

当需要使用包时，使用 import 语句来告诉编译器到哪儿去寻找类。事实上，包名称在包中（比如，abc.financedept）形成类的名称的一部分。你可以引用 Employee 类作为 abc.financedept.Employee，或可以使用 import 语句及仅用类名称 Employee。

```
import abc.financeDept.*;
public class Manager extends Employee {
  String department;
  Employee [] subordinates;
}
```

注—import 语句必须先于所有类声明。

当使用一个包声明时，不必引入同样的包或该包的任何元素。记住，import 语句被用来将其它包中的类带到当前名空间。当前包，不管是显式的还是隐含的，总是当前名空间的一部分。

### 5.10.3 目录布局及 CLASSPATH 环境变量

**目录布局及 CLASSPATH 环境变量**

包被贮存在包含包名称的目录树中。

```
package abc.financedept
public class Employee  {
}

javac –d .  Employee.java
```

Employee.class 的目录路径是什么？

包被贮存在包含包名称分支的目录树中。例如，来自前面页中的 Employee.class 文件必须存在于下述目录中：

```
path\abc\financeDept
```

查寻类文件的包的目录树的根目录是在 CLASSPATH 中。

编译器的-d 选项规定了包层次的根，类文件被放在它里面（前面所示的 path）。

Java 技术编译器创建包目录，并在使用-d 选项时将编译的类文件移到它当中。

```
c:\jdk1.2\source\> javac -d . Employee.java
```

将创建目录结构 abc\financedept 在当前目录中（“.”）。

CLASSPATH 变量以前没有使用过，因为如果它没被设定，那么，工具的缺省行为将自动地包括类分布的标准位置和当前工作目录。如果想访问位于其它地方的包，那么必须设定 CLASSPATH 变量来显式地覆盖缺省行为。

```
c:\jdk1.2\source\>javac -d c:\mypackages Employee.java
```

为了让编译器在编译前页中的 Manager.java 文件时给 abc.financedept. Employee 类定位，CLASSPATH 环境变量必须包括下述包路径：

```
CLASSPATH=c:\mypackages;.
```

# 练习：使用对象及类

练习目的—会写、编译及运行三个程序，通过模仿使用银行帐目使用继承、构造函数及数据隐藏等面向对象概念。

**一、准备**

为了成功地完成该实验，必须理解类和对象的概念。

**二、任务**

一级实验：银行帐目

1. 创建一个类，Account.java，它定义银行帐目。决定应该做什么样的帐目，需要贮存什么样的数据，以及将用什么样的方法。
2. 使用一个包，bank，来包含类

二级实验：帐目类型

1. 修改一级实验，因而会针对 CheckingAccount 类的细节对 Account 划分子类。
2. 允许检查帐目来提供溢出保护。

三级实验：在线帐目服务

1. 创建一个简单的应用程序，Teller.java，它使用一级或二级实验来提供一个在线帐目开户服务。

**三、练习总结**

讨论—花几分钟时间来讨论一下实验练习中的经验、问题或发现。

- 经验 解释 总结 应用

**四、检查进步情况**

在继续下一个模块前，检查一下，确信你能：

- 定义封装、多态性和继承
- 使用访问修饰符 private 和 public
- 开发一个程序段来创建并初始化一个对象

- 在一个特定的对象上调用方法
- 描述构造函数和方法重载
- 描述 this 引用起什么作用
- 讨论为什么 Java 应用程序代码是可重用的

在 Java 软件程序中，确认：

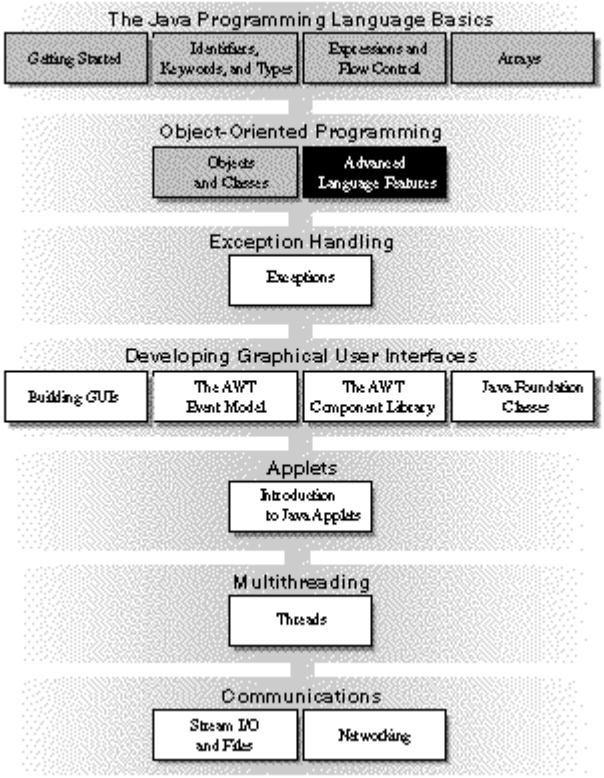
- package 语句
- import 语句
- 类成员函数和变量
- 构造函数
- 重载方法
- 覆盖方法
- 父类构造函数

五、思考

既然理解了对象和类，如何在当前或今后工作中使用这个信息？

# 第六章 高级语言特征

本模块讨论 Java 编程语言更多的面向对象特征。




## 第一节 相关问题

讨论-下述问题与本模块中出现的材料相关：

- 如何保持一个类或方法不被分成子类或被覆盖？
- 如何将数组概念的使用扩展到对象？

## 第二节 目 的

完成本模块的学习后，应该能

- 描述 static 变量，方法和初始程序
- 描述 final 类，方法和变量
- 列出访问控制级别
- 确认降级类并解释如何从 JDK1.0 升迁到 JDK1.1 到 JDK1.2

- 描述如何应用收集和反射
- 在 Java 软件程序中，确认
  - static 方法和变量
  - public，private，protected 和缺省变量
- 使用 abstract 类和方法
- 解释如何以及何时使用内部类
- 解释如何以及何时使用接口
- 描述==和 equals()之间的不同

# 第三节　类(static)变量

**类（static）变量**

在所有类的实例中共享

可以被标记为 public 或 private

如果被标记为 public 而没有该类的实例，可以从该类的外部访问

```
public class Count {
   private int serialNumber;
    private static int counter = 0;
   public Count() {
       counter++;
     serialNumber = counter;
    }
 }
```

有时想有一个可以在类的所有实例中共享的变量。比如，这可以用作实例之间交流的基础或追踪已经创建的实例的数量。

可以用关键字 static 来标记变量的办法获得这个效果。这样的变量有时被叫做 class variable，以便与不共享的成员或实例变量区分开来。

```
public class Count {
        private int serialNumber;
        private static int counter = 0;
        public Count() {
          counter++;
          serialNumber = counter;
        }
      }
```

在这个例子中，被创建的每个对象被赋于一个独特的序号，从 1 开始并继续往上。变量 counter 在所有实例中共享，所以，当一个对象的构造函数增加 counter 时，被创建的下一个对象接受增加过的值。

Static 变量在某种程度上与其它语言中的全局变量相似。Java 编程语言没有这样的全局语言，但 static 变量是可以从类的任何实例访问的单个变量。

如果 static 变量没有被标记成 private，它可能会被从该类的外部进行访问。要这样做，不需要类的实例，可以通过类名指向它。

```
public class StaticVar {
  public static int number;
}
public class OtherClass [
  public void method() {
    int x = StaticVar.number;
  }
```

```
}
```

# 第四节 类(static)方法

**类(static)方法**

**没有它所属的类的任何实例，static 方法可以被调用**

```
public class GeneralFunction {

    public static int addUp(int x, int y) {
     return x + y;
    }
  }

  public class UseGeneral {
    public void method() {
      int a = 9;
      int b = 10;
      int c = GeneralFunction.addUp(a, b);
      System.out.println("addUp() gives " + c);
    }
  }
```

**当没有一个特殊对象变量的实例的时候，有时需要访问程序代码。用关键字 static 标记的方法可以这样使用，有时被称做 class method。static 方法可以用类名而不是引用来访问，如：**

```
public class GeneralFunction {
      public static int addUp(int x, int y) {
        return x + y;
      }
    }
    public class UseGeneral {
      public void method() {
        int a = 9;
        int b = 10;
        int c = GeneralFunction.addUp(a, b);
        System.out.println("addUp() gives " + c);
      }
    }
```

**因为 static 方法不需它所属的类的任何实例就会被调用，因此没有 this 值。结果是，static 方法不能访问与它本身的参数以及 static 变量分离的任何变量。访问非静态变量的尝试会引起编译错误。**

**注-非静态变量只限于实例，并只能通过实例引用被访问。**

```
public class Wrong {
  int x;
  public static void main(String args[]) {
    x = 9; // COMPILER ERROR!
  }
}
```

```
Import points to remember about static methods:
```

**Main()是静态的，因为它必须在任何实例化发生前被顺序地访问，以便应用程序的运行。**

**静态方法不能被覆盖成非静态。**

# 第五节 静态初始化程序

**静态初始化程序**

**在 static block 中，类可以包含方法程序中不存在的代码。**
**当类被装载时，静态代码块只执行一次。**

**方法程序体中不存在的代码在 static block 中类可以包含该代码，这是完全有效的。当类被装载时，静态块代码只执行一次。类中不同的静态块按它们在类中出现的顺序被执行。**

```
public class StaticInitDemo {

static int i = 5;
  static {
    System.out.println("Static code i= "+ i++ );
  }
}
public class Test {
  public static void main(String args[]) {
    System.out.println("Main code: i="
      + StaticInitDemo.i);
  }
}
```

**将打印出：**

```
Static code: i=5
Main code: i=6
```

**Static 方法和数据**

# 第六节 一个完整的例子

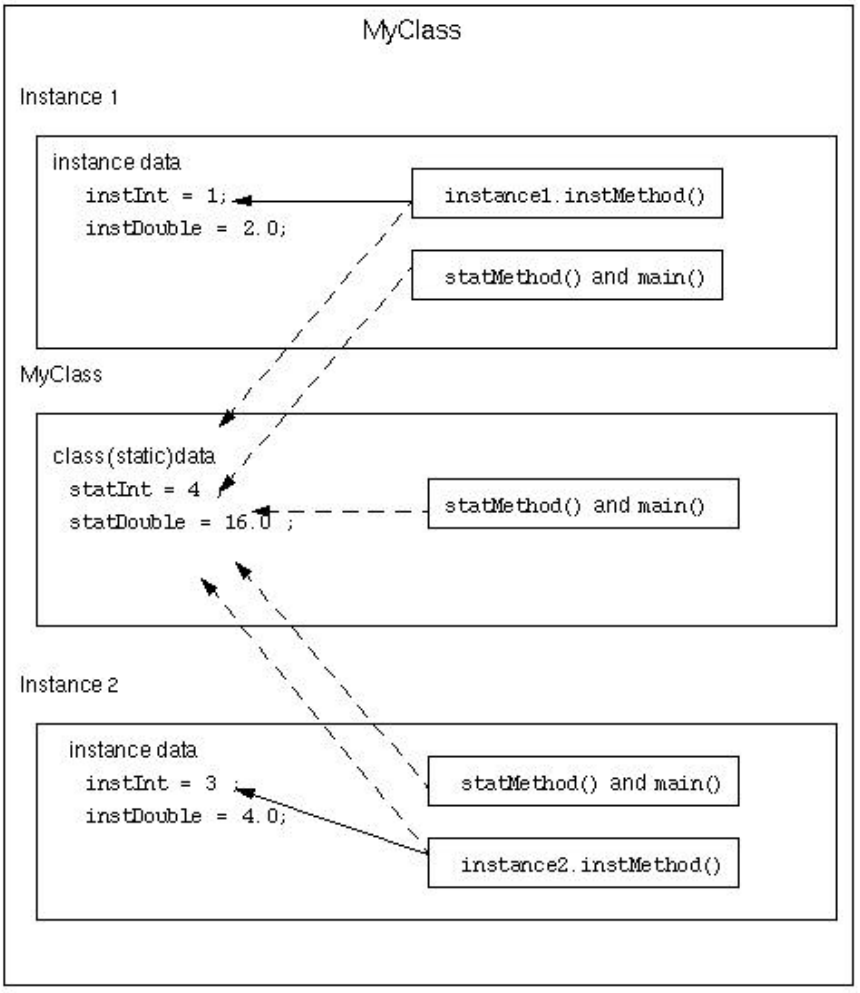
```
class MyClass {
  static int statInt = 4;
  static double statDouble = 16.0;
  int instInt;
  double instDouble;

  public static void statMethod(){
    System.out.println ("statInt="+statInt+
        ";statdouble="+statDouble);
}
public void instMethod(){
  System.out.println("instInt="+instInt+
    ";instdouble="+instDouble);
}
public MyClass(int intArg, double doubleArg){
  instInt = intArg;
  instDouble = doubleArg;
}
```

```
public static void main(String args[]){
  MyClass instance1 = new MyClass(1,2.0);
  MyClass instance2 = new MyClass(3,4.0);

  MyClass.statMethod(); //Outputs:statInt=4;
        //statDouble=16.0

  instance1.instMethod(); //Outputs:instInt=1;
       //instDouble=2.0
  instance1.statMethod(); //Outputs:statInt=4;
       //statDouble=16.0

  instance2.instMethod(); //Outputs:instInt=3;
       //instDouble=4.0
  instance2.statMethod(); //Outputs:statInt=4;
       //statDouble=16.0
}
}
```

图 6-1 是 MyClass 类定义的框图。这个例子阐述了：

1. Static 方法和数据的单个（共享）副本是因为类和该类的所有实例而存在。通过一个实例或通过类本身可以访问 static 成员。
2. 非静态数据只限于实例，只能通过该实例的非静态方法对它进行访问。非静态数据定义对象之间互不相同的特点，非静态方法在它们所作用的非静态数据的基础上对每个对象的行为互不相同。

考虑一下模仿汽车的特殊类型的一个对象的实例。轮子的大小，对该类型的所有汽车来说是个常量，



可能被模仿成一个静态变量。颜色根据对象的不同而不同，其行为也根据对象的不同而不同，在它所作用的非静态数据的基础上对不同对象返回不同的颜色。

图 6-1 Myclass 例子

# 第七节 关键字 final

## 6.7.1 Final 类

关键字 final

Final 类不能被分成子类

Final 方法不能被覆盖

Final 变量是常数

Java 编程语言允许关键字 Final 被应用到类中。如果这样做了，类便不能被子分成子类。比如，类 Java.lang.String 就是一个 final 类。这样做是出于安全原因，因为它保证，如果方法有字符串的引用，它肯定就是类 String 的字符串，而不是某个其它类的字符串，这个类是 String 的被修改过的子类，因为 String 可能被恶意窜改过。

## 6.7.2 Final 方法

个体方法也可以被标记为 final。被标记为 final 的方法不能被覆盖。这是由于安全原因。如果方法具有不能被改变的实现，而且对于对象的一致状态是关键的，那么就要使方法成为 final。

被声明为 final 的方法有时被用于优化。编译器能产生直接对方法调用的代码，而不是通常的涉及运行时查找的虚拟方法调用。

被标记为 static 或 private 的方法被自动地 final，因为动态联编在上述两种情况下都不能应用。

## 6.7.3 Final 变量

如果变量被标记为 final，其结果是使它成为常数。想改变 final 变量的值会导致一个编译错误。下面是一个正确定义 final 变量的例子：

```
public final int MAX_ARRAY_SIZE = 25;
```

注-如果将引用类型（即，任何类的类型）的变量标记为 final，那么该变量不能指向任何其它对象。但可能改变对象的内容，因为只有引用本身是 final。

# 第八节 抽象类

抽象类

声明方法的存在而不去实现它的类被叫做抽象类

可以通过关键字 abstract 进行标记将类声明为抽象

```
public abstract class Drawing {
public abstract void drawDot(int x, int y);
public void drawLine(int x1, int y1,
                     int x2, int y2) {
  // draw using the drawDot() method repeatedly.
 }
}
```

一个 abstract 类可以包含非抽象方法和变量

有时在库开发中，要创建一个体现某些基本行为的类，并为该类声明方法，但不能在该类中实现该行为。取而代之，在子类中实现该方法。知道其行为的其它类可以在类中实现这些方法。

例如，考虑一个 Drawing 类。该类包含用于各种绘图设备的方法，但这些必须以独立平台的方法实现。它不可能去访问机器的录像硬件而且还必须是独立于平台的。其意图是绘图类定义哪种方法应该存在，但实际上，由特殊的从属于平台子类去实现这个行为。

正如 Drawing 类这样的类，它声明方法的存在而不是实现，以及带有对已知行为的方法的实现，这样的类通常被称做抽象类。通过用关键字 abstract 进行标记声明一个抽象类。被声明但没有实现的方法（即，

这些没有程序体或{})，也必须标记为抽象。

```
public abstract class Drawing {
 public abstract void drawDot(int x, int y);
 public void drawLine(int x1, int y1,
                      int x2, int y2) {
   // draw using the drawDot() method repeatedly.
 }
}
```

不能创建 abstract 类的实例。然而可以创建一个变量，其类型是一个抽象类，并让它指向具体子类的一个实例。不能有抽象构造函数或抽象静态方法。

Abstract 类的子类为它们父类中的所有抽象方法提供实现，否则它们也是抽象类。

```
public class MachineDrawing extends Drawing {
  public void drawDot (int mach x, intmach y) {
         // Draw the dot
  }
}

Drawing d = new MachineDrawing();
```

# 第九节 接 口

**接 口**

接口是抽象类的变体。

在接口中，所有方法都是抽象的。

多继承性可通过实现这样的接口而获得。

句法是：

```
public interface Transparency {

  public static final int OPAQUE=1;
  public static final int BITMASK=2;
  public static final int TRANSLUCENT=3;

   public int getTransparency();
 }
```

接口是抽象类的变体。接口中的所有方法都是抽象的，没有一个有程序体。接口只可以定义 static final 成员变量。

接口的好处是，它给出了屈从于 Java 技术单继承规则的假象。当类定义只能扩展出单个类时，它能实现所需的多个接口。

接口的实现与子类相似，除了该实现类不能从接口定义中继承行为。当类实现特殊接口时，它定义（即，将程序体给予）所有这种接口的方法。然后，它可以在实现了该接口的类的任何对象上调用接口的方法。由于有抽象类，它允许使用接口名作为引用变量的类型。通常的动态联编将生效。引用可以转换到接口类型或从接口类型转换，instanceof 运算符可以用来决定某对象的类是否实现了接口。

接口是用关键字 interface 来定义的，如下所述：

```
public interface Transparency {
  public static final int OPAQUE=1;
  public static final int BITMASK=2;
  public static final int TRANSLUCENT=3;
  public int getTransparency();
```

```
        }
```

类能实现许多接口。由类实现的接口出现在类声明的末尾以逗号分隔的列表中，如下所示：

```
 public class MyApplet extends Applet implements
   Runnable, MouseListener{
     "..."
   }
```

下例表示一个简单的接口和实现它的一个类：

```
public interface SayHello {
    public void printMessage();
}

public    class SayHelloImpl implements SayHello {
     public void printMessage() {
         System.out.println("Hello");
      }
   }
```

interface SayHello 强制规定，实现它的所有的类必须有一个称做 printMessage 的方法，该方法带有一个 void 返回类型且没有输入参数。

**接　口**

对于下述情况，界面是有用的：

声明方法，期望一个或更多的类来实现该方法

决定一个对象的编程界面，而不揭示类的实际程序体

捕获无关类之间的相似性，而不强迫类关系

描述“似函数”对象，它可以作为参数被传递到在其它对象上调用的方法中

对于下述情况，接口是有用的：

声明方法，期望一个或更多的类来实现该方法。

揭示一个对象的编程接口，而不揭示类的实际程序体。（当将类的一个包输送到其它开发程序中时它是非常有用的。）

捕获无关类之间的相似性，而不强迫类关系。

描述“似函数”对象，它可以作为参数被传递到在其它对象上调用的方法中。它们是“函数指针”（用在 C 和 C++中）用法的一个安全的替代用法。

# 第十节　高级访问控制

**高级访问控制**

| 修饰符 | 同类 | 同包 | 子类 | 通用性 |
|---|---|---|---|---|
| 公共 | 是 | 是 | 是 | 是 |
| 受保护 | 是 | 是 | 是 | |
| 缺省 | 是 | 是 | | |
| 私有 | 是 | | | |

变量和方法可以处于四个访问级别的一个中；公共，受保护，缺省或私有。类可以在公共或缺省级别。

变量、方法或类有缺省访问性，如果它没有显式受保护修饰符作为它的声明的一部分的话。这种访问性意味着，访问可以来自任何方法，当然这些方法只能在作为目标的同一个包中的成员类当中。

以修饰符 protected 标记的变量或方法实际上比以缺省访问控制标记的更易访问。一个 protected 方法或变量可以从类当中的任何方法进行访问，这个类可以是同一个包中的成员，也可以是从任何子类中的任何方法进行访问。当它适合于一个类的子类但不是不相关的类时，就可以使用这种受保护访问来访问成员。

表 6-1 总结访问性标准

表 6-1　访问性标准

| 修饰符 | 同类 | 同包 | 子类 | 通用性 |
|---|---|---|---|---|
| 公共 | 是 | 是 | 是 | 是 |
| 受保护 | 是 | 是 | 是 | |
| 缺省 | 是 | 是 | | |
| 私有 | 是 | | | |

受保护访问甚至被提供给子类，该子类驻留在与拥有受保护特征的类的不同包中。

# 第十一节　降级

降　　级

降级就是过时的构造函数和方法调用。

过时的方法和构造函数由具有更标准化的命名规则的方法所取代。

当升迁代码时，用-deprecation 标志来编译代码：

javac -deprecation MyFile.java

在 JDK1.1 中，对方法名称的标准化做了重大努力。因此，在 JDK1.2 中，大量的类构造函数和方法调用过时。它们由根据更标准化的命名规则规定的方法名称所取代，总的说来，使程序员的生活简单化。

例如，在 JDK1.1 版本中的 Java.awt.Component 类：

改变或获得组件大小的方法是 resize()和 size()。

改变或获得组件矩形框的方法是 reshape()和 bounds()。

在 JDK1.0 版本中的 Java.awt.Component，这些方法被降级并被以 set 和 get 开头表示该方法的初级运算的方法所代替。

setSize()和 getSize()

setBounds()getBounds()

无论什么时候将代码从 JDK1.0 升迁到 JDK1.1 或更高版本中,或者即使使用以前用在 JDK1.0 中的代码，对用-deprecation 标志来编译代码都是一个好主意。

```
c:\ javac -deprecation MyFile.java
```

-deprecation 标志将报告在降级过的类中使用的任何方法。例如，看一个叫做 DateConverter 的实用类，它将 mm/dd/yy 格式的日期转换成星期几：

```
package myutilities;
import java.util.*;
import java.text.*;
public final class DateConverter {

private static String day_of_the_week [] = {"Sunday", "Monday", "Tuesday",
"Wednesday", "Thursday", "Friday", "Saturday"};

public static String getDayOfWeek (String theDate){
  int month, day, year;
```

```
  StringTokenizer st =
    new StringTokenizer (theDate, "/");
  month = Integer.parseInt(st.nextToken ());
  day = Integer.parseInt(st.nextToken());
  year = Integer.parseInt(st.nextToken());
  Date d = new Date (year, month, day);

  return (day_of_the_week[d.getDay()]);
  }
}
```

**当这个代码用-deprecation 标志在 JDK1.2 中被编译时，会得到：**

```
c:\ javac -deprecation DateConverter.java
DateConverter.java:16: Note: The constructor java.util.Date(int,int,int) has
been deprecated.
          Date d = new Date (year, month, day);
                ^
DateConverter.java:18: Note: The method int getDay() in class
java.util.Date has been deprecated.
               return (day_of_the_week[d.getDay()]);
                                       ^
```

Note: DateConverter.java uses a deprecated API.Please consult the documentation for a better alternative. 3 warnings

**重写的 DateConverter 类看起来象这样：**

```
package myutilities;
import java.util.*;
import java.text.*;
public final class DateConverter {

private static String day_Of_The_Week [] = {"Sunday", "Monday", "Tuesday",
"Wednesday", "Thursday", "Friday", "Saturday"};

public static String getDayOfWeek (String theDate){
   Date d = null;
 SimpleDateFormat sdf =
   new SimpleDateFormat ("MM/dd/yy");
 try {
   d = sdf.parse (theDate);
 } catch (ParseException e) {
   System.out.println (e);
   e.printStackTrace();
 }
 // Create a GregorianCalendar object
Calendar c =
 new GregorianCalendar (TimeZone.getTimeZone ("EST"),        Locale.US);
c.setTime (d);
return(day_Of_The_Week[(c.get(Calendar.DAY_OF_WEEK)-1)]);
```

```
  }
}
```

在这里，1.2 版本使用两个新类：SimpleDateFormat，用来采用任何 String 日期格式并创建一个 Date 对象的类；以及 GregorianCalendar 类，用来创建一个带有当地时区和场所的日历。

## 第十二节 ==运算符与 equals()方法

| ==运算符与 equals()方法 |
| --- |
| equals()和==方法决定引用值是否指向同一对象 |
| equals()在类中被覆盖，为的是当两个分离的对象的内容和类型相配的话，返回真值。 |

Java.lang 包中的 Object 类有 public boolean equals (Object obj)方法。它比较两个对象是否相等。仅当被比较的两个引用指向同一对象时，对象的 equals()方法返回 true。

==运算符也进行等值比较。也就是说，对于任何引用值 X 和 Y，当且仅当 X 和 Y 指向同一对象时， X==Y 返回真。

当两个分离的对象的内容和类型相配的话，String，Date，File 类和所有其它 override equals() 的包装类(Integer，Double，等等)将返回真。

例如，String 类中的 equals()方法返回 true，当且仅当参数不为 null 并且是一个 String 对象，该对象与调用该方法的 String 对象具有相同的字符顺序。

```
String s1=new String("JDK1.2");
String s2=new String("JDK1.2");
```

方法 s1.equals(s2)返回真,尽管 s1 和 s2 指向两个不同的对象。

## 第十三节 toString()方法

| toString()方法 |
| --- |
| 被用来将一个对象转换成 String |
| 被用来将一个基本类型转换成 String |

toString 方法被用来将一个对象转换成 String 表达式。当自动字符串转换发生时，它被用作编译程序的参照。System.out.println()调用下述代码：

```
Date now = new Date()
  System.out.println(now)
```

将被翻译成：

```
System.out.println(now.toString());
```

对象类定义缺省的 toString()方法，它返回类名称和它的引用的地址（通常情况下不是很有用）。许多类覆盖 toString()以提供更有用的信息。例如，所有的包装类覆盖 toString()以提供它们所代表的值的字符串格式。甚至没有字符串格式的类为了调试目的常常实现 toString()来返回对象状态信息。

## 第十四节 内部类

| 内部类 |
| --- |
| 被附加到 JDK1.1 |
| 允许一个类定义被放到另一个类定义里 |
| 把类在逻辑上组织在一起 |
| 可访问它们所嵌套的类的范围 |

内部类，有时叫做嵌套类，被附加到 JDK1.1 及更高版本中。内部类允许一个类定义被放到另一个类定义里。内部类是一个有用的特征，因为它们允许将逻辑上同属性的类组合到一起，并在另一个类中控制一个类的可视性。

### 6.14.1 内部类基础

下述例子表示使用内部类的共同方法：

```
import java.awt.*;
import java.awt.event.*;
public class MyFrame extends Frame{
    Button myButton;
    TextArea myTextArea;
    int count;;

    public MyFrame(){
       super("Inner Class Frame");
     myButton = new Button("click me");
     myTextArea = new TextArea();
     add(myButton,BorderLayout.CENTER);
     add(myTextArea,BorderLayout.NORTH);
     ButtonListener bList = new ButtonListener();
     myButton.addActionListener(bList);
 }
 class ButtonListener implements ActionListener{
   public void actionPerformed(ActionEvent e){
   count ++
     myTextArea.setText("button clicked" + {
         count + "times");
   }
 }// end of innerclass ButtonListener

 public static void main(String args[]){
   MyFrame f = new MyFrame();
   f.setSize(300,300);
   f.setVisible(true);
 }
} // end of class MyFrame
```

前面的例子包含一个类 MyFrame，它包括一个内部类 ButtonListener。编译器生成一个类文件，MyFrame$ButtonListener.class 以及 toMyFrame.class。它包含在 MyFrame.class 中，是在类的外部创建的。

### 6.14.2 如何做内部类工作？

内部类可访问它们所嵌套的类的范围。所嵌套的类的成员的访问性是关键而且很有用。对嵌套类的范围的访问是可能的，因为内部类实际上有一个隐含的引用指向外部类上下文（如外部类“this”）。

```
1.  public class MyFrame extends Frame{
2.      Button myButton;
3.      TextArea myTextarea;
4.      public MyFrame(){
5.        ......................
6.        ......................
7.        MyFrame$ButtonListener bList = new
8.                MyFrame$ButtonListener(this);
9.        myButton.addActionListener(bList);
10.    }
```

```
11.    class MyFrame$ButtonListener implements
12.    ActionListener{
13.    private MyFrame outerThis;
14.    Myframe$ButtonListener(MyFrame outerThisArg){
15.       outerThis = outerThisArg;
16.    }
17.
18.    public void actionPerformed(ActionEvent e) {
19.    outerThis.MyTextArea.setText("buttonclicked");
20.    ......................
21.    ......................
22.    }
23.    public static void main(String args[]){
24.      MyFrame f = new MyFrame();
25.      f.setSize(300,300);
26.      f.setVisible(true);
27.    }
28. }
```

有时可能要从static方法或在没有this的某些其它情况下,创建一个内部类的一个实例(例如,main)。可以如下这么做：

```
public static void main(String args[]){
    MyFrame f = new MyFrame();
    MyFrame.ButtonListener bList =
          f.new ButtonListener();
    f.setSize(50,50);
    f.setVisible(true);
}
```

### 6.14.3 内部类属性

| 内部类属性 |
| --- |
| 类名称只能用在定义过的范围中，除非用限定的名称。<br>内部类的名称必须与所嵌套的类不同。<br>内部类可以被定义在方法中。<br>任何变量，不论是本地变量还是正式参数，如果变量被标记为 final，那么，就可以被内部类中的方法访问。 |

内部类有如下属性：

类名称只能用在定义过的范围中，除非用在限定的名称中。内部类的名称必须与所嵌套的类不同。

内部类可以被定义在方法中。这条规则较简单，它支配到所嵌套类方法的变量的访问。任何变量，不论是本地变量还是正式参数，如果变量被标记为 final，那么，就可以被内部类中的方法访问。

内部类可以使用所嵌套类的类和实例变量以及所嵌套的块中的本地变量。

内部类可以被定义为 abstract.

属性

只有内部类可以被声明为 private 或 protected，以便防护它们不受来自外部类的访问。访问保护不阻止内部类使用其它类的任何成员，只要一个类嵌套另一个。

一个内部类可以作为一个接口，由另一个内部类实现。

被自动地声明为 static 的内部类成为顶层类。这些内部类失去了在本地范围和其它内部类中使用数据或变量的能力。

内部类不能声明任何 static 成员；只有顶层类可以声明 static 成员。因此，一个需求 static 成员的

内部类必须使用来自顶层类的成员。

注-内部类有常常被用作创建事件适配器的方便特征。事件适配器将在后面模块中讨论。

# 第十五节 包装类

包装类

用来将基本数据元素看作对象，包装类可以被用作：

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

Java 编程语言不把基本数据类型看作对象。例如，在基本格式本身当中，数字、布尔及字符型数据都被看作是为了提高效率。Java 编程语言提供包装类来将基本数据元素看作对象。这样的数据元素被包裹在创建于它们周围的对象中，每个 Java 基本数据类型在 Java.lang 包中都有一个相应的 wrapper class。每个包装类对象封装一个基本值。

包装类列表如下：

表 6-2 包装类

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

可以通过将被包裹的值传递到合适的构造函数中来构造包装类对象。例如：

```
int pInt = 500;
Integer wInt = new Integer(pInt);
```

# 第十六节 收集 API

收集 API

收集（或容器）是代表一个对象组的单个对象，被认为是它的元素。

收集类 Vector,Bits,Stack,Hashtable,LinkedList 等等都被支持。

收集 API 包括将对象保持为下述情况的界面：

收集-没有具体顺序的一组对象

设定-没有复制的一组对象

列表-有序对象组，允许复制

收集（或容器）是代表一个对象组的单个对象，被认为是它的元素。收集典型地处理许多对象的类型，所有的类型都有一个特殊的种类（也就是说，它们都是从一个共同父类型延续来的）。Java 编程语言支持收集类 Vector,Bits, Stack,Hashtable ,LinkedList 等等。例如，Stack 实现后进先出（LIFO）的顺序,Hashtable 提供一个相关的对象数组。

收集保持处理 Object 类型的对象。这允许在收集中贮存任何对象。它还可以，在使用对象前、从收集

中检索到它之后，使用正确的类型转换。

收集 API 典型地由将对象保持为下述情况的接口而组成：

收集-没有具体顺序的一组对象

设定-没有复制的一组对象

列表-有序对象组，允许复制

API 还包括诸如 HashSet，ArraySet，ArrayList，LinkedList 和 Vector 等等的类，它们实现这些接口。API 还提供了支持某些算法的方法，如：排序，二进制搜索，评估列表中的最小和最大，以及收集等。

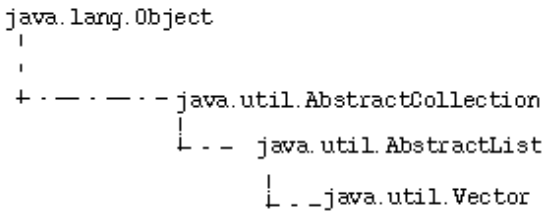
# 第十七节 Vector 类

Vector 类

Vector 类为与各种元素类型的动态数组一起工作提供方法。

```
Java.lang.Object
  |
  └-------Java.util.AbstractCollection
            └---Java.util.AbstractList
                 └----Java.util.Vector
```

Vector 类为与各种元素类型的动态数组一起工作提供方法。




## 6.17.1 提要

提 要

每个矢量保持一个 capacity 和 capacityIncrement。

因为附加了元素，矢量的贮存在信息块上增加到 capacityIncrement 变量的大小。

public class Vector extends AbstractList implements List, Cloneable, Serlializable

每个矢量保持一个 capacity 和 capacityIncrement。因为附加了元素到矢量，矢量的贮存在信息块上增加到 capacityIncrement 变量的大小。一个矢量的容量总是至少与矢量的大小一样大（通常情况下更大）。

## 6.17.2 构造函数

构造函数

```
public Vector()
 public Vector(int initialCapacity)
   public Vector(int initialCapacity,int
    capacityInctrment)
```

Vector 类的构造函数是

public Vector()-构造一个空矢量

public Vector(int initialCapacity)-构造一个具有具体贮存容量的空矢量

public Vector(int initialCapacity,int capacityInctrment)-构造具有具体贮存容量和具体 capacityInctrment 的空矢量。

## 6.17.3 变量

| 变 量 |
| --- |
| protected int capacityIncrement<br>protected int elementCount<br>protected Object elementData[] |

Vector 类包含下述实例变量：

protected int capacityIncrement -增加量。（如为 0，每次需要增加时，缓冲区的大小成倍。）

protected int elementCount -缓冲区中元素的数量。

protected Object elementData[]-元素被贮存的缓冲区。

### 6.17.4 方法

下面是 Vector 类中的一些方法。参照 Java API，看该类中所有方法的描述。

public final int size()-返回矢量中元素的数量。（这与矢量的容量不一样。）

public final boolean contains(Object elem)-如果指定对象是收集的值，返回真。

public final int indexOf (Object elem)-从起始位置搜索指定的对象，然后将一个索引返回到它当中（或，如果元素未找到为-1)。它使用对象的 equals()方法，因此，如果对象不覆盖 Object 的 equals()方法，它只比较对象引用，不比较对象内容。

public finalsynchronized Object elementAt (int index)-在指定的索引中返回元素。如果 index 无效，它抛出 ArrayIndexOutOfVoundsException。

public final synchronized void setElementAt (int index)-以指定对象在指定索引中替代指定元素。如果 index 无效，它抛出 ArrayIndexOutOfVoundsException。

public final synchronized void removeElementAt (int index)-删除指定索引中的元素。如果 index 无效，它抛出 ArrayIndexOutOfVoundsException。

public final synchronized void addElement (Object obj)-附加指定对象作为矢量的最后元素。

public final synchronized void insertElementAt (Object obj,int index)-插入指定对象作为指定索引中的一个元素，上移具有同等或更大索引的所有元素。如果 index 无效，它抛出 ArrayIndexOutOfVoundsException。

### 6.17.5 向量 Vector 模板例子：

下述模板可以用来附加不同的元素类型到矢量中并打印出矢量元素：

---

注-这个程序使用的方法来自本模块中前面讨论的类。

---

```
import java.util.*;

public class MyVector extends Vector {
  public MyVector() {
    super(1,1); // storage capacity & capacityIncrement
  }

 public void addInt(int i) {
   addElement(new Integer(i)); // addElement requires
                  // Object arg
}

public void addFloat(float f) {
  addElement(new Float(f));
}
```

```
public void addString(String s) {
  addElement(s);
}

public void addCharArray(char a[]) {
  addElement(a);
}

public void printVector() {
  Object o;
  int length = size(); // compare with capacity()
  System.out.println("Number of vector elements is
                " + length+ " and they are:");
for (int i = 0; i < length; i++) {
  o = elementAt(i);
  if (o instanceof char[]) {
  //An array's toString() method does not print what we want.
   System.out.println(String.copyValueOf((char[]) o));
  }
  else
     System.out.println(o.toString());
 }
}
public static void main(String args[]) {
  MyVector v = new MyVector() ;
  int digit = 5;
  float real = 3.14f;
  char letters[] = { 'a', 'b', 'c', 'd'};
  String s = new String ("Hi there!");

  v.addInt(digit);
  v.addFloat(real);
  v.addString(s);
  v.addCharArray(letters);

  v.printVector();
 }
}
```

**这个程序产生下列输出：**

```
$ java MyVector
Number of vector elements is 4 and are:
5
3.14
Hi there!
Abcd
```

# 第十八节 反射 API

**反射 API**

可以用作

构造新类实例和新数组

访问并修改对象和类的字段

调用对象和类中的方法

访问并修改数组的元素

Java 反射 API 提供一套类，可以用它们来决定一个类文件的变量和方法。因为被共同用于动态发现和执行代码的目的，因此 API 可以被用于：

构造新类实例和新数组

访问并修改对象和类的字段

调用对象和类中的方法

访问并修改数组的元素

只要安全策略允许，这些操作是可能的。在需要运行时检索并处理信息的情况下，反射 API 是有用的。例如，如果正在编写一个 Java 软件解释程序或调试程序，可以使用它。

# 第十九节 反射 API 特征

**反射 API 特征**

Java.lang.Class

Java.lang.reflect.Field

Java.lang.reflect.Method

Java.lang.reflect.Array

Java.lang.reflect.Constructor

定义类和方法的核心反射 API 的主要特征如下：

Java.lang.Class 类提供方法，该方法可获得有关类及其字段、构造函数以及方法的信息。

Java.lang.reflect.Field 提供方法，该方法设定/获得有关类中的字段的信息。

Java.lang.reflect.Method 提供方法，该方法访问并调用类中的方法，并获得它们的签名。

Java.lang.reflect.Array 能使数组对象自省。

Java.lang.reflect.Constructor 提供反射访问到构造函数。

# 第二十节 反射 API 安全模型

**反射 API 安全模型**

Java 安全管理器一个类接一个类地控制对核心 API 的访问。

当下述情况发生时，标准的 Java 编程语言访问控制得到加强：

Field 被用来获得或设定一个字段值

Method 被用来调用一个方法

Constructor 被用来创建并初始化一个新的类的实例

Java 安全管理器一个类接一个类地控制对核心 API 的访问。当下述情况发生时，标准的 Java 编程语言访问控制得到加强：

Field 被用来获得或设定一个字段值

Method 被用来调用一个方法

Constructor 被用来创建并初始化一个新的类的实例

# 练习：用高级语言特征工作

练习目的—使用银行帐户模型和采用高级面向对象特征，如：内部类，矢量类和接口等，重写、编译并运行三个程序。

## 一、准备

为了成功地完成该实验，必须熟悉本模块及前面模块中所讲的面向对象概念。

## 二、任务

### 一级实验：修改银行帐户问题

1. 定义只包含两个方法 deposit 和 withdraw 的接口 Personal。
2. 从模块 5 中，运用 Personal 接口来定义一套不同的帐户类型，重新定义类 Account.Java。它必须能处理个人帐户，进一步分成支票和存款两个帐户。
3. 设计并开发提供保护的方法。例如，如果一个客户有一个存款和支票帐户，须确保支票帐户受存款帐户保护。

### 二级实验：使用内部类

1. 创建一个叫做 BasicArray 的类，声明并初始化一个叫做 thisArray 的数组，它包含四个整数。
2. 创建一个名为 Factorial 的类，它包含一个计算它的参数的阶乘的方法。
3. 从 BasicArray 的主要方法创建 Factorial 类的一个实例，然后调用其方法来计算四个整数中每一个的阶乘。
4. 编译并测试该程序。
5. 将 Factorial 类中的所有东西都移到 BasicArray 类中。Factorial 现在就是 BasicArray 的一个内部类。
6. 编译并测试该程序。

### 三级实验：将 find 和 delete 方法附加到 MyVector 类中

1. 将 Find 方法附加到 MyVector 类中，它将返回被作为参数传递的元素的位置。
   如果未发现该参数，让方法返回-1。
   例如：

```
$ java MyVectorFind 3.14
3.14 is located at index 1
Number of vector elements is 4 and are:
5
3.14
Hi there!
abcd

$ java MyVectorFind c
args[0]=c, not found in vector
Number of vector elements is 4 and are:
5
3.14
Hi there!
abcd
```

2. 将 delete 方法附加到 MyVector 类中，该类将所有与参数相配的元素移走。
   方法必须返回 true 或 false：如果删除成功，为 true；否则为 false（元素存在或不存在于矢量

中)。

例如：

```
$ java MyVectorDelete 3.14
Elements 3.14 successfully deleted from vector.
Number of vector elements is 3 and are:
5
Hi there!
abcd
```

