第十三章 创建窗口和程序片

在 Java 1.0 中，图形用户接口（GUI）库最初的设计目标是让程序员构建一个通用的 GUI，使其在所有平台上都能正常显示。

但遗憾的是，这个目标并未达到。事实上，Java 1.0 版的“抽象 Windows 工具包”（AWT）产生的是在各系统看来都同样欠佳的图形用户接口。除此之外，它还限制我们只能使用四种字体，并且不能访问操作系统中现有的高级 GUI 元素。同时，Jave1.0 版的 AWT 编程模型也不是面向对象的，极不成熟。这类情况在 Java1.1 版的 AWT 事件模型中得到了很好的改进，例如：更加清晰、面向对象的编程、遵循 Java Beans 的范例，以及一个可轻松创建可视编程环境的编程组件模型。Java1.2 为老的 Java 1.0 AWT 添加了 Java 基础类（AWT），这是一个被称为“Swing”的 GUI 的一部分。丰富的、易于使用和理解的 Java Beans 能经过拖放操作（像手工编程一样的好），创建出能使程序员满意的 GUI。软件业的“3 次修订版”规则看来对于程序设计语言也是成立的（一个产品除非经过第 3 次修订，否则不会尽如人意）。

Java 的主要设计目的之一是建立程序片，也就是建立运行在 WEB 浏览器上的小应用程序。由于它们必须是安全的，所以程序片在运行时必须加以限制。无论怎样，它们都是支持客户端编程的强有力的工具，一个重要的应用便是在 Web 上。

在一个程序片中编程会受到很多的限制，我们一般说它“在沙箱内”，这是由于 Java 运行时一直会有某个东西——即 Java 运行期安全系统——在监视着我们。Jave 1.1 为程序片提供了数字签名，所以可选出能信赖的程序片去访问主机。不过，我们也能跳出沙箱的限制写出可靠的程序。在这种情况下，我们可访问操作系统中的其他功能。在这本书中我们自始至终编写的都是可靠的程序，但它们

成为了没有图形组件的控制台程序。AWT 也能用来为可靠的程序建立 GUI 接口。

在这一章中我们将先学习使用老的 AWT 工具，我们会与许多支持和使用 AWT 的代码程序样本相遇。尽管这有一些困难，但却是必须的，因为我们必须用老的 AWT 来维护和阅读传统的 Java 代码。有时甚至需要我们编写 AWT 代码去支持不能从 Java1.0 升级的环境。在本章第二部分，我们将学习 Java 1.1 版中新的 AWT 结构并会看到它的事件模型是如此的优秀（如果能掌握的话，那么在编制新的程序时就可使用这最新的工具。最后，我们将学习新的能像类库一样加入到 Java 1.1 版中的 JFC/Swing 组件，这意味着不需要升级到 Java 1.2 便能使用这一类库。

大多数的例程都将展示程序片的建立，这并不仅仅是因为这非常的容易，更因为这是 AWT 的主要作用。另外，当用 AWT 创建一个可靠的程序时，我们将看到处理程序的不同之处，以及怎样创建能在命令行和浏览器中运行的程序。

请注意的是这不是为了描述类的所有程序的综合解释。这一章将带领我们从摘要开始。当我们查找更复杂的内容时，请确定我们的信息浏览器通过查找类和方法来解决编程中的问题（如果我们正在使用一个开发环境，信息浏览器也许是内建的；如果我们使用的是 SUN 公司的 JDK 则这时我们要使用 WEB 浏览器并在 Java 根目录下面开始）。附录 F 列出了用于深入学习库知识的其他一些参考资料。

13.1 为何要用 AWT？

对于本章要学习的“老式”AWT，它最严重的缺点就是它无论在面向对象设计方面，还是在 GUI 开发包设计方面，都有不尽如人意的表现。它使我们回到了程序设计的黑暗年代（换成其他话就是“拙劣的”、“可怕的”、“恶劣的”等等）。必须为执行每一个事件编写代码，包括在其他环境中利用“资源”即可轻松完成的一些任务。

许多象这样的问题在 Java 1.1 里都得到了缓解或排除，因为：

(1)Java 1.1 的新型 AWT 是一个更好的编程模型，并向更好的库设计迈出了可喜的一步。而 Java Beans 则是那个库的框架。

(2)“GUI 构建器”（可视编程环境）将适用于所有开发系统。在我们用图形化工具将组件置入窗体的时候，Java Beans 和新的 AWT 使 GUI 构建器能帮我们自动完成代码。其它组件技术如 ActiveX 等也将以相同的形式支持。

既然如此，为什么还要学习使用老的 AWT 呢？原因很简单，因为它的存在是个事实。就目前来说，这个事实对我们来说显得有些不利，它涉及到面向对象库设计的一个宗旨：一旦我们在库中公布一个组件，就再不能去掉它。如去掉它，就会损害别人已存在的代码。另外，当我们学习 Java 和所有使用老 AWT 的程序时，会发现有许多原来的代码使用的都是老式 AWT。

AWT 必须能与固有操作系统的 GUI 组件打交通，这意味着它需要执行一个程序片不可能做到的任务。一个不被信任的程序片在操作系统中不能作出任何直接调用，否则它会对用户的机器做出不恰当的事情。一个不被信任的程序片不能访问重要的功能。例如，“在屏幕上画一个窗口”的唯一方法是通过调用拥有特殊接口和安全检查的标准 Java 库。Sun 公司的原始模型创建的信任库将仅仅供给 Web 浏览器中的 Java 系统信任关系自动授权器使用，自动授权器将控制怎样进入到库中去。

但当我们想增加操作系统中访问新组件的功能时该怎么办？等待Sun来决定我们的扩展被合并到标准的Java库中，但这不一定会解决我们的问题。Java 1.1版中的新模型是“信任代码”或“签名代码”，因此一个特殊服务器将校验我们下载的、由规定的开发者使用的公共密钥加密系统的代码。这样我们就可知道代码从何而来，那真的是Bob的代码，还是由某人伪装成Bob的代码。这并不能阻止Bob犯错误或作某些恶意的事，但能防止Bob逃避匿名制造计算机病毒的责任。一个数字签名的程序片——“被信任的程序片”——在Java 1.1版能进入我们的机器并直接控制它，正像一些其它的应用程序从信任关系自动授权机中得到“信任”并安装在我们的机器上。

这是老AWT的所有特点。老的AWT代码将一直存在，新的Java编程者在从旧的书本中学习时将会遇到老的AWT代码。同样，老的AWT也是值得去学习的，例如在一个只有少量库的例程设计中。老的AWT所包括的范围在不考虑深度和枚举每一个程序和类，取而代之的是给了我们一个老AWT设计的概貌。