三、练习总结

讨论—花几分钟来讨论实验练习中遇到的经验、问题或发现。

经验 解释 总结 应用

四、检查进步情况

在继续学习下个模块前，检查一下你确实能：

描述 static 变量、方法和初始化程序

描述 final 类、方法和变量

列出访问控制级别

确认降级类并解释如何从 JDK1.0 升迁到 JDK1.1、JDK1.2。

在 Java 软件程序中，确认：

static 方法和变量

public，private，protected 和缺省变量

使用 abstract 类和方法

解释如何及何时使用内部类

解释如何及何时使用接口

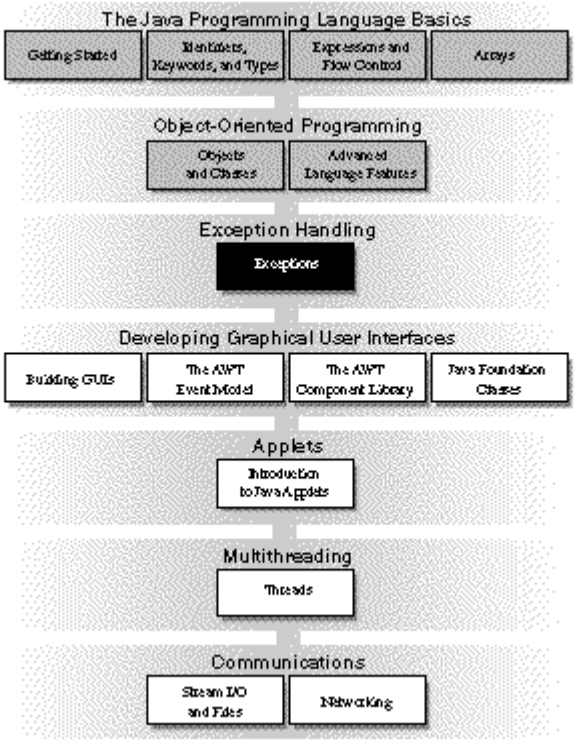
描述= =和 equals()之间的不同

五、思考

Java 编程语言具有什么特征，从而可以直接处理运行时错误情况？

# 第七章 异 常

本模块讲述建立在 Java 编程语言中的错误处理装置。




# 第一节 相关问题

讨论一下述问题与本模块中的材料相关：

- 在大部分编程语言中，如何解决运行时错误？

# 第二节 目 的

本模块学习结束后，能够：

- 定义异常
- 使用 try,catch 和 finally 语句
- 描述异常分类
- 开发程序来处理自己的异常

# 第三节 异 常

异 常

异常类定义程序所遇到的轻微错误

发生下列情况时，会出现异常：

想打开的文件不存在

网络连接中断

受控操作数超出预定范围

非常感兴趣地正在装载的类文件丢失

错误类定义严重的错误条件

## 7.3.1 介绍

什么是异常？在 Java 编程语言中，异常类定义程序中可能遇到的轻微的错误条件。可以写代码来处理异常并继续程序执行，而不是让程序中断。

在程序执行中，任何中断正常程序流程的异常条件就是错误或异常。例如，发生下列情况时，会出现异常：

- 想打开的文件不存在
- 网络连接中断
- 受控操作数超出预定范围
- 非常感兴趣地正在装载的类文件丢失

在 Java 编程语言中，错误类定义被认为是不能恢复的严重错误条件。在大多数情况下，当遇到这样的错误时，建议让程序中断。

Java 编程语言实现 C++异常来帮助建立弹性代码。在程序中发生错误时，发现错误的方法能抛出一个异常到其调用程序，发出已经发生问题的信号。然后，调用方法捕获抛出的异常，在可能时，再恢复回来。这个方案给程序员一个写处理程序的选择，来处理异常。

通过浏览 API，可以决定方法抛出的是什么样的异常。

### 7.3.2 实例

考虑一下 HelloWorld.java 程序版本的简单扩展，它通过信息来循环：

```
1.  public class HelloWorld {
2.    public static void main (String args[]) {
3.      int i = 0;
4.
5.      String greetings [] = {
6.        "Hello world!",
7.        "No, I mean it!",
8.        "HELLO WORLD!!"
9.      };
10.
11.     while (i < 4) {
12.       System.out.println (greetings[i]);
13.       i++;
14.     }
15.   }
16. }
```

正常情况下，当异常被抛出时，在其循环被执行四次之后，程序终止，并带有错误信息，就象前面所示的程序那样。

```
1.  c:\student\> java HelloWorld
2.  Hello world!
3.  No, I mean it!
4.  HELLO WORLD!!
5.  java.lang.ArrayIndexOutOfBoundsException: 3
6.      at HelloWorld.main(HelloWorld.java:12)
```

异常处理允许程序捕获异常，处理它们，然后继续程序执行。它是分层把关，因此，错误情况不会介入到程序的正常流程中。特殊情况发生时，在与正常执行的代码分离的代码块中被处理。这就产生了更易识别和管理的代码。

## 第四节 异常处理

Java 编程语言提供了一个来考虑哪个异常被抛出以及如何来恢复它的机制。

### 7.4.1 try 和 catch 语句

try 和 catch 语句

```
1.   try {
2.     // code that might throw a particular exception
3.   } catch (MyExceptionType e) {
4.     // code to execute if a MyExceptionType exception is thrown
5.   } catch (Exception e) {
6.    // code to execute if a general Exception exception is thrown
7.   }
```

要处理特殊的异常，将能够抛出异常的代码放入 try 块中，然后创建相应的 catch 块的列表，每个可以被抛出异常都有一个。如果生成的异常与 catch 中提到的相匹配，那么 catch 条件的块语句就被执行。在 try 块之后，可能有许多 catch 块，每一个都处理不同的异常。

```
1.  try {
2.    // code that might throw a particular exception
3.  } catch (MyExceptionType e) {
4.    // code to execute if a MyExceptionType exception is thrown
5.  } catch (Exception e) {
6.   // code to execute if a general Exception exception is thrown
7.  }
```

### 7.4.2 调用栈机制

如果方法中的一个语句抛出一个没有在相应的 try/catch 块中处理的异常，那么这个异常就被抛出到调用方法中。如果异常也没有在调用方法中被处理，它就被抛出到该方法的调用程序。这个过程要一直延续到异常被处理。如果异常到这时还没被处理，它便回到 main()，而且，即使 main()不处理它，那么，该异常就异常地中断程序。

考虑这样一种情况，在该情况中 main()方法调用另一个方法（比如，first()），然后它调用另一个（比如，second()）。如果在 second()中发生异常，那么必须做一个检查来看看该异常是否有一个 catch；如果没有，那么对调用栈（first()）中的下一个方法进行检查，然后检查下一个（main()）。如果这个异常在该调用栈上没有被最后一个方法处理，那么就会发生一个运行时错误，程序终止执行。

### 7.4.3 finally 语句

finally 语句

```
1.  try {
2.    startFaucet();
3.    waterLawn();
4.  }
5.  finally {
6.    stopFaucet();
7.  }
```

finally 语句定义一个总是执行的代码块，而不考虑异常是否被捕获。下述样板代码来自 Frank Yellin 弗兰克叶林的白皮书《Java 中的低级安全》：

```
1.  try {
2.    startFaucet();
3.    waterLawn();
4.  }
5.  finally {
6.    stopFaucet();
7.  }
```

在前面的例子中，即使异常在打开开关或给草地浇水时发生，开关也能被关掉。try 后面的括号中的代码被称做保护码。

如果终止程序的 System.exit()方法在保护码内被执行，那么，这是 finally 语句不被执行的唯一情况。这就暗示，控制流程能偏离正常执行顺序，比如，如果一个 return 语句被嵌入 try 块内的代码中，那么，finally 块中的代码应在 return 前执行。

### 7.4.4 重访前例

下面的例子是第 169 页 main()方法的重写。本程序以前的版本中产生的异常被捕获，数组索引重新设定，使下述程序继续运行。

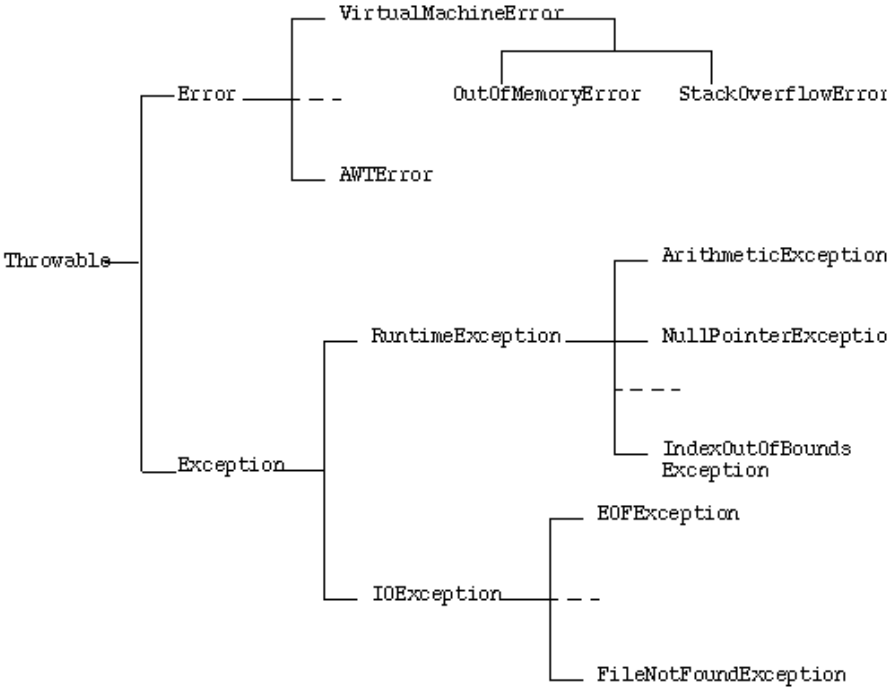
```
1.  public static void main (String args[]) {
2.    int i = 0;
3.    String greetings [] = {
4.      "Hello world!",
5.      "No, I mean it!",
6.      "HELLO WORLD!!"
7.    };
8.    while (i < 4) {
9.      try {
10.     System.out.println (greetings[i]);
11.     } catch (ArrayIndexOutOfBoundsException e){
12.     System.out.println( "Re-setting Index Value");
13.     i = -1;
14.     } finally {
15.     System.out.println("This is always printed");
16.     }
17.     i++;
18.     } // end while()
19.     } // end main()
```

当循环被执行时，下述在屏幕上出现的信息将改变。

```
1.  Hello world!
2.  This is always printed
3.  No, I mean it!
4.  This is always printed
5.  HELLO WORLD!!
6.  This is always printed
7.  Re-setting Index Value
8.  This is always printed
```

# 第五节 异常分类

在 Java 编程语言中，异常有三种分类。Java.lang.Throwable 类充当所有对象的父类，可以使用异常处理机制将这些对象抛出并捕获。在 Throwable 类中定义方法来检索与异常相关的错误信息，并打印显示异常发生的栈跟踪信息。它有 Error 和 Exception 两个基本子类，如下图所示：




Throwable 类不能使用，而使用子类异常中的一个来描述任何特殊异常。每个异常的目的描述如下：

- Error 表示恢复不是不可能但很困难的情况下的一种严重问题。比如说内存溢出。不可能指望程序能处理这样的情况。
- RuntimeException 表示一种设计或实现问题。也就是说，它表示如果程序运行正常，从不会发生的情况。比如，如果数组索引扩展不超出数组界限，那么，ArrayIndexOutOfBoundsException 异常从不会抛出。比如，这也适用于取消引用一个空值对象变量。因为一个正确设计和实现的程序从不出现这种异常，通常对它不做处理。这会导致一个运行时信息，应确保能采取措施更正问题，而不是将它藏到谁也不注意的地方。
- 其它异常表示一种运行时的困难，它通常由环境效果引起，可以进行处理。例子包括文件未找到或无效 URL 异常（用户打了一个错误的 URL），如果用户误打了什么东西，两者都容易出现。这两者都可能因为用户错误而出现，这就鼓励程序员去处理它们。

# 第六节　共同异常

**共同异常**

- ArithmeticException
- NullPointerException
- NegativeArraySizeException
- ArrayIndexoutofBoundsException
- SecurityException

Java 编程语言提供几种预定义的异常。下面是可能遇到的更具共同性的异常中的几种：

- ArithmeticException—整数被 0 除，运算得出的结果。
- int I =12 / 0；
- NullPointerException—当对象没被实例化时，访问对象的属性或方法的尝试：
- Date d= null;
- System.out.println(d.toString());
- NegativeArraySizeException—创建带负维数大小的数组的尝试。
- ArrayIndexoutofBoundsException—访问超过数组大小范围的一个元素的尝试。
- SecurityException—典型地被抛出到浏览器中，SecurityManager 类将抛出 applets 的一个异

常，该异常企图做下述工作（除非明显地得到允许）：

- 访问一个本地文件
- 打开主机的一个 socket，这个主机与服务于 applet 的主机不是同一个。
- 在运行时环境中执行另一个程序

# 第七节 处理或声明规则

处理或声明规则

- 用 try-catch-finally 块来处理异常
- 使用 throws 子句声明代码能引起一个异常

为了写出健壮的代码，Java 编程语言要求，当一个方法在栈（即，它已经被调用）上发生 Exception（它与 Error 或 RuntimeException 不同）时，那么，该方法必须决定如果出现问题该采取什么措施。

程序员可以做满足该要求的两件事：

第一，通过将 Try {} catch () {} 块纳入其代码中，在这里捕获给被命名为属于某个超类的异常，并调用方法处理它。即使 catch 块是空的，这也算是处理情况。

第二，让被调用的方法表示它将不处理异常，而且该异常将被抛回到它所遇到的调用方法中。它是按如下所示通过用 throws 子句标记的该调用方法的声明来实现的：

```
public void troublesome() throws IOException
```

关键字 throws 之后是所有异常的列表，方法可以抛回到它的调用程序中。尽管这里只显示了一个异常，如果有成倍的可能的异常可以通过该方法被抛出，那么，可以使用逗号分开的列表。

是选择处理还是选择声明一个异常取决于是否给你自己或你的调用程序一个更合适的候选的办法来处理异常。

注—由于异常类象其它类一样被组编到层次中，而且由于无论何时想要使用超类都必须使用子类， 因此，可以捕获异常“组”并以相同的捕获代码来处理它们。例如，尽管 IOExceptions（EOFException,FileNotFoundException 等等）有几种不同的类型，通过俘获 IOException，也可以捕获 IOException 任何子类的实例。

# 第八节 创建自己的异常

### 7.8.1 介绍

用户定义异常是通过扩展 Exception 类来创建的。这种异常类可以包含一个“普通”类所包含的任何东西。下面就是一个用户定义异常类例子，它包含一个构造函数、几个变量以及方法：

```
1.  public class ServerTimedOutException extends Exception {
2.      private String reason;
3.      private int port;
4.      public ServerTimedOutException (String reason,int port){
5.        this.reason = reason;
6.        this.port = port;
7.      }
8.      public String getReason() {
9.      return reason;
10.     }
11.     public int getPort() {
12.       return port;
13.     }
14.  }
```

使用语句来抛出已经创建的异常：

```
throw new ServerTimedOutException
("Could not connect", 80);
```

### 7.8.2 实例

考虑一个客户服务器程序。在客户代码中，要与服务器连接，并希望服务器在 5 秒钟内响应。如果服务器没有响应，那么，代码就如下所述抛出一个异常（如一个用户定义的 ServerTimedOutException）。

```
1.  public void connectMe(String serverName) throws
    ServerTimedOutException {
2.    int success;
3.    int portToConnect = 80;
4.    success = open(serverName, portToConnect);
5.    if (success == -1) {
6.    throw new ServerTimedOutException(
7.    "Could not connect", 80);
8.    }
9.    }
```

要捕获异常，使用 try 语句：

```
1.  public void findServer() {
2.  . . .
3.  try {
4.  connectMe(defaultServer);
5.  } catch(ServerTimedOutException e) {
6.  System.out.println("Server timed out, trying alternate");
7.  try {
8.  connectMe(alternateServer);
9.  } catch (ServerTimedOutException e1) {
10. System.out.println("No server currently available");
11. }
12. }
13. .. .
```

注—try 和 catch 块可以如前例所述那样被嵌套。

也可能部分地处理一个异常然后也将它抛出。如：

```
try {
.....
.....
} catch (ServerTimedOutException e) {
  System.out.println("Error caught ");
  throw e;
}
```

## 练习：处理并创建异常

练习目的—通过编写可以创建并处理异常的 Java 软件程序，可以获得异常机制的经验。

一、准备

为了成功地完成该实验，必须理解处理运行时错误的异常的概念。

二、任务

一级实验：处理一个异常

1．使用第 169 页上的样板异常程序在数组索引超出数组大小时创建一个异常。（或修改自己的程序以便创建一个异常。）

2．使用 try 和 catch 语句从异常进行恢复。

二级实验：创建自己的异常

使用模块 5 中创建的 bank 包并附加下述异常：

AccountOverdrawnException—当有了这个要取出比帐户上更多的钱的尝试时。

InvalidDepositException—当无效钱数（小于 0）存入时。

三、练习总结

讨论—花几分钟时间讨论实验练习中所取得的经验、问题或发现。

- 经验 解释 总结 应用

四、检查进步情况

在继续下一个模块前，检查一下，确信能够：

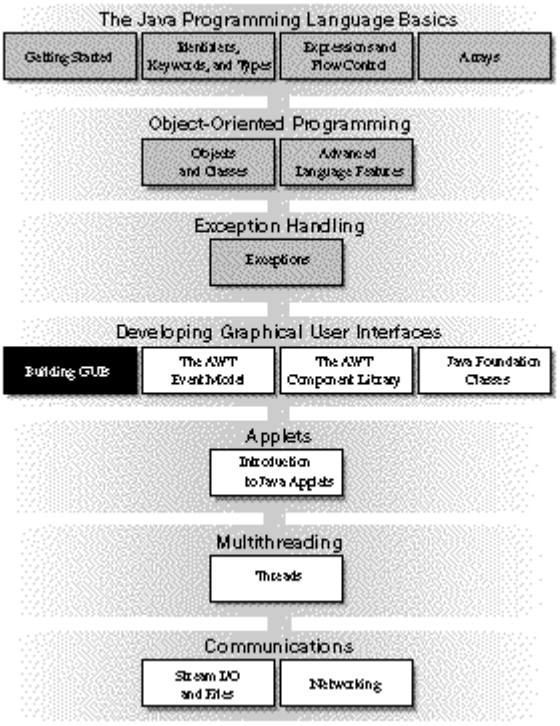
- 定义异常
- 使用 try，catch 和 finally 语句
- 描述异常分类
- 确认共同异常
- 开发程序来处理自己的异常

五、思考

Java 应用环境有什么特征，使它支持用户界面的开发？

# 第八章 建立 GUIs

本模块讲述图形用户界面的建立及布局。它介绍了抽象视窗工具包，一种建立 GUIs 的类包。




## 第一节 相关问题

讨论—下述问题与本模块中出现的材料相关。

- Java 编程语言是一个具有独立平台的编程语言。GUI 环境通常是从属平台。那么，为了使 GUI 平台独立，Java 技术是如何接近这个主题的呢？

# 第二节 目 的

完成本模块学习时，将能：

- 描述 AWT 包及其组件
- 定义 Container、Component 及 Layout Manager 等术语，以及它们是如何在一起来建立 GUI 的
- 使用 Layout Manager
- 使用 Flow、Border、Gird 及 Card 布局管理器来获得期望的动态布局
- 增加组件到 Container
- 正确运用 Frame 及 Panel 容器
- 描述如何使用嵌套容器来完成复杂的布局
- 在 Java 软件程序中，确认如下内容：
    - 容器
    - 相关布局管理器
    - 所有组件的布局层次

# 第三节 AWT

AWT

- 提供基本的 GUI 组件，用在所有的 Java applets 及应用程序中
- 具有可以扩展的超类，它们的属性是继承的，类也可被抽象化
- 确保显示在屏幕上的每个 GUI 组件都是抽象类组件的子类
- Container，它是一个 Component 的抽象子类，而且包括两个子类
    - Panel
    - window

AWT 提供用于所有 Java applets 及应用程序中的基本 GUI 组件，还为应用程序提供与机器的界面。这将保证一台计算机上出现的东西与另一台上的相一致。

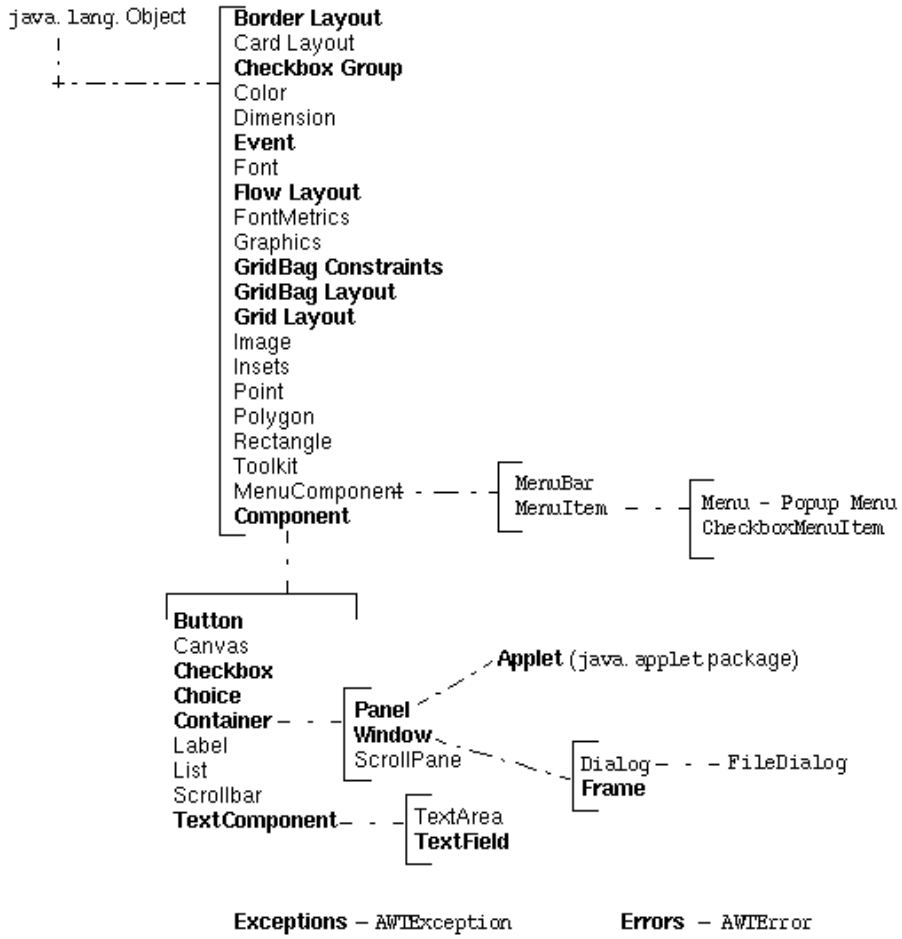
在学 AWT 之前，简单回顾一下对象层次。记住，超类是可以扩展的，它们的属性是可继承的。而且，类可以被抽象化，这就是说，它们是可被分成子类的模板，子类用于类的具体实现。

显示在屏幕上的每个 GUI 组件都是抽象类组件的子类。也就是说，每个从组件类扩展来的图形对象都与允许它们运行的大量方法和实例变量共享。

Container 是 Component 的一个抽象子类，它允许其它的组件被嵌套在里面。这些组件也可以是允许其它组件被嵌套在里面的容器，于是就创建了一个完整的层次结构。在屏幕上布置 GUI 组件，容器是很有用的。Panel 是 Container 的最简单的类。Container 的另一个子类是 Window。

# 第四节 Java.awt 包

Java.awt 包包含生成 WIDGETS 和 GUI 组件的类。该包的基本情况如下图所示。黑体字的类表明该模块的要点。




# 第五节　建立图形用户界面

## 8.5.1　Container

**Container**

- Container 的两个主要类型是 Window 和 Panel
- Window 是 Java.awt.Window.的对象
- Panel 是 Java.awt.Panel 的对象

Container 有两个主要类型：Window 和 Panel

Window 是 Java.awt.Window.的对象。Window 是显示屏上独立的本机窗口，它独立于其它容器。

Window 有两种形式：Frame(框架)和 Dialog（对话框）。Frame 和 Dialog 是 Window 的子类。Frame 是一个带有标题和缩放角的窗口。对话没有菜单条。尽管它能移动，但它不能缩放。

Panel 是 Java.awt.Panel 的对象。Panel 包含在另一个容器中，或是在 Web 浏览器的窗口中。Panel 确定一个四边形，其它组件可以放入其中。Panel 必须放在 Window 之中（或 Window 的子类中）以便能显示出来。

注—容器不但能容纳组件，还能容纳其它容器，这一事实对于建立复杂的布局是关键的，也是基本的。

滚动块也是 Window 的一个子类。它在模块 10“AWT 组件集”里讨论。

## 8.5.2　定位组件

**定位组件**

- 容器里的组件的位置和大小是由布局管理器决定的。
- 可以通过停用布局管理器来控制组件的大小或位置。

然后必须用组件上的 setLocation()，setSize()，或 setBounds()来定位它们在容器里的位置。

容器里的组件的位置和大小是由布局管理器决定的。容器对布局管理器的特定实例保持一个引用。当容器需要定位一个组件时，它将调用布局管理器来做。当决定一个组件的大小时，同样如此。布局管理器完全控制容器内的所有组件。它负责计算并定义上下文中对象在实际屏幕中所需的大小。

### 8.5.3 组件大小

因为布局管理器负责容器里的组件的位置和大小，因此不需要总是自己去设定组件的大小或位置。如果想这样做（使用 setLocation(),setSize()或 setBounds()方法中的任何一种），布局管理器将覆盖你的决定。

如果必须控制组件的大小或位置，而使用标准布局管理器做不到，那就可能通过将下述方法调用发送到容器中来中止布局管理器：

setLayout(null);

做完这一步，必须对所有的组件使用 setLocation(),setSize()或 setBounds()，来将它们定位在容器中。请注意，由于窗口系统和字体大小之间的不同，这种办法将导致从属于平台的布局。更好的途径是创建布局管理器的新子类。

# 第六节 Frames

**Frames**

- 是 Window 的子类
- 具有标题和缩放角
- 从组件继承并以 add 方式添加组件
- 能以字符串规定的标题来创建不可见框架对象
- 能将 Border Layout 当做缺省布局管理器
- 用 setLayout 方式来改变缺省布局管理器

Frames 是 Window 的一个子类。它是带有标题和缩放角的窗口。它继承于 Java.awt.Container，因此，可以用 add()方式来给框架添加组件。框架的缺省布局管理器就是 Border Layout。它可以用 setLayout()方式来改变。

框架类中的构造程序 Frame(String)用由 String 规定的标题来创建一个新的不可见的框架对象。当它还处于不可见状态时，将所有组件添加到框架中。

```
1.  import java.awt.*;
2.  public class MyFrame extends Frame {
3.  public static void main (String args[]) {
6.        MyFrame fr = new MyFrame("Hello Out There!");
7.        fr.setSize(500,500);
4.  fr.setBackground(Color.blue);
5.         fr.setVisible(true);
6.       }
7.      public MyFrame (String str) {
8.          super(str);
9.      }
10. }
```

上述程序创建了下述框架，它有一个具体的标题、大小及背景颜色。


注—在框架显示在屏幕上之前，必须做成可见的（通过调用程序 setVisible(true)），而且其大小是确定的（通过调用程序 setSize()或 pack()）。

# 第七节 Panels

Panels

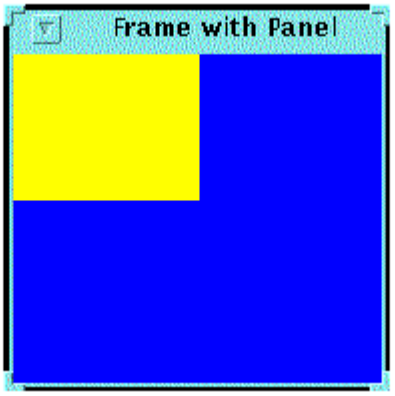
- 为组件提供空间
- 允许子面板拥有自己的布局管理器
- 以 add 方法添加组件

象 Frames 一样，Panels 提供空间来连接任何 GUI 组件，包括其它面板。每个面板都可以有它自己的布管理程序。

一旦一个面板对象被创建，为了能看得见，它必须添加到窗口或框架对象上。用 Container 类中的 add() 方式可以做到这一点。

下面的程序创建了一个小的黄色面板，并将它加到一个框架对象上：

```
1.  import java.awt.*;
2.  public class FrameWithPanel extends Frame {
3.
4.      // Constructor
5.      public FrameWithPanel (String str) {
6.      super (str);
7.      }
8.
9.      public static void main (String args[]) {
10.         FrameWithPanel fr =
11.         new FrameWithPanel ("Frame with Panel");
12.         Panel pan = new Panel();
13.
14.         fr.setSize(200,200);
15.         fr.setBackground(Color.blue);
16.         fr.setLayout(null); //override default layout mgr
17.         pan.setSize (100,100);
18.         pan.setBackground(Color.yellow);
19.
20.         fr.add(pan);
21.         fr.setVisible(true);
22.      }
23.     ....
24.
```




# 第八节 容器布局(Container Layout)

**容器布局**(Container Layout)

- **流程布局**(Flow Layout)
- **边框布局**(Border Layout)
- **网格布局**(Grid Layout)
- **卡布局**(Card Layout)
- **网格包布局**(GridBag Layout)

**容器中组件的布局通常由布局管理器控制。每个** Container（**比如一个** Panel **或一个** Frame）**都有一个与它相关的缺省布局管理器，它可以通过调用** setLayout()**来改变。**

**布局管理器负责决定布局方针以及其容器的每一个子组件的大小。**

# 第九节 布局管理器

**下面的布局管理器包含在** Java **编程语言中：**

- Flow Layout—Panel **和** Applets **的缺省布局管理器**
- Border Layout—Window、Dialog **及** Frame **的缺省管理程序**
- Grid Layout
- Card Layout
- GridBag Layout

GridBag **布局管理器在本模块中不深入讨论。**

Flow Layout **的一个简单例子**

**这个样板代码阐述了几个要点，将在下一节讨论。**

```
1.  import java.awt.*;
2.
3.    public class ExGui {
4.       private Frame f;
5.       private Button b1;
6.       private Button b2;
7.
8.       public static void main(String args[]) {
9.         ExGui guiWindow = new ExGui();
10.        guiWindow.go();
11.      }
12.
```

```
13.      public void go() {
14.        f = new Frame("GUI example");
15.        f.setLayout(new FlowLayout());
16.        b1 = new Button("Press Me");
17.        b2 = new Button("Don't Press Me");
18.        f.add(b1);
19.        f.add(b2);
20.        f.pack();
21.        f.setVisible(true);
22.      }
23. }
```

**main()方法**

本例中第 8 行 main()方法有两个作用。首先，它创建了 ExGui 对象的一个实例。回想一下，直到一个实例存在，还没有被称做 f，b1 和 b2 的真实数据项可以使用。第二，当数据空间被创建时，main()在该实例的上下文中调用实例方法 go()。在 go()中，真正的运行才开始。

**new Frame (“ GUI Example” )**

这个方法创建 Java.awt.Frame 类的一个实例。根据本地协议，在 Java 编程语言中，Frame 是顶级窗口，带有标题条—在这种情况下，标题条由构造程序参数“GUI Example”定义—缩放柄，以及其它修饰。

**f.setLayout (new FlowLayout())**

这个方法创建 Flow 布局管理器的一个实例，并将它安装在框架中。对于每个 Frame、Border 布局来说，都有一个布局管理器，但本例中没有使用。Flow 布局管理器在 AWT 中是最简单的，它在某种程度上象一个页面中的单词被安排成一行一行的那样来定位组件。请注意，Flow 布局缺省地将每一行居中。

**new Button(“ Press Me” )**

这个方法创建 Java.awt.Button 类的一个实例。按钮是从本地窗口工具包中取出的一个标准按钮。按钮标签是由构造程序的字符串参数定义的。

**f.add(b1)**

这个方法告诉框架 f（它是一个容器），它将包容组件 b1。b1 的大小和位置受从这一点向前的 Frame 布局管理器的控制。

**f.pack()**

这个方法告诉框架来设定大小，能恰好密封它所包含的组件。为了确定框架要用多大，f.pack()询问布局管理器，在框架中哪个负责所有组件的大小和位置。

**f.setVisible(true)**

这个方法使框架以及其所有的内容变成用户看得见的东西。

第 190 页代码的最终结果是：

### 8.9.1 Flow 布局管理器

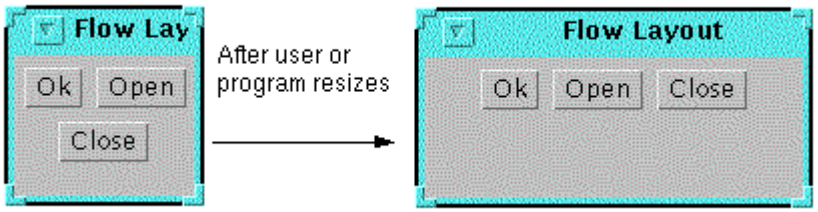
第 190 页例子中所用的 Flow 布局对组件逐行地定位。每完成一行，一个新行便又开始。

与其它布局管理器不一样，Flow 布局管理器不限制它所管理的组件的大小，而是允许它们有自己的最佳大小。

如果想将组件设定缺省居中的话，Flow 布局构造程序参数允许将组件左对齐或右对齐。

如果想在组件之间创建一个更大的最小间隔，可以规定一个界限。

当用户对由 Flow 布局管理的区域进行缩放时，布局就发生变化。如：




**下面的例子就是如何用类容器的** setLayout()**方法来创建** Flow **布局对象并安装它们。**

```
setLayout(new FlowLayout(int align,int hgap, int vgap));
```

**对齐的值必须是** FlowLayout.LEFT, FlowLayout.RIGHT**，或** FlowLayout.CENTER**。例如：**

```
setLayout(new FlowLayout(FlowLayout.RIGHT, 20, 40));
```

**下述程序构造并安装一个新** Flow **布局，它带有规定好的对齐方式以及一个缺省的** 5 **单位的水平和垂直间隙。对齐的值必须是** FlowLayout.LEFT, FlowLayout.RIGHT**，或** FlowLayout.CENTER**。**

```
setLayout(new FlowLayout(int align);
```

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

**下述程序构造并安装一个新** Flow **布局，它带有规定好的居中对齐方式和一个缺省的** 5 **单位的水平和垂直间隙。**

```
setLayout(new FlowLayout());
```

**这个模块代码将几个按钮添加到框架中的一个** Flow **布局中：**

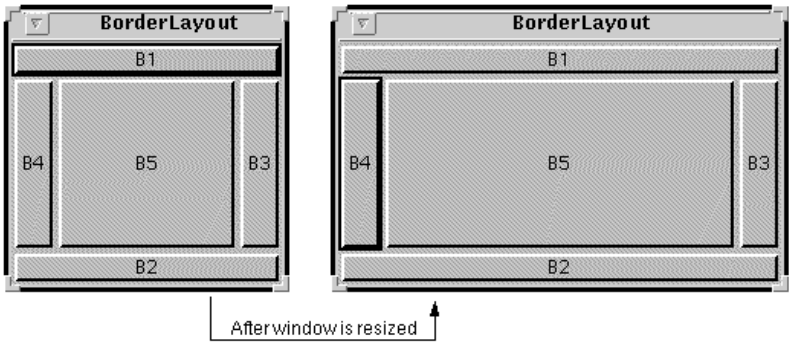
```
1.  import java.awt.*;
2.
3.  public class MyFlow {
4.      private Frame f;
5.      private Button button1, button2, button3;
6.
7.  public static void main (String args[]) {
8.      MyFlow mflow = new MyFlow ();
9.      mflow.go();
10. }
11.
12.   public void go() {
13.     f = new Frame ("Flow Layout");
14.     f.setLayout(new FlowLayout());
15.     button1 = new Button("Ok");
16.     button2 = new Button("Open");
17.     button3 = new Button("Close");
18.     f.add(button1);
19.     f.add(button2);
20.     f.add(button3);
21.     f.setSize (100,100);
22.     f.setVisible(true);
23.   }
24. }
```

### 8.9.2 Border 布局管理器

Border 布局管理器为在一个 Panel 或 Window 中放置组件提供一个更复杂的方案。Border 布局管理器包括五个明显的区域：东、南、西、北、中。

北占据面板的上方，东占据面板的右侧，等等。中间区域是在东、南、西、北都填满后剩下的区域。当窗口垂直延伸时，东、西、中区域也延伸；而当窗口水平延伸时，东、西、中区域也延伸。

Border 布局管理器是用于 Dialog 和 Frame 的缺省布局管理器。下例的代码是在第 193 页上：




---

注—当窗口缩放时，按钮相应的位置不变化，但其大小改变。

---

下面的代码对前例进行了修改，表示出了 Border 布局管理器的特性。可以用从 Container 类继承的 setLayout()方法来将布局设定为 Border 布局。

```
1.  import java.awt.*;
2.
3.  public class ExGui2 {
4.      private Frame f;
5.      private Button bn, bs, bw, be, bc;
6.
7.      public static void main(String args[]) {
8.          ExGui2 guiWindow2 = new ExGui2();
9.          guiWindow2.go();
10.  }
11.
12.  public void go() {
13.      f = new Frame("Border Layout");
14.      bn = new Button("B1");
15.      bs = new Button("B2");
16.      be = new Button("B3");
17.      bw = new Button("B4");
18.      bc = new Button("B5");
19.
20.      f.add(bn, BorderLayout.NORTH);
21.      f.add(bs, BorderLayout.SOUTH);
22.      f.add(be, BorderLayout.EAST);
23.      f.add(bw, BorderLayout.WEST);
24.      f.add(bc, BorderLayout.CENTER);
25.
```

```
26.     f.setSize (200, 200);
27.     f.setVisible(true);
28.  }
29. }
```

下面这一行：

```
setLayout(new BorderLayout());
```

构造并安装一个新 Border 布局，在组件之间没有间隙。

这一行

```
setLayout(new BorderLayout(int hgap, int vgap);
```

构造并安装一个 Border 布局，在由 hgap 和 vgap 规定的组件之间有规定的间隙。

在布局管理器中组件必须被添加到指定的区域，而且还看不见。区域名称拼写要正确，尤其是在选择不使用常量（如 add(button,"Center")）而使用 add(button,BorderLayout.CENTER)时。拼写与大写很关键。

可以使用 Border 布局管理器来产生布局，且带有在缩放时在一个方向、另一方向或双方向上都延伸的元素。

| 注—如果窗口水平缩放，南、北、中区域变化；如果窗口垂直缩放，东、西、中区域变化； |
|---|

如果离开一个 Border 布局未使用的区域，，好象它的大小为 0。中央区域即使在不含组件的情况下仍然呈现为背景。

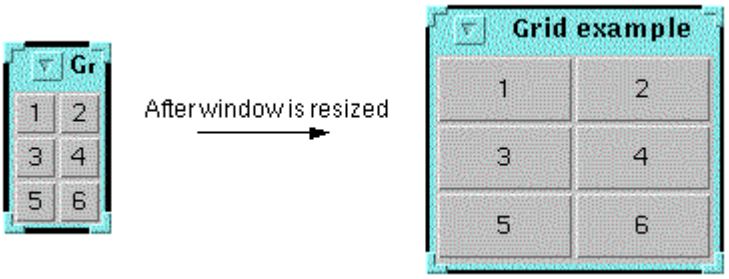
可以仅将单个组件添加到 Border 布局管理器五个区域的每一个当中。如果添加不止一个，只有最后一个看得见。后面的模块将演示如何用中间容器来允许不止一个组件被放在单个 Border 布局管理器区域的空间里。

| 注—布局管理器给予南、北组件最佳高度，并强迫它们与容器一样宽。但对于东、西组件，给予最佳宽度，而高度受到限制。 |
|---|

### 8.9.3 Grid 布局管理器

Grid 布局管理器为放置组件提供了灵活性。用许多行和栏来创建管理程序。然后组件就填充到由管理程序规定的单元中。比如，由语句 new GridLayout(3,2)创建的有三行两栏的 Grid 布局能产生如下六个单元：

因为有 Border 布局管理器，组件相应的位置不随区域的缩放而改变。只是组件的大小改变。




Grid 布局管理器总是忽略组件的最佳大小。所有单元的宽度是相同的，是根据单元数对可用宽度进行平分而定的。同样地，所有单元的高度是相同的，是根据行数对可用高度进行平分而定的。

将组件添加到网格中的命令决定它们占有的单元。单元的行数是从左到右填充，就象文本一样，而页是从上到下由行填充。

行

```
setLayout(new GridLayout());
```

创建并安装一个 Grid 布局，每行中的每个组件有一个栏缺省。

行

```
setLayout(new GridLayout(int rows, int cols));
```

创建并安装一个带有规定好行数和栏数的 Grid 布局。对布局中所有组件所给的大小一样。

下一行：

```
setLayout(new GridLayout(int rows, int cols,  int hgap, int vgap);
```

创建并安装一个带有规定好行数和栏数的网格布局。布局中所有组件所给的大小一样。hgap 和 vgap 规定组件间各自的间隙。水平间隙放在左右两边及栏与栏之间。垂直间隙放在顶部、底部及每行之间。

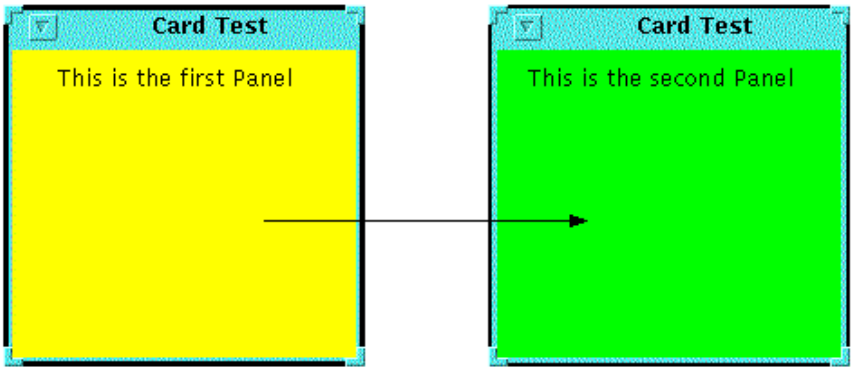
注—行和栏中的一个，不是两个同时，可以为 0。这就是说，任何数量的对象都可以放在一个行或一个栏中。

第 8—27 页上所示的应用程序代码如下：

```
import java.awt.*;
public class GridEx {
    private Frame f;
    private Button b1, b2, b3, b4, b5, b6;

    public static void main(String args[]) {
        GridEx grid = new GridEx();
        grid.go();
    }

    public void go() {
        f = new Frame("Grid example");

        f.setLayout (new GridLayout (3, 2));
        b1 = new Button("1");
        b2 = new Button("2");
        b3 = new Button("3");
        b4 = new Button("4");
        b5 = new Button("5");
        b6 = new Button("6");

        f.add(b1);
        f.add(b2);
        f.add(b3);
        f.add(b4);
        f.add(b5);
        f.add(b6);

        f.pack();
        f.setVisible(true);
    }
}
```

### 8.9.4 Card 布局管理器

Card 布局管理器能将界面看作一系列的卡，其中的一个在任何时候都可见。用 add()方法来将卡添加到 Card 布局中。Card 布局管理器的 show()方法应请求转换到一个新卡中。下例就是一个带有 5 张卡的框架。




**鼠标点击左面板将视图转换到右面板，等等。**

**用来创建上图框架的代码段如下所示：**

```
1. import java.awt.*;
2. import java.awt.event.*;
3.
4. public class CardTest implements MouseListener {
5.
6.     Panel p1, p2, p3, p4, p5;
7.     Label l1, l2, l3, l4, l5;
8.
9.     // Declare a CardLayout object,
10.// to call its methods
11.CardLayout myCard;
12.Frame f;
13.
14.public static void main (String args[]) {
15.    CardTest ct = new CardTest ();
16.    ct.init();
17.}
18.
19.public void init () {
20.    f = new Frame ("Card Test");
21.    myCard = new CardLayout();
22.    f.setLayout(myCard);
23.
24.   // create the panels that I want
25.   // to use as cards
26.   p1 = new Panel();
27.   p2 = new Panel();
28.   p3 = new Panel();
29.   p4 = new Panel();
30.   p5 = new Panel();
31.   // create a label to attach to each panel, and
32.   // change the color of each panel, so they are
33.   // easily distinguishable
34.
35.   l1 = new Label("This is the first Panel");
36.   p1.setBackground(Color.yellow);
```

```
37.   p1.add(l1);
38.
39.   l2 = new Label("This is the second Panel");
40.   p2.setBackground(Color.green);
41.   p2.add(l2);
42.
43.   l3 = new Label("This is the third Panel");
44.   p3.setBackground(Color.magenta);
45.   p3.add(l3);
46.
47.   l4 = new Label("This is the fourth Panel");
48.   p4.setBackground(Color.white);
49.   p4.add(l4);
50.
51.   l5 = new Label("This is the fifth Panel");
52.   p5.setBackground(Color.cyan);
53.   p5.add(l5);
54.
55.   // Set up the event handling here ....
  √
56.   // add each panel to my CardLayout
57.   f.add(p1, "First");
58.   f.add(p2, "Second");
59.   f.add(p3, "Third");
60.   f.add(p4, "Fourth");
61.   f.add(p5, "Fifth");
62.
63.    // display the first panel
64.    myCard.show(f, "First");
65.
66.       f.setSize (200, 200);
67.       f.setVisible(true);
68.    }
```

### 8.9.5 GridBag 布局管理器

**GridBag 布局管理器**

- 复杂布局可以放在网格中
- 单个组件可以采用其最佳大小
- 一个组件能扩展成不止一个单元

除了 Flow、Border、Grid 和 Card 布局管理器外，核心 Java.awt 也提供 GridBag 布局管理器。

GridBag 布局管理器在网格的基础上提供复杂的布局，但它允许单个组件在一个单元中而不是填满整个单元那样地占用它们的最佳大小。网格包布局管理器也允许单个组件扩展成不止一个单元。

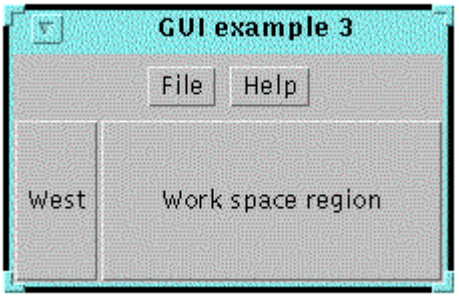
# 第十节 创建面板及复杂布局

下面的程序使用一个面板，允许在一个 Border 布局的北部地区放置两个按钮。这种嵌套法对复杂布局来说是基本的。请注意，就框架本身而论，面板被看作象另一个组件。

```
1.   import java.awt.*;
2.
3.   public class ExGui3 {
4.
5.    private Frame f;
6.    private Panel p;
7.    private Button bw, bc;
8.        private Button bfile, bhelp;.
9.
10.       public static void main(String args[]) {
11.           ExGui3 gui = new ExGui3();
12.           gui.go();
13.       }
14.       public void go() {
15.           f = new Frame("GUI example 3");
16.           bw = new Button("West");
17.           bc = new Button("Work space region");
18.           f.add(bw, BorderLayout.WEST );
19.           f.add(bc, BorderLayout.CENTER );
20.           p = new Panel();
21.           bfile = new Button("File");
22.           bhelp = new Button("Help");
23.           p.add(bfile);
24.           p.add(bhelp);
25.           f.add(p, BorderLayout.NORTH );
26.           f.pack();
27.           f.setVisible(true);
28.       }
29.       }
```

**当这个例子运行时，其结果如下所示：**


**如果缩放窗口，它看起来如下：**



请注意，现在，Border 布局的北部地区有效地保持着两个按钮。事实上，它保持的只是单个面板，但这个面板含有这两个按钮。

面板的大小和位置是由 Border 布局管理器决定的，一个面板的最佳大小是由该面板中组件的最佳大小来决定的。面板中按钮的大小和位置受 Flow 布局的控制，该布局是由缺省与面板相关联的。

# 练习：建立 Java GUIs

练习目的—在这个实验中，开发两个图形用户界面。

一、准备

为了成功地完成这个实验，必须理解图形用户界面的目的，掌握如何用布局管理器来创建图形用户界面。

二、任务

一级实验：创建计算器 GUI

创建下述 GUI：


三级实验：创建帐目 GUI

创建一个 GUI，它能给模块 5 中创建的 Teller.Java 应用程序提供前端用户界面。需要研究一些本模块中没有描述的组件。(它们将在本课程的后面讲解)。

三、练习总结

讨论—花几分钟讨论在实验练习中的经验、问题或发现。

- 经验 解释 总结 应用

四、检查进步情况

在继续下一个模块前，检查一下，你确实能：

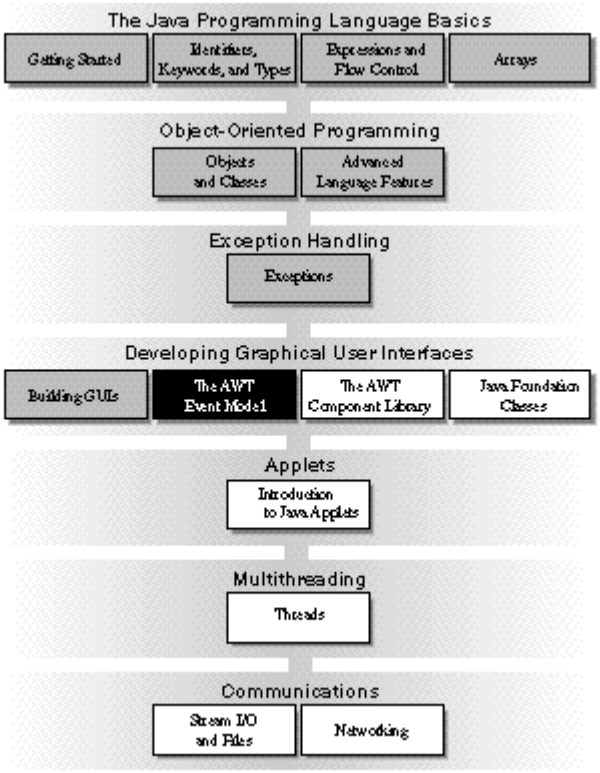
- 描述 AWT 包及其组件
- 定义 Container 组件及布局管理器等术语，以及它们如何一起工作来建立 GUI 的
- 使用布局管理器
- 使用 Flow、Border、Grid 和 Card 布局管理器来获得所需的动态布局
- 添加组件到容器
- 正确使用框架和面板容器
- 描述复杂布局与嵌套容器是如何工作的
- 在 Java 程序中，确认下列：
  - 容器
  - 相关的布局管理器
  - 所有组件的布局层次

五、思考

现在知道如何在计算机屏幕上显示一个 GUI 了，那么，要使 GUI 有用，还需要什么呢？

# 第9章 AWT 事件模型

本模块讨论了事件驱动的图形用户界面(GUI)的用户输入机制。




## 第一节 相关问题

讨论 － 以下为与本模块内容有关的问题：

- 哪些部分对于一个图形用户界面来说是必需的？
- 一个图形化程序如何处理鼠标点击或者其他类型的用户交互？

## 第二节 目 标

完成本模块之后，你应当能够：

- 编写代码来处理在图形用户界面中发生的事件
- 描述 Adapter 类的概念，包括如何和何时使用它们
- 根据事件对象的细节来确定产生事件的用户动作
- 为各种类型的事件创建合适的接口和事件处理器。

## 第三节 什么是事件？

**什么是事件？**

- 事件－描述发生了什么的对象
- 事件源－事件的产生器
- 事件处理器－接收事件、解释事件并处理用户交互的方法

如果用户在用户界面层执行了一个动作(鼠标点击和按键)，这将导致一个事件的发生。事件是描述发生了什么的对象。存在各种不同类型的事件类用来描述各种类型的用户交互。

### 9.3.1 事件源

事件源是一个事件的产生者。例如，在 Button 组件上点击鼠标会产生以这个 Button 为源的一个 ActionEvent. 这个 ActionEvent 实例是一个对象，它包含关于刚才所发生的那个事件的信息的对象。这些信息包括：

- getActionCommand－返回与动作相关联的命令名称。
- GetModifiers－返回在执行动作时持有的修饰符。

### 9.3.2 事件处理器

事件处理器就是一个接收事件、解释事件并处理用户交互的方法。

# 第四节 JDK1.0 的事件模型与 JDK1.2 的事件模型比较

JDK1.0 的事件模型与 JDK1.2 的事件模型比较

- 层次模型(JDK 1.0)
- 委托模型(JDK 1.2)

在 JDK1.1 中，事件接收和处理的方法发生了重要的改变。本节将比较以前的事件模型(JDK1.0)和当前的事件模型(JDK1.1 和 JDK1.2)。

JDK1.0 采用的是层次事件模型，而 JDK1.1 以及更高的版本采用的是委托事件模型。

### 9.4.1 层次模型(JDK1.0)

层次模型是基于容器的。 事件先发送到组件，然后沿容器层次向上传播。没有被组件处理的事件会自动地继续传播到组件的容器。

### JDK1.0 的事件模型与 JDK1.2 的事件模型比较

例如，在下图中，Button 对象(包含在一个 Frame 上的 Panel 中)上的鼠标点击首先向 Button 发送一个动作事件。如果它没有被 Button 处理，这个事件会被送往 Panel，如果它在那儿仍然没有被处理，这个事件会被送往 Frame。

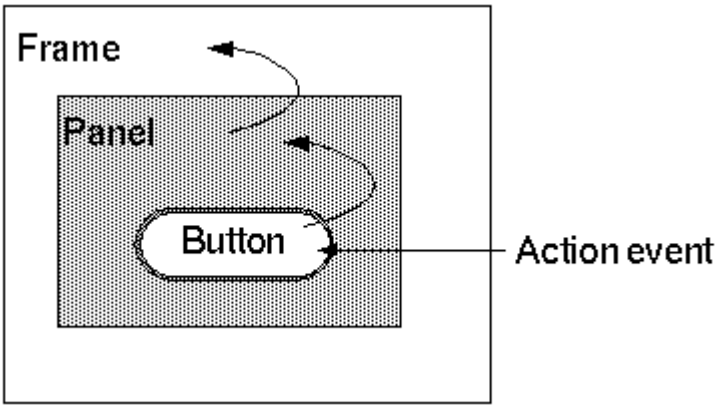
层次模型(JDK1.0)

优点

- 简单，而且非常适合面向对象的编程环境。

缺点

- 事件只能由产生这个事件的组件或包含这个组件的容器处理。
- 为了处理事件，你必须定义接收这个事件的组件的子类，或者在基容器创建 handleEvent()方法。




### 层次模型(JDK1.0)(续)

这种模型有一个显著的优点：

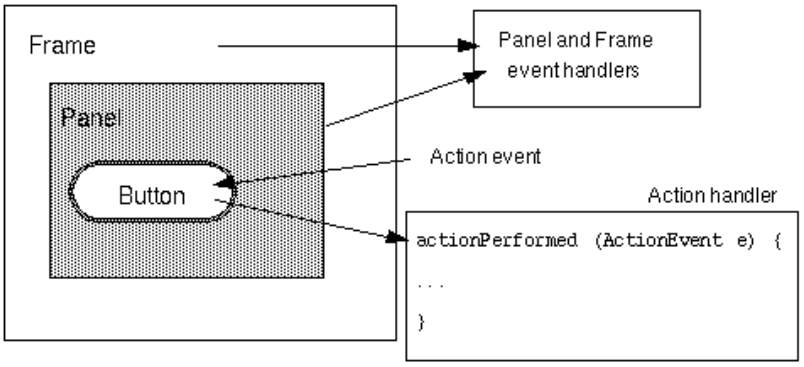
- 它简单，而且非常适合面向对象的编程环境；说到底，所有的组件都继承了 java.awt.Component 类，而 handleEvent()就在 java.awt.Component 类中。

然而，这种模型也存在缺点：

- 事件只能由产生这个事件的组件或包含这个组件的容器处理。这个限制违反了面向对象编程的一个基本原则：功能应该包含在最合适的类中。而最适合处理事件的类往往并不是源组件的容器层次中的成员。

- 大量的 CPU 周期浪费在处理不相关的事件上。任何对于程序来说不相关或者并不重要的事件会沿容器层次一路传播，直到最后被抛弃。不存在一种简单的方法来过滤事件。
- 为了处理事件，你必须定义接收这个事件的组件的子类，或者在基容器创建一个庞大的 handleEvent()方法。

委托事件模型是在 JDK1.1 中引入的。在这个模型中，事件被送往产生这个事件的组件，然而，注册一个或多个称为监听者的类取决于每一个组件，这些类包含事件处理器，用来接收和处理这个事件。采用这种方法，事件处理器可以安排在与源组件分离的对象中。监听者就是实现了 Listener 接口的类。




事件是只向注册的监听者报告的对象。每个事件都有一个对应的监听者接口，规定哪些方法必须在适合接收那种类型的事件的类中定义。实现了定义那些方法的接口的类可以注册为一个监听者。

### 9.4.2 委托模型

从没有注册的监听者的组件中发出的事件不会被传播。

例如，这是一个只含有单个 Button 的简单 Frame。

```
1.import java.awt.*;
2.
3.public class TestButton {
4.public static void main(String args[]) {
5.Frame f = new Frame("Test");
6.Button b = new Button("Press Me!");
7.b.addActionListener(new ButtonHandler());
8.f.add(b,"Center");
9.f.pack();
10.f.setVisible(true);
11.}
12.}
```

ButtonHandler 类是一个处理器类，事件将被委托给这个类。

```
1.import java.awt.event.*;
2.
3.public class ButtonHandler implements ActionListener {
4.public void actionPerformed(ActionEvent e) {
5.System.out.println("Action occurred");
6.System.out.println("Button's label is : '
7.+ e.getActionCommand());
8.}
9.}
```

这两个范例的特征如下：

- Button 类有一个 addActionListner(ActionListener)方法。
- AddActionListner 接口定义了一个方法 actionPerformed，用来接收一个 ActionEvent。

- 创建一个 Button 对象时，这个对象可以通过使用 addActionListener 方法注册为 ActionEvents 的监听者。调用这个方法时带有一个实现了 ActionListener 接口的类的参数。

**委托模型(JDK1.1 或更高版本)**

**优点**

- 事件不会被意外地处理。
- 有可能创建并使用适配器 (adapter)类对事件动作进行分类。
- 委托模型有利于把工作分布到各个类中。

**缺点**

- 不容易将 JDK1.0 代码移植到 JDK1.1 上。

**委托模型(JDK1.1 或更高版本)(续)**

- 在 Button 对象上用鼠标进行点击时，将发送一个 ActionEvent 事件。这个 ActionEvent 事件会被使用 addActionListener()方法进行注册的所有 ActionListener 的 actionPerformed()方法接收。
- ActionEvent 类的 getActionCommand()方法返回与动作相关联的命令名称。以按钮的点击动作为例，将返回 Button 的标签。

这种方法有若干优点：

- 事件不会被意外地处理。在层次模型中，一个事件可能传播到容器，并在非预期的层次被处理。
- 有可能创建并使用适配器(adapter)类对事件动作进行分类。
- 委托模型有利于把工作分布到各个类中。
- 新的事件模型提供对 JavaBeans 的支持。

这种方法也有一个缺点：

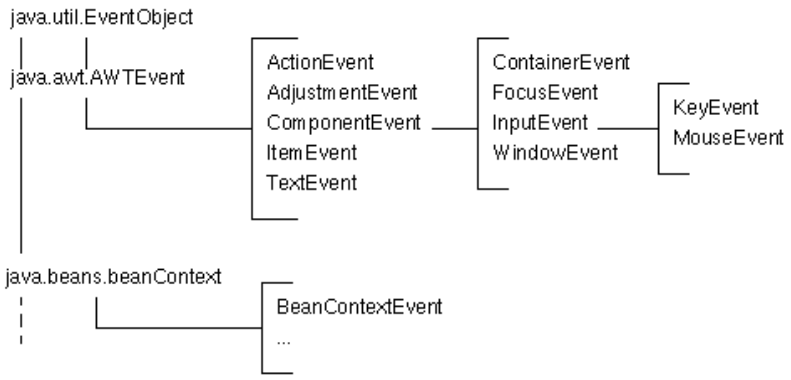
- 尽管当前的 JDK 支持委托模型和 JDK1.0 事件模型，但不能混合使用 JDK1.0 和 JDK1.1。

# 第五节 图形用户界面的行为

## 9.5.1 事件类型

我们已经介绍了在单一类型事件上下文中从组件接收事件的通用机制。事件类的层次结构图如下所示。许多事件类在 java.awt.event 包中，也有一些事件类在 API 的其他地方。

对于每类事件，都有一个接口，这个接口必须由想接收这个事件的类的对象实现。这个接口还要求定义一个或多个方法。当发生特定的事件时，就会调用这些方法。表 9-1 列出了这些(事件)类型，并给出了每




个类型对应的接口名称，以及所要求定义的方法。这些方法的名称是易于记忆的，名称表示了会引起这个方法被调用的源或条件。

表 9-1 方法类型和接口

| Category | Interface Name | Methods |
|---|---|---|
| Action | `ActionListener` | `actionPerformed(ActionEvent)` |
| Item | `ItemListener` | `itemStateChanged(ItemEvent)` |
| Mouse motion | `MouseMotionListener` | `mouseDragged(MouseEvent)` |
| | | `mouseMoved(MouseEvent)` |

| | | |
|---|---|---|
| Mouse button | MouseListener | mousePressed(MouseEvent) |
| | | mouseReleased(MouseEvent) |
| | | mouseEntered(MouseEvent) |
| | | mouseExited(MouseEvent) |
| | | mouseClicked(MouseEvent) |
| Key | KeyListener | keyPressed(KeyEvent) |
| | | keyReleased(KeyEvent) |
| | | keyTyped(KeyEvent) |
| Focus | FocusListener | focusGained(FocusEvent) |
| | | focusLost(FocusEvent) |
| Adjustment | AdjustmentListener | adjustmentValueChanged |
| | | (AdjustmentEvent) |
| Component | ComponentListener | componentMoved(ComponentEvent) |
| | | componentHidden(ComponentEvent) |
| | | componentResized(ComponentEvent) |
| | | componentShown(ComponentEvent) |

**表 9-1 方法类型和接口(续)**

| Category | Interface Name | Methods |
|---|---|---|
| Window | WindowListener | windowClosing(WindowEvent) |
| | | windowOpened(WindowEvent) |
| | | windowIconified(WindowEvent) |
| | | windowDeiconified(WindowEvent) |
| | | windowClosed(WindowEvent) |
| | | windowActivated(WindowEvent) |
| | | windowDeactivated(WindowEvent) |
| Container | ContainerListener | componentAdded(ContainerEvent) |
| | | componentRemoved(ContainerEvent) |
| Text | TextListener | textValueChanged(TextEvent) |

### 9.5.2 复杂的范例

本节将考察一个更复杂的 Java 软件范例。它将跟踪鼠标被按下时，鼠标的移动情况(鼠标拖动)。这个范例还将监测当鼠标没有按下时，鼠标的移动情况(鼠标移动)。

当鼠标按下或没有按下时，移动鼠标产生的事件会被实现了 MouseMotionListener 接口的类的对象检取。这个接口要求定义两个方法，mouseDragged()和 mouseMoved()。即使你只对鼠标拖动感兴趣，也必须提供这两个方法。然而，mouseMoved()的体可以是空的。

要检取其他鼠标事件，包括鼠标点击，必须定义 MouseListener 接口。这个接口包括若干个事件，即：mouseEntered, mouseExited, mousePressed, mouseReleased 和 mouseClicked。

发生鼠标或键盘事件时，有关鼠标的位置和所按下的键的信息可以从事件中得到。代码如下：

```
1.import java.awt.*;
2.import java.awt.event.*;
3.
4.public class TwoListen implements
5.MouseMotionListener, MouseListener {
6.private Frame f;
7.private TextField tf;
8.
```

```
 9.public static void main(String args[]) {
10.TwoListen two = new TwoListen();
11.two.go();
12.}
13.
```

**复杂的范例(续)**

```
 1.public void go() {
 2.f = new Frame("Two listeners example");
 3.f.add (new Label ("Click and drag the mouse"),
 4."BorderLayout.NORTH");
 5.tf = new TextField (30);
 6.f.add (tf, "BorderLayout.SOUTH");
 7.
 8.f.addMouseMotionListener(this);
 9.f.addMouseListener (this);
10.f.setSize(300, 200);
11.f.setVisible(true);
12.}
13.
14.// These are MouseMotionListener events
15.public void mouseDragged (MouseEvent e) {
16.String s =
17."Mouse dragging: X = " + e.getX() +
18." Y = " + e.getY();
19.tf.setText (s);
20.}
21.
22.public void mouseMoved (MouseEvent e) {
23.}
24.
25.// These are MouseListener events
26.public void mouseClicked (MouseEvent e) {
27.}
28.
29.public void mouseEntered (MouseEvent e) {
30.String s = "The mouse entered";
31.tf.setText (s);
32.}
```

**复杂的范例(续)**

```
 1.public void mouseExited (MouseEvent e) {
 2.String s = "The mouse has left the building";
 3.tf.setText (s);
 4.}
 5.
 6.public void mousePressed (MouseEvent e) {
 7.}
 8.
```

```
 9.public void mouseReleased (MouseEvent e) {
10.}
11.}
```

这个范例中的一些地方值得注意。它们将在以下的几节中讨论。

定义多重接口

这个类由第 4 行中的如下代码声明：

```
implements MouseMotionListener, MouseListener
```

声明多个接口时，可以用逗号隔开。

监听多个源

如果你在第 11 行和第 12 行中，调用如下方法：

两种类型的事件都会引起 TwoListen 类中的方法被调用。一个对象可以“监听”任意数量的事件源；它的类只需要实现所要求的接口。

```
f.addMouseListener(this);
f.addMouseMotionListener(this);
```

**复杂的范例(续)**

获取关于事件的细节

调用处理器方法(如 mouseDragged())时所带的事件参数可能会包含关于源事件的重要信息。为了确定可以获取关于某类事件的哪些细节时，你应当查看 java.awt.event 包中合适的类文档。

### 9.5.3 多监听者

| 多监听者 |
| --- |
| - 多监听者可以使一个程序的不相关部分执行同样的动作。<br>- 事件发生时，所有被注册的监听者的处理器都会被调用 |

AWT 事件监听框架事实上允许同一个组件带有多个监听者。一般地，如果你想写一个程序，它基于一个事件而执行多个动作，把那些行为编写到处理器的方法里即可。然而，有时一个程序的设计要求同一程序的多个不相关的部分对于同一事件作出反应。这种情况是有可能的，例如，将一个上下文敏感的帮助系统加到一个已存在的程序中。

监听者机制允许你调用 addXXXlistener 方法任意多次，而且，你可以根据你的设计需要指定任意多个不同的监听者。事件发生时，所有被注册的监听者的处理器都会被调用。

---

注－没有定义调用事件处理器方法的顺序。通常，如果调用的顺序起作用，那么处理器就不是不相关的。在这种情况下，只注册第一个监听者，并由它直接调用其它的监听者。

---

### 9.5.4 事件 Adapters(适配器)

你定义的 Listener 可以继承 Adapter 类，而且只需重写你所需要的方法。例如：

实现每个 Listener 接口的所有方法的工作量是非常大的，尤其是 MouseListener 接口和 ComponentListener 接口。

以 MouseListener 接口为例，它定义了如下方法：

- `mouseClicked(MouseEvent)`
- `mouseEntered(MouseEvent)`
- `mouseExited(MouseEvent)`
- `mousePressed(MouseEvent)`
- `mouseReleased(MouseEvent)`

为了方便起见，Java 语言提供了 Adapters 类，用来实现含有多个方法的类。这些 Adapters 类中的方法是空的。

你可以继承 Adapters 类，而且只需重写你所需要的方法。例如：

```
1.import java.awt.*;
```

```
 2.import java.awt.event.*;
 3.
 4.public class MouseClickHandler extends MouseAdapter {
 5.
 6.// We just need the mouseClick handler, so we use
 7.// the Adapter to avoid having to write all the
 8.// event handler methods
 9.public void mouseClicked (MouseEvent e) {
10.// Do something with the mouse click...
11.}
12.}
```

### 9.5.5 匿名类

**可以在一个表达式的域中，包含整个类的定义。这种方法定义了一个所谓的匿名类并且立即创建了实例。匿名类通常和 AWT 事件处理一起使用。例如：**

```
 1.import java.awt.*;
 2.import java.awt.event.*;
 3.public class AnonTest {
 4.private Frame f;
 5.private TextField tf;
 6.
 7.public static void main(String args[])
 8.{
 9.AnonTest obj = new AnonTest();
10.obj.go();
11.}
12.
```

**匿名类（续）**

```
 1.public void go() {
 2.f = new Frame("Anonymous classes example");
 3.f.add(new Label("Click and drag the " +
 4." mouse",BorderLayout.NORTH);
 5.tf = new TextField (30);
 6.f.add(tf,BorderLayout.SOUTH);
 7.
 8.f.addMouseMotionListener ( new
 9.MouseMotionAdapter() {
10.public void mouseDragged (MouseEvent e) {
11.String s =
12."Mouse dragging: X = " + e.getX() +
13." Y = " + e.getY();
14.tf.setText (s);
15.}
16.} ); // <- note the closing parenthesis
17.
18.f.addMouseListener (new MouseClickHandler());
19.f.setSize(300, 200);
20.f.setVisible(true);
```

```
21.}
22.}
```

# 练习：事件

**练习目标－你将编写、编译和运行包含事件处理器的** Calculator **图形用户界面 和** Account **图形用户界面的修改版本。**

**一、准备**

**为了很好地完成这个练习，你必须对事件模型是如何工作的有一个清晰的了解。**

**二、任务**

**水平** 1**：创建一个** Calculator **图形用户界面，第二部分**

1. **使用你在模块** 8 **中创建的图形用户界面代码，编写一段事件代码，用来连接计算器的用户界面和处理计算器上的函数的事件处理器。**

**水平** 3**：创建一个** Account **图形用户界面，第二部分**

1. **创建一个** AccountEvent **类，类的对象在帐目发生改变时被激活。然后激活送往** Bankmanager **类的事件。根据你在模块** 8 **中创建的** Teller.java GUI **代码为起点进行练习。**

**三、练习小结**

**讨论 － 花几分钟时间讨论一下，在本实验练习过程中你都经历、提出和发现了什么。**

**经验 解释 总结 应用**

**四、检查一下你的进度**

**在进入下一个模块的学习之前，请确认你能够：**

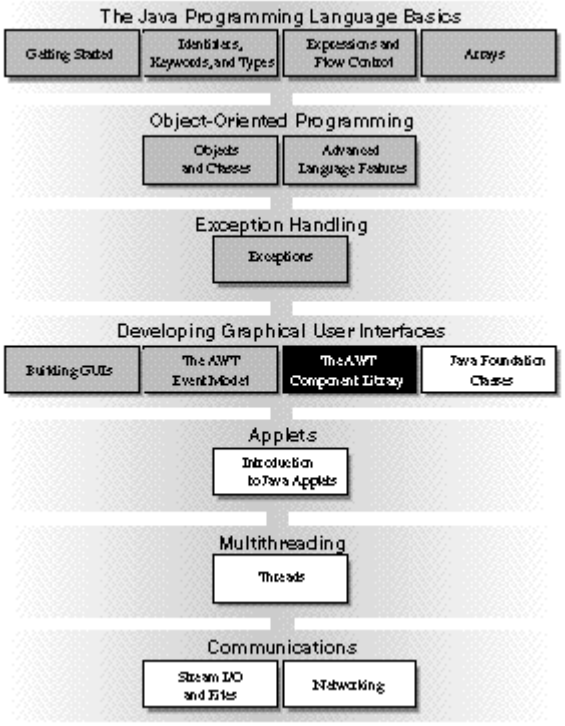
- **编写代码来处理在图形用户界面中发生的事件**
- **描述** Adapter **类的概念，包括如何和何时使用它们**
- **根据事件对象的细节来确定产生事件的用户动作**
- **为各种类型的事件创建合适的接口和事件处理器。**

**五、思考**

**你现在已经知道如何为图形化输出和交互式用户输入来创建** Java **图形用户界面。然而，我们只介绍了能创建图形用户界面的一些组件。其他的组件在图形用户界面中有什么用处呢？**

# 第10章 AWT 组件库

JDK **提供了能创建图形用户界面的许多组件。本模块考察这些** AWT **组件，以及非组件的** AWT **类，例如** Color、Font **和图形用户界面的打印。**




## 第一节 相关问题

讨论一下面的问题和本模块中的内容有关：

你现在已经知道如何为图形化输出和交互式用户输入来创建 Java 图形用户界面。然而，我们只介绍了能创建图形用户界面的一些组件。其他的组件在图形用户界面中有什么用处呢？

## 第二节 目 标

完成本模块之后，你应该能够：

- 认识关键的 AWT 组件。
- 给你一个用户界面的描述，能够用 AWT 组件来创建一个用户界面。
- 给你一个 AWT 程序，能够改变 AWT 组件的颜色和字体。
- 使用 Java 打印机制来打印一个用户界面。

## 第三节 AWT 的特点

| AWT 的特性 |
| --- |
| - AWT 组件提供了控制界面外观，包括颜色和字体的机制 |
| - AWT 还支持打印(这是在 JDK1.1 版中引入的) |

AWT 提供了各种标准特性。本模块将介绍你可以使用的组件，并且概述了你需要知道的一些特殊情形。

首先将描述 AWT 的各个组件。它们用来创建用户界面。你需要知道所有图形用户界面组件，这样你就可以在创建你自己的界面时选择合适的组件。

AWT 组件提供了控制界面外观的机制，包括用于文本显示的颜色和字体。

此外，AWT 还支持打印。这个功能是在 JDK1.1 版中引入的。

### 10.3.1 按钮(Button)

你已经比较熟悉 Button 组件了。这个组件提供了“按下并动作”的基本用户界面。可以构造一个带文本标签的按钮，用来告诉用户它的作用。

```
Button b = new Button("Sample");
b.addActionListener(this);
add(b);
```


任何实现了被注册为监听者的 ActionListener 接口的类，它的 actionPerformed()方法将在一个按钮被鼠标点击“按下”时被调用。

```
public void actionPerformed(ActionEvent ae) {
  System.out.println("Button press received.");
  System.out.println("Button's action command is: " +
ae.getActionCommand());
}
```

按钮被按下时调用的 getActionCommand()方法在缺省情况下将返回标签字符串。用按钮的 setActionCommand()方法改变动作命令和标签。

```
Button b = new Button("Sample");
b.setActionCommand("Action Command Was Here!");
b.addActionListener(this);
add(b);
```

---

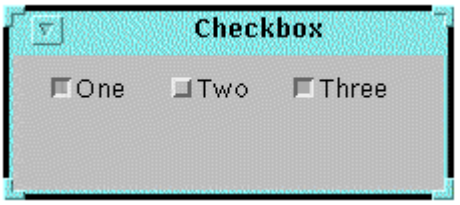
注－SampleButton 和 ActionCommandButton 的完整源代码可以在 course_examples 目录下找到。

---

### 10.3.2 复选框(Checkbox)

Checkbox 组件提供一种简单的“开/关”输入设备，它旁边有一个文本标签。

```
Frame f = new Frame("Checkbox")
Checkbox one = new Checkbox("One", true);
Checkbox two = new Checkbox("Two", false);
Checkbox three = new Checkbox("Three", true);
one.addItemListener(this);
two.addItemListener(this);
three.addItemListener(this);
f.add(one);
f.add(two);
f.add(three);
```




选取或不选取(取消)一个复选框的事件将被送往 ItemListener 接口。所传递的 ItemEvent 包含 getStatechange()方法，它根据实际情况返回 ItemEvent.DESELECTED 或 ItemEvent.SELECTED。getItem()方法将受到影响的复选框作为一个表示这个复选框标签的 String 对象返回。

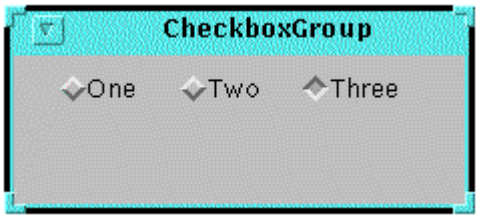
```
class Handler implements ItemListener {
   public void itemStateChanged(ItemEvent ev) {
      String state = "deselected";
      if (ev.getStateChange() == ItemEvent.SELECTED){
            state = "selected";
         }
      System.out.println(ev.getItem() + " " + state);
```

```
        }
    }
```

### 10.3.3 复选框组－单选框(Checkbox group-Radio Button)

复选框组提供了将多个复选框作为互斥的一个集合的方法，因此在任何时刻，这个集合中只有一个复选框的值是 true。值为 true 的复选框就是当前被选中的复选框。你可以使用带有一个额外的 CheckboxGroup 参数的构造函数来创建一组中的每个复选框。正是这个 CheckBoxGroup 对象将各个复选框连接成一组。如果你这么做的话，那么复选框的外观会发生改变，而且所有和一个复选框组相关联的复选框将表现出“单选框”的行为。

```
Frame f = new Frame("Checkbox Group");
CheckboxGroup cbg = new CheckboxGroup();
Checkbox one = new Checkbox("One", false, cbg);
Checkbox two = new Checkbox("Two", false, cbg);
Checkbox three = new Checkbox("Three", true, cbg);
one.addItemListener(this);
two.addItemListener(this);
three.addItemListener(this);
f.add(one);
f.add(two);
f.add(three);
```

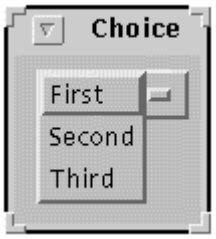



### 10.3.4 下拉列表(Choice)

下拉列表组件提供了一个简单的“从列表中选取一个”类型的输入。例如：

```
Frame f = new Frame("Choice");
Choice c = new Choice();
c.add("First");
c.add("Second");
c.add("Third");
c.addItemListener(this);
f.add(c);
```


点击下拉列表组件时，它会显示一个列表，列表中包含了所有加入其中的条目。注意所加入的条目是 String 对象。




ItemListener 接口用来观察下拉列表组件的变化，其细节和复选框的相同。

### 10.3.5 画布(Canvas)

画布提供了一个空白(背景色)的空间。除非你用 setSize()显式地定义它的大小，否则它的大小就是 0×0。

画布的空间可以用来绘图、显示文本、接收键盘或鼠标的输入。后面的模块将告诉你如何有效地在 AWT 中绘图。

通常，画布用来提供一个一般的绘图空间或者为客户组件提供工作区域。


画布可以“监听”所有适用于一个普通组件的事件。特别地，你还可能想增加 KeyListener、MouseMotionListener 和 MouseListener 对象，以允许某种方式对用户输入作出反应。

---

注－如要在画布中接收键盘事件，必须调用画布的 requestFocus()方法。如果缺少这个步骤，一般是不能将击键“导向”画布的。键盘事件会送往另一个组件，或整个地丢失了。示例代码 overleaf(第 228 页)表明了这一点。

---

下面是画布的一个范例。每击一次键，这个程序就改变一次画布的颜色。

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
public class MyCanvas extends Canvas
implements KeyListener {
    int index;
    Color colors[] = {Color.red, Color.green, Color.blue };
    public void paint(Graphics g) {
         g.setColor(colors[index]);
         g.fillRect(0,0,getSize().width,getSize().height);
      }

    public static void main(String args[]) {
       Frame f = new Frame("Canvas");
       MyCanvas mc = new MyCanvas();
       f.add(mc,BorderLayout.CENTER);
       f.setSize(150, 150);
```

```
            mc.requestFocus();
            mc.addKeyListener(mc);
            f.setVisible(true);
        }

        public void keyTyped(KeyEvent ev) {
             index++;
             if (index == colors.length) {
                    index =0;
             }
             repaint();
        }

        public void keyPressed(KeyEvent ev) {
        }

        public void keyReleased(KeyEvent ev) {
        }
    }
```

### 10.3.6 标签(Label)

一个标签对象显示一行静态文本。程序可以改变文本，但用户不能改变。标签没有任何特殊的边框和装饰。

```
Label l = new Label( " Hello " );
add(l);
```


标签通常不处理事件，但也可以按照和画布相同的方式来处理事件。也就是说，只有调用了requestFocus()方法后，才能可靠地检取击键事件。

### 10.3.7 文本域(Textfield)

文本域是一个单行的文本输入设备。例如：

```
TextField f = new TextField("Single line" , 30);
f.addActionListener(this);
add(f);
```


因为只允许有一行，所以当按下 Enter 或 Return 键时，ActionListener 可以通过 actionPerformed()知道这个事件。如果需要，还可以增加其他的组件监听者。

除了注册一个 ActionListener，你还可以注册一个 TextListener 来接收关于个别击键的通知。它的回调方法是 textValueChanged(TextEvent)。

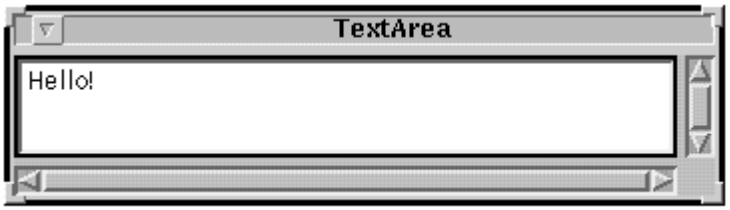
### 10.3.8 文本区(TextArea)

文本区是一个多行多列的文本输入设备。你可以用 setEditable(boolean)将它设置成只读的。文本区将显

示水平和垂直的滚动条。

下面这个范例创建了一个 4 行×30 字符的文本，最初它含有“Hello!”。

```
TextArea t = new TextArea( " Hello! " , 4, 30);
t.addTextListener(this);
add(t);
```




你用 addTexListener 指定的监听者将以和文本域相同的方式接收到关于击键的通知。

你可以给文本区增加一般的组件监听者，然而，由于文本是多行的，按下 Enter 键将导致把另一个字符送入缓冲。如果你需要识别“输入的结束”，你可以在文本区旁放置一个“应用”或“确认”按钮，以便用户指明“输入的结束”。

### 10.3.9 文本组件(Text Components)

**文本组件**

- 文本区和文本域是子类
- 文本组件实现了 TextListener

文本区和文本域的文档都分为两个部分。如果你查找一个称为文本组件的类，你会找到若干个文本区和文本域共有的方法。例如，文本区和文本域都是文本组件的子类。

你已经知道文本区和文本域类的构造函数允许你指定显示所用的列数。记住所显示的组件大小是由布局管理器决定的，因此这些设置可能被忽略。进而，列数是按所用字体的平均宽度计算的。如果使用一种空间比例字体，实际显示的字符数可能相差很大。

由于文本组件实现了 TextListener，诸如文本域、文本区及其它子类都有对击键事件的内置支持。

### 10.3.10 列表(list)

一个列表将各个文本选项显示在一个区域中，这样就可以在同时看到若干个条目。列表可以滚动，并支持单选和多选两种模式。例如：

```
List l = new List(4, true);
l.add("Hello");
l.add("there");
l.add("how");
l.add("are");
```


构造函数的数值参数定义了按可见列计算的列表高度。这个值也可能被布局管理器覆盖。一个值为 true 的布尔型参数表明这个列表允许用户作多个选择。


选取或取消一个条目时，AWT 将一个 ItemEvent 的实例送往列表。用户双击滚动列表中的一个条目时，单选模式和多选模式的列表都会产生一个 ActionEvent。根据每个平台的约定来决定列表中的条目是否被选取。对于 UNIX/Motif 环境，单击会加亮列表中的一个条目，只有双击才会触发列表事件(选取条目)。

### 10.3.11 对话框(Dialog)

对话框组件与一个框架相关联。它是一个带有一些装饰的自由窗口。它与框架的区别在于所提供的一些装饰，而且你可以生成一个“模式”对话框，它在被关闭前将存储所有窗口的输入。

对话框可以是无模式和模式的。对于无模式对话框，用户可以同时与框架和对话框交互。“模式”对话


框在被关闭前将阻塞包括框架在内的其他所有应用程序的输入。

由于对话框是窗口的子类，所以它的缺省布局管理器是 Border Layout。

```
Dialog d = new Dialog(f, "Dialog", false);
d.add(new Label("Hello, I'm a Dialog",Border.Layout.CENTER));
d.pack();
```

对话框在创建时通常是不可见的。通常在对按下按钮等用户输入作出反应时，才显示对话框。

```
public void actionPerformed(ActionEvent ev) {
   d.setVisible(true);
}
```

---

注－可以把对话框作为一个可重用设备。也就是说，在显示上关闭对话框，并不销毁这个对象；保存这个对象以便以后使用。垃圾回收站使得浪费内存变得非常容易对付。记住，创建和初始化对象耗费时间，所以不应该不加考虑就进行创建和初始化。

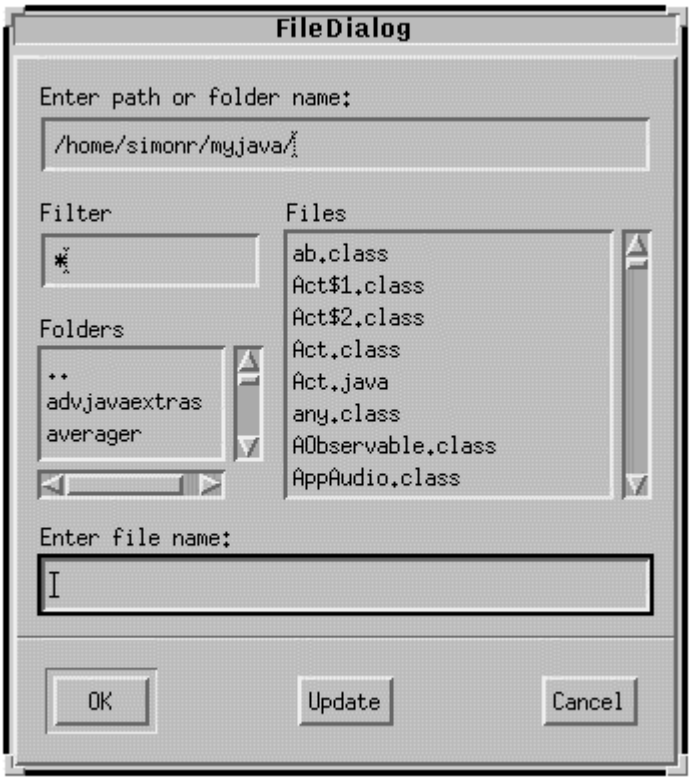
---

要隐藏对话框，你必须调用 setVisible(false)。典型的做法是对它添加一个 WindowListener，并等待对那个监听者的 windowClosing()调用。这和处理一个框架的关闭是平行的。

### 10.3.12 文件对话框(File Dialog)

文件对话框是文件选择设备的一个实现。它有自己的自由窗口，以及窗口元素，并且允许用户浏览文件系统，以及为以后的操作选择一个特定的文件。例如：