13.2 基本程序片

库通常按照它们的功能来进行组合。一些库，例如使用过的，便中断搁置起来。标准的Java库字符串和矢量类就是这样的一个例子。其他的库被特殊地设计，例如构建块去建立其它的库。库中的某些类是应用程序的框架，其目的是协助我们构建应用程序，在提供类或类集的情况下产生每个特定应用程序的基本活动状况。然后，为我们定制活动状况，必须继承应用程序类并且废弃程序的权益。应用程序框架的默认控制结构将在特定的时间调用我们废弃的程序。应用程序的框架是“分离、改变和中止事件”的好例子，因为它总是努力去尝试集中在被废弃的所有特殊程序段。

程序片利用应用程序框架来建立。我们从类中继承程序片，并且废弃特定的程序。大多数时间我们必须考虑一些不得不运行的使程序片在WEB页面上建立和使用的重要方法。这些方法是：

591页表

方法 作用

init() 程序片第一次被创建，初次运行初始化程序片时调用

start() 每当程序片进入Web浏览器中，并且允许程序片启动它的常规操作时调用（特殊的程序片被stop()关闭）；同样在init()后调用

paint() 基础类Component的一部分（继承结构中上溯三级）。作为update()的一部分调用，以便对程序片的画布进行特殊的描绘

stop() 每次程序片从Web浏览器的视线中离开时调用，使程序片能关闭代价高昂的操作；同样在调用destroy()前调用

destroy() 程序片不再需要，将它从页面中卸载时调用，以执行资源的最后清除工作

现在来看一看paint()方法。一旦Component（目前是程序片）决定自己需要更新，就会调用这个方法——可能是由于它再次回转屏幕，首次在屏幕上显示，

或者是由于其他窗口临时覆盖了你的 Web 浏览器。此时程序片会调用它的 update()方法（在基础类 Component 中定义），该方法会恢复一切该恢复的东西，而调用 paint()正是这个过程的一部分。没必要对 paint()进行过载处理，但构建一个简单的程序片无疑是方便的方法，所以我们首先从 paint()方法开始。

update()调用 paint()时，会向其传递指向 Graphics 对象的一个句柄，那个对象代表准备在上面描绘（作图）的表面。这是非常重要的，因为我们受到项目组件的外观的限制，因此不能画到区域外，这可是一件好事，否则我们就会画到线外去。在程序片的例子中，程序片的外观就是这界定的区域。

图形对象同样有一系列我们可对其进行的操作。这些操作都与在画布上作图有关。所以其中的大部分都要涉及图像、几何菜状、圆弧等等的描绘（注意如果有兴趣，可在 Java 文档中找到更详细的说明）。有些方法允许我们画出字符，而其中最常用的就是 drawString()。对于它，需指出自己想描绘的 String（字串），并指定它在程序片作图区域的起点。这个位置用像素表示，所以它在不同的机器上看起来是不同的，但至少是可以移植的。

根据这些信息即可创建一个简单的程序片：

592 页上程序

注意这个程序片不需要有一个 main()。所有内容都封装到应用程序框架中；我们将所有启动代码都放在 init()里。

必须将这个程序放到一个 Web 页中才能运行，而只能在支持 Java 的 Web 浏览器中才能看到此页。为了将一个程序片置入 Web 页，需要在那个 Web 页的代码中设置一个特殊的标记(注释①)，以指示网页装载和运行程序片。这就是 applet 标记，它在 Applet1 中的样子如下：

592 页下程序

①：本书假定读者已掌握了 HTML 的基本知识。这些知识不难学习，有许多书籍和网上资源都可以提供帮助。

其中，code 值指定了.class 文件的名字，程序片就驻留在那个文件中。width 和 height 指定这个程序片的初始尺寸（如前所述，以像素为单位）。还可将另一些东西放入 applet 标记：用于在因特网上寻找其他.class 文件的位置（codebase）、对齐和排列信息（align）、使程序片相互间能够通信的一个特殊标识符（name）以及用于提供程序片能接收的信息的参数。参数采取下述形式：

<Paramname=标识符 value ="信息">

可根据需要设置任意多个这样的参数。

在简单的程序片中，我们要做的唯一事情是按上述形式在 Web 页中设置一个程序片标记（applet），令其装载和运行程序片。

13.2.1 程序片的测试

我们可在不必建立网络连接的前提下进行一次简单的测试，方法是启动我们的 Web 浏览器，然后打开包含了程序片标签的 HTML 文件（Sun 公司的 JDK 同样包括一个称为“程序片观察器”的工具，它能挑出 html 文件的<applet>标记，

并运行这个程序片，不必显示周围的 HTML 文本——注释②）。html 文件载入后，浏览器会发现程序片的标签，并查找由 code 值指定的.class 文件。当然，它会先在 CLASSPATH（类路径）中寻找，如果在 CLASSPATH 下找不到类文件，就在 WEB 浏览器状态栏给出一个错误信息，告知不能找到.class 文件。

②；由于程序片观察器会忽略除 APPLET 标记之外的任何东西，所以可将那些标记作为注释置入 Java 源码：

// <applet code=MyApplet.class width=200 height=100></applet>

这样就可直接执行“appletviewer MyApplet.java”，不必再创建小的 HTML 文件来完成测试。

若想在 Web 站点上试验，还会碰到另一些麻烦。首先，我们必须有一个 Web 站点，这对大多数人来说都意味着位于远程地点的一家服务提供商（ISP）。然后必须通过某种途径将 HTML 文件和.class 文件从自己的站点移至 ISP 机器上正确的目录（WWW 目录）。这一般是通过采用“文件传输协议”（FTP）的程序来做成的，网上可找到许多这样的免费程序。所以我们要做的全部事情似乎就是用 FTP 协议将文件移至 ISP 的机器，然后用自己的浏览器连接网站和 HTML 文件；假如程序片正确装载和执行，就表明大功告成。但真是这样吗？

但这儿我们可能会受到愚弄。假如 Web 浏览器在服务器上找不到.class 文件，就会在你的本地机器上搜寻 CLASSPATH。所以程序片或许根本不能从服务器上正确地装载，但在你看来却是一切正常的，因为浏览器在你的机器上找到了它需要的东西。但在其他人访问时，他们的浏览器就无法找到那些类文件。所以在测试时，必须确定已从自己的机器删除了相关的.class 文件，以确保测试结果的真实。