```
FileDialog d = new FileDialog(parentFrame, "FileDialog");
d.setVisible(true);// block here until OK selected
String fname = d.getFile();
```

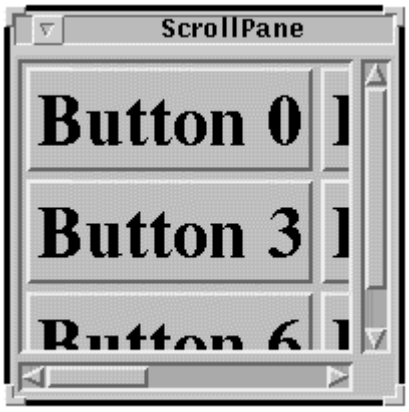



通常并不需要处理 FileDialog 的事件。调用 setVisible(true)将阻塞事件，直至用户选择 OK，这时会请求用户选择的文件名。这个信息将作为一个 String 返回。

### 10.3.13 滚动面板(Scroll Pane)

滚动面板提供了一种不能作为自由窗口的通用容器。它应当总是和一个容器相关联(例如，框架)。它提供了到一个更大的区域的视窗以及操纵这个视窗的滚动条。例如：

```
Frame f = new Frame("ScrollPane");
Panel p = new Panel();
ScrollPane sp = new ScrollPane();
p.setLayout(new GridLayout(3, 4));

sp.add(p);
f.add(sp, "Center");
f.setSize(200, 200);
f.setVisible(true);
```




滚动面板创建和管理滚动条，并持有一个组件。你不能控制它所用的布局管理器。你可以将一个面板加入到滚动面板中，配置面板的布局管理器，并在那个面板中放置你的组件。

通常，你不处理滚动面板上的事件；这些事件通过滚动面板所包含的组件进行处理。

# 第四节 菜 单

菜单(Menus)
必须添加到菜单管理器中。
包括一个帮助菜单
setHelpMenu(Menu)

菜单与其他组件有一个重要的不同：你不能将菜单添加到一般的容器中，而且不能使用布局管理器对它们进行布局。你只能将菜单加到一个菜单容器中。然而，你可以将一个 Jmenuswing 组件加到一个 Jcontainer 中。你可以通过使用 setMenuBar()方法将菜单放到一个框架中，从而启动一个菜单“树”。从那个时刻之后，你可以将菜单加到菜单条中，并将菜单或菜单项加到菜单中。

弹出式菜单是一个例外，因为它们可以以浮动窗口形式出现，因此不需要布局。

## 10.4.1 帮助菜单

菜单条的一个特性是你可以将一个菜单指定为帮助菜单。这可以用 setHelpMenu(Menu)来做到。要作为帮助菜单的菜单必须加入到菜单条中；然后它就会以和本地平台的帮助菜单同样的方式被处理。对于 X/Motif 类型的系统，这涉及将菜单条放置在菜单条的最右边。

## 10.4.2 菜单条(MenuBar)

一个菜单条组件是一个水平菜单。它只能加入到一个框架中，并成为所有菜单树的根。在一个时刻，一个框架可以显示一个菜单条。然而，你可以根据程序的状态修改菜单条，这样在不同的时刻就可以显示不同的菜单。例如：

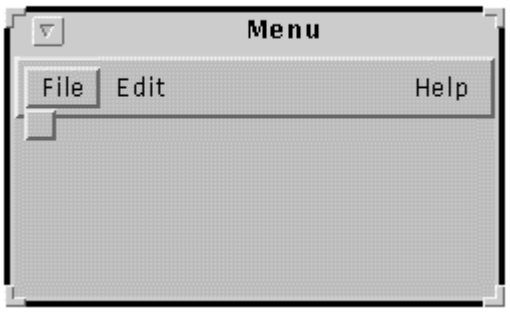
```
Frame f = new Frame("MenuBar");
MenuBar mb = new MenuBar();
f.setMenuBar(mb);
```


菜单条不支持监听者。作为普通菜单行为的一部分，在菜单条的区域中发生的预期事件会被自动处理。

## 10.4.3 菜单

菜单组件提供了一个基本的下拉式菜单。它可以加入到一个菜单条或者另一个菜单中。例如：

```
MenuBar mb = new MenuBar();
Menu m1 = new Menu("File");
Menu m2 = new Menu("Edit");
Menu m3 = new Menu("Help");
mb.add(m1);
mb.add(m2);
mb.setHelpMenu(m3);
f.setMenuBar(mb);
```




---

注－这里显示的菜单是空的，这正是 File 菜单的外观。

你可以将一个 ActionListener 加入到菜单对象，但这种做法是罕见的。正常情况下，菜单用来显示和控制菜单条，这将在后面讨论。

---

### 10.4.4 菜单项(MenuItem)

菜单项组件是菜单树的文本“叶”结点。它们通常被加入到菜单中，以构成一个完整的菜单。例如：

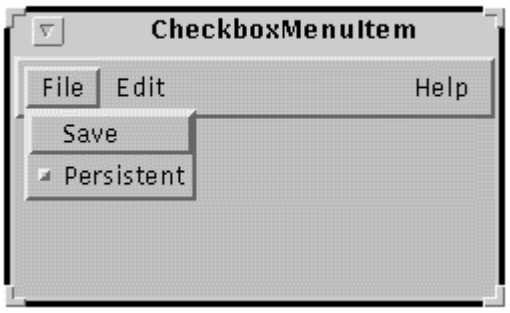
```
Menu m1 = new Menu("File");
MenuItem mi1 = new MenuItem("New");
MenuItem mi2 = new MenuItem("Load");
MenuItem mi3 = newMenuItem ("Save");
MenuItem mi4 = new MenuItem("Quit");
mi1.addActionListener(this);
mi2.addActionListener(this);
mi3.addActionListener(this);
m1.add(mi1);
m1.add(mi2);
m1.addSeparator();
m1.add(mi3);
```

通常，将一个 ActionListener 加入到一个菜单项对象中，以提供菜单的行为。

### 10.4.5 复选菜单项(CheckboxMenuItem)

复选菜单项是一个可复选的菜单项，所以你可以在菜单上有选项(“开”或“关”)。例如：

```
Menu m1 = new Menu("File");
MenuItem mi1 = new MenuItem("Save");
CheckboxMenuItem mi2 =
new CheckboxMenuItem("Persistent");
mi1.addItemListener(this);
mi2.addItemListener(this);
m1.add(mi1);
m1.add(mi2);
```




应当用 ItemListener 接口来监视复选菜单。因此当复选框状态发生改变时，就会调用 itemStateChanged() 方法。

### 10.4.6 弹出式菜单(PopupMenu)

弹出式菜单提供了一种独立的菜单，它可以在任何组件上显示。你可以将菜单条目和菜单加入到弹出式菜单中去。

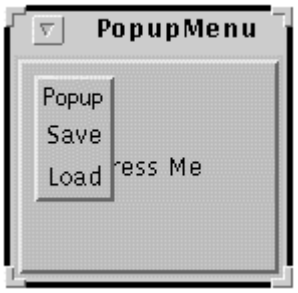
例如：

```
Frame f = new Frame("PopupMenu");
Button b = new Button("Press Me");
PopupMenu p = new PopupMenu("Popup");
MenuItem s = new MenuItem("Save");
MenuItem l = new MenuItem("Load");
b.addActionListener(this);
f.add(b,Border.Layout.CENTER);
p.add(s);
p.add(l);
f.add(p);
```

为了显示弹出式菜单，你必须调用显示方法。显示需要一个组件的引用，作为 x 和 y 坐标轴的起点。通常，你要为此使用组件的触发器。在上面这个范例中，触发器是 Button b。

### 弹出式菜单（续）

```
public void actionPerformed(ActionEvent ev) {
   p.show(b, 10, 10); // display popup
  // at (10,10) relative to b
}
```




---

注－弹出式菜单必须加入到一个“父”组件中。这与将组件加入到容器中是不同的。在上面这个范例中，弹出式菜单被加入到周围的框架中。

---

## 第五节 控制外观

**控制外观**

**颜色**

- setForeground()
- getForeground()

**字体**

- setFont()方法可以用来控制所显示的文本的颜色。
- Dialog, Helvetica, Palatino Roman 和 Courier 都是有效的字体名称

你可以控制在 AWT 组件中所显示的文本的前景背景颜色、背景颜色和字体。

### 10.5.1 颜色

有两个方法用来设置组件的颜色：

- setForeground()
- getForeground()

这两个方法都带有一个参数，参数是 java.awt.Color 类的实例。你可以使用常数颜色，如 Color.red,Color.blue 等。所有预定义的颜色列在 Color 类的文档中。

此外，你可以创建一个特定的颜色，例如：

```
int r = 255, g = 255, b = 0;
Color c = new Color(r, g, b);
```

上述构造函数根据指定的红色、绿色和蓝色的亮度(它们的范围都是 0~255)来创建一个颜色。

### 10.5.2 字体

在组件中显示时所用的字体可以用 setFont()方法来设置。这个方法的参数应当是 java.awt.Font 类的实例。

没有为字体定义常数，但你可以根据字体的名称，风格，磅值来创建一个字体。

```
Font f = new Font("TimesRoman", Font.PLAIN, 14);
```

有效的字体名称包括：

- Dialog
- Helvetica
- TimesRoman
- Courier

可以通过调用 Toolkit 对象的 getFontlist()方法来得到完整的列表。GetToolkit()方法是用来在显示 toolkit 后获得 toolkit 的。还有另外一种方法，你可以使用缺省的 toolkit，它可以通过调用 Toolkit.getDefaultToolkit()来获得。

字体风格常数实际上是 int 值，即：

- Font.BOLD
- Font.ITALIC
- Font.PLAIN
- Font.BOLD+ Font.ITALIC

磅值必须使用 int 值来指定。

## 第六节 打 印

**打印**

- 为了允许使用本地打印机的约定，必须使用下面范例中的方法：

```
Frame f = new Frame("Print test");
1.Toolkit t = f.getToolkit();
2.PrintJob job = t.getPrintJob(f, "MyPrintJob", null);
3.Graphics g = job.getGraphics();
```

- 为了为每一页获得一个新的图形，使用：

```
f.printComponents(g);
```

在 JDK1.2 中的打印处理方式和屏幕显示是几乎平行的。会获取一种特殊的 java.awt.Graphics 对象，这样任何送往图形的绘图指令实际上会最终被送往打印机。

JDK1.2 打印系统允许使用本地打印机的控制约定。选择一个打印操作时，用户会看到一个打印机选择对话框。然后用户可以设置各种选项，如：纸张大小、打印质量和使用哪个打印机。例如：

```
Frame f = new Frame("Print test");
Toolkit t = f.getToolkit();
PrintJob job = t.getPrintJob(f, "MyPrintJob", null);
Graphics g = job.getGraphics();
```

这些代码创建了一个 Graphics 对象，它将连接到用户选择的打印机。

你可以使用任何 Graphics 类绘图方法来使用打印机。另一种方法，如下所示，你可以让一个组件在图形上绘制自身。

```
f.printComponents(g);
```

print()方法以这种方式让组件绘制自身，但它只和所要求的组件相关联。如果是容器，你可以使用 printcomponents()方法使容器和它所包含的全部组件绘制在打印机上。

```
g.dispose();
job.end();
```

创建输出页之后，使用 dispose()方法将页面提交给打印机作业对象。

完成作业以后，调用打印作业对象的 end()方法。这表明打印作业已经完成，并允许打印假脱机系统运行作业，以及释放打印机以供其他作业使用。

# 练　习

练习目标－在这个练习中，你将创建一个使用许多组件的复杂应用程序。

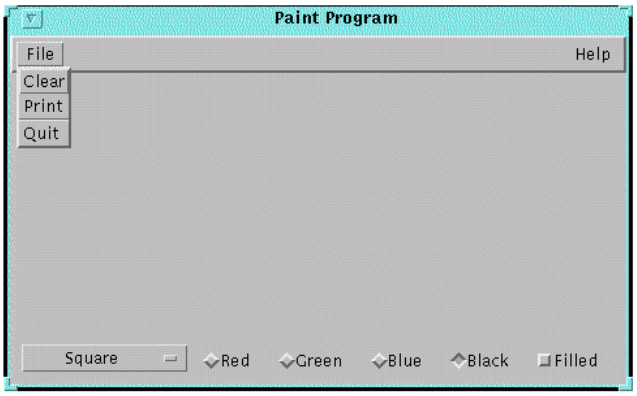
## 一、准备

为了很好地完成这个练习，你必须理解 AWT 的目的、它的事件处理器以及它的图形特性。

## 二、任务

水平 1： 创建绘图程序布局

1. 用 java.awt 包，创建一个 Java 应用程序——PaintGUI.java，它将添加显示在下图中的组件。




2. 使用帮助菜单的对话框。

水平 3：创建绘图程序

1. 以上图中的布局为指导，创建一个简单的绘图程序。包含一个使用 GUI 的 PaintHandler 类；使用事件处理来完成这个目标。
2. 选取打印菜单选项时，使用 java.awt.PrintJob 来打印创建的图形。

## 三、练习小结

讨论 － 花几分钟时间讨论一下，在本实验练习过程中你都经历、提出和发现了什么。

- 经验 解释 总结 应用

四、检查一下你的进度

在进入下一个模块的学习之前，请确认你能够：

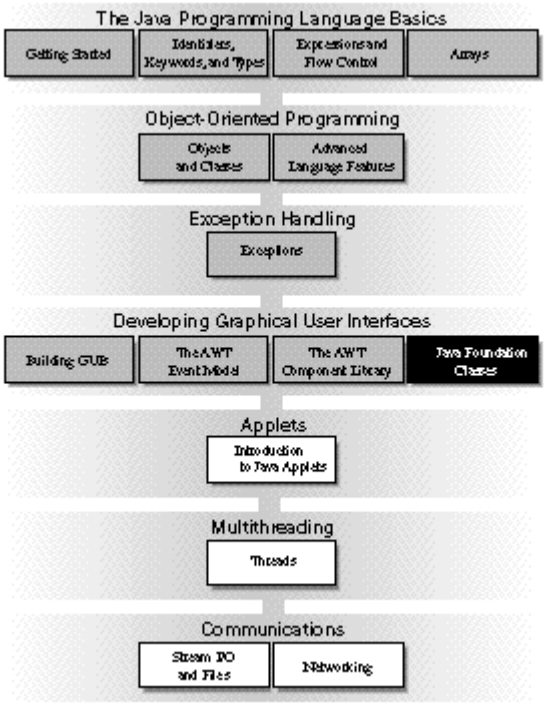
- 认识关键的 AWT 组件
- 使用 AWT 组件来创建真实程序的用户界面
- 控制 AWT 组件使用的颜色和字体
- 使用 Java 打印机制

五、思考

怎样才能使 AWT 更好地工作？

# 第11章 Java 基础类

JDK1.2 提供了 Java 基础类，其中的一部分就是 Swing。Swing 是构筑在 AWT 上层的一些组件的集合(为了保证平台独立性，它是用 100%的纯 Java 编写)。本模块介绍了 JFC 和 Swing 图形用户界面的实现。




## 第一节 相关问题

讨论－以下为与本模块内容有关的问题：

- AWT 本身是非常有用的，它是一个新的类集合的一部分。这个新的类集合称为 Java 基础类（JFC），它作为一个整体，将 GUI 提升到了一个新的水平层次。JFC 究竟是什么，特别地，什么是 Swing？什么事 Swing 可以做但 AWT 不能？

# 第二节 目 标

在完成了本模块的学习后，你应当能够：

- 认识 Java 基础类的关键特性
- 描述 com.sun.java.swing 包的关键特性
- 认识 Swing 组件
- 定义容器和组件，并解释如何联合使用它们来构造一个 Swing GUI
- 编写，编译并运行一个基本的 Swing 应用程序
- 高效地使用诸如 Jframe 和 Japplet 等顶层容器

**参考文献**

以下参考文献可提供有关本模块论题的其他细节内容：

- *The Java Tutorial*，这是 Sun Microsystems 的一本在线教材，可以从 http://java.sun.com/docs/books/tutorial 得到。

# 第三节 介 绍

**介绍**

- Java 基础类包含 5 个 API
  - AWT
  - Java2D
  - Accessibility
  - Drag & Drop
  - Swing

Java 基础类是关于 GUI 组件和服务的完整集合，它大大简化了健壮 Java 应用程序的开发和实现。

JFC，作为 JDK1.2 的一个有机部分，主要包含 5 个 API：AWT，Java2D，Accessibility，Drag & Drop，Swing。它提供了帮助开发人员设计复杂应用程序的一整套应用程序开发包。

正如前面那些模块中所讨论的那样，AWT 组件为各类 Java 应用程序提供了多种 GUI 工具。

Java2D 是一图形 API，它为 Java 应用程序提供了一套高级的有关二维（2D）图形图像处理的类。Java2D API 扩展了 java.awt 和 java.awt. image 类，并提供了丰富的绘图风格，定义复杂图形的机制和精心调节绘制过程的方法和类。这些 API 使得独立于平台的图形应用程序的开发更加简便。

Accessibility API 提供了一套高级工具，用以辅助开发使用非传统输入和输出的应用程序。它提供了一个辅助的技术接口，如：屏幕阅读器，屏幕放大器，听觉文本阅读器（语音处理）等等。

Drag & Drop 技术提供了 Java 和本地应用程序之间的互操作性，用来在 Java 应用程序和不支持 Java 技术的应用程序之间交换数据。

JFC 模块的重点在 Swing。Swing 用来进行基于窗口的应用程序开发，它提供了一套丰富的组件和工作框架，以指定 GUI 如何独立于平台地展现其视觉效果。

### 11.3.1 Swing 介绍

Swing 介绍

- 可插的外观和感觉
  - 应用程序看上去是与平台有关的
  - 有客户化的 Swing 组件
- Swing 的体系结构
  - 它是围绕着实现 AWT 各个部分的 API 构筑的
  - 大多数组件不象 AWT 那样使用与平台相关的实现

Swing 提供了一整套 GUI 组件，为了保证可移植性，它是完全用 Java 语言编写的。

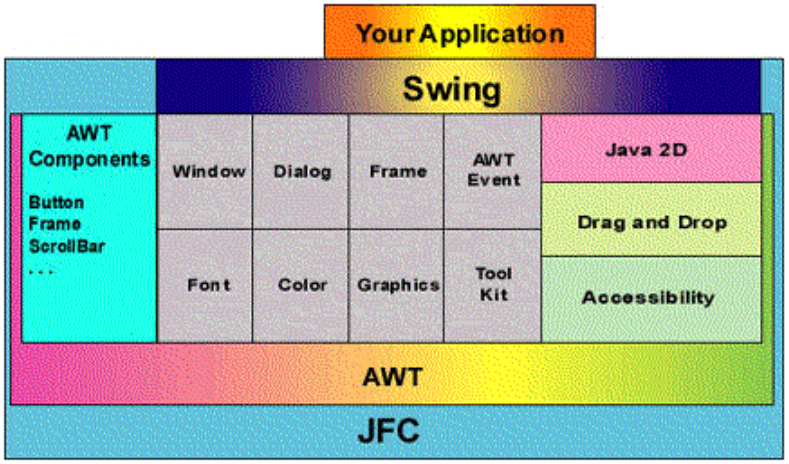
**可插的外观和感觉**

可插的外观和感觉使得开发人员可以构建这样的应用程序：它们可以在任何平台上执行，而且看上去就象是专门为那个特定的平台而开发的。一个在 Windows 环境中执行的程序，似乎是专为这个环境而开发的；而同样的程序在 Unix 平台上执行，它的行为又似乎是专为 Unix 环境开发的。

开发人员可以创建自己的客户化 Swing 组件，带有他们想设计出的任何外观和感觉。这增加了用于跨平台应用程序和 Applet 的可靠性和一致性。一个完整应用程序的 GUI 可以在运行时刻从一种外观和感觉切换到另一种。

**Swing 的体系结构**

与 AWT 比较，Swing 提供了更完整的组件，引入了许多新的特性和能力。Swing API 是围绕着实现 AWT 各个部分的 API 构筑的。这保证了所有早期的 AWT 组件仍然可以使用。AWT 采用了与特定平台相关的实现，而绝大多数 Swing 组件却不是这样做的，因此 Swing 的外观和感觉是可客户化和可插的。

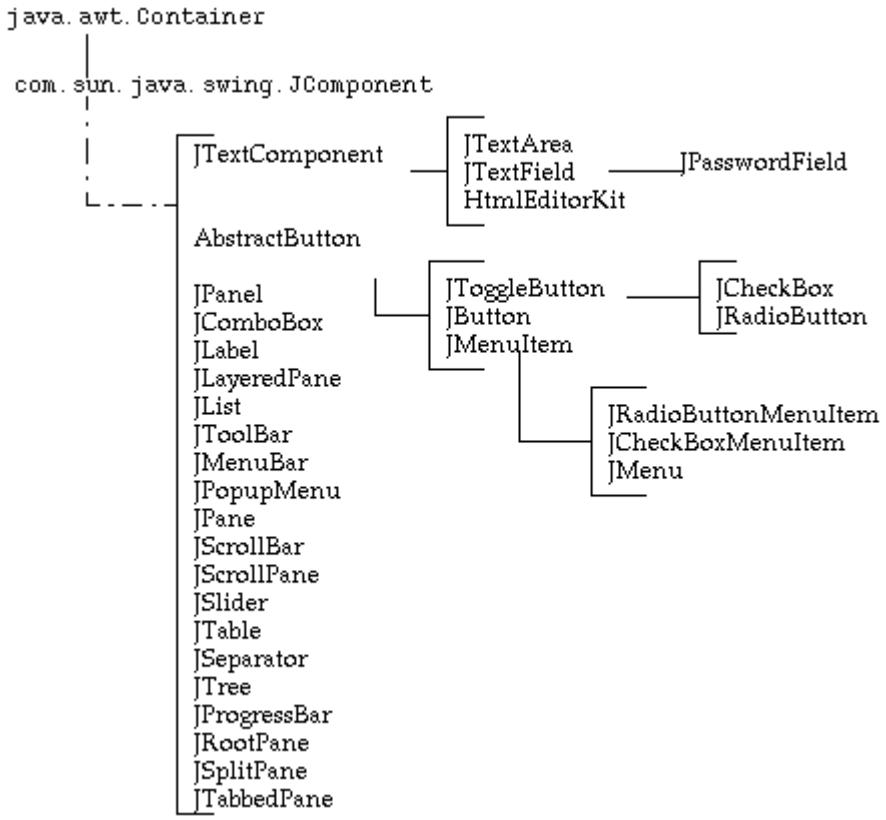



上图显示了 JFC 各个部分之间的相互关系。Java2D，Accessibility，Drag & Drop，和 Accessibility API 是 AWT 和 JFC 的一部分，但它们不属于 Swing。这是因为，这些组件使用了一些本地代码，而 Swing 却不是这样的。

Swing 是围绕着一个称为 JComponent 的新组件构建的，而 JComponent 则由 AWT 的容器类扩展而来。

**Swing 的层次结构**

下图说明了 Swing 组件的层次结构：




Swing GUI **使用两种类型的类，即** GUI **类和非** GUI **支持类。**GUI **类是可视的，它从** JComponent **继承而来，因此称为“** J **”类。非** GUI **类为** GUI **类提供服务，并执行关键功能；因此它们不产生任何可视的输出。**

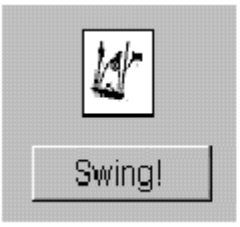
---

**注－Swing 的事件处理类是非 GUI 类的一例。**

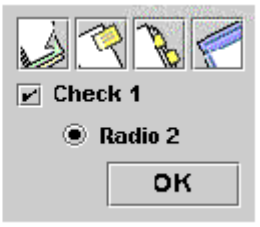
---

**Swing 组件**

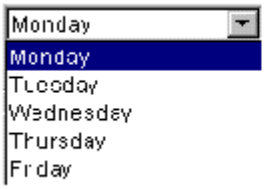
Swing **组件主要为文本处理、按钮、标签、列表、**pane**、组合框、滚动条、滚动** pane**、菜单、表格和树提供了组件。其中一些组件如下所示：**



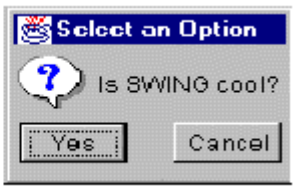

JApplet




JButton

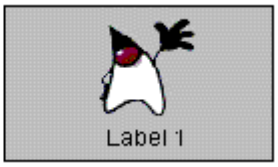



JComboBox




JOptionPane

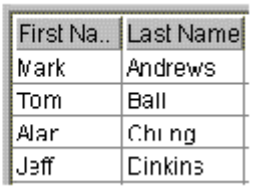

JList




JLabel

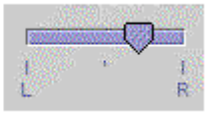
**Swing 组件（续）**


JScrollPane




JTable

JScrollBar



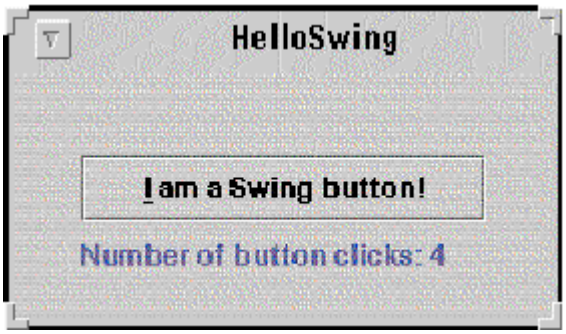

JSlider


JTooltip


JTree

# 第四节　基本的 Swing 应用程序

HelloSwing **应用程序的输出产生下图所示的窗口：**




**每次用户点击按钮时，标签就会更新。**

### 11.5.1　HelloSwing

```
1.import  java.awt.*;
2.import  java.awt.event.*;
```

```
3.import com.sun.java.swing.*;
4.import java.awt.accessibility.*;
5.
6.public class HelloSwing implements ActionListener {
7.private JFrame jFrame;
8.private JLabel jLabel;
9.private JPanel jPanel;
10.private JButton jButton;
11.private AccessibleContext accContext;
12.
13.private String labelPrefix =
14."Number of button clicks: ";
15.private int numClicks = 0;
16.
17.public void go() {
18.
19.// Here is how you can set up a particular
20.// lookAndFeel. Not necessary for default.
21.//
22.// try {
23.// UIManager.setLookAndFeel(
24.// UIManager.getLookAndFeel());
25.// } catch (UnsupportedLookAndFeelException e) {
26.// System.err.println("Couldn't use the " +
27.// "default look and feel " + e);
28.// }
29.
30.jFrame = new JFrame("HelloSwing");
31.jLabel = new JLabel(labelPrefix + "0");
32.
33.jButton = new JButton("I am a Swing button!");
34.
35.// Create a shortcut: make ALT-A be equivalent
36.// to pressing mouse over button.
37.jButton.setMnemonic('i');
38.
39.jButton.addActionListener(this);
40.
```

**HelloSwing（续）**

```
1.// Add support for accessibility.
2.accContext = jButton.getAccessibleContext();
3.accContext.setAccessibleDescription(
4."Pressing this button increments " +
5."the number of button clicks");
6.
7.// Set up pane.
8.// Give it a border around the edges.
9.jPanel = new JPanel();
```

```
10.jPanel.setBorder(
11.BorderFactory.createEmptyBorder(
12.30,30,10,30));
 13.
 14.// Arrange for compts to be in a single column.
15.jPanel.setLayout(new  GridLayout(0, 1));
 16.
17.// Put compts in pane, not in JFrame directly.
 18.jPanel.add(jButton);
 19.jPanel.add(jLabel);
 20.jFrame.setContentPane(jPanel);
 21.
22.// Set up a WindowListener inner class to handle
23.// window's quit button.
 24.WindowListener wl = new WindowAdapter() {
25.public void windowClosing(WindowEvent e) {
 26.System.exit(0);
 27.}
 28.};
 29.jFrame.addWindowListener(wl);
 30.
 31.jFrame.pack();
 32.jFrame.setVisible(true);
 33.}
 34.
```

**HelloSwing（续）**

```
 1.// Button handling.
 2.public void actionPerformed(ActionEvent e) {
  3.numClicks++;
 4.jLabel.setText(labelPrefix + numClicks);
  5.}
  6.
 7.public static void main(String[] args) {
  8.
 9.HelloSwing helloSwing = new HelloSwing();
10.helloSwing.go();
11.}
 12.}
```

### 11.4.2 导入 Swing 包

- **导入** Swing **包**
- **选择外观和感觉**
  - getLookAndFeel()
- **设置窗口容器**
  - JFrame **与** Frame **相似**
  - **你不能直接将组件加入到** JFrame **中**
  - **一个** content pane **包含了除菜单条外所有** Frame **的可视组件**

语句行 import com.sun.java.swing.*装入整个 Swing 包，它包括了标准 Swing 组件和功能。选择外观和感觉

Hello Swing 的第 22－28 行给定了应用程序外观和感觉的格式。getLookAndFeel()方法返回在 Windows 环境中的外观和感觉。在运行 Solaris 操作系统的机器上，这个方法则返回一个公共桌面环境（CDE）/Motif 的外观和感觉。因为都是缺省值，所以对本例来说，这些行都不是必需的。

### 11.4.3 建立窗口

Swing 程序用 JFrame 对象实现了它们的窗口。JFrame 类是 AWT Frame 类的一个子类。它还加入了一些 Swing 所独有的特性。Hello Swing 中，处理 JFrame 的代码如下：

```
public HelloSwing() {
JFrame jFrame;
JPanel jPanel;
        .....
jFrame = new JFrame("HelloSwing");
        jPanel = new JPanel();
        .......
        jFrame.setContentPane(jPanel);
```

这段代码与使用 Frame 的代码十分相似。唯一的区别在于，你不能将组件加入到 JFrame 中。你可以或者将组件加入到 JFrame 的 content pane 中，或者提供一个新的 content pane。

一个 content pane 是一个包含除菜单条（如果有的话）外所有框架的可视组件的容器。要获得一个 JFrame 的 content pane，可使用 getContentPane()方法。要设置它的 content pane（如前面本例所示），则可使用 set ContentPane()方法。

### 11.4.4 建立 Swing 组件

Swing 应用程序基础

- 建立 Swing 组件
- Hello Swing.java 示例实例化了 4 个 Swing 组件，这 4 个组件是：JFrame，JButton，JLabel 和 JPanel
- 支持辅助技术
  - Hello Swing.java 示例代码支持辅助技术

```
accContext = jButton.getAccessibleContext();
        accContext.setAccessibleDescription(
          "Pressing this button increments " +
           " the number of button clicks.");
```

Hello Swing 程序显式地实例化了 4 个组件：JFrame，JButton，JLabel 和 JPanel。Hello Swing 用第 33－45 行中的代码来初始化 JButton。

第 33 行创建了按钮。第 37 行将 ACT－I 键组合设置为快捷键，用来模拟按钮的点击。第 39 行为点击注册了一个事件处理器。第 41－45 行描述了一个按钮，使得辅助技术可以提供有关按钮功能的信息。

第 49－59 行初始化了 JPanel。这些代码创建了 JPanel 对象，设置它的边框，并将它的

布局管理器设置为单列地放置 panel 的内容。最后，将一个按钮和一个标签加入到 Panel 中。Hello Swing 中的 Panel 使用了一个不可见的边框，用来在它周围放入额外的填充。

### 11.4.5 支持辅助技术

Hello Swing.java 中唯一支持辅助技术的代码是：

```
accContext = jButton.getAccessibleContext();
    accContext.setAc cessibleDescription(
    "Pressing this button increments " +
    " the number of button clicks.");
```

下列信息集也可由辅助技术使用：

```
jButton = new JButton("I'm a Swing button!");
jLabel = new JLabel(labelPrefix + "0"); jLabel.setText(labelPrefix + numClicks);
```

在 JFrame，JButton，JLabel 和其他所有组件中，都有内建的 Accessibility 支持。辅助技术可以很容易地获得文本，甚至与一组件某特定部分相关的文本。

## 第五节 构造一个 Swing GUI

**构造一个 Swing GUI**

- 顶层容器（JFrame，JApplet，JDialog，和 JWindow）
- 轻质组件（如 JButton，JPanel 和 JMenu）
- 将 Swing 组件加入到与顶层容器相关联的 content pane 中。

Swing 包定义了两种类型的组件：

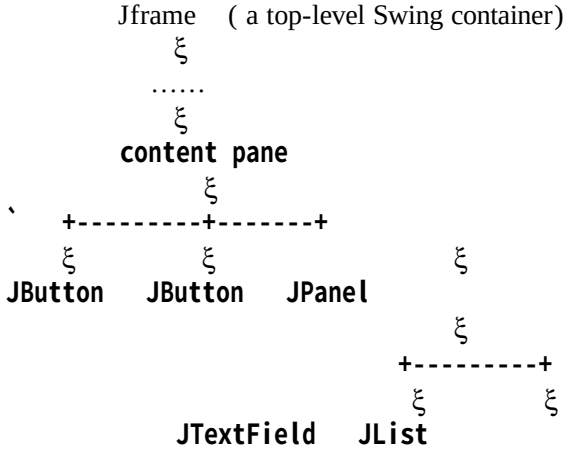
- 顶层容器（JFrame，JApplet，JWindow，和 JDialog）
- 轻质组件（其他的 J…，如 JButton，JPanel 和 JMenu）

顶层容器定义了可以包含轻质组件的框架。特别地，一个顶层 Swing 容器提供了一个区域，轻质组件可在这个区域中绘制自身。顶层容器是它们对应的重质 AWT 组件的 Swing 子类。这些 Swing 容器依靠它们的 AWT 超类的本地方法与硬件进行适当的交互。

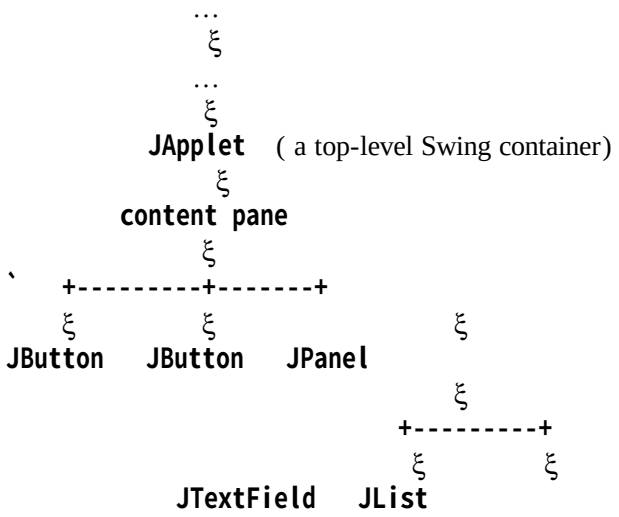
通常，每个 Swing 组件在其容器层次结构中都应当有一个位于组件上面的顶层 Swing 容器。例如，每个包含 Swing 组件的 Applet 都应作为 JApplet（而它自身又是 java.applet.Applet 的一个子类）的子类来实现。相似地，每个包含 Swing 组件的主窗口都应用 JFrame 来实现。典型地，如果你在使用 Swing 组件，你将只能使用 Swing 组件和 Swing 容器。

Swing 组件可以加入到一个与顶层容器关联的 content pane 中，但绝不能直接加入到顶层容器中。content pane 是一个轻质 Swing 组件，如 JPanel。

下面是一个典型 Swing 程序的 GUI 容器层次结构图，这个程序实现了一个包含 2 个按钮，一个文本域和一个列表：



**下面是关于同样的 GUI 的另一个容器层次结构，只是在这里，GUI 是在浏览器中运行的一个 Applet。**




**下面是构造如上图所示的 GUI 层次结构的代码：**

```
1.import com.sun.java.swing.*;
2.import java.awt.*;
3.
4.public class SwingGUI {
5.
6.JFrame topLevel;
7.JPanel jPanel;
8.JTextField jTextField;
9.JList jList;
10.
11.JButton b1;
12.JButton b2;
13.Container contentPane;
14.
15.Object listData[] = {
16.new String("First selection"),
17.new String("Second selection"),
18.new String("Third selection")
19.};
20.
21.public static void main (String args[]) {
22.SwingGUI swingGUI = new SwingGUI();
23.swingGUI.go();
24.}
25.
26.public void go() {
27.topLevel = new JFrame("Swing GUI");
28.
29.// Set up the JPanel, which contains the text field
30.// and list.
31.jPanel = new JPanel();
32.jTextField = new JTextField(20);
```

```
33.jList = new JList(listData);
 34.
35.contentPane = topLevel.getContentPane();
36.contentPane.setLayout(new BorderLayout());
 37.
38.b1 = new JButton("1");
39.b2 = new JButton("2");
 40.contentPane.add(b1, BorderLayout.NORTH);
41.contentPane.add(b2, BorderLayout.SOUTH);
 42.
43.jPanel.setLayout(new FlowLayout());
44.jPanel.add(jTextField);
45.jPanel.add(jList);
46.contentPane.add(jPanel, BorderLayout.CENTER);
 47.
48.topLevel.pack();
49.topLevel.setVisible(true);
50.}
 51.}
```

# 第六节 JComponent 类

JComponent 类

- Swing 组件是 JComponent 的子类
- 边框
- 双缓冲
- 提示框
- 键盘导航
- 应用程序范围的可插式外观和感觉

所有 Swing 都作为 JComponent 的子类来实现，而 JComponent 类又是从 Container 类继承而来。Swing 组件从 JComponent 继承了如下功能：

- 边框

  你可以用 setBorder()方法来指定在组件周围显示的边框。还可用一个 EmptyBorder 的实例来指定一个组件在其周围留有一定的额外空间。

- 双缓冲

  双缓冲可以改善一个频繁被改变的组件的外观。现在你不需要编写双缓冲代码——Swing 已为你提供了。缺省情况下，Swing 组件是双缓冲的。

- 提示框

  通过用 setToolTipText()方法来指定一个字符串，你可以提供给用户有关某个组件的帮助信息。当光标暂停在组件上时，所指定的字符串就会在组件附近的一个小窗口中显示出来。

- 键盘导航

  使用 registerKeyboardAction()方法，你可以让用户以键盘代替鼠标来操作 GUI。用户为启动一个动作所必须按下的修饰键与字符的组合，由一个 KeyStroke 对象来表示。

- 应用程序范围的可插式外观和感觉

  每个 Java 应用程序在运行时刻有一个 GUIManager 对象，它用于确定运行时刻

Swing 组件的外观和感觉。由于安全性的限制，你可以通过调用 UIManager.setLookAndFeel()方法选择所有 Swing 组件的外观和感觉。在你所看见的东西背后，每个 JComponent 对象都有一个对应的 ComponentGUI 对象，它用来执行所有关于该 JComponent 的绘制、事件处理、大小判定等任务。

# 练习：熟悉 Swing

**练习目标－在本实验中，你将编写、编译和执行两个在 GUI 中使用 Swing 组件的程序。**

## 一、准备

为了更好地完成这个练习，你必须理解 Swing 组件和 AWT 组件的关系。

## 二、任务

**水平 1：创建一个基本的 Swing 应用程序**

1. 使用文本编辑器，创建一个与前面所讨论的 HelloSwing 类似的应用程序。
2. 将一个图标与按钮相关联。（提示－你可能需要使用 ImageIcon 类。）
3. 将一个提示框与按钮相关联，这样当鼠标移动到按钮之上时，会显示一个“ JFC Button ”的提示框。

**水平 2：用 Swing 组件创建一个文本编辑器**

1. 创建一个初始的 JFrame，它包含一个 JToolBar，TextArea 和 JLabel。
2. 将一个 JMenuBar 与 JFrame 关联起来。
3. 创建 JMenuBar 上的第一个菜单。创建一个标记为 JMenu，其 JMenuItems 包括 New，Open，Save 和 Close。
4. 为每个条目增加一个加速键。使用标签的第一个字母。
5. 为每个 JMenuItem 创建一个匿名的 ActionListener，用来处理事件并调用与每个事件对应的方法。
6. 将带有 About JMenuItem 的 HelpJMenu 加入到 JMenuBar。分别为 H 和 A 增加快捷键。
7. 在与 About JMenuItem 相关联的事件处理器中创建一个模式对话框。
8. 在工具条上创建 4 个 JButton，标为 New，Open，Save 和 About。
9. 为工具条上的每个按钮增加一个带有适当消息的提示框。此外，创建一个匿名 ActionListener 来处理适当的事件。
10. 保存并编译程序。

## 三、练习小结

讨论 － 花几分钟时间讨论一下，在本实验练习过程中你都经历、提出和发现了什么。

- 经验
- 解释
- 总结
- 应用

## 四、检查你的进度

在进入下一个模块的学习之前，请确认你能够：

- 认识 Java 基础类的关键特性
- 描述 com.sun.java.swing 包的关键特性
- 认识 Swing 组件

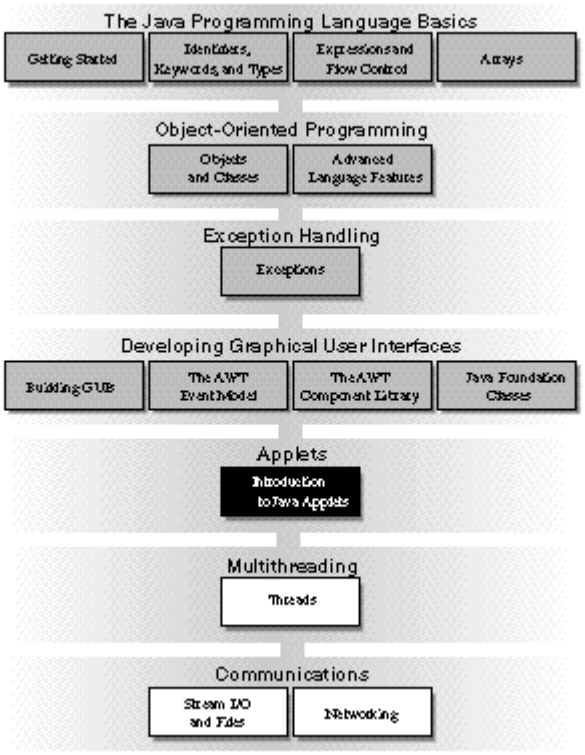
- 定义容器和组件，并解释如何联合使用它们来构造一个 Swing GUI
- 编写，编译并运行一个基本的 Swing 应用程序
- 高效地使用诸如 JFrame 和 JApplet 等顶层容器

**五、思考题**

**你现在已经知道了如何编写 GUI 应用程序。假设你想在一个 Web 浏览器中运行一个 GUI 应用程序，如何做到这点？**

# 第12章 Jvav 小程序介绍

**本模块讨论了 JDK 对 Applet 的支持，以及 Applet 在编程方式、操作上下文和如何开始等方面与应用程序的区别。**




# 第一节 相关问题

**讨论 － 以下为与本模块内容有关的问题：**

- Applet **有那些优点？**

# 第二节 目 标

**在完成了本模块的学习后，你应当能够：**

- **区分独立应用程序和** Applet
- **编写一个** HTML **标记来调用** Java Applet
- **描述** Applet **和** AWT **的类层次**
- **创建** HelloWorld.Java Applet
- **列出** Applet **的主要方法**
- **描述和使用** AWT **的绘图模型**
- **使用** Applet **方法从** URL **读取图像和文件**
- **使用**<param>**标记配置** Applet

# 第三节 什么是 Applet？

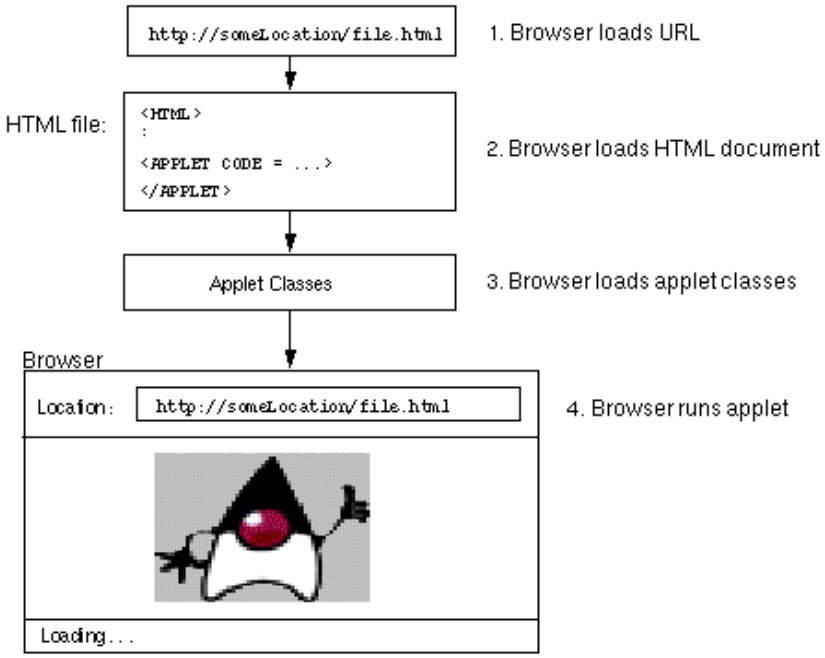
**什么是** Applet**？**

- **能嵌入到一个** HTML **页面中且可通过** Web **浏览器下载**

Applet 是能够嵌入到一个 HTML 页面中，且可通过 Web 浏览器下载和执行的一种 Java 类。它是 Java 技术容器(container)的一种特定类型，其执行方式不同于应用程序。一个应用程序是从它的 main()方法被调用开始的，而一个 Applet 的生命周期在一定程度上则要复杂得多。本模块分析了 Applet 如何运行，如何被装载到浏览器中，以及它是如何编写的。

### 12.3.1 装入 Applet

由于 Applet 在 Web 浏览器环境中运行，所以它并不直接由键入的一个命令启动。你必须要创建一个 HTML 文件来告诉浏览器需装载什么以及如何运行它。




1. 浏览器装入 URL
2. 浏览器装入 HTML 文档
3. 浏览器装入 Applet 类
4. 浏览器运行 Applet

### 12.3.2 Applet 的安全限制

Applet 的安全限制

- 多数浏览器禁止以下操作：
  - 运行时执行另一程序
  - 任何文件的输入/输出
  - 调用任何本地方法
  - 尝试打开除提供 Applet 的主机之外的任何系统的 Socket

由于通过网络装载，Applet 的代码具有一种内在的危险性。如果有人编写了一个恶意的类来读取你的密码文件，并把它通过 Internet 传送，会产生怎样的后果呢？

所能够控制的安全程度是在浏览器层次上实现的。大多数浏览器（包括 Netscape Nevigator）缺省地禁止以下操作：

- 运行时执行另一程序
- 任何文件的输入/输出
- 调用任何本地方法
- 尝试打开除提供 Applet 的主机之外的任何系统的 Socket

这些限制的关键在于，通过限制 Applet 对系统文件的存取来阻止它侵犯一个远程系统的隐私或破坏该系统。禁止执行另一程序和不允许调用本地方法限制了 Applet 启动未经 JVM 检查的代码。对 Socket 的限制则禁止了与另一个可能有危害性的程序的通信。

JDK1.2 提供了一种方式，它指定了一个特殊的“保护域”或一个特殊 Applet 运行的安全性环境。远程系统检查原始的 URL 以及它下载的 Applet 的签名，和一个含有从特殊的 Applet 到特殊保护域的映射入口的本地文件进行比较。因此，来自特别位置的特殊 Applet 具有一些运行特权。

# 第四节 编写一个 Applet

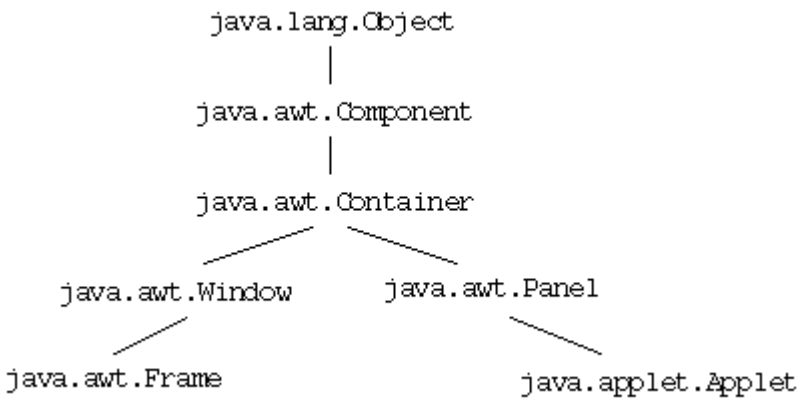
要编写一个 Applet，必须首先用以下方式创建一个类：

```
import java.applet.*;
public class HelloWorld extends Applet {
```

Applet 的类必须为 public，且它的名称必须与它所在的文件名匹配；在这里，就是 HelloWorld.java。而且，该类必须为 java.applet.Applet 的子类。

Applet 类的层次

Java.applet.Applet 类实际上是 java.awt.Panel 的子类。Applet 和 AWT 类的层次如下：




这种层次关系显示，一个 Applet 可直接用作一个 AWT 布局的起始点。因为 Applet 为一 Panel，所以它有一个缺省的流(flow)布局管理器。Component，Container 和 Panel 类的方法被 Applet 类继承了下来。

### 12.4.1 主要的 Applet 方法

**主要的 Applet 方法**

- init()
- start()
- stop()
- destroy()

在一个应用程序中，程序由 main()方法处进入，而在一个 Applet 中却不是这样。在构造函数完成了它的任务后，浏览器调用 init()对 Applet 进行基本的初始化操作。init()结束后，浏览器调用另一个称为 start()的方法。本模块稍后将对 start()做更细致的剖析；start()通常在 Applet 成为可见时被调用。

方法 init()和 start()都是在 Applet 成为“活动的”之前运行完成的，正因为这样，它们都不能用来编写 Applet 中继续下去的动作。实际上，与一个简单应用程序中的方法 main()不同的是，没有什么方法的执行是贯穿于 Applet 的整个生命过程中的。你在后面将看到如何使用线程来实现这一特色。此外，你在编写 Applet 子类时可用的方法还有：stop()，destroy()和 paint()。

### 12.4.2 Applet 显示

Applet 显示
- Applet 在本质上是图形方式的
- 方法 paint()由浏览器环境调用

Applet 本质上是图形方式的，所以尽管你可以提出 System.out.println()的调用请求，通常也不能这样做，而是应该在图形环境中创建你的显示。

你可以通过创建一个 paint()方法在 Applet 的 panel 上绘图。只要 Applet 的显示需要刷新，paint()方法就会被浏览器环境调用。例如，当浏览器窗口被最小化或被要求以图标方式显示时，这种调用就会发生。

你应该编写自己的 paint()方法，以使它在任何时候被调用时都能正常地工作。对它的调用是异步产生的，且由环境而不是程序来驱动。

### 12.4.3 paint()方法和图形对象

paint()方法带有一个参数，它是 java.awt.Graphics 类的一个实例。这个参数总是建立该 Applet 的 panel 的图形上下文。你能用这个上下文在你的 Applet 中绘图或写入文本。下面是使用 paint()方法写出文字的一例。

```
1.import java.awt.*;
2.import java.applet.*;
3.
4.public class HelloWorld extends Applet {
5.
6.public void paint(Graphics g){
7.g.drawString("Hello World!", 25, 25);
8.}
9.}
```

---

注－drawString 方法的数字型参数为文本起始处的 x 和 y 的象素坐标。(0，0) 表示左上角。这些坐标是针对字体的基线来讲的，所以在 y 坐标为 0 处写的结果是：文字的大部分在显示器顶部的上方，只有象字母 y 尾部那样的下面部分是可见的。

---

# 第五节 Applet 的方法和 Applet 的生命周期

Applet 的方法和 Applet 的生命周期

- init()
  - 在 Applet 创建时被调用
  - 可用于初始化数据值
- start()
  - 当 Applet 成为可见时运行
- stop()
  - 当 Applet 成为不可见时运行

Applet 的生命周期比所讨论的要稍微复杂一些。与其生命周期相关的有三个主要方法：init()，start()和 stop()。

### 12.5.1 init()

本成员函数在 Applet 被创建并装入一个能支持 Java 技术的浏览器（如 appletviewer）时被调用。Applet 可用这个方法来初始化数据的值。本方法只在 Applet 首次装入时被调用，并且在调用 start()之前执行完成。

### 12.5.2 start()

init()方法一完成，start()就开始执行。它的执行使得 Applet 成为“活动”的。无论 Applet 何时成为可见的，它同样要执行一次，如：当浏览器在被图标化后又恢复时，或者当浏览器在链接到另一个 URL 后又返回含有这个 Applet 的页面时。这一方法的典型用法是启动动画和播放声音。

```
1.public void start() {
2.musicClip.play();
3.}
```

### 12.5.3 stop()

stop()方法是在 Applet 成为不可见时被调用的，这种情况一般在浏览器被图标化或链接到另一个 URL 时会出现。Applet 用该方法使动画停止。

```
1.public void stop() {
2.musicClip.stop();
3.}
```

start()和 stop()形成一对动作：典型地，start()激活 Applet 中的某一行为，而 stop()则可将它禁止。

# 第六节 AWT 绘图

AWT 绘图

- paint (Graphics g)
- repaint()
- update(Graphics g)

除了基本的生命周期外，Applet 还有与其显示有关的一些重要的方法。这些方法的声明和文档在 AWT 组件类中。使用 AWT 做显示处理时遵循正确的模型是非常重要的。

更新显示由一种被称为 AWT 线程的独立的线程来完成。这个线程可用来处理与显示更新相关的两种情况。

第一种情况是显露（exposure），它或在首次显示时，或在部分显示已被破坏而必须刷新时出现。显示的破坏可能发生在任何时刻，因此，你的程序必须能在任意时刻更新显示。

第二种情况是在程序重画带有新内容的画面时。这种重画可能会要求首先擦除原来的图像。

### 12.6.1 Paint(Graphics g)方法

显露处理自动地发生，且导致对 paint()方法的一次调用。一种 Graphics 类的被称为裁剪矩形的设备常用于对 paint()方法进行优化。除非必要，更新不会完全覆盖整个图形区域，而是严格限制在被破坏的范围内。

### 12.6.2 repaint()方法

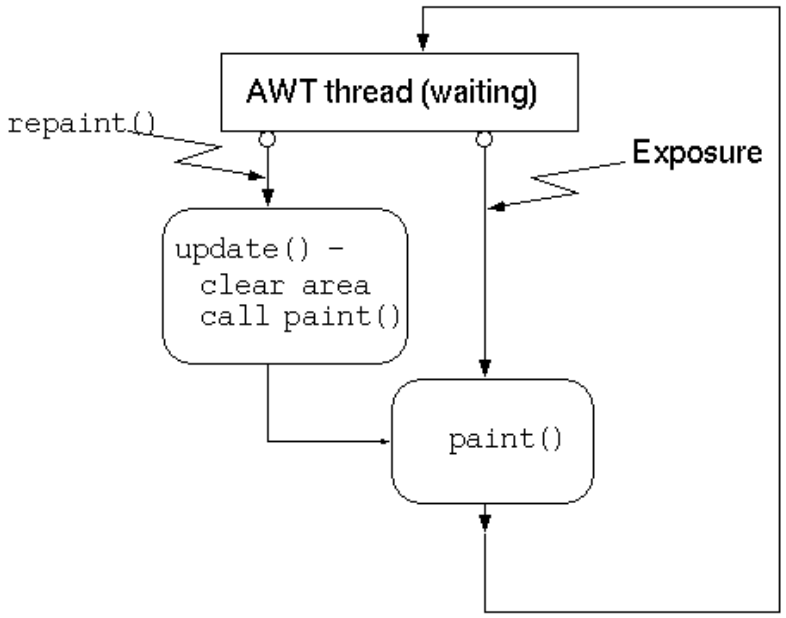
对 repaint()的调用可通知系统：你想改变显示，于是系统将调用 paint()。

### 12.6.3 update(Graphics g)方法

repaint()实际上产生了一个调用另一方法 update()的 AWT 线程。update 方法通常清除当前的显示并调用 paint()。update()方法可以被修改，如：为了减少闪烁可不清除显示而直接调用 paint()。

### 12.6.4 方法的交互

下面的框图描述了 paint()，update()和 repaint()方法间的内在关系。




### 12.6.5 Applet 的显示策略

Applet 的显示策略

- 维护一个显示模型
- 使 paint()提供仅仅基于这个模型的显示
- 更新这个模型并调用 repaint()来改变显示

Applet 模型要求你采取一种特定的策略来维护你的显示：

- 维护一个显示模型。这个模型是对为再次提供显示而所需做的事情的一个定义。关于如何去做的指令在 paint()方法中被具体化；这些指令所用的数据通常是全局成员变量。
- 使 paint()提供仅仅基于该模型的显示。这使得无论 paint()何时被调用，它都能以一致的方法再生该显示，并正确地处理显露问题。
- 使得程序对显示的改变，通过更新该模型而调用 repaint()方法来进行，以使 update()方法（最终是 paint()方法）被 AWT 线程调用。

---

注－一个单一 AWT 线程处理所有的绘图组件和输入事件的分发。应保持 paint()和 update()的简单性，以避免它们使 AWT 线程发生故障的可能性更大；在极端情况下，你将需要其他线程的帮助以达到这一目的。有关线程的编程是模块 14 的主题。

---

# 第七节 什么是 appletviewer？

**什么是 appletviewer？**

- 使你不需 Web 浏览器就能运行 Applet 的一个 Java 应用程序
- 它把 HTML 文件作为一个参数来装载

  appletViewer HelloWorld.html
- 它至少需要以下 HTML 代码：

```
<html>
  <applet code=HelloWorld.class width=100 height=100>
  </applet>
  </html>
```

Applet 通常运行于一个 Web 浏览器中，如 HotJava TM 或 Netscape Navigator，它们有支持 Java 软件程序运行的能力。为了简化和加速开发过程，JDK 应运而生，它附带有一个专为查看 Applet 而设计但不支持 HTML 页面查看的工具。这个工具就是 appletviewer。

appletviewer 是使你不必使用 Web 浏览器即可运行 Applet 的一个 Java 应用程序。它犹如一个“最小化的浏览器”。

appletviewer 读取命令行中 URL 所指定的 HTML 文件。这个文件必须包含装入及执行一个或多个 Applet 的指令。appletviewer 忽略了所有其他的 HTML 代码。它不能显示普通的 HTML 或嵌人在一个文本页中的 Applet。

### 12.7.1 用 appletviewer 启动 Applet

appletviewer 将一个框架样式的区域粘贴在屏幕上，然后实例化该 Applet 并将这个 Applet 实例贴在已有的框架中。

appletviewer 带有一个命令行参数形式的 URL，它指向一个含有 Applet 引用的 HTML 文件。这个 Applet 引用是一个指定了 appletviewer 要装载的代码的 HTML 标记。

```
<html>
<applet code=HelloWorld.class width=100 height=100>
</applet>
</html>
```

注意，这个标记的通用格式与任何其他的 HTML 相同，即，用<和>两个符号来分隔指令。上例中显示的所有部分都是必需的，你必须使用<applet . . .>和</applet>。<applet . . .>

部分指明了代码的入口，以及宽度和高度。

---

注－通常，你应该把 Applet 当作是固定大小的，并且使用<applet>标记中所指定的大小。

---

### 12.7.2 使用 appletviewer

**提要**

appletviewer 带有一个指向包含<applet>标记的 HTML 文件的 URL，这个 URL 被作为命令行参数。

appletviewer [-debug] URLs ...

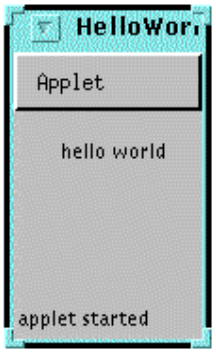
appletviewer 仅有的合法选项是 –debug，它使得 Applet 在 Java 调试器 jdb 中启动。若用带 –g 选项的方式编译你的 Java 代码，则可在调试器中看到源代码。

**范例**

以如下所示的 appletviewer 命令启动 appletviewer：

c:\jdk1.2\source> appletviewer HelloWorld.html

于是它创建并显示出如下的小窗口：




# 第八节 Applet 标记

### 12.8.1 句法

以下为 Applet 标记的完整句法：

```
<applet
[archive= archiveList]
code= appletFile. class
width= pixels height= pixels
[codebase= codebaseURL ]
[alt= alternateText ]
[nam e= appletInstanceName ]
[alig n= alignment ]
[vspace = pixels ] [hspace= pixels ]
>
[<param name= appletAttribute1 value= value >]
[<param name= appletAttribute2 value= value >]
. . .
[alternateHTML]
</applet>
```

其中

- archive = archiveList － 这一可选属性描述了一个或多个含有将被“预装”的类和其他资源的 archives。类的装载由带有给定 codebase 的 AppletClassLoader 的一个实例来完成。ArchiveList 中的 archives 以逗号（,）分隔。
- code = appletFile.class － 这是一个必需的属性，它给定了含有已编译好的 Applet 子类的文件名。也可用 package.appletFile.class 的格式来表示。

---

注－这个文件与你要装入的 HTML 文件的基 URL 有关，它不能含有路径名。要改变 Applet 的基 URL，可使用<codebase>。

---

- width = pixels height = pixels － 这些必需的属性给出了 Applet 显示区域的初始宽度和高度（以象素为单位），不包括 Applet 所产生的任何窗口或对话框。

### 12.8.2 描述

- codebase = codebaseURL － 这一可选属性指定了 Applet 的基 URL——包含有 Applet 代码的目录。如果这一属性未指定，则采用文档的 URL。
- alt = alternateText － 这一可选属性指定了当浏览器能读取 Applet 标记但不能执行 Java Applet 时要显示的文本。
- name = appletInstanceName － 这个可选属性为 Applet 实例指定有关名称，从而使得在同一页面上的 Applet 可找到彼此（以及互相通信）。
- align = alignment － 这个可选属性指定了 Applet 的对齐方式。它的可取值与基本的 HTML 中 IMG 标记的相应属性相同，为：left，right，top，texttop，middle，absmiddle，baseline，bottom 和 absbottom。
- vspace = pixels hspace = pixels － 这些可选属性指定了在 Applet 上下（vspace）及左右（hspace）的象素数目。其用法与 IMG 标记的 vspace 和 hspace 属性相同。
- <param name = appletAttribute1 value = value> － 这个标记提供了一种可带有由“外部”指定的数值的 Applet，它对一个 Java 应用程序的作用与命令行参数相同。Applet 用 getParameter()方法来存取它们的属性，该方法将在本模块稍后作更详细的讨论。
- 不支持 Java 程序执行的浏览器将显示被包括在<applet>和</applet>标记之间的任何常规的 HTML；而可支持 Java 技术的浏览器则忽略介于这两个标记之间的 HTML 代码。

# 第九节 其他的 Applet 工具

**其他的 Applet 工具**

- getDocumentBase() － 返回一个描述当前浏览器页面目录的 URL 对象
- getCodeBase() － 返回一个描述 Applet 类源目录的 URL 对象

在 Applet 中有若干其他特色。

所有的 Java 软件程序都具有访问网络的特色，这可使用模块 15 中所讲到的 java.net 包中的类来实现。此外，Applet 还有些其他的方法可允许它们取得有关自己启动时所在的浏览器环境的信息。

类 java.net.URL 描述了 URL，并可用于它们之间的连接。在 Applet 类中有两个方法决定了 URL 的重要的值：

- getDocumentBase()返回一个描述当前浏览器中带有 Applet 标记的 HTML 文件所属页面目录的 URL 对象
- getCodeBase()返回一个描述 Applet 类文件本身源目录的 URL 对象。它通常与 HTML 文件目录相同，但并不是一定要这样。

其他的 Applet 特色

用 URL 对象作为一个起始点，你可以将声音和图像取回到你的 Applet 中。

- getImage(URL base, String target)从被命名为 target 且位于由 base 所指定目录的文件中取回一幅图像。其返回值是类 Image 的一个实例。
- getAudioClip(URL base, String target) 从被命名为 target 且位于由 base 所指定目录的文件中取回一声音。其返回值是类 Audio Clip 的一个实例。

---

注－getImage(URL, String)和 getAudioClip(URL, String)方法中的 String target 能包括一个来自于 URL 的相对目录路径。但是请注意，在目录层次中向上的相对路径名，在某些系统上可能是不允许的。

---

## 第十节 一个简单的图像测试

下面的 Applet 获得了相对于 getDocumentBase 方法返回的目录路径为 graphics/joe.gif 的图像文件，并将它显示在 appletviewer 中：

```
1.// HelloWorld extended to draw an image
2.// Assumes existence of
3.//"graphics/SurferDuke.gif"
4.//
5.import java.awt.*;
6.import java.applet.Applet;
7.
8.public class HwImage extends Applet {
9.Image duke;
10.
11.public void init() {
12.duke = getImage(getDocumentBase(),
13."graphics/SurferDuke.gif");
14.}
15.
16.public void paint(Graphics g) {
17.g.drawImage(duke, 25, 25, this);
```

```
18.}
19.}
```

**drawImage()方法的参数是：**

- **将要被绘出的** Image **对象**
- **绘图的** x **轴坐标**
- **绘图的** y **轴坐标**
- **图像观察者。图像观察者是可以得知该图像的状态是否改变的一个接口（如：在装入过程中发生了什么）。**

**由** getImage()**装载的图像在调用首次提出后过一段时间将会改变，这是由于装载是在后台完成的。每次，图像的更多部分被装入，**paint()**方法被又一次调用。这种对** paint()**方法调用的发生是因为** Applet **将自己作为** drawImage()**的第四个参数传递给了自己，从而使自己被注册为一个观察者。**

# 第二章第十一节 Audio Clips

Java **编程语言也具有播放** Audio Clips **的方法。这些方法在** java.applet.AudioClip **类中。为了播放** Audio Clips**，你将需要为你的计算机装配适当的硬件。**

### 12.11.1 播放一段 Clip

**欣赏一段** audio clip **的最简单的方式是通过** Applet **的** play **方法：**

```
play(URL soundDirectory, String soundFile);
```

**或更简单的：**

```
play(URL soundURL);
```

**例如：**

```
play(getDocumentBase(), "bark.au");
```

**将播放存放在与** HTML **文件相同目录的** bark.au**。**

### 12.11.2 一个简单的 Audio 测试

**以下的** Applet **在** appletviewer **中打印出消息“** Audio Test **”，然后播放** audio **文件** sounds/cuckoo.au**：**

```
1.//
2.// HelloWorld extended to play an Audio sound
3.// Assumes existence of "sounds/cuckoo.au" file
4.//
5.
6.import java.awt.Graphics;
7.import java.applet.Applet;
8.
9.public class HwAudio extends Applet {
10.
11.public void paint(Graphics g) {
12.g.drawString("Audio Test", 25, 25);
13.play(getDocumentBase(),"sounds/cuckoo.au");
14.}
```

```
15.}
```

### 12.11.3 循环播放一段 Audio Clip

**循环播放一段** Audio Clip

- **装入一段** Audio Clip
- **播放一段** Audio Clip
- **停止一段** Audio Clip

**你可以用与装入图像相同的方式装入** audio clip。**在将它们装载之后进行播放。**

**装入一段 Audio Clip**

**为了装入一段** Audio Clip，**可使用来自** java.applet.Applet **类的** getAudioClip **方法：**

```
AudioClip sound;
sound = getAudioClip(getDocumentBase(), "bark.au");
```

**一旦一段** clip **被装载，可选择与之相关的三个方法之一：**play，loop，**或** stop。

**播放 Audio Clip**

**使用** java.applet.AudioClip **接口中的** play **方法将已装入的** audio clip **播放一遍：**

```
sound.play();
```

**为了启动** clip **的播放并使它不断循环(自动重放),可使用** java.applet.AudioClip **中的** loop **方法：**

```
sound. loop();
```

**停止** Audio Clip

**要停止一段正在播放的** clip，**可用** java.applet.AudioClip **中的** stop **方法：**

```
sound. stop();
```

### 12.11.4 一个简单的 Audio 循环测试

**下例中将一段装入的** audio clip **自动循环播放：**

```
1.//
2.// HelloWorld extended to loop an audio track
3.// Assumes existence of "sounds/cuckoo.au"
4.//
5.
6.import java.awt.Graphics;
7.import java.applet.*;
8.
9.public class HwLoop extends Applet {
10.AudioClip sound;
11.
12.public void init() {
13.sound = getAudioClip(getDocumentBase(),
14."sounds/cuckoo.au");
15.}
16.
17.public void paint(Graphics g) {
18.g.drawString("Audio Test", 25, 25);
19.}
20.
21.public void start() {
```

```
22.sound.loop();
23.}
 24.
25.public void stop() {
26.sound.stop();
 27.}
```

---

注－JDK1.2 支持一种新的声音引擎，这个引擎提供了对 MIDI 文件和全部 .wav，aiff 及 .au 文件的回放功能。它给出了一个新方法 newAudioClip(URL url)，这个方法从给定的 URL 获取一段 audio clip，参数 URL 指向该 audio clip。第 13 行中的 getAudioClip 方法可用这个方法替换。NewAudioClip 方法不需要第二个参数 String，只有 URL 参数要求被传递。

---

# 第十二节　鼠标输入

Java 编程语言所支持的最有用的特色之一是直接的交互动作。Java Applet，同应用程序一样，能注意到鼠标，并对鼠标事件作出反应。在这里，我们将对鼠标的支持作一次快速的回顾，以帮助理解下面的例子。

回想一下模块 9 中，JDK1.2 事件模型对每一类交互动作都支持一种事件类型。鼠标事件由实现 MouseListener 接口的类来接收，它们可接收的事件为：

- mouseClicked －鼠标已被点击（鼠标按钮被按下然后被释放，作为一个动作）
- mouseEntered －鼠标光标进入一个组件
- mouseExited －鼠标光标离开一个组件
- mousePressed －鼠标按钮被按下
- mouseReleased －鼠标按钮被释放

## 12.12.1　一个简单的 Mouse 测试

下面的程序显示了鼠标在 Applet 中点击的位置：

```
 1.//
2.// HelloWorld extended to watch for mouse input
3.// "Hello World!" is reprinted at the location of
 4.// the mouse click.
 5.//
 6.
 7.import java.awt.*;
 8.import java.awt.event.*;
 9.import java.applet.Applet;
10.
11.public class HwMouse extends Applet
12.implements MouseListener {
 13.
14.int mouseX=25;
```

```
15.int mouseY=25;
16.
17.// Register this applet instance to catch // MouseListener events
18.public void init () {
19.addMouseListener (this);
20.}
21.
22.public void paint(Graphics g) {
23.g.drawString("Hello World!", mouseX, mouseY);
24.}
25.
26.// Process the mousePressed MouseListener event
27.public void mousePressed(MouseEvent evt){
28.mouseX = evt.getX();
29.mouseY = evt.getY();
30.repaint();
31.}
32.
33.// We are not using the other mouse events
34.public void mouseClicked (MouseEvent e) {}
35.public void mouseEntered (MouseEvent e) {}
36.public void mouseExited (MouseEvent e) {}
37.public void mouseReleased (MouseEvent e) {}
38.
39.}
```

# 第十三节　读取参数

**在一个** HTML **文件中，上下文为**<applet>**的**<param>**标记能够为** Applet **传递配置信息。例如：**

```
<applet code=DrawAny.class width=100 height=100>
<param name=image value=duke.gif>
</applet>
```

**在这个** Applet **内部，你可用方法** getParameter()**来读取这些值。**

```
1.import java.awt.*;
2.import java.applet.*;
3.
4.public class DrawAny extends Applet {
5.Image im;
6.
7.public void init() {
8.URL url = getDocumentBase();
9.String imageName = getParameter( " image " );
10.im = getImage(url, imageName);
11.}
12.
13.public void paint(Graphics g) {
```

```
14.g.drawImage(im, 0, 0, this);
15.}
16.}
```

**读取参数**

**方法** getParameter()**搜索匹配的名称，并将与之相关的值以字符串的形式返回。**

**如果这个参数名称在位于**<applet></applet>**标记对中的任何**<param>**标记中都未找到，则** getParameter()**返回** null**。一个商业化程序应该很好地处理这种情况。**

**参数的类型都是** String**。如果你需要其他类型的参数，则必须自己做一些转换处理；例如，读取应为** int **类型的参数：**

```
int speed = Integer.parseInt (getParameter ( " speed " ));
```

**由于** HTML **的本性，参数名称对大小写不敏感；但是，使它们全部为大写或小写是一种良好的风格。如果参数值的字符串中含有空格，则应把整个字符串放入双引号中。值的字符串对大小写敏感；不论是否使用双引号，它们的大小写都保持不变。**

# 第十四节　双重目的代码

**是可以在一个单一的类文件中创建既可用作** Java Applet**，又可用作** Java **应用程序的** Java **软件代码。为了理解应用程序的要求，需要做较多的工作，但是一旦已经创建，**Applet/**应用程序代码可作为一个更复杂程序的模板来使用。**

```
1.// Applet/Application which shows an image of Duke in
2.// surfing mode
3.import java.applet.Applet;
4.import java.awt.*;
5.import java.awt.event.*;
6.import java.util.*;
7.
8.public class AppletApp extends Applet {
9.
10.Date date;
11.
12.// An application will require a main()
13.public static void main (String args[]) {
14.
15.// Create a Frame to house the applet
16.Frame frame = new Frame("Application");
17.
18.// Create an instance of the class (applet)
19.AppletApp app = new AppletApp();
20.
21.// Add it to the center of the frame
22.frame.add(app, BorderLayout.CENTER);
23.frame.setSize (250, 150);
24.
25.// Register the AppletApp class as the
26.// listener for a Window Destroy event
27.frame.addWindowListener (new WindowAdapter() {
```

```
28.public void windowClosing (WindowEvent e) {
 29.System.exit(0);
 30.}
31.} );
 32.
33.// Call the applet methods

  1.app.init();
  2.app.start();
  3.frame.setVisible(true);  // Invokes paint()
  4.}
  5.
 6.public void init() {
 7.date = new Date();
  8.}
  9.
10.public void paint (Graphics g) {
11.g.drawString("This Java program started at", 25, 25);
12.g.drawString(date.toString(), 25, 60);
 13.}
 14.}
```

---

**注－应用程序没有浏览器所提供的资源，因此不能使用 getImage()或 getAudioClip()。**

---

# 练习：创建 Applet

**练习目标 － 在本实验中，你将熟悉** Applet **编程，尤其是用于屏幕更新和刷新的** paint() **方法。**

## 一、准备

**为了成功地完成本实验，你必须能够用浏览器来显示一个** Applet。

## 二、任务

**水平** 1：**编写一个** Applet

1. **打开一个新的外壳程序或** Command Tool **窗口。**
2. **用一个文本编辑器，键入** HwMouse.java **程序或从** course_example **目录拷贝它。**
3. **修改这个程序，使得你在** Applet **中点击时，它可以循环显示三种不同的消息。**
4. **编译** HwMouse.java.java **程序**

   c:\student> HwMouse.java
5. **用一个文本编辑器，创建一个** HwMouse.html **文件，文件中含有调用** HwMouse.class **程序的**<applet>**标记。**
6. **用** appletviewer **命令测试你的** Applet。

   c:\student> HwMouse.html

**水平** 2：**创建同心的正方形**

1. **创建一个** Applet，Squares.java，**它产生一系列如下图的同心正方形（或圆形）：**

2. 试图使每个正方形（或圆形）为一种不同的颜色。如果你导入 java.awt.Color 类，则可用 setColor 方法对 Java applet 加入色彩。

```
import java.awt.Color;
. . .
public void paint(Graphics g) {
g.setColor(Color.blue);
g.drawRect(5, 5, 50, 50);
. . .
}
```

**水平 3：创建一个滚动的 Java applet**

1. 编写一个 Applet，显示一幅图像，并在鼠标经过该图像时播放一个声音。

## 三、练习小结

**讨论 － 花几分钟时间讨论一下，在本实验练习过程中你都经历、提出和发现了什么。**

- 经验
- 解释
- 总结
- 应用

## 四、检查你的进度

**在进入下一个模块的学习之前，请确认你能够：**

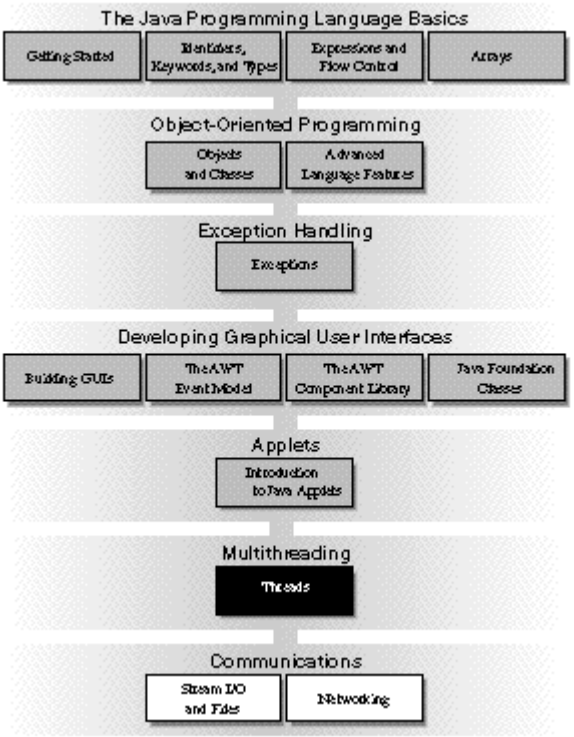
- 区分独立应用程序和 Applet
- 编写一个 HTML 标记来调用 Java Applet
- 描述 Applet 和 AWT 的类层次树
- 创建 HelloWorld.Java Applet
- 列出 Applet 的主要方法
- 描述和使用 AWT 的绘图模型
- 使用 Applet 方法从 URL 读取图像和文件
- 使用<param>标记配置 Applet

## 五、思考题

**如何把 Applet 应用在你公司的 Web 页面上，以改进它的整体表现效果？**

# 第13章 线 程

**本模块讨论多线程，它允许一个程序同时执行多个任务。**




## 第一节 相关问题

**讨论 － 以下为与本模块内容有关的问题：**

- **我如何使我的程序执行多个任务？**

# 第二节 目 标

在完成了本模块的学习后，你应当能够：

- 定义一个线程
- 在一个 Java 程序中创建若干分离的线程，控制线程使用的代码和数据
- 控制线程的执行，并用线程编写独立于平台的代码
- 描述在多个线程共享数据时可能会碰到的困难
- 使用 synchronized 关键字保护数据不受破坏
- 使用 wait()和 notify()使线程间相互通信
- 解释为什么在 JDK1.2 中不赞成使用 suspend()、resume()和 stop()方法？

# 第三节 线 程

线 程
- 什么是线程？
  - 虚拟处理机

### 13.3.1 什么是线程？

一个关于计算机的简化的视图是：它有一个执行计算的处理机、包含处理机所执行的程序的 ROM(只读存储器)、包含程序所要操作的数据的 RAM(只读存储器)。在这个简化视图中，只能执行一个作业。一个关于最现代计算机比较完整的视图允许计算机在同时执行一个以上的作业。

你不需关心这一点是如何实现的，只需从编程的角度考虑就可以了。如果你要执行一个以上的作业，这类似有一台以上的计算机。在这个模型中，线程或执行上下文，被认为是带有自己的程序代码和数据的虚拟处理机的封装。java.lang.Thread 类允许用户创建并控制他们的线程。

---

注－在这个模块中，使用“Thread”时是指 java.lang.Thread 而使用“thread”时是指执行上下文。

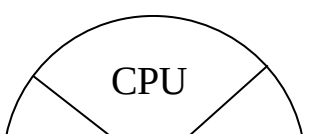
---

### 13.3.2 线程的三个部分

线程的三个部分
- 处理机
- 代码
- 数据

进程是正在执行的程序。一个或更多的线程构成了一个进程。一个线程或执行上下文由三个主要部分组成

- 一个虚拟处理机




A thread or
execution context

- CPU 执行的代码
- 代码操作的数据

代码可以或不可以由多个线程共享，这和数据是独立的。两个线程如果执行同一个类的实例代码，则它们可以共享相同的代码。

类似地，数据可以或不可以由多个线程共享，这和代码是独立的。两个线程如果共享对一个公共对象的存取，则它们可以共享相同的数据。

在 Java 编程中，虚拟处理机封装在 Thread 类的一个实例里。构造线程时，定义其上下文的代码和数据是由传递给它的构造函数的对象指定的。

# 第四节 Java 编程中的线程

### 13.4.1 创建线程

| 创建线程 |
| --- |
| ● 多线程编程 |
| ● 从同一个 Runnbale 实例派生的多线程 |
| ● 线程共享数据和代码。 |

本节介绍了如何创建线程,以及如何使用构造函数参数来为一个线程提供运行时的数据和代码。

一个 Thread 类构造函数带有一个参数，它是 Runnable 的一个实例。一个 Runnable 是由一个实现了 Runnable 接口(即，提供了一个 public void run()方法)的类产生的。

例如：

```
1.public class ThreadTest {
2.public static void main(String args[]) {
3.Xyz r = new Xyz();
4.Thread t = new Thread(r);
5.}
6.}
7.
8.class Xyz implements Runnable {
9.int i;
10.
11.public void run() {
12.while (true) {
13.System.out.println("Hello " + i++);
14.if (i == 50) break;
```

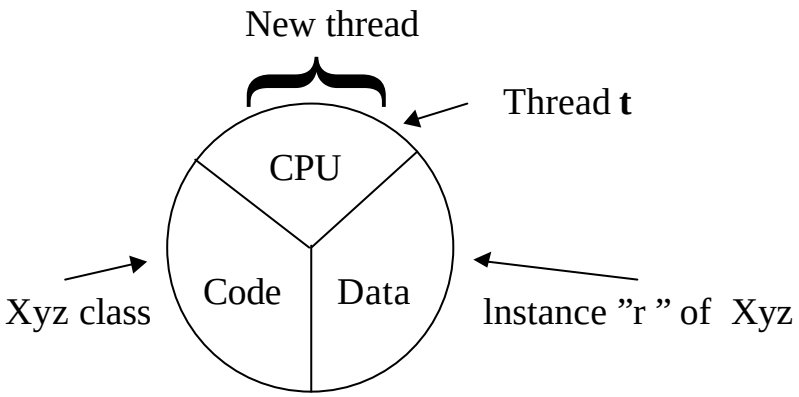
```
15.}
16.}
17.}
```

**首先，**main()**方法构造了** Xyz **类的一个实例** r。**实例** r **有它自己的数据，在这里就是整数** i。**因为实例** r **是传给** Thread **的类构造函数的，所以** r **的整数** i **就是线程运行时刻所操作的数据。线程总是从它所装载的** Runnable **实例(在本例中，这个实例就是** r。**)的** run()**方法开始运行。**

**一个多线程编程环境允许创建基于同一个** Runnable **实例的多个线程。这可以通过以下方法来做到：**

```
Thread t1= new Thread(r);
Thread t2= new Thread(r);
```

**此时，这两个线程共享数据和代码。**




**总之，线程通过** Thread **对象的一个实例引用。线程从装入的** Runnble **实例的** run()**方法开始执行。线程操作的数据从传递给** Thread **构造函数的** Runnable **的特定实例处获得。**

### 13.4.2 启动线程

**启动线程**

- **使用** start()**方法**
- **使线程置于可运行状态**

**一个新创建的线程并不自动开始运行。你必须调用它的** start()**方法。例如，你可以发现上例中第** 4 **行代码中的命令：**

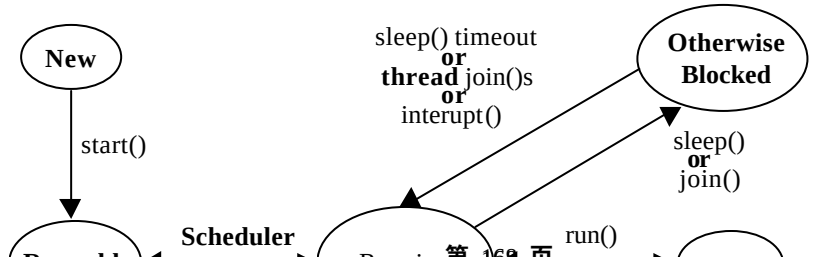
```
t.start();
```

**调用** start()**方法使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由** JVM **调度并执行。这并不意味着线程就会立即运行。**

### 13.4.3 线程调度

**一个** Thread **对象在它的生命周期中会处于各种不同的状态。下图形象地说明了这点：**

***Thread states***



尽管线程变为可运行的，但它并不立即开始运行。在一个只带有一个
处理机的机器上，在一个时刻只能进行一个动作。下节描述了如果有一个以上可运行线程时，如何分配处理机。

在 Java 中，线程是抢占式的，但并不一定是分时的（一个常见的错误是认为“抢占式”只不过是“分时”的一种新奇的称呼而已）。

抢占式调度模型是指可能有多个线程是可运行的，但只有一个线程在实际运行。这个线程会一直运行，直至它不再是可运行的，或者另一个具有更高优先级的线程成为可运行的。对于后面一种情形，低优先级线程被高优先级线程抢占了运行的机会。

一个线程可能因为各种原因而不再是可运行的。线程的代码可能执行了一个 Thread.sleep()调用，要求这个线程暂停一段固定的时间。这个线程可能在等待访问某个资源，而且在这个资源可访问之前，这个线程无法继续运行。

所有可运行线程根据优先级保存在池中。当一个被阻塞的线程变成可运行时，它会被放回相应的可运行池。优先级最高的非空池中的线程会得到处理机时间(被运行)。

因为 Java 线程不一定是分时的，所有你必须确保你的代码中的线程会不时地给另外一个线程运行的机会。这可以通过在各种时间间隔中发出 sleep()调用来做到。