我自己就遇到过这样的一个问题。当时是将程序片置入一个 package（包）中。上载了 HTML 文件和程序片后，由于包名的问题，程序片的服务器路径似乎陷入了混乱。但是，我的浏览器在本地类路径（CLASSPATH）中找到了它。这样一来，我就成了能够成功装载程序片的唯一一个人。后来我花了一些时间才发现原来是 package 语句有误。一般地，应该将 package 语句置于程序片的外部。

13.2.2 一个更图形化的例子

这个程序不会太令人紧张，所以让我们试着增加一些有趣的图形组件。

594 页程序

这个程序用一个方框将字符串包围起来。当然，所有数字都是“硬编码”的（指数字固定于程序内部），并以像素为基础。所以在一些机器上，框会正好将字串围住；而在另一些机器上，也许根本看不见这个框，因为不同机器安装的字体也会有所区别。

对 Graphic 类而言，可在帮助文档中找到另一些有趣的内容。大多数涉及图形的活动都是很有趣的，所有我将更多的试验留给读者自己去进行。

13.2.3 框架方法的演示

观看框架方法的实际运作是相当有趣的（这个例子只使用 init()，start()和

stop()，因为 paint()和 destroy()非常简单，很容易就能掌握）。下面的程序片将跟踪这些方法调用的次数，并用 paint()将其显示出来：

595 页程序

正常情况下，当我们过载一个方法时，需检查自己是否需要调用方法的基础类版本，这是十分重要的。例如，使用 init()时可能需要调用 super.init()。然而，Applet 文档特别指出 init()、start()和 stop()在 Applet 中没有用处，所以这里不需要调用它们。

试验这个程序片时，会发现假如最小化 WEB 浏览器，或者用另一个窗口将其覆盖，那么就不能再调用 stop()和 start()（这一行为会随着不同的实现方案变化；可考虑将 Web 浏览器的行为同程序片观察器的行为对照一下）。调用唯一发生的场合是在我们转移到一个不同的 Web 页，然后返回包含了程序片的那个页时。

13.3 制作按钮

制作一个按钮非常简单：只需要调用 Button 构建器，并指定想在按钮上出现的标签就行了（如果不想要标签，亦可使用默认构建器，但那种情况极少出现）。可参照后面的程序为按钮创建一个句柄，以便以后能够引用它。

Button 是一个组件，象它自己的小窗口一样，会在更新时得以重绘。这意味着我们不必明确描绘一个按钮或者其他任意种类的控件；只需将它们纳入窗体，以后的描绘工作会由它们自行负责。所以为了将一个按钮置入窗体，需要过载 init()方法，而不是过载 paint()：

596 页程序

但这还不足以创建 Button（或其他任何控件）。必须同时调用 Applet add()方法，令按钮放置在程序片的窗体中。这看起来似乎比实际简单得多，因为对 add()的调用实际会（间接地）决定将控件放在窗体的什么地方。对窗体布局的控件马上就要讲到。

13.4 捕获事件

大家可注意到假如编译和运行上面的程序片，按下按钮后不会发生任何事情。必须进入程序片内部，编写用于决定要发生什么事情的代码。对于由事件驱动的程序设计，它的基本目标就是用代码捕获发生的事件，并由代码对那些事件作出响应。事实上，GUI 的大部分内容都是围绕这种事件驱动的程序设计展开的。

经过本书前面的学习，大家应该有了面向对象程序设计的一些基础，此时可能会想到应当有一些面向对象的方法来专门控制事件。例如，也许不得不继承每个按钮，并过载一些“按钮按下”方法（尽管这显得非常麻烦有有限）。大家也可能认为存在一些主控“事件”类，其中为希望响应的每个事件都包含了一个方法。

在对象以前，事件控制的典型方式是 switch 语句。每个事件都对应一个独一无二的整数编号；而且在主事件控制方法中，需要专门为那个值写一个 switch。

Java 1.0 的 AWT 没有采用任何面向对象的手段。此外，它也没有使用 switch

语句，没有打算依靠那些分配给事件的数字。相反，我们必须创建 if 语句的一个嵌套系列。通过 if 语句，我们需要尝试做的事情是侦测到作为事件“目标”的对象。换言之，那是我们关心的全部内容——假如某个按钮是一个事件的目标，那么它肯定是一次鼠标点击，并要基于那个假设继续下去。但是，事件里也可能包含了其他信息。例如，假如想调查一次鼠标点击的像素位置，以便画一条引向那个位置的线，那么 Event 对象里就会包含那个位置的信息（也要注意 Java 1.0 的组件只能产生有限种类的事件，而 Java 1.1 和 Swing/JFC 组件则可产生完整的一系列事件）。

Java 1.0 版的 AWT 方法串联的条件语句中存在 action()方法的调用。虽然整个 Java 1.0 版的事件模型不兼容 Java 1.1 版，但它在还不支持 Java1.1 版的机器和运行简单的程序片的系统中更广泛地使用，忠告您使用它会变得非常的舒适，包括对下面使用的 action()程序方法而言。

action()拥有两个自变量：第一个是事件的类型，包括所有的触发调用 action()的事件的有关信息。例如鼠标单击、普通按键按下或释放、特殊按键按下或释放、鼠标移动或者拖动、事件组件得到或丢失焦点，等等。第二个自变量通常是我们忽略的事件目标。第二个自变量封装在事件目标中，所以它像一个自变量一样的冗长。

需调用 action()时情况非常有限：将控件置入窗体时，一些类型的控件（按钮、复选框、下拉列表单、菜单）会发生一种“标准行动”，从而随相应的 Event 对象发起对 action()的调用。比如对按钮来说，一旦按钮被按下，而且没有再多按一次，就会调用它的 action()方法。这种行为通常正是我们所希望的，因为这正是我们对一个按钮正常观感。但正如本章后面要讲到的那样，还可通过 handleEvent()方法来处理其他许多类型的事件。

前面的例程可进行一些扩展，以便象下面这样控制按钮的点击：

598 页程序

为了解目标是什么，需要向 Event 对象询问它的 target（目标）成员是什么，然后用 equals()方法检查它是否与自己感兴趣的目标对象句柄相符。为所有感兴趣的对象写好句柄后，必须在末尾的 else 语句中调用 super.action(evt, arg)方法。我们在第 7 章已经说过（有关多形性的那一章），此时调用的是我们过载过的方法，而非它的基础类版本。然而，基础类版本也针对我们不感兴趣的所有情况提供了相应的控制代码。除非明确进行，否则它们是不会得到调用的。返回值指出我们是否已经处理了它，所以假如确实与一个事件相符，就应返回 true；否则就返回由基础类 event()返回的东西。