```
1.public class Xyz implements Runnable {
2.public void run() {
3.while (true) {
4.// do lots of interesting stuff
  5.:
6.// Give other threads a chance
  7.try {
 8.Thread.sleep(10);
9.} catch (InterruptedException e) {
10.// This thread's sleep was interrupted
11.// by another thread
12.}
13.}
14.}
15.}
```

注意 try 和 catch 块的使用。Thread.sleep()和其它使线程暂停一段时间的方法是可中断的。线程可以调用另外一个线程的 interrupt()方法，这将向暂停的线程发出一个 InterruptedException。

注意 Thread 类的 sleep()方法对当前线程操作，因此被称作 Thread.sleep(x)，它是一个静态方法。sleep()的参数指定以毫秒为单位的线程最小休眠时间。除非线程因为中断而提早恢复执行，否则它不会在这段时间之前恢复执行。

Thread 类的另一个方法 yield()，可以用来使具有相同优先级的线程获得执行的机会。如果具有相同优先级的其它线程是可运行的，yield()将把调用线程放到可运行池中并使另一个线程运行。如果没有相同优先级的可运行进程，yield()什么都不做。

注意 sleep()调用会给较低优先级线程一个运行的机会。yield()方法只会给相同优先级线程一个执行的机会。

# 第五节 线程的基本控制

### 13.5.1 终止一个线程

当一个线程结束运行并终止时，它就不能再运行了。

可以用一个指示 run()方法必须退出的标志来停止一个线程。

```
1.public class Xyz implements Runnable {
2.private boolean timeToQuit=false;
3.
4.public void run() {
5.while(! timeToQuit) {
6....
7.}
8.// clean up before run() ends
9.}
10.
11.public void stopRunning() {
12.timeToQuit=true;
13.}
14.}
15.
16.public class ControlThread {
17.private Runnable r = new Xyz();
18.private Thread t = new Thread(r);
19.
20.public void startThread() {
21.t.start();
22.}
23.
24.public void stopThread() {
25.// use specific instance of Xyz
26.r.stopRunning();
27.}
28.}
```

在一段特定的代码中，可以使用静态 Thread 方法 currentThread()来获取对当前线程的引用，例如：

```
1.public class Xyz implements Runnable {
2.public void run() {
3.while (true) {
4.// lots of interesting stuff
```

```
5.// Print name of the current thread
6.System.out.println("Thread" +
7.Thread.currentThread().getName()+
8."completed");
9.}
10.}
11.}
```

### 13.5.2 测试一个线程

- 测试一个线程
  - isAlive()
- sleep()
- join()

**有时线程可处于一个未知的状态。**isAlive()**方法用来确定一个线程是否仍是活的。活着的线程并不意味着线程正在运行；对于一个已开始运行但还没有完成任务的线程，这个方法返回** true**。**

### 13.5.3 延迟线程

**存在可以使线程暂停执行的机制。也可以恢复运行，就好象什么也每发生过一样,线程看上去就象在很慢地执行一条指令。**

**sleep()**

sleep()**方法是使线程停止一段时间的方法。在** sleep **时间间隔期满后，线程不一定立即恢复执行。这是因为在那个时刻，其它线程可能正在运行而且没有被调度为放弃执行，除非**

(a)**“醒来”的线程具有更高的优先级**

(b)**正在运行的线程因为其它原因而阻塞**

```
1.public class Xyz implements Runnable {
2.public void run() {
3.while (true) {
4.// lots of interesting stuff
5.// Print name of the current thread
6.System.out.println("Thread" +
7.Thread.currentThread().getName()+
8."completed");
9.}
10.}
11.}
```

**join()**

join()**方法使当前线程停下来等待，直至另一个调用** join **方法的线程终止。例如：**

```
public void doTask() {
TimerThread tt = new TimerThread (100);
tt.start ();
...
// Do stuff in parallel with the other thread for
// a while
...
// Wait here for the timer thread to finish
```

```
try {
tt.join ();
} catch (InterruptedException e) {
// tt came back early
}
...
// Now continue in this thread
...
}
```

**可以带有一个以毫秒为单位的时间值来调用 join 方法，例如：**

```
void join (long timeout);
```

**其中 join()方法会挂起当前线程。挂起的时间或者为 timeout 毫秒，或者挂起当前线程直至它所调用的线程终止。**

# 第六节　创建线程的其它方法

**到目前为止，你已经知道如何用实现了 Runnable 的分离类来创建线程上下文。事实上，这不是唯一的方法。Thread 类自身实现了 Runnable 接口，所以可以通过扩展 Thread 类而不是实现 Runnable 来创建线程。**

```
1.public class MyThread extends Thread {
2.public void run() {
3.while (running) {
4.// do lots of interesting stuff
5.try {
6.sleep(100);
7.} catch (InterruptedException e) {
8.// sleep interrupted
9.}
10.}
11.}
12.
13.public static void main(String args[]) {
14.Thread t = new MyThread();
15.t.start();
16.}
17.}
```

### 13.6.1　使用那种方法？

**使用那种方法？**

- **实现 Runnable**
- **更符合面向对象的设计**
- **单继承**
- **一致性**
- **扩展 Thread**
- **代码更简单**

给定各种方法的选择，你如何决定使用哪个？每种方法都有若干优点。

**实现 Runnable 的优点**

- 从面向对象的角度来看，Thread 类是一个虚拟处理机严格的封装，因此只有当处理机模型修改或扩展时，才应该继承类。正因为这个原因和区别一个正在运行的线程的处理机、代码和数据部分的意义，本教程采用了这种方法。
- 由于 Java 技术只允许单一继承，所以如果你已经继承了 Thread，你就不能再继承其它任何类，例如 Applet。在某些情况下，这会使你只能采用实现 Runnable 的方法。
- 因为有时你必须实现 Runnable，所以你可能喜欢保持一致，并总是使用这种方法。

**继承 Thread 的优点**

- 当一个 run()方法体现在继承 Thread 类的类中，用 this 指向实际控制运行的 Thread 实例。因此，代码不再需要使用如下控制：

  Thread.currentThread().join();

  而可以简单地用：

  join();

因为代码简单了一些，许多 Java 编程语言的程序员使用扩展 Thread 的机制。注意：如果你采用这种方法，在你的代码生命周期的后期，单继承模型可能会给你带来困难。

# 第七节 使用 Java 技术中的 synchronized

本节讨论关键字 synchronized 的使用。它提供 Java 编程语言一种机制，允许程序员控制共享数据的线程。

### 13.7.1 问题

想象一个表示栈的类。这个类最初可能象下面那样：

```
1.public class MyStack {
2.
3.int idx = 0;
4.char [] data = new char[6];
5.
6.public void push(char c) {
7.data[idx] = c;
8.idx++;
9.}
10.
11.public char pop() {
12.idx--;
13.return data[idx];
14.}
15.}
```

注意这个类没有处理栈的上溢和下溢，所以栈的容量是相当有限的。这些方面和本讨论无关。

这个模型的行为要求索引值包含栈中下一个空单元的数组下标。“先进后出”方法用来

产生这个信息。

现在想象两个线程都有对这个类里的一个单一实例的引用。一个线程将数据推入栈，而另一个线程，或多或少独立地，将数据弹出栈。通常看来，数据将会正确地被加入或移走。然而，这存在着潜在的问题。

假设线程 a 正在添加字符，而线程 b 正在移走字符。线程 a 已经放入了一个字符，但还没有使下标加 1。因为某个原因，这个线程被剥夺(运行的机会)。这时，对象所表示的数据模型是不一致的。

```
buffer |p|q|r| | | |
idx  =  2       ^
```

特别地，一致性会要求 idx=3，或者还没有添加字符。

如果线程 a 恢复运行，那就可能不造成破坏，但假设线程 b 正等待移走一个字符。在线程 a 等待另一个运行的机会时，线程 b 正在等待移走一个字符的机会。

pop()方法所指向的条目存在不一致的数据，然而 pop 方法要将下标值减 1。

```
buffer |p|q|r| | | |
idx  =  1    ^
```

这实际上将忽略了字符“r”。此后，它将返回字符“q”。至此，从其行为来看，就好象没有推入字母“r”，所以很难说是否存在问题。现在看一看如果线程 a 继续运行，会发生什么。

线程 a 从上次中断的地方开始运行，即在 push()方法中，它将使下标值加 1。现在你可以看到：

```
buffer |p|q|r| | | |
idx  =  2    ^
```

注意这个配置隐含了：“ q ”是有效的，而含有“ r ”的单元是下一个空单元。也就是说，读取“ q ”时，它就象被两次推入了栈，而字母“ r ”则永远不会出现。

这是一个当多线程共享数据时会经常发生的问题的一个简单范例。需要有机制来保证共享数据在任何线程使用它完成某一特定任务之前是一致的。

---

注－有一种方法可以保证线程 a 在执行完成关键部分的代码时不被调出。这种方法常用在底层的机器语言编程中，但不适合多用户系统。

---

---

注－另外一种方法，它可以被 Java 技术采用。这种方法提供精细地处理数据的机制。这种方法允许无论线程是否会在执行存取的中间被调出，线程对数据的存取都是不可分割的，

---

### 13.7.2 对象锁标志

**对象锁标志**

- 每个对象都有一个标志，它可以被认为是“ 锁标志 ”。
- synchronized 允许和锁标志交互。

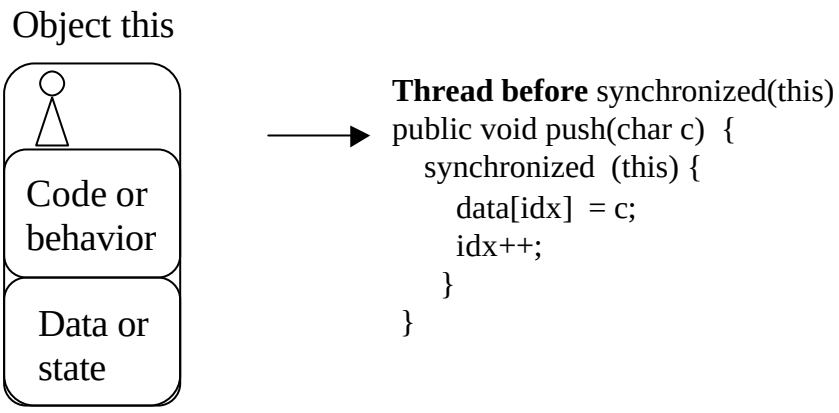
在 Java 技术中，每个对象都有一个和它相关联的标志。这个标志可以被认为是“ 锁标志 ”。 synchronized 关键字使能和这个标志的交互，即允许独占地存取对象。看一看下面修改过的代码片断：

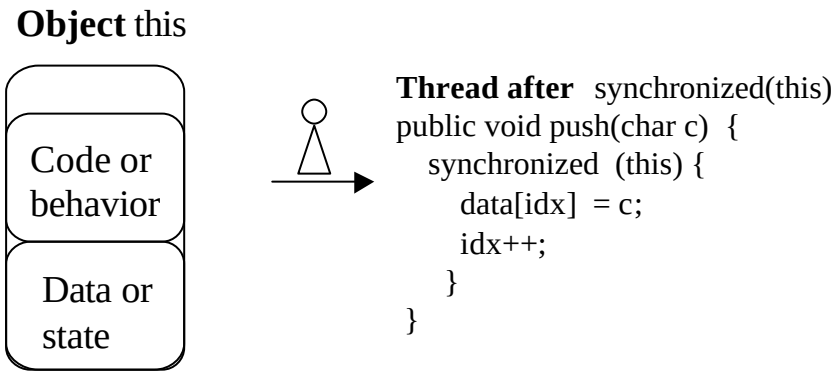
```
public void push(char c) {
synchronized(this) {
data[idx] = c;
idx++;
```

```
    }
  }
```

**当线程运行到** synchronized **语句，它检查作为参数传递的对象，并在继续执行之前试图从对象获得锁标志。**

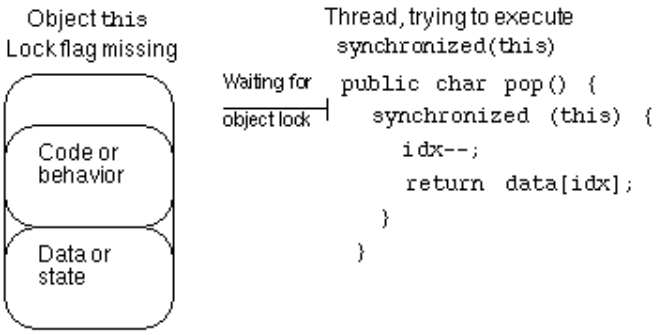
**对象锁标志**




**意识到它自身并没有保护数据是很重要的。因为如果同一个对象的** pop()**方法没有受到** synchronized **的影响，且** pop()**是由另一个线程调用的，那么仍然存在破坏** data **的一致性的危险。如果要使锁有效，所有存取共享数据的方法必须在同一把锁上同步。**

**下图显示了如果** pop()**受到** synchronized **的影响，且另一个线程在原线程持有那个对象的锁时试图执行** pop()**方法时所发生的事情：**




**当线程试图执行** synchronized(this)**语句时，它试图从** this **对象获取锁标志。由于得不到标志，所以线程不能继续运行。然后，线程加入到与那个对象锁相关联的等待线程池中。当标志返回给对象时，某个等待这个标志的线程将得到这把锁并继续运行。**

### 13.7.3 释放锁标志

| 释放锁标志 |
| --- |
| ● 线程执行到 synchronized()代码块末尾时释放 |

由于等待一个对象的锁标志的线程在得到标志之前不能恢复运行,所以让持有锁标志的线程在不再需要的时候返回标志是很重要的。

锁标志将自动返回给它的对象。持有锁标志的线程执行到 synchronized()代码块末尾时将释放锁。Java 技术特别注意了保证即使出现中断或异常而使得执行流跳出 synchronized()代码块，锁也会自动返回。此外，如果一个线程对同一个对象两次发出 synchronized 调用，则在跳出最外层的块时，标志会正确地释放，而最内层的将被忽略。

这些规则使得与其它系统中的等价功能相比，管理同步块的使用简单了很多。

### 13.7.4 synchronized——放在一起

synchronized－放在一起

- 所有对易碎数据的存取应当同步。
- 由 synchronized 保护的易碎数据应当是 private 的。

正如所暗示的那样，只有当所有对易碎数据的存取位于同步块内，synchronized()才会发生作用。

所有由 synchronized 块保护的易碎数据应当标记为 private。考虑来自对象的易碎部分的数据的可存取性。如果它们不被标记为 private，则它们可以由位于类定义之外的代码存取。这样，你必须确信其他程序员不会省略必需的保护。

一个方法，如果它全部属于与这个实例同步的块，它可以把 synchronized 关键字放到它的头部。下面两段代码是等价的：

```
public void push(char c) {
synchronized(this) {
:
:
}
}

public synchronized void push(char c) {
:
:
}
```

为什么使用另外一种技术？

如果你把 synchronized 作为一种修饰符，那么整个块就成为一个同步块。这可能会导致不必要地持有锁标志很长时间，因而是低效的。

然而，以这种方式来标记方法可以使方法的用户由 javadoc 产生的文档了解到：正在同步。这对于设计时避免死锁(将在下一节讨论)是很重要的。注意 javadoc 文档生成器将 synchronized 关键字传播到文档文件中，但它不能为在方法块内的 synchronized(this)做到这点。

### 13.7.5 死锁

**死锁**

- 两个线程相互等待来自对方的锁
- 它不能被监测到或避免
- 它可以通过以下方法来避免
- 决定获取锁的次序
- 始终遵照这个次序
- 按照相反的次序释放锁

如果程序中有多个线程竞争多个资源，就可能会产生死锁。当一个线程等待由另一个线程持有的锁，而后者正在等待已被第一个线程持有的锁时，就会发生死锁。在这种情况下，除非另一个已经执行到 synchronized 块的末尾，否则没有一个线程能继续执行。由于没有一个线程能继续执行，所以没有一个线程能执行到块的末尾。

Java 技术不监测也不试图避免这种情况。因而保证不发生死锁就成了程序员的责任。避免死锁的一个通用的经验法则是：决定获取锁的次序并始终遵照这个次序。按照与获取相反的次序释放锁。

## 第八节 线程交互－wait()和 notify()

**线程交互－wait()和 notify()**

- 场景
  - 把你和出租车司机当作两个线程
- 问题
  - 如何决定你已经到达了你的终点
- 解决方案
  - 你把你的终点和休息时间告诉出租车司机
  - 出租车司机在到达终点时通知你

经常创建不同的线程来执行不相关的任务。然而，有时它们所执行的任务是有某种联系的，为此必须编写使它们交互的程序。

### 13.8.1 场景

把你自己和出租车司机当作两个线程。你需要出租车司机带你到终点，而出租车司机需要为乘客服务来获得车费。所以，你们两者都有一个任务。

### 13.8.2 问题

你希望坐到出租车里，舒服地休息，直到出租车司机告诉你已经到达终点。如果每 2 秒就问一下“我们到了哪里？”，这对出租车司机和你都会是很烦的。出租车司机想睡在出租车里，直到一个乘客想到另外一个地方去。出租车司机不想为了查看是否有乘客的到来而每 5 分钟就醒来一次。所以，两个线程都想用一种尽量轻松的方式来达到它们的目的。

### 13.8.3 解决方案

出租车司机和你都想用某种方式进行通信。当你正忙着走向出租车站时，司机正在车中安睡。当你告诉司机你想坐他的车时，司机醒来并开始驾驶，然后你开始等待并休息。到达终点时，司机会通知你，所以你必须继续你的任务，即走出出租车，然后去工作。出租车司机又开始等待和休息，直到下一个乘客的到来。

### 13.8.4 wait()和 notify()

**线程交互**

- wait()和 notify()
- 池
  - 等待池
  - 锁池

java.lang.Object 类中提供了两个用于线程通信的方法：wait()和 notify()。如果线程对一个同步对象 x 发出一个 wait()调用，该线程会暂停执行，直到另一个线程对同一个同步对象 x 发出一个 notify()调用。

在上个场景中，在车中等待的出租车司机被翻译成执行 cab.wait()调用的“出租车司机”线程，而你使用出租车的需求被翻译成执行 cab.notify()调用的“你”线程。

为了让线程对一个对象调用 wait()或 notify()，线程必须锁定那个特定的对象。也就是说，只能在它们被调用的实例的同步块内使用 wait()和 notify()。对于这个实例来说，需要一个以 synchronized(cab)开始的块来允许执行 cab.wait()和 cab.notify()调用。

关于池

当线程执行包含对一个特定对象执行 wait()调用的同步代码时，那个线程被放到与那个对象相关的等待池中。此外，调用 wait()的线程自动释放对象的锁标志。可以调用不同的 wait()：

wait() 或 wait(long timeout);

对一个特定对象执行 notify()调用时，将从对象的等待池中移走一个任意的线程，并放到锁池中，那里的对象一直在等待，直到可以获得对象的锁标记。 notifyAll()方法将从等待池中移走所有等待那个对象的线程并放到锁池中。只有锁池中的线程能获取对象的锁标记，锁标记允许线程从上次因调用 wait()而中断的地方开始继续运行。

在许多实现了 wait()/notify()机制的系统中，醒来的线程必定是那个等待时间最长的线程。然而，在 Java 技术中，并不保证这点。

注意，不管是否有线程在等待，都可以调用 notify()。如果对一个对象调用 notify()方法，而在这个对象的锁标记等待池中并没有阻塞的线程，那么 notify()调用将不起任何作用。对 notify()的调用不会被存储。

### 13.8.5 同步的监视模型

**同步的监视模型**

- 使共享数据处于一致的状态
- 保证程序不死锁
- 不要将期待不同通知的线程放到同一个等待池

协调两个需要存取公共数据的线程可能会变得非常复杂。你必须非常小心，以保证可能有另一个线程存取数据时，共享数据的状态是一致的。因为线程不能在其他线程在等待这把锁的时候释放合适的锁，所以你必须保证你的程序不发生死锁，

在出租车范例中，代码依赖于一个同步对象——出租车，在其上执行 wait()和 notify()。如果有任何人在等待一辆公共汽车，你就需要一个独立的公共汽车对象，在它上面施用 notify()。记住，在同一个等待池中的所有线程都因来自等待池的控制对象的通知而满足。

永远不要设计这样的程序：把线程放在同一个等待池中，但它们却在等待不同条件的通知。

### 13.8.6 放在一起

下面将给出一个线程交互的实例，它说明了如何使用 wait()和 notify()方法来解决一个经典的生产者－消费者问题。

我们先看一下栈对象的大致情况和要存取栈的线程的细节。然后再看一下栈的详情，以及基于栈的状态来保护栈数据和实现线程通信的机制。

实例中的栈类称为 SyncStack，用来与核心 java.util.Stack 相区别，它提供了如下公共的 API：

```
public synchronized void push(char c);
public synchronized char pop();
```

生产者线程运行如下方法：

```
public void run() {
char c;
for (int i = 0; i < 200; i++) {
c = (char)(Math.random() * 26 + 'A');
theStack.push(c);
System.out.println("Producer" + num + ": " + c);
try {
Thread.sleep((int)(Math.random() * 300));
} catch (InterruptedException e) {
// ignore it
}
}
}
```

这将产生 200 个随机的大写字母并将其推入栈中，每个推入操作之间有 0 到 300 毫秒的随机延迟。每个被推入的字符将显示到控制台上，同时还显示正在执行的生产者线程的标识。
消费者

消费者线程运行如下方法：

```
public void run() {
char c;
for (int i = 0; i < 200; i++) {
c = theStack.pop();
System.out.println(" Consumer" + num + ": " + c);
try {
Thread.sleep((int)(Math.random() * 300));
} catch (InterruptedException e) {
// ignore it
}
}
}
```

上面这个程序从栈中取出 200 个字符，每两个取出操作的尝试之间有 0 到 300 毫秒的随机延迟。每个被弹出的字符将显示在控制台上，同时还显示正在执行的消费者线程的标识。

现在考虑栈类的构造。你将使用 Vector 类创建一个栈，它看上去有无限大的空间。按照这种设计，你的线程只要在栈是否为空的基础上进行通信即可。

**SyncStack 类**

一个新构造的 SyncStack 对象的缓冲应当为空。下面这段代码用来构造你的类：

```
public class SyncStack {
private Vector buffer = new Vector(400,200);
public synchronized char pop() {
}
public synchronized void push(char c) {
}
}
```

请注意，其中没有任何构造函数。包含有一个构造函数是一种相当好的风格，但为了保持简洁，这里省略了构造函数。

现在考虑 push()和 pop()方法。为了保护共享缓冲，它们必须均为 synchronized。此外，如果要执行 pop()方法时栈为空，则正在执行的线程必须等待。若执行 push()方法后栈不再为空，正在等待的线程将会得到通知。

pop()方法如下：

```
public synchronized char pop() {
char c;
while (buffer.size() == 0) {
try {
this.wait();
} catch (InterruptedException e) {
// ignore it
}
}
c = ((Character)buffer.remove(buffer.size()-1)).charValue();
return c;
}
```

注意这里显式地调用了栈对象的 wait()，这说明了如何对一个特定对象进行同步。如果栈为空，则不会弹出任何数据，所以一个线程必须等到栈不再为空时才能弹出数据。

由于一个 interrupt()的调用可能结束线程的等待阶段，所以 wait()调用被放在一个 try/catch 块中。对于本例，wait()还必须放在一个循环中。如果 wait()被中断，而栈仍为空，则线程必须继续等待。

栈的 pop()方法为 synchronized 是出于两个原因。首先，将字符从栈中弹出影响了共享数据 buffer。其次，this.wait()的调用必须位于关于栈对象的一个同步块中，这个块由 this 表示。

你将看到 push()方法如何使用 this.notify()方法将一个线程从栈对象的等待池中释放出来。一旦线程被释放并可随后再次获得栈的锁，该线程就可以继续执行 pop()完成从栈缓冲区中移走字符任务的代码。

---

注 － 在 pop()中，wait()方法在对栈的共享数据作修改之前被调用。这是非常关键的一点，因为在对象锁被释放和线程继续执行改变栈数据的代码之前，数据必须保持一致的状态。你必须使你所设计的代码满足这样的假设：在进入影响数据的代码时，共享数据是处于一致的状态。

---

需要考虑的另一点是错误检查。你可能已经注意到没有显式的代码来保证栈不发生下溢。这不是必需的，因为从栈中移走字符的唯一方法是通过 pop()方法，而这个方法导致正在执行的线程在没有字符的时候会进入 wait()状态。因此，错误检查不是必要的。push()在影响共享缓冲方面与此类似，因此也必须被同步。此外，由于 push()将一个字符加入缓冲区，所以由它负责通知正在等待非空栈的线程。这个通知的完成与栈对象有关。

push()方法如下：

```
public synchronized void push(char c) {
this.notify();
Character charObj = new Character(c);
buffer.addElement(charObj);
}
```

对 this.notify()的调用将释放一个因栈空而调用 wait()的单个线程。在共享数据发生真正的改变之前调用 notify()不会产生任何结果。只有退出该 synchronized 块后，才会释放对象的锁，所以当栈数据在被改变时，正在等待锁的线程不会获得这个锁。

### 13.8.7 SyncStack 范例

**完整的代码**

现在，生产者、消费者和栈代码必须组装成一个完整的类。还需要一个测试工具将这些代码集成为一体。特别要注意，SyncTest 是如何只创建一个由所有线程共享的栈对象的。

SyncTest.java

```
1.package mod14;
2.public class SyncTest {
3.public static void main(String args[]) {
4.
5.SyncStack stack = new SyncStack();
6.
7.Producer p1 = new Producer(stack);
8.Thread prodT1= new Thread(p1);
9.prodT1.start();
10.
11.Producer p2 = new Producer(stack);
12.Thread prodT2= new Thread(p2);
13.prodT2.start();
14.
15.Consumer c1 = new Consumer(stack);
16.Thread consT1 = new Thread(c1);
17.consT1.start();
18.
19.Consumer c2 = new Consumer(stack);
20.Thread consT2 = new Thread(c2);
21.constT2.start();
22.}
23.}
```

Producer.java

```
1.package mod14;
2.public class Producer implements Runnable {
3.private SyncStack theStack;
4.private int num;
5.private static int counter = 1;
```

Producer.java（续）

```
1.
2.public Producer (SyncStack s) {
3.theStack = s;
4.num = counter++;
5.}
6.
7.public void run() {
8.char c;
9.
10.for (int i = 0; i < 200; i++) {
11.c = (char)(Math.random() * 26 + `A');
12.theStack.push(c);
13.System.out.println("Producer" + num + ": " + c);
14.try {
15.Thread.sleep((int)(Math.random() * 300));
16.} catch (InterruptedException e) {
17.// ignore it
18.}
19.}
20.}
21.}
```

Consumer.java

```
1.package mod14;
2.public class Consumer implements Runnable {
3.private SyncStack theStack;
4.private int num;
5.private static int counter = 1;
6.
7.public Consumer (SyncStack s) {
8.theStack = s;
9.num = counter++;
10.}
11.
12.public void run() {
```

Consumer.java（续）

```
1.char c;
2.
3.for (int i=0; i < 200; i++) {
4.c = theStack.pop();
5.System.out.println("Consumer" + num + ": " + c);
6.try {
7.Thread.sleep((int)(Math.random() * 300));
8.} catch (InterruptedException e) {
9.// ignore it
10.}
11.}
```

```
12.}
13.}
```

SyncStack.java

```
1.package mod14;
2.
3.import java.util.Vector;
4.
5.public class SyncStack {
6.private Vector buffer = new Vector(400,200);
7.
8.public synchronized char pop() {
9.char c;
10.
11.while (buffer.size() == 0) {
12.try {
13.this.wait();
14.} catch (InterruptedException e) {
15.// ignore it
16.}
17.}
18.
19. c = ((Character)buffer.remove(buffer.size()- 1).charValue();
```

SyncStack.java（续）

```
1.return c;
2.}
3.
4.public synchronized void push(char c) {
5.this.notify();
6.
7.Character charObj = new Character(c);
8.buffer.addelement(charObj);
9.}
10.}
```

**运行** javamodB.SyncTest **的输出如下。请注意每次运行线程代码时，结果都会有所不同。**

```
Producer2: F

Consumer1: F

Producer2: K

Consumer2: K

Producer2: T

Producer1: N
```

Producer1: V

Consumer2: V

Consumer1: N

Producer2: V

Producer2: U

Consumer2: U

Consumer2: V

Producer1: F

Consumer1: F

Producer2: M

Consumer2: M

Consumer2: T

# 第九节 JDK1.2 中的线程控制

### 13.9.1 suspend()和 resume()方法

suspend()和 resume()方法
- JDK1.2 不赞成使用它们
- 应当用 wait()和 notify()来代替它们

JDK1.2 中不赞成使用 suspend()和 resume()方法。resume()方法的唯一作用就是恢复被挂起的线程。所以，如果没有 suspend()，resume()也就没有存在的必要。从设计的角度来看，有两个原因使 suspend()非常危险：它容易产生死锁；它允许一个线程控制另一个线程代码的执行。下面将分别介绍这两种危险。

假设有两个线程：threadA 和 threadB。当正在执行它的代码时，threadB 获得一个对象的锁，然后继续它的任务。现在 threadA 的执行代码调用 threadB.suspend()，这将使 threadB 停止执行它的代码。

如果 threadB.suspend()没有使 threadB 释放它所持有的锁，就会发生死锁。如果调用 threadB.resume()的线程需要 threadB 仍持有的锁，这两个线程就会陷入死锁。

假设 threadA 调用 threadB.suspend()。如果 threadB 被挂起时 threadA 获得控制，那么 threadB 就永远得不到机会来进行清除工作，例如使它正在操作的共享数据处于稳定状态。为了安全起见，只有 threadB 才可以决定何时停止它自己的代码。

你应该使用对同步对象调用 wait()和 notify()的机制来代替 suspend()和 resume()进行线程控制。这种方法是通过执行 wait()调用来强制线程决定何时“挂起”自己。这使得同步对象的锁被自动释放，并给予线程一个在调用 wait()之前稳定任何数据的机会。

### 13.9.2 stop()方法

stop()方法
- 在终止前释放锁。
- 可能使共享数据处于不一致的状态。
- 应当用 wait()和 notify()来代替它们

stop()方法的情形是类似的，但结果有所不同。如果一个线程在持有一个对象锁的时候被停止，它将在终止之前释放它持有的锁。这避免了前面所讨论的死锁问题，但它又引入了其他问题。

在前面的范例中，如果线程在已将字符加入栈但还没有使下标值加 1 之后被停止，你在释放锁的时候会得到一个不一致的栈结构。

总会有一些关键操作需要不可分割地执行，而且在线程执行这些操作时被停止就会破坏操作的不可分割性。

一个关于停止线程的独立而又重要的问题涉及线程的总体设计策略。创建线程来执行某个特定作业，并存活于整个程序的生命周期。换言之，你不会这样来设计程序：随意地创建和处理线程，或创建无数个对话框或 socket 端点。每个线程都会消耗系统资源，而系统资源并不是无限的。这并不是暗示一个线程必须连续执行；它只是简单地意味着应当使用合适而安全的 wait()和 notify()机制来控制线程。

### 13.9.3 合适的线程控制

既然你已经知道如何来设计具有良好行为的线程，并使用 wait()和 notify()进行通信，而不需要再使用 suspend()和 stop()，那就可以考察下面的代码。注意：其中的 run()方法保证了在执行暂停或终止之前，共享数据处于一致的状态，这是非常重要的。

```
1.public class ControlledThread extends Thread {
2.static final int SUSP=1;
3.static final int STOP=2;
4.static final int RUN=0;
5.private int state = RUN;
6.
7.public synchronized void setState( int s){
8.state = s;
9.if (s == RUN)
10.notify();
11.}
12.
13.public synchronized boolean checkState() {
14.while(state == SUSP) {
15.try {
16.wait();
17.} catch (InterruptedException e) { }
18.}
19.if (state == STOP){
20.return false;
21.}
22.return true;
23.}
```

```
 24.
25.public void run() {
26.while(true) {
27.doSomething();
28.// be sure shared data is in
29.// consistent state in case the
30.// thread is waited or marked for
31.// exiting from run().
 32.if (!checkState())
 33.break;
 34.}
35.}//of run
36.}//of producer
```

一个要挂起、恢复或终止生产者线程的线程用合适的值来调用生产者线程的 setState()方法。当生产者线程确定进行上述操作是安全的时候，它会挂起自己(通过使用 wait()方法)或者停止自己(通过退出 run()方法)。

关于此问题更详细的讨论已超出本模块的范围。

# 练习：使用多线程编程

练习目标－在这个练习中，你将通过编写一些多线程的程序来熟悉多线程的概念。创建一个多线程的 Applet。

## 一、准备

为了很好地完成这个练习，你必须理解本模块中讨论的多线程概念。

## 二、任务

### 水平 1：创建三个线程

1. 创建简单的程序 ThreeThreads.java，它将创建三个线程。每个线程应当显示它所运行的时间。(考虑使用 Date()类)

### 水平 2：使用动画

1. 创建一个 Applet ThreadedAnimation.java，它读取 10 幅 DukeTM waving 图像(在 graphics/Duke 目录中)并按照 Duke 波动的顺序来显示它们。
2. 用 MediaTracker 类使这些图像的装载更平滑。
3. 允许用户连续点击鼠标来停止和启动动画。

## 三、练习小结

讨论－花几分钟时间讨论一下，在本实验练习过程中你都经历、提出和发现了什么。

- 经验
- 解释
- 总结
- 应用

## 四、检查你的进度

在进入下一个模块的学习之前，请确认你能够：

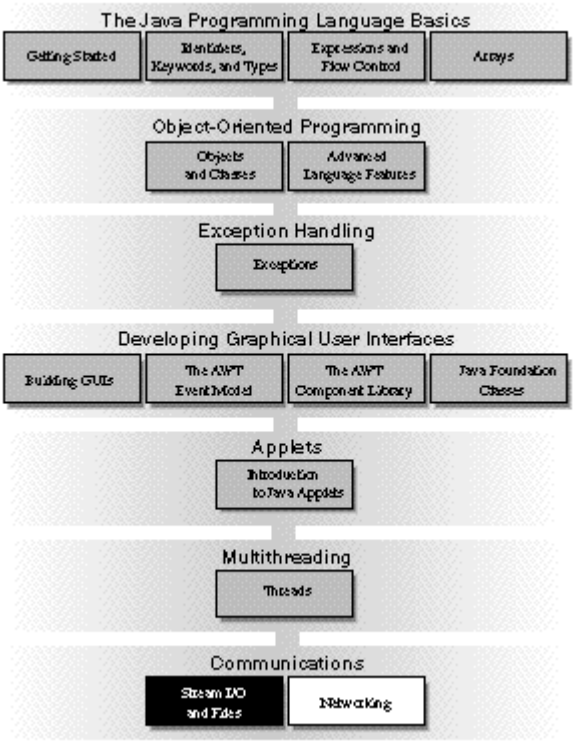
- 定义一个线程
- 在一个 Java 程序中创建若干分离的线程，控制线程使用的代码和数据
- 控制线程的执行，并用线程编写独立于平台的代码
- 描述在多个线程共享数据时可能会碰到的困难
- 使用 synchronized 关键字保护数据不受破坏
- 使用 wait()和 notify()使线程间相互通信
- 使用 synchronized 关键字保护数据不受破坏
- 解释为什么在 JDK1.2 中不赞成使用 suspend()、resume()和 stop()方法？

**五、思考**

**你是否有受益于多线程的应用程序？**

# 第14章　流式 I/O 和文件

**本模块讨论文件，**socket **和其他数据源使用的流式** I/O **机制。**




# 第一节 相关问题

**讨论 － 以下为与本模块内容有关的问题：**

- Java **编程语言中使用什么机制来读写文件？**

# 第二节 目 标

**在完成了本模块的学习后，你应当能够：**

- **描述和使用** java.io **包的流式思想**
- **构造文件和过滤器流，并恰当地使用它们**
- **区别流与读者和作者，并进行合适的选择**
- **考察并操作文件和目录**
- **读、写和更新文本和数据文件**
- **使用** Serialization **接口来保持对象的状态**

# 第三节 流式 I/0

**流式** I/O

- **流是字节的源或目的。**
- **两种基本的流是：**

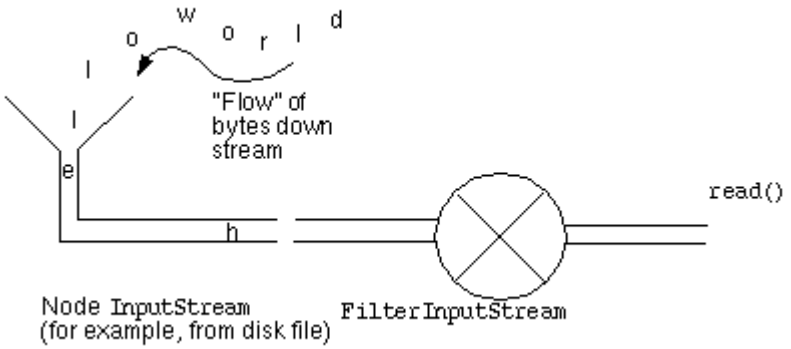
本模块考察了 Java 编程语言如何使用流来处理字节和字符 I/O(包括 stdio，stdout 和 stderr)。下面几节将考察有关处理文件和操作它们所包含的数据的特定细节。

### 14.3.1 流的基础知识

一个流是字节的源或目的。次序是有意义的。例如，一个需要键盘输入的程序可以用流来做到这一点。

两种基本的流是：输入流和输出流。你可以从输入流读，但你不能对它写。要从输入流读取字节，必须有一个与这个流相关联的字符源。

在 java.io 包中，有一些流是结点流，即它们可以从一个特定的地方读写，例如磁盘或者一块内存。其他流称作过滤器。一个过滤器输入流是用一个到已存在的输入流的连接创建的。此后，当你试图从过滤输入流对象读时，它向你提供来自另一个输入流对象的字符。




### 14.3.2 InputStream 方法

InputStream 方法

- 三个基本的 read()方法
  - int read()
  - int read(byte [])
  - int read(byte[], int ,int )
- 其他方法
  - void close()
  - int available()
  - skip(long)
  - boolean markSupported()
  - void mark(int)
  - void reset(int)

- int read()
- int read(byte [])
- int read(byte[], int ,int )

这三个方法提供对输入管道数据的存取。简单读方法返回一个 int 值，它包含从流里读出的一个字节或者-1，其中后者表明文件结束。其它两种方法将数据读入到字节数组中，并返回所读的字节数。第三个方法中的两个 int 参数指定了所要填入的数组的子范围。

---

注－考虑到效率，总是在实际最大的块中读取数据。

---

**void close()**

你完成流操作之后，就关闭这个流。如果你有一个流所组成的栈，使用过滤器流，就关闭栈顶部的流。这个关闭操作会关闭其余的流。

**int available()**

这个方法报告立刻可以从流中读取的字节数。在这个调用之后的实际读操作可能返回更多的字节数。

**skip(long)**

这个方法丢弃了流中指定数目的字符。

boolean markSupported()
void mark(int)
void reset()

如果流支持“回放”操作，则这些方法可以用来完成这个操作。如果 mark()和 reset()方法可以在特定的流上操作，则 markSupported()方法将返回 ture。mark(int)方法用来指明应当标记流的当前点和分配一个足够大的缓冲区，它最少可以容纳参数所指定数量的字符。在随后的 read()操作完成之后，调用 reset()方法来返回你标记的输入点。

### 14.3.3 OutputStream 方法

OutputStream 方法

- 三个基本的 write()方法
  - int write()
  - int write(byte [])
  - int write(byte[], int ,int )
- 其他方法
  - void close()
  - void flush()

- void write(int)
- void write(byte [])
- void write(byte [], int, int)

这些方法写输出流。和输入一样，总是尝试以实际最大的块进行写操作。

**void close()**

当你完成写操作后，就关闭输出流。如果你有一个流所组成的栈，就关闭栈顶部的流。

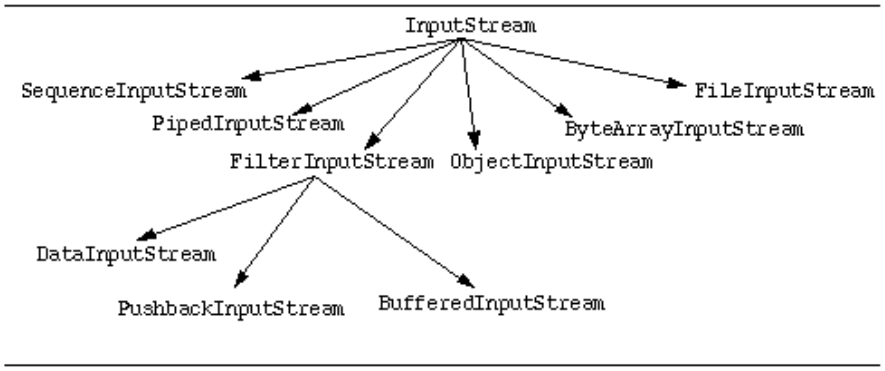
这个关闭操作会关闭其余的流。

**void flush()**

有时一个输出流在积累了若干次之后才进行真正的写操作。flush()方法允许你强制执行写操作。

# 第四节 基本的流类

在 java.io 包中定义了一些流类。下图表明了包中的类层次。一些更公共的类将在后面介绍。




**基本的流类**

- FileInputStream 和 FileOutputStream
- BufferInputStream 和 BufferOutputStream
- DataInputStream 和 DataOutputStream
- PipedInputStream 和 PipedOutputStream

### 14.4.1 FileInputStream 和 FileOutputStream

这些类是结点流，而且正如这个名字所暗示的那样，它们使用磁盘文件。这些类的构造函数允许你指定它们所连接的文件。要构造一个 FileInputStream，所关联的文件必须存在而且是可读的。如果你要构造一个 FileOutputStream 而输出文件已经存在，则它将被覆盖。

```
FileInputStream infile =
new FileInputStream("myfile.dat");
FileOutputStream outfile =
new FileOutputStream("results.dat");
```

### 14.4.2 BufferInputStream 和 BufferOutputStream

这些是过滤器流，它们可以提高 I/O 操作的效率。

### 14.4.3 DataInputStream 和 DataOutputStream

这些过滤器通过流来读写 Java 基本类。例如：

**DataInputStream 方法**

byte readByte()
long readLong()
double readDouble()

**DataOutputStream 方法**

void writeByte(byte)
void writeLong(long)
void writeDouble(double)

**注意** DataInputStream **和** DataOutputStream **的方法是成对的。**

**这些流都有读写字符串的方法，但不应当使用这些方法。它们已经被后面所讨论的读者和作者所取代。**

### 14.4.4 PipedInputStream 和 PipedOutputStream

**管道流用来在线程间进行通信。一个线程的** PipedInputStream **对象从另一个线程的** PipedOutputStream **对象读取输入。要使管道流有用，必须有一个输入方和一个输出方。**

## 第五节 URL 输入流

URL **输入流**

```
java.net.URL imageSource;
try {
imageSource = new URL("http://mysite.com/~info");
} catch ( MalformedURLException e) {}
images[0] = getImage(imageSource, "Duke/T1.gif");
```

**除了基本的文件访问之外，**Java **技术提供了使用统一资源定位器**(URL)**来访问网络上的文件。当你使用** Applet **的** getDocumentBase()**方法来访问声音和图象时，你已经隐含地使用了** URL **对象。**

```
String imageFile = new String ("images/Duke/T1.gif");
images[0] = getImage(getDocumentBase(), imageFile);
```

**然而，你必须象下面的程序那样提供一个直接的** URL

```
java.net.URL imageSource;
try {
imageSource = new URL("http://mysite.com/~info");
} catch ( MalformedURLException e) {}
images[0] = getImage(imageSource, "Duke/T1.gif");
```

### 14.5.1 打开一个输入流

**你可以通过存储文档基目录下的一个数据文件来打开一个合适的** URL **输入流。**

```
1.InputStream is = null;
```

```
2.String datafile = new String("Data/data.1-96");
3.byte buffer[] = new byte[24];
4.try {
5.// new URL throws a MalformedURLException
6.// URL.openStream() throws an IOException
7.is = (new URL(getDocumentBase(),
    datafile)).openStream();
8.} catch (Exception e) {}
```

**现在，你可以就象使用** FileInputStream **对象那样来用** it **来读取信息：**

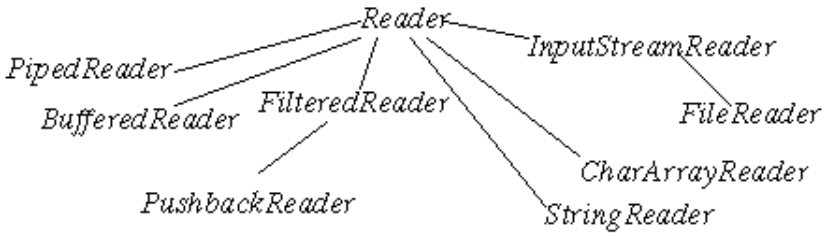
```
1.try {
2.is.read(buffer, 0, buffer.length);
3.} catch (IOException e1) {}
```

---

**警告－记住大多数用户进行了浏览器的安全设置，以防止 Applet 存取文件。**

---

# 第六节 读者和作者




### 14.6.1 Unicode

Java **技术使用** Unicode **来表示字符串和字符，而且它提供了** 16 **位版本的流，以便用类似的方法来处理字符。这些** 16 **位版本的流称为读者和作者。和流一样，它们都在** java.io **包中。**

**读者和作者中最重要的版本是** InputStreamReader **和** OutputStreamWriter**。这些类用来作为字节流与读者和作者之间的接口。**

**当你构造一个** InputStreamReader **或** OutputStreamWriter **时，转换规则定义了** 16 **位** Unicode **和其它平台的特定表示之间的转换。**

### 14.6.2 字节和字符转换

**缺省情况下，如果你构造了一个连接到流的读者和作者，那么转换规则会在使用缺省平台所定义的字节编码和** Unicode **之间切换。在英语国家中，所使用的字节编码是** ISO 8859-1**。**

**你可以使用所支持的另一种编码形式来指定其它的字节编码。在** native2ascii **工具中，你可以找到一个关于所支持的编码形式的列表。**

**使用转换模式，**Java **技术能够获得本地平台字符集的全部灵活性，同时由于内部使用** Unicode**，所以还能保持平台独立性。**

### 14.6.3 缓冲读者和作者

**因为在各种格式之间进行转换和其它** I/O **操作很类似，所以在处理大块数据时效率最高。在** InputStreamReader **和** OutputStreamWriter **的结尾链接一个** BufferedReader **和**

BufferedWriter 是一个好主意。记住对 BufferedWriter 使用 flush()方法。

### 14.6.4 读入字符串输入

下面这个例子说明了从控制台标准输入读取字符串所应当使用的一个技术。

```
1.import java.io.*;
2.public class CharInput {
3.public static void main (String args[]) throws
4.java.io.IOException {
5.String s;
6.InputStreamReader ir;
7.BufferedReader in;
8.ir = new InputStreamReader(System.in);
9.in = new BufferedReader(ir);
10.
11.while ((s = in.readLine()) != null) {
12.System.out.println("Read: " + s);
13.}
14.}
15.}
```

### 14.6.5 使用其它字符转换

如果你需要从一个非本地(例如，从连接到一个不同类型的机器的网络连接读取)的字符编码读取输入，你可以象下面这个程序那样，使用显式的字符编码构造 ir=new InputStreamReader(System.in, “8859_1”);

---

注－如果你通过网络连接读取字符，就应该使用这种形式。否则，你的程序会总是试图将所读取的字符当作本地表示来进行转换，而这并不总是正确的。ISO 8859-1 是映射到 ASCII 的 Latin-1 编码模式。

---

## 第七节 文 件

### 14.7.1 创建一个新的 File 对象

**创建一个新的 File 对象**

```
● File myFile;
myFile = new File("mymotd");
● myFile = new File("/", "mymotd");
// more useful if the directory or filename is
// a variable
● File myDir = new File("/");
myFile = new File(myDir, "mymotd");
```

File 类提供了若干处理文件和获取它们基本信息的方法。

```
● File myFile;
myFile = new File("mymotd");
● myFile = new File("/", "mymotd");
```

```
// more useful if the directory or filename is
// a variable
● File myDir = new File("/");
myFile = new File(myDir, "mymotd");
```

**你所使用的构造函数经常取决于你所使用的其他文件对象。例如，如果你在你的应用程序中只使用一个文件，那么就会使用第一个构造函数。如果你使用一个公共目录中的若干文件，那么使用第二个或者第三个构造函数可能更容易。**

File **类提供了独立于平台的方法来操作由本地文件系统维护的文件。然而它不允许你存取文件的内容。**

---

**注－你可以使用一个 File 对象来代替一个 String 作为 FileInputStream 和 FileOutputStream 对象的构造函数参数。这是一种推荐方法，因为它独立于本地文件系统的约定。**

---

# 第八节 文件测试和工具

**文件测试和工具**

- **文件名**
  - String getName()
  - String getPath()
  - String getAbsolutePath()
  - String getParent()
  - boolean renameTo(File newName)
- **文件测试**
  - boolean exists()
  - boolean canWrite()
  - boolean canRead()
  - boolean isFile()
  - boolean isDirectory()
  - boolean isAbsolute()

**当你创建一个** File **对象时，你可以使用下面任何一种方法来获取有关文件的信息：**

### 14.8.1 文件名

- String getName()
- String getPath()
- String getAbsolutePath()
- String getParent()
- boolean renameTo(File newName)

### 14.8.2 文件测试

- boolean exists()
- boolean canWrite()
- boolean canRead()
- boolean isFile()
- boolean isDirectory()

- boolean isAbsolute()

### 14.8.3 通用文件信息和工具

- long lastModified()
- long length()
- boolean delete()

### 14.8.4 目录工具

- boolean mkdir()
- String[] list()

# 第九节 随机存取文件

### 14.9.1 创建一个随机存取文件

**创建一个随机存取文件**

- **用文件名**

  myRAFile = new RandomAccessFile(String name, String mode);
- **用文件对象**

  myRAFile = new RandomAccessFile(File file, String mode);

**你经常会发现你只想读取文件的一部分数据，而不需要从头至尾读取整个文件。你可能想访问一个作为数据库的文本文件，此时你会移动到某一条记录并读取它的数据，接着移动到另一个记录，然后再到其他记录——每一条记录都位于文件的不同部分。**Java **编程语言提供了一个** RandomAccessFile **类来处理这种类型的输入输出。**

**你可以用如下两种方法来打开一个随机存取文件：**

- **用文件名**

  myRAFile = new RandomAccessFile(String name, String mode);
- **用文件对象**

  myRAFile = new RandomAccessFile(File file, String mode);

mode **参数决定了你对这个文件的存取是只读**(r)**还是读/写**(rw)**。**

**例如，你可以打开一个打开一个数据库文件并准备更新：**

```
RandomAccessFile myRAFile;
myRAFile = new RandomAccessFile("db/stock.dbf","rw");
```

### 14.9.2 存取信息

**随机存取文件**

- long getFilePointer()；
- void seek(long pos)；
- long length()；

RandomAccessFile **对象按照与数据输入输出对象相同的方式来读写信息。你可以访问在** DataInputStrem **和** DataOutputStream **中所有的** read()**和** write()**操作。**

Java **编程语言提供了若干种方法，用来帮助你在文件中移动。**

- long getFilePointer();
  返回文件指针的当前位置。
- void seek(long pos);
  设置文件指针到给定的绝对位置。这个位置是按照从文件开始的字节偏移量给出的。位置 0 标志文件的开始。
- long length()
  返回文件的长度。位置 length()标志文件的结束。

### 14.9.3 添加信息

你可以使用随机存取文件来得到文件输出的添加模式。

```
myRAFile = new RandomAccessFile("java.log","rw");
myRAFile.seek(myRAFile.length());
// Any subsequent write()s will be appended to the file
```

## 第十节 串行化

**串行化**

- 将一个对象存放到永久存储器上称为保持。
- 只有对象的数据被串行化。
- 标记为 transient 关键字的数据不被串行化。

从 JDK1.1 开始具有的新特性包括导入 java.io.Serializable 接口和改变 JVM 使之支持将一个 Java 技术对象存放到一个流的能力。

将一个对象存放到某种类型的永久存储器上称为保持。如果一个对象可以被存放到磁盘或磁带上，或者可以发送到另外一台机器并存放到存储器或磁盘上，那么这个对象就被称为可保持的。

java.io.Serializable 接口没有任何方法，它只作为一个“标记者”，用来表明实现了这个接口的类可以考虑串行化。类中没有实现 Serializable 的对象不能保存或恢复它们的状态。

### 14.10.1 对象图

当一个对象被串行化时，只有对象的数据被保存；方法和构造函数不属于串行化流。如果一个数据变量是一个对象，那么这个对象的数据成员也会被串行化。树或者对象数据的结构，包括这些子对象，构成了对象图。

因为有些对象类所表示的数据在不断地改变，所以它们不会被串行化；例如，java.io.FileInputStream 、java.io.FileOutputStream 和 java.lang.Thread 等流。如果一个可串行化对象包含对某个不可串行化元素的引用，那么整个串行化操作就会失败，而且会抛出一个 NotSerializableException。

如果对象图包含一个不可串行化的引用，只要这个引用已经用 transient 关键字进行了标记，那么对象仍然可以被串行化。

```
public class MyClass implements Serializable {
public transient Thread myThread;
private String customerID;
private int total;
```

域存取修饰符（public,protected,default and private）对于被串行化的对象没有任何作用。写入到流的数据是字节格式，而且字符串被表示为 UTF(文件系统安全的通用字符集转换格式)。transient 关键字防止对象被串行化。

```
public class MyClass implements Serializable {
```

```
public transient Thread myThread;
private transient String customerID;
private int total;
```

# 第十一节 读写一个对象流

### 14.11.1 写

**对一个文件流读写对象是一个简单的过程。考虑如下代码段，它将一个** java.util.Data **对象的实例发送到一个文件：**

```
1.public class SerializeDate {
2.SerializeDate() {
3.Date d = new Date ();
4.try {
5.FileOutputStream f = new
6.FileOutputStream("date.ser");
7.ObjectOutputStream s = new
8.ObjectOutputStream(f);
9.s.writeObject (d);
10.f.close ();
11.} catch (IOException e) {
12.e.printStackTrace ();
13.}
14.}
15.
16.public static void main (String args[]) {
17.new SerializeDate();
18.}
19.}
```

### 14.11.2 读

**读对象和写对象一样简单，只需要说明一点－**readObject()**方法将流作为一个** Object **类型返回，而且在使用那个类的方法之前，必须把它转换成合适的类名。**

```
1.public class UnSerializeDate {
2.UnSerializeDate () {
3.Date d = null;
4.try {
5.FileInputStream f = new
6.FileInputStream("date.ser");
7.ObjectInputStream s = new
8.ObjectInputStream(f);
9.d = (Date) s.readObject ();
10.f.close ();
11.} catch (Exception e) {
12.e.printStackTrace ();
13.}
14.
```

```
15.System.out.println("Unserialized Date object from date.ser");
16.System.out.println("Date: "+d);
17.}
18.
19.public static void main (String args[]) {
20.new UnSerializeDate();
21.}
22.}
```

# 练习：熟悉 I/O

**练习目标－在这个练习中，你将熟悉通过编写执行文件 I/O 的程序来熟悉流式 I/O。**

## 一、准备

**你应当理解数据库和向流写入数据的基本概念。**

## 二、任务

**水平 1：打开文件**

1. **创建一个称为 DisplayFile.java 的 Java 应用程序，它将打开、读取并显示任何可读文件的内容。**
2. **在你的应用程序中包含合适的异常处理，使之在不能显示文件时提示合适的出错信息。**

**水平 2：创建一个简单的数据库程序**

1.**创建一个称为 DBTest.java 的应用程序，它模仿了一个能存储和获取产品记录的小型数据库程序。使用 RandomAccessFile 类和平坦式文件。**

- **数据库中的记录应当由字符串名称和整数量组成。**
- **你的程序应当允许用户显示、更新和添加记录。**

**水平 3：使用保持**

1.**采用模块 9 中的 Paint 程序，并将画布的状态保存到一个文件中。**

## 三、练习小结

**讨论 － 花几分钟时间讨论一下，在本实验练习过程中你都经历、提出和发现了什么。**

- **经验**
- **解释**
- **总结**
- **应用**

## 四、检查一下你的进度

**在进入下一个模块的学习之前，请确认你能够：**

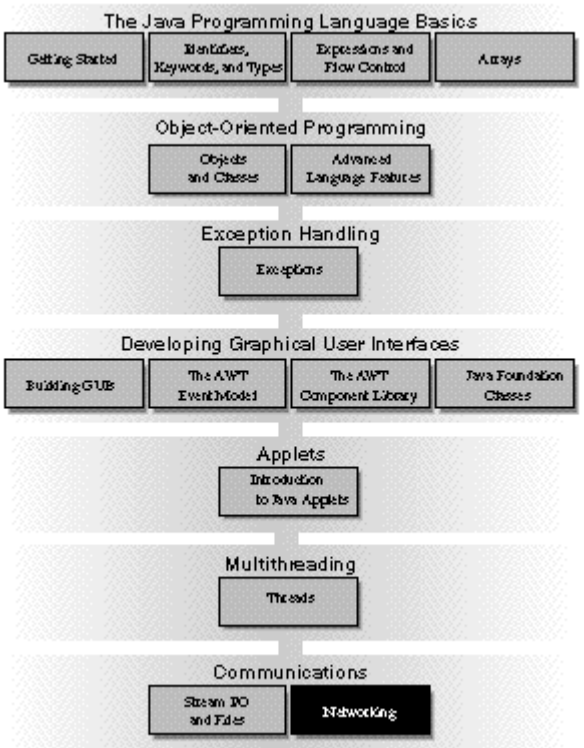
- **描述和使用 java.io 包的流式思想**
- **构造文件和过滤器流，并恰当地使用它们**
- **区别流与读者和作者，并进行合适的选择**
- **考察并操作文件和目录**
- **读、写和更新文本和数据文件**
- **使用 Serialization 接口来保持对象的状态**

**五、思考题**

**你是否有需要 I/O 的应用程序？**

# 第15章 网 络

**本模块讨论了 JDK 对 socket 和 socket 编程的支持。socket 编程用来与在相同的网络上的另一台计算机上运行的程序进行通信。**




## 第一节 相关问题

**讨论 － 以下为与本模块内容有关的问题：**

- **如何在网络上建立客户机与服务器之间的通信链路？**

## 第二节 目 标

**在完成本模块之后，你应当能够：**

- 开发代码来建立网络连接
- 理解 TCP/IP 和 UDP 协议
- 用 ServerSocket 和 Socket 类来实现 TCP/IP 客户和服务器
- 用 DatagramPacket 和 DatagramSocket 来有效地进行基于 UDP 的网络通信。

# 第三节 网 络

网络

- socket
- socket 包含两个流
- 建立连接
- 建立过程与电话系统相似

### 15.3.1 socket

socket 是指在一个特定编程模型下，进程间通信链路的端点。因为这个特定编程模型的流行，socket 这个名字在其他领域得到了复用，包括 Java 技术。

当进程通过网络进行通信时，Java 技术使用它的流模型。一个 socket 包括两个流：一个输入流和一个输出流。如果一个进程要通过网络向另一个进程发送数据，只需简单地写入与 socket 相关联的输出流。一个进程通过从与 socket 相关联的输入流读来读取另一个进程所写的数据。

建立网络连接之后，使用与 socket 相关联的流和使用其他流是非常相似的。

### 15.3.2 建立连接

如果要建立连接，一台机器必须运行一个进程来等待连接，而另一台机器必须试图到达第一台机器。这和电话系统类似；一方必须发起呼叫，而另一方在此时必须等待电话呼叫。

# 第四节 Java 技术中的网络

Java 技术中的网络

- 连接的地址
- 远程机器的地址和名字
- 端口号表明目的
- 端口号
- 范围从 0~65535

### 15.4.1 连接的地址

你发起电话呼叫时，你必须知道所拨的电话号码。如果要发起网络连接，你需要知道远程机器的地址或名字。此外，每个网络连接需要一个端口号，你可以把它想象成电话的分机号码。一旦你和一台计算机建立连接，你需要指明连接的目的。所以，就如同你可以使用一个特定的分机号码来和财务部门对话那样，你可以使用一个特定的端口号来和会计程序通信。

### 15.4.2 端口号

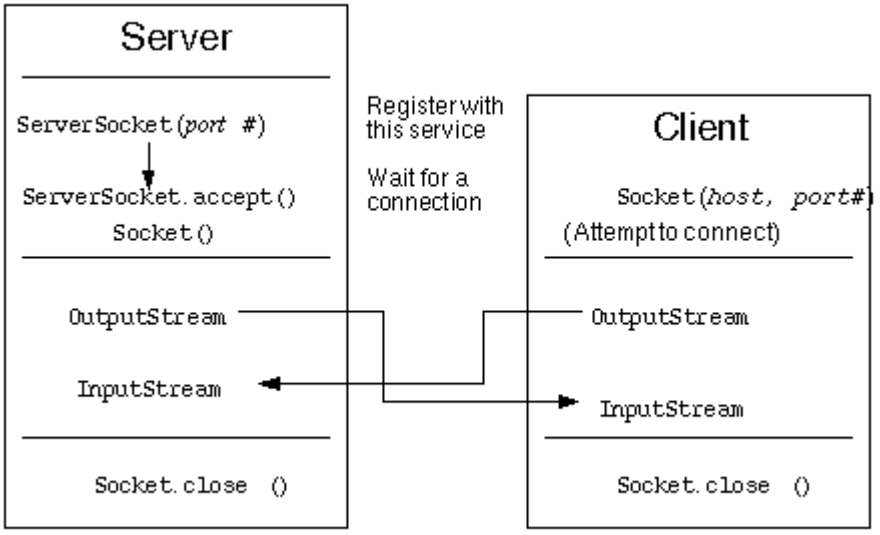
TCP/IP 系统中的端口号是一个 16 位的数字，它的范围是 0~65535。实际上，小于 1024

的端口号保留给预定义的服务，而且除非要和那些服务之一进行通信(例如 telnet，SMTP 邮件和 ftp 等)，否则你不应该使用它们。

客户和服务器必须事先约定所使用的端口。如果系统两部分所使用的端口不一致，那就不能进行通信。

### 15.4.3 Java 网络模型

在 Java 编程语言中，TCP/IP socket 连接是用 java.net 包中的类实现的。下图说明了服务器和客户端所发生的动作。




- 服务器分配一个端口号。如果客户请求一个连接，服务器使用 accept()方法打开 socket 连接。
- 客户在 host 的 port 端口建立连接。
- 服务器和客户使用 InputStream 和 OutputStream 进行通信。

### 15.4.4 最小 TCP/IP 服务器

TCP/IP 服务器应用程序依靠 Java 技术语言提供的网络类。ServerSocket 类完成了建立一个服务器所需的大部分工作。

```
1.import java.net.*;
2.import java.io.*;
3.
4.public class SimpleServer {
5.public static void main(String args[]) {
6.ServerSocket s = null;
7.Socket s1;
8.String sendString = "Hello Net World!";
9.OutputStream s1out;
10.DataOutputStream dos;
11.
12.// Register your service on port 5432
13.try {
14.s = new ServerSocket(5432);
15.} catch (IOException e) { }
16.
17.// Run the listen/accept loop forever
```

```
18.while (true) {
19.try {
20.// Wait here and listen for a connection
21.s1=s.accept();
22.
23.// Get a communication stream for soocket
24.s1out = s1.getOutputStream();
25.dos = new DataOutputStream (s1out);
26.
27.// Send your string! (UTF provides machine-independent format)
28.dos.writeUTF(sendString);
29.
30.// Close the connection, but not the server socket
31.s1out.close();
32.s1.close();
33.} catch (IOException e) { }
34.}
35.}
36.}
```

### 15.4.5 最小 TCP/IP 客户

**一个** TCP/IP **应用程序的客户方依靠** Socket **类。**Socket **类完成了建立一个连接所需的大部分工作。客户连接到上一页所示的服务器上，并将服务器发送的所有数据显示在控制台上。**

```
1.import java.net.*;
2.import java.io.*;
3.
4.public class SimpleClient {
5.public static void main(String args[]) throws IOException {
6.int c;
7.Socket s1;
8.InputStream s1In;
9.DataInputStream dis;
10.
11.// Open your connection to sunbert, at port 5432
12.s1 = new Socket("sunbert",5432);
13.
14.// Get an input file handle from the socket and read the input
15.s1In = s1.getInputStream();
16.dis = new DataInputStream(s1In);
17.
18.String st = new String (dis.readUTF());
19.System.out.println(st);
20.
21.// When done, just close the connection and exit
22.s1In.close();
23.s1.close();
24.}
25.}
```

# 第三章第五节 UDP socket

UDP socket

- 它们是无连接的协议。
- 不保证消息的可靠传输。
- 它们由 Java 技术中的 DatagramSocket 和 DatagramPacket 类支持。

TCP/IP 是面向连接的协议。而用户数据报协议(UDP)是一种无连接的协议。要区分这两种协议，一种很简单而又很贴切的方法是把它们比作电话呼叫和邮递信件。

电话呼叫保证有一个同步通信；消息按给定次序发送和接收。而对于邮递信件，即使能收到所有的消息，它们的顺序也可能不同。

用户数据报协议(UDP)由 Java 软件的 DatagramSocket 和 DatagramPacket 类支持。包是自包含的消息，它包括有关发送方、消息长度和消息自身。

### 15.5.1 DatagramPacket

DatagramPacket

DatagramPacket 有两个构造函数：一个用来接收数据，另一个用来发送数据。

DatagramPacket(byte [] recvBuf, int readLength)

DatagramPacket(byte [] sendBuf, int sendLength, InetAddress iaddr, int iport)

DatagramPacket 有两个构造函数：一个用来接收数据，另一个用来发送数据：

- DatagramPacket(byte [] recvBuf, int readLength)－用来建立一个字节数组以接收 UDP 包。byte 数组在传递给构造函数时是空的，而 int 值用来设定要读取的字节数(不能比数组的大小还大)。
- DatagramPacket(byte [] sendBuf, int sendLength, InetAddress iaddr, int iport)－用来建立将要传输的 UDP 包。sendLength 不应该比 sendBuf 字节数组的大小要大。

### 15.5.2 DatagramSocket

DatagramSocket

DatagramSocket 有三个构造函数：

DatagramSocket 用来读写 UDP 包。这个类有三个构造函数，允许你指定要绑定的端口号和 internet 地址：

- DatagramSocket()－绑定本地主机的所有可用端口
- DatagramSocket(int port)－绑定本地主机的指定端口
- Datagra

mSocket(int port, InetAddress iaddr)－绑定指定地址的指定端口

### 15.5.3 最小 UDP 服务器

最小 UDP 服务器在 8000 端口监听客户的请求。当它从客户接收到一个 DatagramPacket 时，它发送服务器上的当前时间。

```
1.import java.io.*;
2.import java.net.*;
3.import java.util.*;
4.
5.public class UdpServer{
6.
7.//This method retrieves the current time on the server
8.public byte[] getTime(){
9.Date d= new Date();
10.return d.toString().getBytes();
11.}
12.
13.// Main server loop.
14.public void go() throws IOException {
15.
16.DatagramSocket datagramSocket;
17.DatagramPacket inDataPacket; // Datagram packet from the client
18.DatagramPacket outDataPacket; // Datagram packet to the client
19.InetAddress clientAddress; // Client return address
20.int clientPort; // Client return port
21.byte[] msg= new byte[10]; // Incoming data buffer. Ignored.
22.byte[] time; // Stores retrieved time
23.
24.// Allocate a socket to man port 8000 for requests.
25.datagramSocket = new DatagramSocket(8000);
26.System.out.println("UDP server active on port 8000");
27.
28.// Loop forever
29.while(true) {
30.
31.// Set up receiver packet. Data will be ignored.
32.inDataPacket = new DatagramPacket(msg, msg.length);
```

最小 UDP 服务器(续)

```
1.
2.// Get the message.
3.datagramSocket.receive(inDataPacket);
4.
5.// Retrieve return address information, including InetAddress
6.// and port from the datagram packet just recieved.
```

```
 7.
 8.clientAddress = inDataPacket.getAddress();
 9.clientPort = inDataPacket.getPort();
10.
11.// Get the current time.
12.time = getTime();
13.
14.//set up a datagram to be sent to the client using the
15.//current time, the client address and port
16.outDataPacket = new DatagramPacket
17.(time, time.length, clientAddress, clientPort);
18.
19.//finally send the packet
20.datagramSocket.send(outDataPacket);
21.}
22.}
23.
24.public static void main(String args[]) {
25.UdpServer udpServer = new UdpServer();
26.try {
27.udpServer.go();
28.} catch (IOException e) {
29.System.out.println ("IOException occured with socket.");
30.System.out.println (e);
31.System.exit(1);
32.}
33.}
34.}
```

### 15.5.4 最小 UDP 客户

**最小** UDP **客户向前面创建的客户发送一个空包并接收一个包含服务器实际时间的包。**

```
 1.import java.io.*;
 2.import java.net.*;
 3.
 4.public class UdpClient {
 5.
 6.public void go() throws IOException, UnknownHostException {
 7.DatagramSocket datagramSocket;
 8.DatagramPacket outDataPacket; // Datagram packet to the server
 9.DatagramPacket inDataPacket; // Datagram packet from the server
10.InetAddress serverAddress; // Server host address
11.byte[] msg = new byte[100]; // Buffer space.
12.String receivedMsg; // Received message in String form.
13.
14.// Allocate a socket by which messages are sent and received.
15.datagramSocket = new DatagramSocket();
16.
17.// Server is running on this same machine for this example.
18.// This method can throw an UnknownHostException.
```

```
19.serverAddress = InetAddress.getLocalHost();
 20.
21.// Set up a datagram request to be sent to the server.
22.// Send to port 8000.
23.outDataPacket = new DatagramPacket(msg, 1, serverAddress, 8000);
 24.
25.// Make the request to the server.
 26.datagramSocket.send(outDataPacket);
 27.
28.// Set up a datagram packet to receive server's response.
29.inDataPacket = new DatagramPacket(msg, msg.length);
 30.
```

**最小 UDP 客户(续)**

```
  1.// Receive the time data from the server
 2.datagramSocket.receive(inDataPacket);
  3.
 4.// Print the data received from the server
 5.receivedMsg = new String
 6.(inDataPacket.getData(), 0, inDataPacket.getLength());
  7.System.out.println(receivedMsg);
  8.
 9.//close the socket
10.datagramSocket.close();
11.}
 12.
13.public static void main(String args[]) {
14.UdpClient udpClient = new UdpClient();
15.try {
16.udpClient.go();
17.} catch (Exception e) {
18.System.out.println ("Exception occured with socket.");
19.System.out.println (e);
20.System.exit(1);
21.}
22.}
23.}
```

# 练习：使用 Socket 编程

**练习目标－通过实现一个使用** socket **进行通信的客户和服务器来获得关于使用** socket **的经验。**

## 一、准备工作

**为了很好地完成这个练习，你必须对** HTML **和网络有清晰的理解。**

## 二、任务

水平 1：创建 socket

成对地进行这个练习，这样你可以使用其他人的机器的名字。你将要创建一个服务器和客户对，还有一个从其中之一请求一个文件的程序。

1. 服务器模板(FileServer.java)位于 templates 目录下。编写一个方法，它能使服务器接收来自客户的文件名字符串，试图打开这个文件并通过 socket 将它传回到客户。
2. 客户模板(ReadFile.java)也位于 templates 目录下。客户程序将文件名字符串作为一个参数并将它发给服务器，然后等待服务器发送错误响应或文件。

水平 3：创建一个多线程的服务器(MultiServer.java)

1. 扩展客户代码，使客户能请求多个文件。
2. 扩展客户，使它在没有错误返回时，将文件存放到磁盘。
3. 使用线程扩展服务器，这样多个客户就可以同时连接到服务器。

## 三、练习小结

讨论 － 花几分钟时间讨论一下，在本实验练习过程中你都经历、提出和发现了什么。

- 经验
- 解释
- 总结
- 应用

## 四、检查一下你的进度

在进入下一个模块的学习之前，请确认你能够：

- 开发代码来建立网络连接
- 理解 TCP/IP 和 UDP 协议
- 用 ServerSocket 和 Socket 类来实现 TCP/IP 客户和服务器
- 用 DatagramPacket 和 DatagramSocket 来有效地进行基于 UDP 的网络通信。

## 五、思考题

有若干关于 Java 平台的高级话题，它们中的许多将在其他 Sun 教育课程中讨论。附录 A 给出了其中一些的简短描述。你还可以查看 JavaSoft 的 Web 站点(www.javasoft.com)。

# 附录 A Java 高级编程的元素

## 一、目标

完成这个附录之后，你应当能够：

- 理解分布式计算的二层体系结构和三层体系结构
- 理解 Java 编程语言作为数据库应用程序的前台程序的作用
- 使用 JDBC API
- 理解使用对象代理的数据交换方法
- 解释 JavaBeans 的组件模型

# 二、二层体系结构和三层体系结构

客户/服务器包括两个或更多的计算机共享一个与完整应用程序相关的任务。理想情况下，每台计算机执行的逻辑适合它的设计和所声称的功能。

使用最广泛的客户/服务器实现是二层体系结构。这包括一个前台的客户应用程序与在另外一台计算机上运行的后台数据库引擎通信。客户程序向数据库服务器发送 SQL 语句。服务器返回合适的结果，而客户负责处理数据。

应用程序所采用的基本二层客户/服务器模型得到了许多流行数据库系统的支持，包括：Oracle、Sybase 和 Informix。

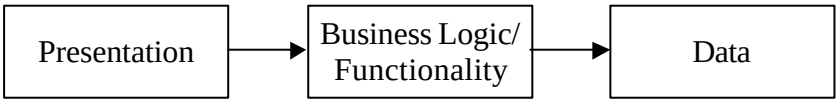
二层客户/服务器结构在性能上损失很大。因为客户方要处理大部分的逻辑，所以客户方软件变得越来越大，越来越复杂。而服务器方的逻辑仅限于数据库操作。这里的客户称为肥客户。

肥客户可能为了访问远程数据库而产生频繁的网络流量。这对于 Intranet 和基于局域网的网络拓扑来说是不错的，并对台式机在内存和磁盘方面的需求产生了重要的影响。此外，后台的数据库在服务器所提供的逻辑方面并不都是相同的，而且它们都有自己的 API 集合，程序员必须使用这些 API 来优化和缩放性能。下面要介绍的三层客户/服务器结构，以更有效的方式来处理可伸缩性、性能和逻辑分割。

# 三、三层体系结构

三层体系结构是最先进的客户/服务器软件结构。三层体系结构一开始要求很陡的开发曲线，特别是当你必须支持不同的平台和网络环境。它的好处在于减少网络流量，优秀的 Internet 和 intranet 性能以及对系统的扩展和增长有更多的控制。

**三层客户/服务器定义**



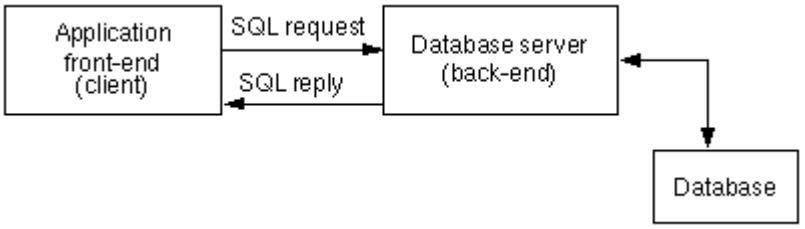

三层客户/服务器环境的三个组件或三个层次是表示，商业逻辑或功能，以及数据。它们是分离的，因此这样其中任何一层的软件都可以用一种不同的实现来代替而不影响其它层。例如，如果你想把一个面向字符的屏幕替换成一个图形用户界面(表示层)，你只要使用已建立的 API 或接口来编写图形用户界面，用它来存取面向字符的屏幕中的功能程序。商业逻辑提供了定义所有商业规则的功能，而操作数据是通过商业规则进行的。商业政策的改变可以只对这一层产生影响，而不影响数据库。第三层，也就是数据层，包括系统、应用程序和数据，其中数据被封装起来，这是为了利用这个体系结构的优点，即最小的程序移植量。

# 四、数据库前台

Java 编程语言为软件工程师创建面向数据库系统的前台应用程序提供了很多便利。由于它的“一次编写，到处使用”[SM] 的特点，Java 编程语言具有可适用于多种硬件和操作系统的优点。甚至在多平台环境中，程序员也不需要编写与特点平台有关的前台应用程序。

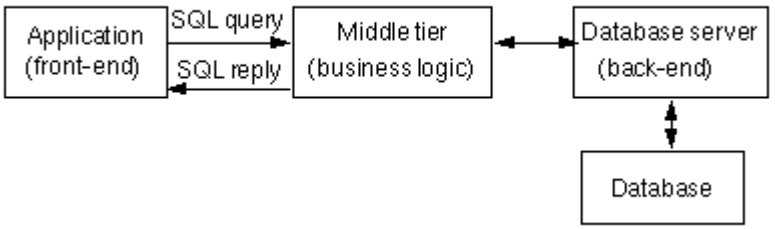
Java 技术支持大量用于前台开发的类，这使得通过 JDBC API 来和数据库进行交互成为可能。JDBC API 提供了到后台数据库的连接，它可以可以返回查询结果，并由前台处理。

在二层模型中，数据库位于数据库服务器。客户执行的前台应用程序打开一个 socket，用来进行网络通信。socket 提供了客户应用程序和后台服务器之间的通信。在下图中，客户程序向客户服务器发送 SQL 数据库请求。服务器将结果返回给客户，由客户格式化结果并进行表示。




经常使用的数据操作机制经常被作为嵌入的“存储过程”。在操作数据库的过程中，满足一定条件就会自动执行触发器。这个模型的主要缺点是所有的商业规则都在客户应用程序中实现，创建了庞大的客户方运行时(程序)以及增加客户端代码的重写量。

在三层模型中，所有的商业规则都嵌入到客户(前台)层。它和中间服务器交互，后者提供了关于后台应用程序的抽象。中间层管理商业规则，而商业规则通过应用程序的管理条件来操纵数据。中间服务器也接受基于各种通信协议的从若干客户到一个或多个服务器的连接。中间层为应用程序提供了一个与数据库无关的接口，并使得前台更加健壮。



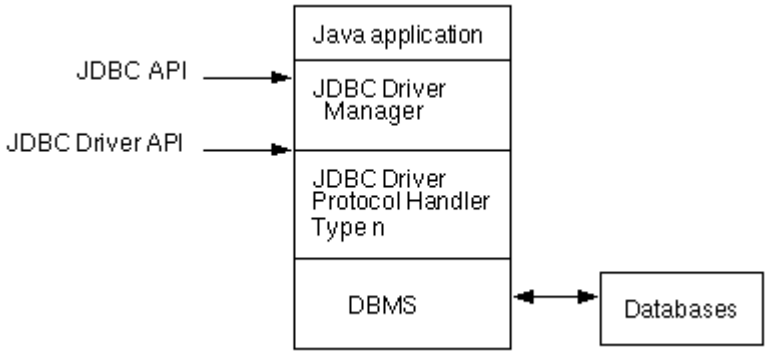

# 五、JDBC API 介绍

创建健壮且平台无关的应用程序和基于 Web 的 Applet 的能力促使开发者提供具有前台连接性的解决方案。JavaSoft 与数据库和数据库工具厂商合作创造了一个与数据库管理系统无关的机制，它使得开发者可以编写能在各种数据库上运行的客户应用程序。这种努力的成果就是 Java 数据库连接应用程序编程接口(JDBC API)。

**1.JDBC，概述**

JDBC 提供了访问数据库的标准接口。JDBC 的模型对开放数据库连接(ODBC)进行了改进，它包含一套发出 SQL 语句、更新表和调用存储过程的类和方法。

如下图所示，Java 编程语言前台应用程序使用 JDBC API 来和 JDBC 驱动管理器进行交互。JDBC 驱动管理器使用 JDBC Driver API 来装载合适的 JDBC 驱动。JDBC 可以从不同的数据库厂商处得到，它用来和底层的 DBMS 通信。

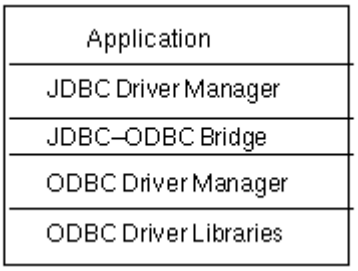



**2.JDBC 驱动**

Java 应用程序使用 JDBC API，并通过数据库驱动和数据库连接。大多数数据库引擎带有与它们关联的 JDBC 驱动。JavaSoft 定义了四种类型的驱动。关于更多的细节，可以参考：
http://java.sun.com/products/jdbc/jdbc.drivers.html

**3.JDBC ODBC 桥**

JDBC－ODBC 桥是一个 JDBC 驱动，它把 JDBC 调用转换为 ODBC 操作。这个桥使得所有支持 ODBC 的 DBMS 都可以和 Java 应用程序交互。




JDBC－ODBC 桥接口作为一套共享动态 C 库提供的。ODBC 提供了客户方一套适合于客户方操作系统的库和驱动。这些 ODBC 调用都是 C 调用，而且客户必须带有 ODBC 驱动和相关的客户方库的本地副本。这限制了它在基于 web 的应用程序中的使用。

# 六、分布式计算

Java 技术可用于创建分布式计算环境。两种流行的技术是远程方法调用(RMI)和公共对象请求代理体系结构(CORBA)。RMI 和远过程调用(RPC)类似，它很受 Java 编程语言的程序员青睐。CORBA 提供了在异构开发环境中的灵活性。

**1.RMI**

RMI 特性使在客户计算机上运行的程序可以调用远程服务器机器上的对象的方法。它提供了程序员在网络环境进行分布式计算的能力。面向对象的设计要求每个任务都是由最适合那个任务的对象执行。RMI 扩展了这个概念，它允许任务由最适合它的机器执行。RMI 定义了一套可以创建远程对象的远程接口。客户可以用和调用本地对象的方法相同的句法来调用远程对象的方法。RMI API 提供了处理所有底层通信和访问远程方法所需的参数引用的类。

在所有分布式体系结构上，一个应用程序进程或对象服务器(监护者)向世界广播自身，这是通过使用本地机器(结点)上的名字服务进行注册来做到的。使用 RMI 时，一个称为 RMI 注册器的名字服务监护者在 RMI 端口上运行，它缺省地监听主机的 1099IP 端口。

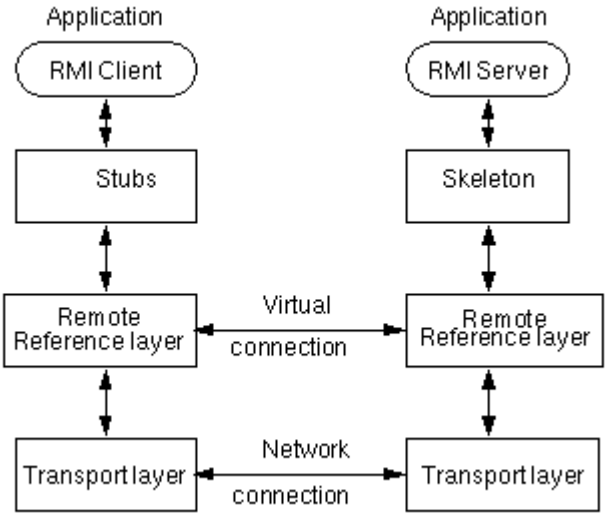
RMI 注册器包含了一张关于远程对象引用的内部表。对于每个远程对象，表中包含一个注册名和这个对象的引用。通过实例化和用不同的名字将对象多次绑定到注册器，就可以在表中存放一个对象的多个实例。

当一个 RMI 客户通过注册器绑定一个远程对象时，它将通过接口接收到关于远程实例化对象的本地引用，并通过这个引用和对象通信。

通过导入远程的 RMI 包和创建远程对象的引用，Applet 开始运行。一旦 Applet 建立链接，它会调用远程对象的方法，就好象这些方法可以在本地得到。

**2.RMI 的体系结构**

RMI 的体系结构提供了三个层次：存根(stub)/骨架(skeleton)、远程引用和传输层。




传输层创建并维护客户和服务器之间的物理链接。它处理在客户和服务器之上的远程/引用层(RRL)中传输的数据流。

远程引用层提供了一个用户客户和服务器之间虚拟网络的独立引用协议。它提供了下面的传输层和上面的存根/骨架层之间的接口。

存根是表示远程对象的客户方代理。客户通过接口和存根交互。存根对于客户就象一个本地对象。服务器方的骨架作为 RRL 和在服务器方实现的对象之间的一个接口。

**3.创建 RMI 应用程序**

这一节指导你有关创建、编译和运行一个 RMI 应用程序的步骤。这个过程包括以下步骤：

- 定义远程类的接口
- 创建并编译远程类的实现类
- 使用 rmic 命令创建存根和骨架类
- 创建并编译服务器应用程序
- 启动 RMI 注册器和服务器应用程序
- 创建并编译客户程序来存取远程对象
- 测试客户

# 七、CORBA

CORBA 是一个规范，它定义远程对象如何进行互操作。CORBA 规范由对象管理小组(OMG)控制，它是一个有 700 多家公司共同参与，制定分布式计算开放标准的国际性开放组织。有关更多的细节，可以参考以下 URL：

http://www.omg.org

CORBA 对象可以用任何编程语言重写，包括 C 和 C++。这些对象可以存在于任何平台之上，包括 Solaris、Windows95/NT、openVMS、Digital UNIX、HP_UX 和其它许多平台。这意味着,在一个 Windows95 平台上运行的 Java 应用程序可以动态装载并使用由 UNIX Web 服务器存放在 Internet 上的 C++对象。

通过使用接口定义语言（IDL）构造对象的接口，可以获得语言无关性。IDL 允许用同样的方式描述所有的 CORBA 对象；唯一的要求是本地语言（C / C++，COBOL，Java）和 IDL 之间的一座“桥”。

CORBA 的核心是对象资源代理（ORB）。ORB 是在 CORBA 应用程序的客户和服务器之间传输信息的主要组件。ORB 管理调度请求，建立到服务器的连接，发送数据，并在服

务器上执行请求。服务器返回操作结果的过程与之类似。来自不同厂商的 ORB 在 TCP/IP 上通信时，使用的是 Internet 内部 ORB 协议（IIOP），它是 CORBA 2.0 标准的一部分。

## 八、Java IDL

Java IDL 给 Java 编程语言增加了 CORBA 能力，它提供了基于标准的互操作性和连接性。Java IDL 允许分布式 Java web 应用程序使用工业标准 IDL 和 IIOP 来透明地请求远程网络服务器上的操作。

Java IDL 不是 OMG 的 IDL 的一个实现。事实上，它是一个使用 IDL 来定义接口的 CORBA ORB。idltojava 编译器生成可移植的客户存根和服务器骨架。通过使用名字服务存取引用对象，这使得 CORBA 客户能够与在远程服务器上运行的另一个对象交互。与 RMI 注册器类似，名字服务是在远程服务器上运行的一个后台进程它包含一张关于名字服务和用来解析客户请求的远程对象引用的表。

建立一个 CORBA 对象的步骤可以概述如下：

- 使用接口定义语言（IDL）来创建对象的接口。
- 使用 javatoidl 编译器将接口转换成存根和骨架。
- 实现骨架对象，创建 CORBA 服务器对象。
- 编译并执行服务器对象，将它绑定到名字服务。
- 执行客户对象，通过 CORBA 名字服务存取服务器对象。

## 九、RMI 与 CORBA 之比较

RMI 最大的优点源于以下事实：它被设计成一个安全的解决方案。这意味着构筑 RMI 应用程序非常简单，而且所有的远程对象和本地对象有相同的特性，对于一个多层解决方案，有可能组合 JDBC 和 RMI 两者的精华部分。

与此同时，CORBA 受益于如下事实，它是一个语言无关的解决方案，但它在开发周期中引入了不可忽视的复杂性，并妨碍了垃圾回收特性。对于数据库应用程序的开发人员，CORBA 为异构环境提供了最好的灵活性。服务器可以用 C 或 C++开发，而客户可以是一个 Java applet。

## 十、Java Beans 组件模型

Java Beans 是一个集成技术和组件框架，它允许可重用的组件对象（称为 Beans）相互通信，以及与框架通信。

Java Beans 是一个独立的、可重用的软件组件，可以用构造工具来直观地操纵它。Bean 可以象 AWT 组件那样是可视的对象，也可以象队列和栈那样是不可视的对象。一个构造器/综合工具操作 Beans 来创建 Applet 和应用程序。组件模型由 Java Beans 1.00－A 规范指定，这个规范定义了五个主要的服务：

- 内省

这个过程给出了 Java Bean 组件支持的特性、方法和事件。这个服务在运行时刻且正在使用可视化构造工具创建 Java Bean 时被使用。

- 通信

事件处理机制创建一个事件,这个事件将作为到其他组件的消息。

- 保持

保持是存储组件状态的一种途径。支持保持的最简单的方法是利用 Java 对象串行化，而使用 bean 来真正地保存 Bean 的状态取决于浏览器或应用程序。

- 属性

属性是 Bean 的特性，它通过名称来引用。这些特性通常由专为这一目的而创建的 Bean 的方法来读写。某些特性除了会影响特性所属的 Bean 之外，还可能影响邻近的组件。

**自定义**

Bean 的一个主要特征就是可重用性。Bean 框架提供了将已存在的 Bean 自定义为新 Bean 的方法。

**Bean 的体系结构**

Bean 由用户可见的接口来表示。如果环境想和这个 Bean 交互，它必须连接到这个接口。Bean 由三个通用用途的接口组成：事件、属性和方法。由于 Bean 依赖于它们的状态，所以它们必须保持一致。

**事件**

Bean 事件是在 Bean 之间或 Bean 和容器之间传送异步消息的机制。Bean 用事件通知另一个 Bean 做某个动作或者告诉某个 Bean 状态已经发生改变。事件允许你的 Bean 在发生某些感兴趣的事时进行通信；为了做到这点，它们使用 JDK1.1 的事件模型。这个通信包含三部分：事件对象，事件监听者和事件源。

Java Beans 主要使用继承了 EventListener 的事件监听者接口来通信。

Bean 的开发者可以使用它们自己的事件类型和事件监听者接口，并可以使他们的 Bean 作为实现 addXXXListener（EventObject e）和 removeXXXListener（EventObject e）方法的源，其中 XXX 是事件类型的名称。然后开发者可以通过实现 XXXListener 接口使其他 Bean 作为事件目标。调用 sourceBean.addXXXListner(targetBean)，将 sourceBean 和 targetBean 结合在一起。

**属性**

属性定义了 Bean 的特征。在运行时刻可以通过它们的 get 和 set 方法来改变它们。

属性可以用来在 Bean 之间进行两路同步通信。Bean 还可以通过特殊事件通信支持异步特性改变。

**方法**

方法是一种用于和 Bean 交互的操作。由合适的事件源方法来调用 Bean，可以使它们接收到事件通知。某些方法是特殊的，用来处理属性和事件。这些方法必须遵循 Bean 规范中的命名规定。其他方法可能和事件或属性没有关系。Bean 的所有公共方法对于 Bean 框架是可存取的，且能用来将一个 Bean 连接到其他 Bean。

**Bean 内省**

Java Bean 内省过程显示了 Bean 的属性、方法和事件。如果有设置或获取特性类型的方法，Bean 类就被认为含有属性。

Java BeanAPI 提供的 BeanInfo 接口，使 Bean 的设计者能显示 Bean 的属性、事件和方法以及任何全局信息。BeanInfo 接口提供了一系列方法来存取 Bean 的信息，但一个 Bean 的开发者也可以包含一个 BeanInfo 类用来定义 Bean 信息的私有描述文件。缺省地，在 Bean 上运行内省时创建 BeanInfo 对象。

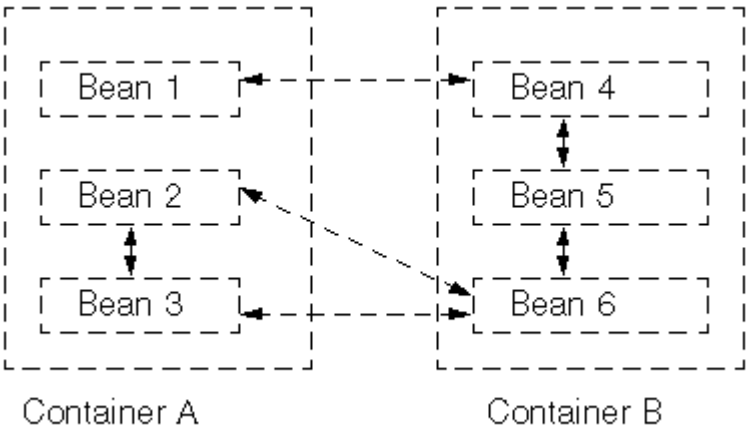



图 A-1 一个 Bean 交互的示例

**一个 Bean 交互的示例**

在图 A－1 中，容器 A 和容器 B 包含 6 个 Bean。一个 Bean 可以和位于同一个容器中的 Bean 以及位于其他容器中的 Bean 交互。在这个示例中，Bean1 和 Bean4 交互。它不与位于同一个容器中的 Bean2 和 Bean3 交互，这表明一个 Bean 可以和任何其他 Bean 通信，而不限于是否在同一个容器中。然而，Bean4 和位于同一个容器中的 Bean5 通信。源 Bean1 向目标 Bean4 发送一个事件，这使得它在事件监听者上监听消息。所有其他的容器内与容器间的 Bean 交互可以用类似的方式进行解释。

**Bean 开发箱（BDK）**

BDK 是 JavaSoft 开发的 Java 应用程序，它允许 Java 开发者使用 Bean 模型创建可重用的组件。它是一个完整的系统，包括所有示例的源代码和文档。BDK 还带有一个示例性的 Bean 构造器和称为 BeanBox 的客户化应用程序。BeanBox 是一个测试者容器，可用来做以下事情：

- 改变 Bean 的大小和移动 Bean。
- 用属性表改变 Bean。
- 用自定义应用程序来自定义 Beans
- 将 Beans 连接在一起
- 将 Beans 放到一个复合窗口中
- 通过串行化保存 Beans
- 恢复 Beans

它带有 12 个示例 Bean，覆盖 Java Beans API 的所有方面。

# 十一、JAR 文件

JAR（库）是一个独立于平台的文件格式，它允许将许多文件集成为一个文件。多个 Java applet 和它们所需的组件（.class 文件、图像和声音）可以捆绑到一个 JAR 文件中，然后在单个超文本传输协议（HTTP）的传输中下载到浏览器，这大大提高了下载速度。JAR 格式还支持压缩，减少了文件大小，从而进一步加快了下载速度。此外，Applet 的作者可以在 JAR 文件中采取数字签名来认证文件的起源。这对于已存在的 Applet，完全可以保证向后兼容，而且可以被执行。

改变你的 HTML 页面中的 applet 标记来包含一个 JAR 文件是很容易的。服务器上的 JAR 由 ARCHIVE 参数标识，它包含了 JAR 文件相对于 HTML 页面的目录位置。例如：