对这个例子来说，最简单的行动就是打印出到底是什么按钮被按下。一些系统允许你弹出一个小消息窗口，但 Java 程序片却防碍窗口的弹出。不过我们可以用调用 Applet 方法的 getAppletContext()来访问浏览器，然后用 showStatus()在浏览器窗口底部的状态栏上显示一条信息（注释③）。还可用同样的方法打印出对事件的一段完整说明文字，方法是调用 getAppletConext().showStatus(evt + "")。空字串会强制编译器将 evt 转换成一个字符串。这些报告对于测试和调试特别有用，因为浏览器可能会覆盖我们的消息。

③：ShowStatus()也属于 Applet 的一个方法，所以可直接调用它，不必调用

getAppletContext()。

尽管看起来似乎很奇怪，但我们确实也能通过 event()中的第二个参数将一个事件与按钮上的文字相配。采用这种方法，上面的例子就变成了：

599 页程序

很难确切知道 equals()方法在这儿要做什么。这种方法有一个很大的问题，就是开始使用这个新技术的 Java 程序员至少需要花费一个受挫折的时期来在比较按钮上的文字时发现他们要么大写了要么写错了（我就有这种经验）。同样，如果我们改变了按钮上的文字，程序代码将不再工作（但我们不会得到任何编译时和运行时的信息）。所以如果可能，我们就得避免使用这种方法。

13.5 文本字段

“文本字段”是允许用户输入和编辑文字的一种线性区域。文本字段从文本组件那里继承了让我们选择文字、让我们像得到字符串一样得到选择的文字，得到或设置文字，设置文本字段是否可编辑以及连同我们从在线参考书中找到的相关方法。下面的例子将证明文本字段的其它功能；我们能注意到方法名是显而易见的：

600－601 页程序

有几种方法均可构建一个文本字段；其中之一是提供一个初始字符串，并设置字符域的大小。

按下按钮 1 是得到我们用鼠标选择的文字就是得到字段内所有的文字并转换成字符串 S。它也允许字段被编辑。按下按钮 2 放一条信息和字符串 s 到 Text fields，并且阻止字段被编辑（尽管我们能够一直选择文字）。文字的可编辑性是通过 setEditable()的真假值来控制的。

13.6 文本区域

“文本区域”很像文字字段，只是它拥有更多的行以及一些引人注目的更多的功能。另外你能在给定位置对一个文本字段追加、插入或者修改文字。这看起来对文本字段有用的功能相当不错，所以设法发现它设计的特性会产生一些困惑。我们可以认为如果我们处处需要“文本区域”的功能，那么可以简单地使用一个线型文字区域在我们将另外使用文本字段的地方。在 Java 1.0 版中，当它们不是固定的时候我们也得到了一个文本区域的垂直和水平方向的滚动条。在 Java 1.1 版中，对高级构建器的修改允许我们选择哪个滚动条是当前的。下面的例子演示的仅仅是在 Java1.0 版的状况下滚动条一直打开。在下一章里我们将看到一个证明 Java 1.1 版中的文字区域的例程。

601-602 页程序

程序中有几个不同的“文本区域”构建器，这其中的一个在此处显示了一个初始字符串和行号和列号。不同的按钮显示得到、追加、修改和插入文字。

13.7 标签

标签准确地运作：安放一个标签到窗体上。这对没有标签的 TextFields 和 Text areas 来说非常的重要，如果我们简单地想安放文字的信息在窗体上也能同样的使用。我们能像本章中第一个例程中演示的那样，使用 drawString()里边的 paint()在确定的位置去安置一个文字。当我们使用的标签允许我们通过布局管理加入其它的文字组件。（在这章的后面我们将进入讨论。）

使用构建器我们能创建一条包括初始化文字的标签（这是我们典型的作法），一个标签包括一行 CENTER（中间）、LEFT（左）和 RIGHT(右)（静态的结果取整定义在类标签里）。如果我们忘记了可以用 getText()和 getalignment()读取值，我们同样可以用 setText()和 setAlignment()来改变和调整。下面的例子将演示标签的特点：

603-604 页程序

首先是标签的最典型的用途：标记一个文本字段或文本区域。在例程的第二部分，当我们按下“test 1”按钮通过 setText()将一串空的空格插入到的字段里。因为空的空格数不等于同样的字符数（在一个等比例间隔的字库里），当插入文字到标签里时我们会看到文字将被省略掉。在例子的第三部分保留的空的空格在我们第一次按下“test 2”会发现标签是空的（trim()删除了每个字符串结尾部分的空格）并且在开头的左列插入了一个短的标签。在工作的其余时间中我们按下按钮进行调整，因此就能看到效果。

我们可能会认为我们可以创建一个空的标签，然后用 setText()安放文字在里面。然而我们不能在一个空标签内加入文字－这大概是因为空标签没有宽度－所以创建一个没有文字的空标签是没有用处的。在上面的例子里，“blank”标签里充满空的空格，所以它足够容纳后面加入的文字。

同样的，setAlignment()在我们用构建器创建的典型的文字标签上没有作用。这个标签的宽度就是文字的宽度，所以不能对它进行任何的调整。但是，如果我们启动一个长标签，然后把它变成短的，我们就可以看到调整的效果。

这些导致事件连同它们最小化的尺寸被挤压的状况被程序片使用的默认布局管理器所发现。有关布局管理器的部分包含在本章的后面。

13.8 复选框

复选框提供一个制造单一选择开关的方法；它包括一个小框和一个标签。典型的复选框有一个小的“X”（或者它设置的其它类型）或是空的，这依靠项目是否被选择来决定的。

我们会使用构建器正常地创建一个复选框，使用它的标签来充当它的自变量。如果我们在创建复选框后想读出或改变它，我们能够获取和设置它的状态，同样也能获取和设置它的标签。注意，复选框的大写是与其它的控制相矛盾的。

无论何时一个复选框都可以设置和清除一个事件指令，我们可以捕捉同样的方法做一个按钮。在下面的例子里使用一个文字区域枚举所有被选中的复选框：

605 页程序

trace()方法将选中的复选框名和当前状态用 appendText()发送到文字区域中去，所以我们看到一个累积的被选中的复选框和它们的状态的列表。

13.9 单选钮

单选钮在 GUI 程序设计中的概念来自于老式的电子管汽车收音机的机械按钮：当我们按下一个按钮时，其它的按钮就会弹起。因此它允许我们强制从众多选择中作出单一选择。

AWT 没有单独的描述单选钮的类；取而代之的是复用复选框。然而将复选框放在单选钮组中（并且修改它的外形使它看起来不同于一般的复选框）我们必须使用一个特殊的构建器象一个自变量一样的作用在 checkboxGroup 对象上。（我们同样能在创建复选框后调用 setCheckboxGroup()方法。）

一个复选框组没有构建器的自变量；它存在的唯一理由就是聚集一些复选框到单选钮组里。一个复选框对象必须在我们试图显示单选钮组之前将它的状态设置成 true，否则在运行时我们就会得到一个异常。如果我们设置超过一个的单选钮为 true，只有最后的一个能被设置成真。

这里有个简单的使用单选钮的例子。注意我们可以像其它的组件一样捕捉单选钮的事件：

606-607 页程序

显示的状态是一个文字字段在被使用。这个字段被设置为不可编辑的，因为它只是用来显示数据而不是收集。这演示了一个使用标签的可取之道。注意字段内的文字是由最早选择的单选钮“Radio button 2”初始化的。

我们可以在窗体中拥有相当多的复选框组。

13.10 下拉列表

下拉列表像一个单选钮组，它是强制用户从一组可实现的选择中选择一个对象的方法。而且，它是一个实现这点的相当简洁的方法，也最易改变选择而不至使用户感到吃力（我们可以动态地改变单选钮，但那种方法显然不方便）。Java 的选择框不像 Windows 中的组合框可以让我从列表中选择或输入自己的选择。在一个选择框中你只能从列表中选择仅仅一个项目。在下面的例子里，选择框从一个确定输入的数字开始，然后当按下一个按钮时，新输入的数字增加到框里。你将可以看到选择框的一些有趣的状态：

607-608 页程序

文本字字段中显示的“selected index,"也就是当前选择的项目的序列号，在事件中选择的字符串就像 action()的第二个自变量的字串符描述的一样好。

运行这个程序片时，请注意对 Choice 框大小的判断：在 windows 里，这个大小是在我们拉下列表时确定的。这意味着如果我们拉下列表，然后增加更多的项目到列表中，这项目将在那，但这个下拉列表不再接受（我们可以通过项目来滚动观察——注释④）。然而，如果我们在第一次拉下下拉列表前将所的项目装入下拉列表，它的大小就会合适。当然，用户在使用时希望看到整个的列表，所以会在下拉列表的状态里对增加项目到选择框里加以特殊的限定。

④：这一行为显然是一种错误，会 Java 以后的版本里解决。

13.11 列表框

列表框与选择框有完全的不同，而不仅仅是当我们在激活选择框时的显示不同，列表框固定在屏幕的指定位置不会改变。另外，一个列表框允许多个选择：如果我们单击在超过一个的项目上，未选择的则表现为高亮度，我们可以选择象我们想要的一样的多。如果我们想察看项目列表，我们可以调用 getSelectedItem()来产生一个被选择的项目列表。要想从一个组里删除一个项目，我们必须再一次的单击它。列表框，当然这里有一个问题就是它默认的动作是双击而不是单击。单击从组中增加或删除项目，双击调用 action()。解决这个问题的方法是象下面的程序假设的一样重新培训我们的用户。

609-610 页程序

按下按钮时，按钮增加项目到列表的顶部（因为 addItem()的第二个自变量为零）。增加项目到列表框比到选择框更加的合理，因为用户期望去滚动一个列表框（因为这个原因，它有内建的滚动条）但用户并不愿意像在前面的例子里不得不去计算怎样才能滚动到要要的那个项目。

然而，调用 action()的唯一方法就是通过双击。如果我们想监视用户在我们的列表中的所作所为（尤其是单击），我们必须提供一个可供选择的方法。

13.11.1 handleEvent()

到目前为止，我们已使用了 action()，现有另一种方法 handleEvent()可对每一事件进行尝试。当一个事件发生时，它总是针对单独事件或发生在单独的事件对象上。该对象的 handleEvent()方法是自动调用的，并且是被 handleEvent()创建并传递到 handleEvent()里。默认的 handleEvent()（handleEvent()定义在组件里，基础类的所有控件都在 AWT 里）将像我们以前一样调用 action()或其它同样的方法去指明鼠标的活动、键盘活动或者指明移动的焦点。我们将会在本章的后面部分看到。

如果其它的方法－特别是 action()－不能满足我们的需要怎么办呢？至于列表框，例如，如果我想捕捉鼠标单击，但 action()只响应双击怎么办呢？这个解答是过载 handleEvent()，毕竟它是从程序片中得到的，因此可以过载任何非确定的方法。当我们为程序片过载 handleEvent()时，我们会得到所有的事件在它们发送出去之前，所以我们不能假设“这里有我的按钮可做的事件，所以我们可以假设按钮被按下了”从它被 action()设为真值。在 handleEvent()中按钮拥有焦点且某人对它进行分配都是可能的。不论它合理与否，我们可测试这些事件并遵照 handleEvent()来进行操作。

为了修改列表样本，使它会响应鼠标的单击，在 action()中按钮测试将被过载，但代码会处理的列表将像下面的例子被移进 handleEvent()中去：

611-612 页程序

这个例子同前面的例子相同除了增加了 handleEvent()外简直一模一样。在程

序中做了试验来验证是否列表框的选择和非选择存在。现在请记住，handleEvent()被程序片所过载，所以它能在窗体中任何存在，并且被其它的列表当成事件来处理。因此我们同样必须通过试验来观察目标。（虽然在这个例子中，程序片中只有一个列表框所以我们能假设所有的列表框事件必须服务于列表框。这是一个不好的习惯，一旦其它的列表框加入，它就会变成程序中的一个缺陷。）如果列表框匹配一个我们感兴趣的列表框，像前面的一样的代码将按上面的策略来运行。注意 handleEvent()的窗体与 action()的相同：如果我们处理一个单独的事件，将返回真值，但如果我们对其它的一些事件不感兴趣，通过 handleEvent()我们必须返回 super.handleEvent()值。这便是程序的核心，如果我们不那样做，其它的任何一个事件处理代码也不会被调用。例如，试注解在上面的代码中返回 super.handleEvent(evt)的值。我们将发现 action()没有被调用，当然那不是我们想得到的。对 action()和 handlEvent()而言，最重要的是跟着上面例子中的格式，并且当我们自己不处理事件时一直返回基础类的方法版本信息。（在例子中我们将返回真值）。（幸运的是，这些类型的错误的仅属于 Java 1.0 版，在本章后面将看到的新设计的 Java 1.1 消除了这些类型的错误。）