```
<applet code=Animator.class
archive="jars/animator.jar"
width=460 height=160>
```

```
<param name=foo value="bar">
</applet>
```

## 十二、检查一下你的进度

**在进入下一个模块的学习之前，请确认你能够：**

- **理解分布式计算的二层体系结构和三层体系结构**
- **理解 Java 编程语言作为数据库应用程序的前台程序的作用**
- **使用 JDBC API**
- **理解使用对象代理的数据交换方法**
- **解释 JavaBeans 的组件模型**

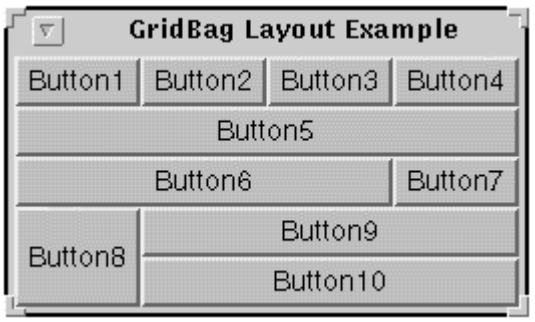
# 附录 B GridBagLayout 管理器

## 一、目标

**本附录详细探讨 GridBagLayout 管理器。**

## 二、GridBagLayout 管理器

GridBagLayout **使得在一个网格布局中能够对独立单元的大小和位置进行更精细的控制。每个网格区域或组件与一个特别的** GridBagConstraints **实例相关，该实例指定了如何在该显示区域中设计组件的布局。**GridBagLayout **涉及比前面所描述的布局更多的工作，但是你可以获得你想要的内容。**




GridBagLayout **是标准** Java **平台上，目前为止所提供的最灵活的布局管理器。它对于你的整体应用程序是一个非常有用的补充。**

**GridBagLayout 的功能**

**与** GridLayout **类似，**GridBagLayout **将组件在一个矩形的‘ 单元 ’网格中对齐。**GridLayout **和** GridBagLayout **的区别在于，**GridBagLayout **使用了一种称为** GridBagConstraints **的对象。**

**通过在** GridBagConstraints **对象中设置值，组件可水平或垂直地对齐（带有或不带有小插图和填充），被告知扩展以填充给定的区域，以及被指示如何对窗口大小的改变采取相应的行动。**

**依照如下所示，可初始化并利用一个** GridBagLayout **以及与之相关的** GridBagConstraints **对象。**

```
private GridBagLayout guiLayout;
```

```
private GridBagConstraints guiConstraints;
...
guiLayout = new GridBagLayout();
setLayout(guiLayout);
guiConstraints = new GridBagConstraints();
guiConstraints.anchor = GridBagConstraints.CENTER;
```

**使用 GridBagConstraints**

**前一页代码片断中的前** 5 **个语句对于你并不陌生，因为在** Java **编程语言学习的前些日子里你已实例化过对象了。你也已多次使用** setLayout()**建立过布局管理器以控制你的组件的布局。**

**这一代码片断中的新内容是最后一个语句：**

```
guiConstraints.anchor = GridBagConstraints.CENTER;
```

**与** GridBagLayout **布局管理器相关的** GridBagConstraints **对象，包含着可被设置以控制组件布局的实例变量。上面的** Java **代码语句设置了实例变量热区。这一实例变量用于指导** GridBagLayout**，在它小于它的单元时如何放置一个组件。有效值为：**

```
GridBagConstraints.CENTER (default)
GridBagConstraints.NORTH
GridBagConstraints.NORTHEAST
GridBagConstraints.EAST
GridBagConstraints.SOUTHEAST
GridBagConstraints.SOUTH
GridBagConstraints.SOUTHWEST
GridBagConstraints.WEST
GridBagConstraints.NORTHWEST
```

## 三、GridBagConstraints 实例变量

**关于** GridBagLayout **的** API **页解释了用于控制组件布局的实例变量。它们是：**

*** gridx 和 gridy**

**这些变量指定了位于组件显示区域左上方的单元的特征，其中，最左上方的单元具有地址** gridx=0**，**gridy=0**。你可用** GridBagConstraints.RELATIVE**（缺省值）来指定某组件的相对位置，即，它相对于在它之前被加入到容器中组件的右方（**gridx**）或下方（**gridy**）的位置。**

*** gridwidth 和 gridheight**

**这些变量指定了在组件的显示区域中，行（**gridwidth**）或列（**gridheight**）中的单元数目。缺省值为** 1**。你可用** GridBagConstraints.REMAINDER **来指定某组件在其行（**gridwidth**）或列（**gridheight**）中为最后一个。用** GridBagConstraints.RELATIVE **可指定某组件在其行（**gridwidth**）或列（**gridheight**）中与最后一个相邻。**

*** fill**

fill **在某组件的显示区域大于它所要求的大小时被使用；**fill **决定了是否（和怎样）改变组件的大小。有效值为** GridBagConstraints.NONE**（缺省），**GridBagConstraints.HORIZONTAL**（使该组件足够大，以填充其显示区域的水平方向，但不改变其高度），**GridBagConstraints.VERTICAL**（使该组件足够大，以填充其显示区域的垂直方向，但不改变其宽度），**GridBagConstraints.BOTH**（使该组件填充其整个显示区域）。**

*** ipadx 和 ipady**

这些变量指定了内部填充；即，在该组件的最小尺寸基础上还需增加多少。组件的宽度必须至少为其最小宽度加 ipadx*2 个象素（因为填充作用于组件的两边）。同样地，组件的高度必须至少为其最小高度加 ipady*2 个象素。

*** insets**

insets 指定了组件的外部填充；即，组件与其显示区域边界之间的最小空间大小。

*** anchor**

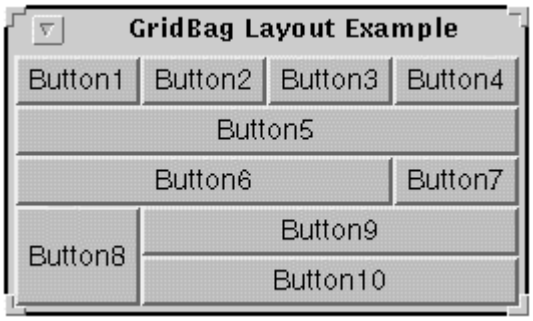
本变量在组件小于其显示区域时使用；anchor 决定了把组件放置在该区域中的位置。有效值为 GridBagConstraints.CENTER（缺省），.NORTH，.NORTHEAST，.EAST，.SOUTHEAST，.SOUTH，.SOUTHWEST，.WEST，和.NORTHWEST。

*** weightx 和 weighty**

这些变量用来决定如何分布空白和改变它的大小。除非你为一行（weightx）和一列（weighty）中至少一个组件指定了重量，否则，所有组件都会集中在容器的中央。这是因为，当重量为 0（缺省值）时，GridBagLayout 在其单元网格间以及容器边界加入一些额外的空白。

## 四、GridBagConstraints 范例

下面的讨论将基于产生如下 GUI 的代码：




在接下来的描述之后，我们将给出本例的完整代码。

前面的 GUI 使用了几个 GridBagConstraints 的实例变量：

```
public void init() {
GridBagLayout gridbag = new GridBagLayout();
GridBagConstraints c = new GridBagConstraints();
setFont(new Font("Helvetica", Font.PLAIN, 14));
setLayout(gridbag);
c.fill = GridBagConstraints.BOTH;
c.weightx = 1.0;
makebutton("Button1", gridbag, c);
makebutton("Button2", gridbag, c);
makebutton("Button3", gridbag, c);
```

上面的第 2 行中，创建了 GridBagLayout 的一个实例，称之为 gridbag，对它的使用贯穿于本例后面的代码中。

第 3 行创建了一个称为 c 的 GridBagConstraints 的实例。你每次要指定如何设计一个组件的某些方面时，都要设置 c 内部的变量。

第 6 行把 GridBagLayout 建立为布局管理器。

第 8 行指示 GridBagLayout，在一个组件未充满它的整个容器的任何时候，填充容器在

水平及垂直方向的空白。由于 c.fill 在范例代码的剩余部分一直未被改变，因此每个组件都保持在两个方向上填充的模式。

第 9 行将 x 重量赋值为 1.0。这影响了如何在任意给定的行中填充，并决定了如何改变位于该行中组件的大小。

第 10 到 12 行调用 makebutton()方法以创建按钮。

在 GridBagLayout 的控制下，每次加入一个组件都按照如下的基本序列进行：

* 创建组件。
* 设置相应的 GridBagConstraints。
* 用 add()添加该组件。

如果把这些步骤收集到一个方法 makebutton()中，则代码会具有更好的可读性，代码冗余度也会大大降低。

```
protected void makebutton(String name,
GridBagLayout gridbag,
GridBagConstraints c) {
Button button = new Button(name);
gridbag.setConstraints(button, c);
add(button);
}
```

第 4 行创建了一个组件。

在前一页的代码中，你用 GridBagConstraints 设置了 c 中的一些变量。现在 GridBagLayout 布局管理器必须得知关于你想要添加的组件的限制。这在第 5 行中使用 GridBagLayout 的 setConstraints()方法来完成。

第 6 行加入了该组件。GridBagLayout 管理器对它的布局进行控制。

范例代码的剩余部分为每个新按钮设置属性，并调用 makebutton 创建这些按钮。

```
c.gridwidth = GridBagConstraints.REMAINDER; //end row
makebutton("Button4", gridbag, c);
c.weightx = 0.0; //reset to the default
makebutton("Button5", gridbag, c); //another row
c.gridwidth = GridBagConstraints.RELATIVE; //next-to-last in row
makebutton("Button6", gridbag, c);
c.gridwidth = GridBagConstraints.REMAINDER; //end row
makebutton("Button7", gridbag, c);
c.gridwidth = 1; //reset to the default
c.gridheight = 2;
c.weighty = 1.0;
makebutton("Button8", gridbag, c);
```

第 1 和第 2 行创建 Button4。第 1 行指示 GridBagLayout 填充本按钮所在行的剩余空间。它还有一个额外的作用，即，使得 Button5 出现在下一行。

第 4 和第 5 行创建 Button5。由于 c.gridwidth 仍保持着在第 1 行中被设置的 GridBagConstraints.REMAINDER，所以该按钮独占一行。同时，第 4 行把 x weight 重置为缺省值。

第 7 和第 8 行创建 Button6。第 7 行指示 GridBagLayout 使本按钮成为本行中紧邻最后按钮的一个。

第 10 和第 11 行创建 Button7。第 10 行指示 GridBagLayout 填充本按钮所在行中的剩余空间。因而，Button8 显示在下一行上。

第 13 到 16 行创建 Button8。第 14 行使得本按钮有两个单位的高度，而第 15 行则指示 GridBagLayout 重置这一按钮垂直向上的大小。

以下为 GridBagLayout 示例的完整代码：

```
import java.awt.*;
import java.util.*;
import java.applet.Applet;
public class GridBagEx1 extends Applet {
protected void makebutton(String name,
GridBagLayout gridbag,
GridBagConstraints c) {
Button button = new Button(name);
gridbag.setConstraints(button, c);
add(button);
}

public void init() {
GridBagLayout gridbag = new GridBagLayout();
GridBagConstraints c = new GridBagConstraints();
setFont(new Font("Helvetica", Font.PLAIN, 14));
setLayout(gridbag);
c.fill = GridBagConstraints.BOTH;
c.weightx = 1.0;
makebutton("Button1", gridbag, c);
makebutton("Button2", gridbag, c);
makebutton("Button3", gridbag, c);
c.gridwidth = GridBagConstraints.REMAINDER; //end row
makebutton("Button4", gridbag, c);
c.weightx = 0.0; //reset to the default
makebutton("Button5", gridbag, c); //another row
c.gridwidth = GridBagConstraints.RELATIVE; //next-to-last in row
makebutton("Button6", gridbag, c);
c.gridwidth = GridBagConstraints.REMAINDER; //end row
makebutton("Button7", gridbag, c);

c.gridwidth = 1; //reset to the default
c.gridheight = 2;
c.weighty = 1.0;
makebutton("Button8", gridbag, c);
c.weighty = 0.0; //reset to the default
c.gridwidth = GridBagConstraints.REMAINDER; //end row
c.gridheight = 1; //reset to the default
makebutton("Button9", gridbag, c);
makebutton("Button10", gridbag, c);
resize(300, 100);
}
public static void main(String args[]) {
Frame f = new Frame("GridBag Layout Example");
GridBagEx1 ex1 = new GridBagEx1();
ex1.init();
f.add("Center", ex1);
f.pack();
f.resize(f.preferredSize());
```

```
f.show();
}
}
```

# 附录 C Java 本地接口

## 一、本地方法

现在，你已经了解了 Java 编程语言中的许多特性和基本功能。你可能希望用 Java 编程语言来编写应用程序，但 Java 编程语言却不能独立支持你所要执行的任务。在这种情况下，你可以使用本地方法，并将它与处理你的特殊需求的 C 语言相链接。

这一附加功能将带来一定的代价——你的应用程序将不再易于移植。其他共享你体系结构的机器必须有一个已编译过的 C 程序的本地拷贝。具有不同体系结构的其他机器会要求把你的 C 语言程序移植到它们自己的体系结构中，使其能由它的一种本地编译器进行编译。

这一过程对结构复杂的程序来说是困难的，而对依赖于下层操作系统中独有特性的程序来说则是不可能的。但是，如果你所面对的是一个单一类型体系结构的环境，或者只想加入具有良好适应性的特性，本地方法无疑可成为满足你要求的最佳选择。

## 二、本地的 HelloWorld

第一个任务将是从 Java 程序中调用一个本地方法，因此首先要为 HelloWorld 程序创建一个本地方法。

以下为将本地方法集成到你的 Java 程序中的四个基本步骤的概述：

* 用适当的本地方法声明定义一个 Java 类。
* 为你的 C 模块的使用创建一个头文件，用 javah 工具来完成。
* 编写包含该本地方法的 C 模块。
* 将这段 C 代码编译成一个动态装载库。

**定义本地方法**

与其他方法相似，你必须声明要用的所有本地方法，并且必须将它们定义在一个类中。

你可将你的本地 HelloWorld 方法定义为如下形式：

```
public native void nativeHelloWorld();
```

这与你所写的其他 public void 方法比较有两个变化：

* 关键字 native 被用作方法修饰符。
* 方法的主体部分（实际实现部分）在这里没有定义，而是代之以一个分号（;）。

你必须把本地方法的声明放到类的定义中。含有这个本地方法的类还包含一个静态代码块，它负责装载带有该方法实现的动态库。这里是为简单的 nativeHelloWorld()方法而写的类定义的一个例子：

```
class NativeHello {
  1.public native void nativeHelloWorld();
  2.static {
  3.System.loadLibrary("hello1");
  4.}
  5.}
```

Java 运行环境在类被装载时执行定义的 static 代码块。在上例中，当类 NativeHello 被装载时，库 hello1 被装入。

**调用本地方法**

一旦你已将本地方法放入到一个类中，就可以为该类创建对象以存取这个本地方法，这与处理普通的类方法相似。这里，我们举例说明程序是如何创建一个新的 NativeHello 对象并调用你的 nativeHelloWorld 方法的：

```
1.class UseNative {
2.public static void main (String args[]) {
3.NativeHello nh = new NativeHello();
4.nh.nativeHelloWorld();
5.}
6.}
```

用 javac 来编译.java 文件。.class 文件在创建头文件时会被使用。

**javah 工具**

你可用 javah 工具来创建基于 NativeHello.class 文件的 C 头文件。对 javah 的调用方式如下：

```
% javah -jni NativeHello
```

所产生的文件，NativeHello.h，为你提供了编写 C 程序所需的消息。这里给出了对于本例 javah 所产生的文件：

```
/* DO NOT EDIT THIS FILE - it is machine generated */
1.#include <jni.h>
2./* Header for class NativeHello */
3.
4.#ifndef _Included_NativeHello
5.#define _Included_NativeHello
6.#ifdef __cplusplus
7.extern "C" {
8.#endif
9./*
10.* Class: NativeHello
11.* Method: nativeHelloWorld
12.* Signature: ()V
13.*/
14.JNIEXPORT void JNICALL Java_NativeHello_nativeHelloWorld
15.(JNIEnv *, jobject);
16.
17.#ifdef __cplusplus
18.}
19.#endif
20.#endif
21.
```

其中的黑体字符部分给出了将要实现的本地方法的签名。

**为本地方法编写 C 函数代码**

到此为止，C 程序是唯一缺少的代码部分。你所编写的 C 代码必须包含上面的头文件，以及在＄JAVA_HOME/include 目录中由 JDK 所提供的 jni.h。（＄JAVA_HOME 指 JDK 的“根”目录。）当然，也要包含你的函数所必需的其他头文件。

对每一个在头文件中声明的函数，你都要提供函数体。对本例来说，称为

MyNativeHello.c 的 C 文件如下：

```
 #include <jni.h>
1.#include "NativeHello.h"
2.#include <stdio.h>
 3.
4.void Java_NativeHello_nativeHelloWorld
5.(JNIEnv *env, jobject obj) {
6.printf ("Hello from C");
 7.}
 8.
```

# 三、将它合并

现在你已准备好了所有片断，必须告诉系统如何将它们集合起来。首先，编译你的 C 程序。也许你必须指定 include 文件的位置。

下例使用了 Solaris 操作系统中的 C 编译器：

```
% cc -I$JAVA_HOME/include -I$JAVA_HOME/include/solaris -G MyNativeHello.c -o
libhello1.so
```

为了编译代码，你必须存取的两个包含目录是 $JAVA_HOME/include 和 $JAVA_HOME/include/solaris。你可以在命令行上指定它们，或者采用修改 INCLUDE 环境变量的方式。

```
C:\> cl MyNativeHello.c -Fehello1.dll -MD -LD javai.lib
```

一旦创建完库文件，你就可运行你的本地方法测试文件了：

```
% java UseNative
Hello Native World!
```

或

```
C:\> java UseNative
Hello Native World!
```

如果你被给出 java.lang.UnsatisfiedLinkError 信息，则需要更新系统的 LD_LIBRARY_PATH 变量来包含当前目录（使得 JVM 可找到 libhello1.so）。

# 四、向本地方法传递信息

在前面的例子中，本地方法既不处理从定义的类中所存取的信息，也不接受任何参数。然而，以下的任务在编程过程中却经常遇到：

将一个 Java 原语作为参数传递

在 Java 程序中，可以象其他方法一样为本地方法提供参数。假设在一个称为 NativeHello2.java 的文件中存在如下的代码声明，该声明定义了打印 count 次的本地方法：

```
public native void nativeHelloWorld2(int count);
```

这一声明将在头文件 NativeHello2.h 中产生如下的入口：

```
JNIEXPORT void JNICALL Java_NativeHello2_nativeHelloWorld2
   1.(JNIEnv *, jobject, jint);
```

现在重写你的 C 方法，使它循环所给定的次数（它作为该方法的第三个参数出现）。

```
 #include <jni.h>
1.#include "NativeHello2.h"
2.#include <stdio.h>
 3.
4.JNIEXPORT void JNICALL Java_NativeHello2_nativeHelloWorld2
5.(JNIEnv *env, jobject obj, jint countMax) {
6.int count;
```

```
7.for (count = 0; count < countMax; count++) {
8.printf ("Hello from C, count = %d\n",count);
 9.}
10.}
```

**将一个 Java 基本类型作为对象数据成员存取**

**在一个本地方法中，最常见的需求是对类数据成员的存取。**jni.h **文件包含几个接口函数，以便使用本地代码模块内部的对象。**

**例如：考虑编写一个具有两个** int **变量，其中之一为** static **的类，用一个本地方法来存取和修改这些变量：**

```
class NativeHello4 {
 1.static int statInt = 2;
 2.int instInt = 4;
 3.public native int nativeHelloWorld4();
 4.static {
 5.System.loadLibrary("hello4");
 6.}
 7.}
```

**在你的** C **语言内部，你可用**<jni.h>**中的函数来存取这些变量：**

```
#include <jni.h>
 1.#include "NativeHello4.h"
 2.#include <stdio.h>
 3.
4./* The names of the Java object fields to be accessed. */
 5.#define STAT_FIELD_NAME "statInt"
 6.#define INST_FIELD_NAME "instInt"
 7.
 8./* This method displays the statInt and instInt fields and
 9.returns the product of the two. */
10.jint Java_NativeHello4_nativeHelloWorld4
11.(JNIEnv *env, jobject obj) {
 12.
13./* Class object. Used to find all fields and access
 14.static ones.
 15.jclass class = (*env)->GetObjectClass(env,obj);
 16.
17.jfieldID fid; /* A field reference. */
18.jint staticInt; /* A C copy of the static int. */
19.jint instanceInt; /* A C copy of the int. */
 20.
```

**将一个 Java 基本类型作为对象数据成员存取（续）**

```
 1.
2./* Get reference to the static field. The "class"
3.argument connects the field to a class. The third
4.argument is the field's name, and the last argument is
5.the field's type. See the union jvalue entry in jni.h,
6.then capitalize for the proper primitive value. */
 7.fid = (*env)->GetStaticFieldID(env, class,
 8.STAT_FIELD_NAME, "I");
 9.if (fid == 0)
```

```
10.return;
 11.
12./* Get that field's data. */
13.staticInt = (*env)->GetStaticIntField(env, class, fid);
 14.
15./* Process it, change it... */
 16.printf
17.("In C, doubling original %s value of %d to %d\n",
18.STAT_FIELD_NAME, staticInt, staticInt*2);
19.staticInt *= 2;
 20.
21./* ... and store it back into the class object. */
 22.(*env)->SetStaticIntField(env, class, fid, staticInt);
 23.
 24.
25./* Now for the nonstatic int, part of the current
26.object. Get the field reference as before... */
 27.fid = (*env)->GetFieldID
28.(env, class, INST_FIELD_NAME, "I");
29.if (fid == 0)
 30.return;
 31.
32./* Get the field. Refer to the object, not the class. */
33.instanceInt = (*env)->GetIntField(env, obj, fid);
 34.
35./* Process it, change it... */
 36.printf
37.("In C, tripling original %s value of %d to %d\n",
38.INST_FIELD_NAME, instanceInt, instanceInt*3);
39.instanceInt *= 3;
 40.
```

**存取字符串**

**可以回想一下，**Java **编程语言中的字符串由** 16-bit **的** Unicode **字符构成。**

**但是，**C **的字符串则由** 8-bit **的美国标准信息交换编码（**ASCII**）字符构成。由于字符串是** Java **代码与本地代码间传递的最常见的对象，所以，在** jni.h **中定义了几个函数来帮助实现对字符串操作的简单性。在** Java **编程语言中，字符串的** C **数据类型是** jstring**。**

**假设在** Java **程序中声明了如下打印一字符串的本地方法：**

```
   #include <jni.h>
1.#include "NativeHello3.h"
2.#include <stdio.h>
 3.
4.void Java_NativeHello3_nativeHelloWorld3 (
5.JNIEnv *env, jobject obj, jstring javaString) {
 6.
7.const char * CString;
 8.
9./* Convert java string to C string. */
10.CString = (*env)->GetStringUTFChars(env, javaString, 0);
 11.
```

```
12.printf ("In C, string is %s\n", CString);
 13.
14./* Tell VM to release mem for CString as done w/ it. */
 15.(*env)->ReleaseStringUTFChars(env, javaString, CString);
 16.}
```

**下面的** C **函数为其实现了本地代码。请注意完成时调用** ReleaseStringUTFChars **来释放为** C **字符串所分配内存的重要性。**

**作为最后的一例，考虑将字符串作为一个对象数据成员来存取。假设所定义的类中带有一个** String **域和本地方法，如下所示：**

```
 class NativeHello5 {
1.String stringField = "original";
2.public native String nativeHelloWorld5();
 3.static {
  4.System.loadLibrary("hello5");
 5.}
  6.}
```

**在下面的程序中，这个类被实例化，并且调用了该本地方法：**

```
 class UseNative5 {
1.public static void main (String args[]) {
 2.String changedString;
3.NativeHello5 nh = new NativeHello5();
  4.
 5.System.out.println ("In Java, nh's string says '"
6.+ nh.stringField + "'");
  7.
 8./* Call native method to print and change string in
 9.the current object, and return a third string. */
10.changedString = nh.nativeHelloWorld5();
 11.
12.System.out.println ("In Java, nh's string says '"
13.+ nh.stringField + "'");
 14.
15.System.out.println ("Native method returned '" +
16.changedString + "'");
 17.
 18.}
 19.}
```

**下面的本地代码将字符串从这个对象中抽取出来，并打印；然后改变并存入新的版本；最后，在函数的返回值中给出另一个新的字符串：**

```
 #include <jni.h>
1.#include "NativeHello5.h"
 2.#include <stdio.h>
```

**存取字符串（续）**

```
  1.
 2.#define NEW_STRING1 "Revised"
 3.#define NEW_STRING2 "Revised again"
  4.
 5.jstring Java_NativeHello5_nativeHelloWorld5
 6.(JNIEnv *env, jobject obj) {
```

```
 7.
 8.jclass class = (*env)->GetObjectClass(env,obj);
 9.jfieldID fid;
10.jstring javaString;
11.const char *CString;
12.
13.fid = (*env)->GetFieldID
14.(env, class, "stringField", "Ljava/lang/String;");
15.if (fid == 0)
16.return;
17.
18./* Get the field reference for the java string. */
19.javaString = (*env)->GetObjectField(env, obj, fid);
20.
21./* Retrieve the C string from the object. */
22.CString = (*env)->GetStringUTFChars(env, javaString, 0);
23.
24.printf ("In C, changing string from %s to %s\n",
25.CString, NEW_STRING1);
26.
27./* Tell VM to release memory for CString as done w/it. */
28.(*env)->ReleaseStringUTFChars(env, javaString, CString);
29.
30./* Alloc a new Java string to store back in the obj. */
31.javaString = (*env)->NewStringUTF(env, NEW_STRING1);
32.
33./* Store it back. */
34.(*env)->SetObjectField(env, obj, fid, javaString);
35.
36./* Allocate one more new Java string to return. */
37.javaString = (*env)->NewStringUTF(env, NEW_STRING2);
38.
39.return (javaString);
40.}
41.
```

# 五、小结

**集成本地方法的精确过程是：**

1. **创建一个包含本地方法声明和装载动态库的静态代码的程序：**

   vi Native.java
2. **创建一个包含对该本地方法调用的程序：**

   vi UseNative.java
3. **编译**.java **文件：**

   javac NativeHello.java UseNative.java
4. **创建** C **头文件：**

   javah –jni NativeHello
5. **创建实现你的本地方法的** C **程序：**

   vi MyNativeHello.c

6. 编译动态库：

   cc –I$JAVA_HOME/include
   I$JAVA_HOME/include/solaris –G MyNativeHello.c –o
   libhello.so

7. 设置 LD_LIBRARY_PATH 变量：

   setenv LD_LIBRARY_PATH .:$LD_LIBRARY_PATH

8. 运行程序：

   java UseNative

关于其他 JNI 功能的信息，请参见 JavaSoft Web 站点上的 JNI tutorial。它可从你本地装载的 master Java API 索引通过超级链接存取。

# 附录 D 事件 1.0.x 到事件 1.1 的转换

本附录给出了在 JDK1.0.x 和 JDK1.1 下有关事件处理的概述，并提供了一张 1.0.x 事件和相应方法与其 1.1 的对应部分的映射表。

**参考文献**

其他资源 － 本附录的内容可由 Web 页面“如何将程序转换到 1.1 AWT API”获得[在线]。可从下列 URL 得到：

http://www.javasoft.com/products/JDK/1.1/docs/guide/awt/HowToUpgrade.html。

## 一、事件处理

**JDK1.1 之前的事件处理**

在 JDK1.1 之前，由组件处理事件方法（以及它所调用的方法，如 action 方法）是事件处理的中心。只有组件对象能处理事件，而且处理一事件的组件必须满足如下条件：或者该事件发生于其中，或者为在组件容器层次中位于其上的组件。

**JDK1.1 中的事件处理**

在 JDK1.1 中，事件处理不再被限制于组件容器层次中的对象，handleEvent 方法也不再是事件处理的中心。取而代之的是，任何类型的对象都可注册为事件监听者。事件监听者只能接收与它们所注册的兴趣范围相关的事件类型。你不必创建一个组件子类来处理事件。

当升级到 JDK1.1 版本时，转换事件处理代码的最简单的方法是，把它留在同一个类中，并使其成为该事件类型的监听者。

另一个可行的方案是，在一个或多个非组件监听者（适配器或过滤器）中使事件处理代码集中化。这种方法允许你把程序的 GUI 与实现细节分离。它要求你修改已有的代码，以使监听者能获得它们从组件所请求的任何信息。如果你试图保持程序体系结构的清晰，这种方法不失为一个好的选择。

把大多数采用 AWT 的 1.0 程序转换为 1.1 API 需改动最多的部分，是转换事件处理代码。一旦你计划好了程序要处理哪些事件，以及这些事件由哪些组件产生，整个过程就能很顺利地进行下去。在源文件中搜索“Event”可帮助你找到这些事件处理代码。

---

注 － 在寻找这些代码时，你应注意某些类的存在是否只是为了事件处理；你可将这样的类清除。

表 D－1 可用于帮助转换过程，它给出了 1.0 的事件和方法到 1.1 相应部分的映射。

* 第一栏列出了每一个 1.0 的事件类型，以及（如果有）为该事件所指定的方法名称。未列出方法的事件，总是由 handleEvent 方法来处理的。
* 第二栏为能够产生该事件类型的 1.0 组件。
* 第三栏为监听者接口，它可帮助你处理所列出的 1.1 的等价事件。
* 第四栏列出了每个监听者接口中的方法。

表 D－1 事件转换表

| 1.0. x | | 1.1 | |
|---|---|---|---|
| Event/Method | Generated by | Interface | Methods |
| ACTION_EVENT/action | Button<br>List<br>MenuItem<br>TextField | ActionListener | actionPerformed(ActionEvent) |
| | Checkbox<br>CheckboxMenuItem<br>Choice | ItemListener | itemStateChanged(ItemEvent) |

表 D－1 事件转换表（续）

| | | | |
|---|---|---|---|
| WINDOW_DESTROY<br>WINDOW_EXPOSE<br>WINDOW_ICONIFY<br>WINDOW_DEICONIFY | Dialog<br>Frame | WindowListener | windowClosing(WindowEvent)<br>windowOpened(WindowEvent)<br>windowIconified(WindowEvent)<br>windowDeiconified(WindowEvent)<br>windowClosed(WindowEvent) * 1<br>windowActivated(WindowEvent) *<br>windowDeactivated(WindowEvent) * |

| | | | |
|---|---|---|---|
| WINDOW_MOVED | Dialog<br>Frame | ComponentListener | componentMoved(ComponentEvent)<br>ComponentHidden(ComponentEvent)*<br>componentResized(ComponentEvent)*<br>componentShown(ComponentEvent)* |
| SCROLL_LINE_UP<br>SCROLL_LINE_DOWN<br>SCROLL_PAGE_UP<br>SCROLL_PAGE_DOWN<br>SCROLL_ABSOLUTE<br>SCROLL_BEGIN<br>SCROLL_END | Scrollbar | AdjustmentListener (Or use the newScrollPane class) | adjustmentValueChanged(AdjustmentEvent) |

**表 D－1　事件转换表（续）**

| | | | |
|---|---|---|---|
| LIST_SELECT<br>LIST_DESELECT | Checkbox<br>CheckboxMenuItem<br>Choice<br>List | ItemListener | itemStateChanged(ItemEvent) |
| MOUSE_DRAG/mouseDrag<br><br>MOUSE_MOVE/mouseMove | Canvas<br>Dialog<br>Frame<br>Panel<br>Window | MouseMotionListener | mouseDragged(MouseEvent)<br>mouseMoved(MouseEvent) |

| | | | |
|---|---|---|---|
| MOUSE_DOWN/mouseDown<br>MOUSE_UP/mouseUp<br>MOUSE_ENTER/mouseEnter<br>MOUSE_EXIT/mouseExit | Canvas<br>Dialog<br>Frame<br>Panel<br>Window | MouseListener | mousePressed(MouseEvent)<br>mouseReleased(MouseEvent)<br>moseEntered(MouseEvent)<br>mouseExited(MouseEvent)<br>mouseClicked(MouseEvent)* |
| KEY_PRESS/keyDown<br>KEY_RELEASE/keyUp<br>KEY_ACTION/keyDown<br>KEY_ACTION_RELEASE/keyUp | Component | KeyListener | keyPressed(KeyEvent)<br>keyReleased(KeyEvent)<br>keyTyped(KeyEvent) * |
| GOT_FOCUS/gotFocus<br>LOST_FOCUS/lostFocus | Component | FocusListener | focusGained(FocusEvent)<br>focusLost(FocusEvent) |
| No 1.0 equivalent | | ContainerListener | componentAdded(ContainerEvent)<br>componentRemoved(ContainerEvent) |
| No 1.0 equivalent | | TextListener | textValueChanged(TextEvent) |

a.no 1.0 equivalent

# 二、使组件成为监听者

**按照以下的通用步骤，可将一个** 1.0 **的组件转换为** 1.1 **监听者：**

1. **改变源文件，使它导入** java.awt.event **包：**

   import java.awt.event.*

2. **利用表** D－1 **确定每种事件类型是由哪些组件产生的。**

**例如，如果你正在转换** action **方法中的事件代码，则应该查找** Button，List，MenuItem，TextField，Checkbox，CheckboxMenuItem，**和** Choice **对象。**

3. **改变该类的声明，以便按照表** D－1 **的指示实现适当的监听者接口。**

**例如：如果你正试图处理由** Button **产生的** action **事件，表** D－1 **告诉你必须实现** ActionListener **接口。**

```
public class MyClass extends SomeComponent
implements ActionListener {
```

4. **决定产生该事件的组件在哪里创建。在创建每个组件的代码之后，应紧接着注册** this **为适当的监听者类型。例如：**

```
newComponentObject .addActionListener( this );
```

5. **在你的类必须实现的监听者接口中创建所有方法的空实现。拷贝事件处理代码**

到适当的方法中。

例如，ActionListener 只有一个方法——actionPerformed。创建新的方法和拷贝事件处理代码到该方法中的一种较简单的方法是，把 action 方法的签名由

public boolean action(Event event, Object arg) {

改变为

public void actionPerformed(ActionEvent event) {

6. 依照下面的方法修改事件处理代码：
   a. 删除所有的 return 语句。
   b. 把对 event.target 的引用改变为 event.getSource()。
   c. 删除关于事件来自于哪个组件的不必要测试。（现在，仅当产生它的组件有一监听者时，该事件才向前进行，因此你不必担心收到来自非所希望组件的事件。）
   d. 做一些可使程序编译简洁和正确执行所要求的其他修改。

签名未验证

数字签名人vanstone
DN：cn=vanstone, o=niming.com, ou=niming.yeah.net, c=CN
日期：2001.12.06 21:06:43 +08'00'
原因：该文档可供审阅
位置：Beijing