在 windows 里，如果我们按下 shift 键，列表框自动允许我们做多个选择。这非常的棒，因为它允许用户做单个或多个的选择而不是编程期间固定的。我们可能会认为我们变得更加的精明，并且当一个鼠标单击被 evt.shiftdown()产生时如果 shift 键是按下的将执行我们自己的试验程序。AWT 的设计妨碍了我们－我们不得不去了解哪个项目被鼠标点击时是否按下了 shift 键，所以我们能取消其余部分所有的选择并且只选择那一个。不管怎样，我们是不可能在 Java 1.0 版中做出来的。（Java 1.1 将所有的鼠标、键盘、焦点事件传送到列表中，所以我们能够完成它。）

13.12 布局的控制

在 Java 里该方法是安一个组件到一个窗体中去，它不同我们使用过的其它 GUI 系统。首先，它是全代码的；没有控制安放组件的“资源”。其次，该方法的组件被安放到一个被“布局管理器”控制的窗体中，由“布局管理器”根据我们 add()它们的决定来安放组件。大小，形状，组件位置与其它系统的布局管理器显著的不同。另外，布局管理器使我们的程序片或应用程序适合窗口的大小，所以，如果窗口的尺寸改变（例如，在 HTML 页面的程序片指定的规格），组件的大小，形状和位置都会改变。

程序片和帧类都是来源于包含和显示组件的容器。（这个容器也是一个组件，所以它也能响应事件。）在容器中，调用 setLayout()方法允许我选择不同的布局管理器。

在这节里我们将探索不同的布局管理器，并安放按钮在它们之上。这里没有捕捉按钮的事件，正好可以演示如何布置这些按钮。

13.12.1 FlowLayout

到目前为止，所有的程序片都被建立，看起来使用一些不可思议的内部逻辑来布置它们的组件。那是因为程序使用一个默认的方式：FlowLayout。这个简单的“Flow”的组件安装在窗体中，从左到右，直到顶部的空格全部再移去一行，并继续循环这些组件。

这里有一个例子明确地（当然也是多余地）设置一个程序片的布局管理器去

FlowLayout，然后在窗体中安放按钮。我们将注意到 FlowLayout 组件使用它们本来的大小。例如一个按钮将会变得和它的字串符一样的大小。

614 页上程序

所有组件将在 FlowLayout 中被压缩为它们的最小尺寸，所以我们可能会得到一些奇怪的状态。例如，一个标签会合适它自己的字符串的尺寸，所以它会右对齐产生一个不变的显示。

13.12.2 BorderLayout

布局管理器有四边和中间区域的概念。当我们增加一些事物到使用 BorderLayout 的面板上时我们必须使用 add()方法将一个字符串对象作为它的第一个自变量，并且字符串必须指定（正确的大写）“North”（上），“South”（下），“west”（左），“East”（右）或者“Center”。如果我们拼写错误或没有大写，就会得到一个编译时的错误，并且程序片不会像你所期望的那样运行。幸运的是，我们会很快发现在 Java 1.1 中有了更多改进。

这是一个简单的程序例子：

614-615 页程序

除了“Center”的每一个位置，当元素在其它空间内扩大到最大时，我们会把它压缩到适合空间的最小尺寸。但是，“Center”扩大后只会占据中心位置。

BorderLayout 是应用程序和对话框的默认布局管理器。

13.12.3 GridLayout

GridLayout 允许我们建立一个组件表。添加那些组件时，它们会按从左到右、从上到下的顺序在网格中排列。在构建器里，需要指定自己希望的行、列数，它们将按正比例展开。

615 页下程序

在这个例子里共有 21 个空位，但却只有 20 个按钮，最后的一个位置作留空处理；注意对 GridLayout 来说，并不存在什么“均衡”处理。

13.12.4 CardLayout

CardLayout 允许我们在更复杂的拥有真正的文件夹卡片与一条边相遇的环境里创建大致相同于“卡片式对话框”的布局，我们必须压下一个卡片使不同的对话框带到前面来。在 AWT 里不是这样的：CardLayout 是简单的空的空格，我们可以自由地把新卡片带到前面来。（JFC/Swing 库包括卡片式的窗格看起来非常的棒，且可以我们处理所有的细节。）

1. 联合布局（Combining layouts)

下面的例子联合了更多的布局类型，在最初只有一个布局管理器被程序片或应用程序操作看起来相当的困难。这是事实，但如果我们创建更多的面板对象，

每个面板都能拥有一个布局管理器，并且像被集成到程序片或应用程序中一样使用程序片或应用程序的布局管理器。这就象下面程序中的一样给了我们更多的灵活性：

616-617 页程序

这个例子首先会创建一种新类型的面板：BottonPanel（按钮面板）。它包括一个单独的按钮，安放在 BorderLayout 的中央，那意味着它将充满整个的面板。按钮上的标签将让我们知道我们在 CardLayout 上的那个面板上。

在程序片里，面板卡片上将存放卡片和布局管理器 CL 因为 CardLayout 必须组成类，因为当我们需要处理卡片时我们需要访问这些句柄。

这个程序片变成使用 BorderLayout 来取代它的默认 FlowLayout，创建面板来容纳三个按钮（使用 FlowLayout），并且这个面板安置在程序片末尾的“North”。卡片面板增加到程序片的“Center”里，有效地占据面板的其余地方。

当我们增加 BottonPanels(或者任何其它我们想要的组件)到卡片面板时，add()方法的第一个自变量不是“North”，“South”等等。相反的是，它是一个描述卡片的字符串。如果我们想轻击那张卡片使用字符串，我们就可以使用，虽然这字符串不会显示在卡片的任何地方。使用的方法不是使用 action()；代之使用 first()、next()和 last()等方法。请查看我们有关其它方法的文件。

在 Java 中，使用的一些卡片式面板结构十分的重要，因为（我们将在后面看到）在程序片编程中使用的弹出式对话框是十分令人沮丧的。对于 Java 1.0 版的程序片而言，CardLayout 是唯一有效的取得很多不同的“弹出式”的窗体。

13.12.5 GridBagLayout

很早以前，人们相信所有的恒星、行星、太阳及月亮都围绕地球公转。这是直观的观察。但后来天文学家变得更加的精明，他们开始跟踪个别星体的移动，它们中的一些似乎有时在轨道上缓慢运行。因为天文学家知道所有的天体都围绕地球公转，天文学家花费了大量的时间来讨论相关的方程式和理论去解释天体对象的运行。当我们试图用 GridBagLayout 来工作时，我们可以想像自己为一个早期的天文学家。基础的条例是（公告：有趣的是设计者居然在太阳上(这可能是在天体图中标错了位置所致，译者注)）所有的天体都将遵守规则来运行。哥白尼日新说（又一次不顾嘲讽，发现太阳系内的所有的行星围绕太阳公转。）是使用网络图来判断布局，这种方法使得程序员的工作变得简单。直到这些增加到 Java 里，我们忍耐（持续的冷嘲热讽）西班牙的 GridBagLayout 和 GridBagConstraints 狂热宗教。我们建议废止 GridBagLayout。取代它的是，使用其它的布局管理器和特殊的在单个程序里联合几个面板使用不同的布局管理器的技术。我们的程序片看起来不会有什么不同；至少不足以调整 GridBagLayout 限制的麻烦。对我而言，通过一个例子来讨论它实在是令人头痛（并且我不鼓励这种库设计）。相反，我建议您从阅读 Cornell 和 Horstmann 撰写的《核心 Java》（第二版，Prentice-Hall 出版社，1997 年）开始。

在这范围内还有其它的：在 JFC/Swing 库里有一个新的使用 Smalltalk 的受人欢迎的“Spring and Struts”布局管理器并且它能显著地减少 GridBagLayout 的需要。

13.13 action 的替代品

正如早先指出的那样，action()并不是我们对所有事进行分类后自动为handleEvent()调用的唯一方法。有三个其它的被调用的方法集，如果我们想捕捉某些类型的事件（键盘、鼠标和焦点事件），因此我们不得不过载规定的方法。这些方法是定义在基础类组件里，所以他们几乎在所有我们可能安放在窗体中的组件中都是有用的。然而，我们也注意到这种方法在 Java 1.1 版中是不被支持的，同样尽管我们可能注意到继承代码利用了这种方法，我们将会使用 Java 1.1 版的方法来代替（本章后面有详细介绍）。

组件方法 何时调用

action(Event evt, Object what) 当典型的事件针对组件发生（例如，当按下一个按钮或下拉列表项目被选中）时调用

keyDown(Event evt, int key） 当按键被按下，组件拥有焦点时调用。第二个自变量是按下的键并且是冗余的是从 evt.key 处复制来的

keyup(Event evt, int key) 当按键被释放，组件拥有焦点时调用

lostFocus(Event evt, Object what) 焦点从目标处移开时调用。通常，what 是从 evt.arg 里冗余复制的

gotFocus(Event evt, Object what) 焦点移动到目标时调用

mouseDown(Event evt, int x，int y) 一个鼠标按下存在于组件之上，在 X，Y 座标处时调用

mouseUp(Event evt, int x, int y) 一个鼠标升起存在于组件之上时调用

mouseMove(Event evt, int x, int y) 当鼠标在组件上移动时调用

mouseDrag(Event evt, int x, int y） 鼠标在一次 mouseDown 事件发生后拖动。所有拖动事件都会报告给内部发生了 mouseDown 事件的那个组件，直到遇到一次 mouseUp 为止

mouseEnter(Event evt, int x, int y) 鼠标从前不在组件上方，但目前在

mouseExit(Event evt, int x, int y) 鼠标曾经位于组件上方，但目前不在

当我们处理特殊情况时——一个鼠标事件，例如，它恰好是我们想得到的鼠标事件存在的座标，我们将看到每个程序接收一个事件连同一些我们所需要的信息。有趣的是，当组件的 handleEvent()调用这些方法时（典型的事例），附加的自变量总是多余的因为它们包含在事件对象里。事实上，如果我们观察 component.handleEvent()的源代码，我们能发现它显然将增加的自变量抽出事件对象（这可能是考虑到在一些语言中无效率的编码，但请记住 Java 的焦点是安全的，不必担心。）试验对我们表明这些事件事实上在被调用并且作为一个有趣的尝试是值得创建一个过载每个方法的程序片，（action()的过载在本章的其它地方）当事件发生时显示它们的相关数据。

这个例子同样向我们展示了怎样制造自己的按钮对象，因为它是作为目标的所有事件权益来使用。我可能会首先（也是必须的）假设制造一个新的按钮，我们从按钮处继承。但它并不能运行。取而代之的是，我们从画布组件处（一个非常普通组件）继承，并在其上不使用 paint()方法画出一个按钮。正如我们所看到的，自从一些代码混入到画按钮中去，按钮根本就不运行，这实在是太糟糕了。（如果您不相信我，试图在例子中为画布组件交换按钮，请记住调用称为 super

的基础类构建器。我们会看到按钮不会被画出，事件也不会被处理。）

myButton 类是明确说明的：它只和一个自动事件（AutoEvent）“父窗口”一起运行（父窗口不是一个基础类，它是按钮创建和存在的窗口。）。通过这个知识，myButton 可能进入到父窗口并且处理它的文字字段，必然就能将状态信息写入到父窗口的字段里。当然这是一种非常有限的解决方法，myButton 仅能在连结 AutoEvent 时被使用。这种代码有时称为“高度结合”。但是，制造 myButton 更需要很多的不是为例子（和可能为我们将写的一些程序片）担保的努力。再者，请注意下面的代码使用了 Java 1.1 版不支持的 API。

621-624 页程序

我们可以看到构建器使用利用自变量同名的方法，所以自变量被赋值，并且使用 this 来区分：

this.label = label;

paint()方法由简单的开始：它用按钮的颜色填充了一个“圆角矩形”，然后画了一个黑线围绕它。请注意 size()的使用决定了组件的宽度和长度（当然，是像素）。这之后，paint()看起来非常的复杂，因为有大量的预测去计算出怎样利用“font metrics”集中按钮的标签到按钮里。我们能得到一个相当好的关于继续关注方法调用的主意，它将程序中那些相当平凡的代码挑出，当我们想集中一个标签到一些组件里时，我们正好可以对它进行剪切和粘贴。

您直到注意到 AutoEvent 类才能正确地理解 keyDown(),keyUp()及其它方法的运行。这包含一个 Hashtable（译者注：散列表）去控制字符串来描述关于事件处理的事件和 TextField 类型。当然，这些能被静态的创建而不是放入 Hashtable 但我认为您会同意它是更容易使用和改变的。特别是，如果我们需要在 AutoEvent 中增加或删除一个新的事件类型，我们只需要简单地在事件列队中增加或删除一个字符串——所有的工作都自动地完成了。

我们查出在 keyDown()，keyup()及其它方法中的字符串的位置回到 myButton 中。这些方法中的任何一个都用父句柄试图回到父窗口。父类是一个 AutoEvent，它包含 Hashtable h 和 get()方法，当拥有特定的字符串时，将对一个我们知道的 TextField 对象产生一个句柄（因此它被选派到那）。然后事件对象修改显示在 TextField 中的字符串陈述。从我们可以真正注意到举出的例子在我们的程序中运行事件时以来，可以发现这个例子运行起来颇为有趣的。

13.14 程序片的局限

出于安全缘故，程序片十分受到限制，并且有很多的事我们都不能做。您一般会问：程序片看起来能做什么，传闻它又能做什么：扩展浏览器中 WEB 页的功能。自从作为一个网上冲浪者，我们从未真正想了解是否一个 WEB 页来自友好的或者不友好的站点，我们想要一些可以安全地行动的代码。所以我们可能会注意到大量的限制：

(1) 一个程序片不能接触到本地的磁盘。这意味着不能在本地磁盘上写和读，我们不想一个程序片通过 WEB 页面阅读和传送重要的信息。写是被禁止的，当然，因为那将会引起病毒的侵入。当数字签名生效时，这些限制会被解除。

(2) 程序片不能拥有菜单。（注意：这是规定在 Swing 中的）这可能会减少关于安全和关于程序简化的麻烦。我们可能会接到有关程序片协调利益以作为

WEB 页面的一部分的通知；而我们通常不去注意程序片的范围。这儿没有帧和标题条从菜单处弹出，出现的帧和标题条是属于 WEB 浏览器的。也许将来设计能被改变成允许我们将浏览器菜单和程序片菜单相结合起来——程序片可以影响它的环境将导致太危及整个系统的安全并使程序片过于的复杂。

(3) 对话框是不被信任的。在 Java 中，对话框存在一些令人难解的地方。首先，它们不能正确地拒绝程序片，这实在是令人沮丧。如果我们从程序片弹出一个对话框，我们会在对话框上看到一个附上的消息框“不被信任的程序片”。这是因为在理论上，它有可能欺骗用户去考虑他们在通过 WEB 同一个老顾客的本地应用程序交易并且让他们输入他们的信用卡号。在看到 AWT 开发的那种 GUI 后，我们可能会难过地相信任何人都会被那种方法所愚弄。但程序片是一直附着在一个 Web 页面上的，并可以在浏览器中看到，而对话框没有这种依附关系，所以理论上是可能的。因此，我们很少会见到一个使用对话框的程序片。

在较新的浏览器中，对受到信任的程序片来说，许多限制都被放宽了（受信任程序片由一个信任源认证）。

涉及程序片的开发时，还有另一些问题需要考虑：

■程序片不停地从一个适合不同类的单独的服务器上下载。我们的浏览器能够缓存程序片，但这没有保证。在 Java 1.1 版中的一个改进是 JAR（Java ARchive）文件，它允许将所有的程序片组件（包括其它的类文件、图像、声音）一起打包到一个的能被单个服务器处理下载的压缩文件。“数字签字”（能校验类创建器）可有效地加入每个单独的 JAR 文件。

■因为安全方面的缘故，我们做某些工作更加困难，例如访问数据库和发送电子邮件。另外，安全限制规则使访问多个主机变得非常的困难，因为每一件事都必须通过 WEB 服务器路由，形成一个性能瓶颈，并且单一环节的出错都会导致整个处理的停止。

■浏览器里的程序片不会拥有同样的本地应用程序运行的控件类型。例如，自从用户可以开关页面以来，在程序片中不会拥有一个形式上的对话框。当用户对一个 WEB 页面进行改变或退出浏览器时，对我们的程序片而言简直是一场灾难——这时没有办法保存状态，所以如果我们在处理和操作中时，信息会被丢失。另外，当我们离开一个 WEB 页面时，不同的浏览器会对我们的程序片做不同的操作，因此结果本来就是不确定的。

13.14.1 程序片的优点

如果能容忍那些限制，那么程序片的一些优点也是非常突出的，尤其是在我们构建客户 / 服务器应用或者其它网络应用时：

■没有安装方面的争议。程序片拥有真正的平台独立性（包括容易地播放声音文件等能力）所以我们不需要针对不同的平台修改代码也不需要任何人根据安装运行任何的“tweaking”。事实上，安装每次自动地将 WEB 页连同程序片一起，因此安静、自动地更新。在传统的客户机/服务器系统中，建立和安装一个新版本的客户端软件简直就是一场恶梦。

■因为安全的原因创建在核心 Java 语言和程序片结构中，我们不必担心坏的代码而导致毁坏某人的系统。这样，连同前面的优点，可使用 Java（可从 JavaScript 和 VBScript 中选择客户端的 WEB 编程工具）为所谓的 Intrant（在公司内部使用而不向 Internet 转移的企业内部网络）客户机/服务器开发应用程序。

■由于程序片是自动同 HTML 集成的，所以我们有一个内建的独立平台文件

系统去支持程序片。这是一个很有趣的方法，因为我们惯于拥有程序文件的一部分而不是相反的拥有文件系统。

13.15 视窗化应用

出于安全的缘故，我们会看到在程序片我们的行为非常的受到限制。我们真实地感到，程序片是被临时地加入在 WEB 浏览器中的，因此，它的功能连同它的相关知识，控件都必须加以限制。但是，我们希望 Java 能制造一个开窗口的程序去运行一些事物，否则宁愿安放在一个 WEB 页面上，并且也许我们希望它可以运行一些可靠的应用程序，以及夸张的实时便携性。在这本书前面的章节中我们制造了一些命令行应用程序，但在一些操作环境中（例如：Macintosh）没有命令行。所以我们有很多的理由去利用 Java 创建一个设置窗口，非程序片的程序。这当然是一个十分合理的要求。

一个 Java 设置窗口应用程序可以拥有菜单和对话框(这对一个程序片来说是不可能的和很困难的)，可是如果我们使用一个老版本的 Java，我们将会牺牲本地操作系统环境的外观和感受。JFC/Swing 库允许我们制造一个保持原来操作系统环境的外观和感受的应用程序。如果我们想建立一个设置窗口应用程序，它会合理地运作，同样，如果我们可以使用最新版本的 Java 并且集合所有的工具，我们就可以发布不会使用户困惑的应用程序。如果因为一些原因，我们被迫使用老版本的 Java，请在毁坏以建立重要的设置窗口的应用程序前仔细地考虑。

13.15.1 菜单

直接在程序片中安放一个菜单是不可能的（Java 1.0,Java1.1 和 Swing 库不允许)，因为它们是针对应用程序的。继续，如果您不相信我并且确定在程序片中可以合理地拥有菜单，那么您可以去试验一下。程序片中没有 setMenuBar()方法，而这种方法是附在菜单中的（我们会看到它可以合理地在程序片产生一个帧，并且帧包含菜单)。

有四种不同类型的 MenuComponent（菜单组件)，所有的菜单组件起源于抽象类：菜单条（我们可以在一个事件帧里拥有一个菜单条)，菜单去支配一个单独的下拉菜单或者子菜单、菜单项来说明菜单里一个单个的元素，以及起源于 MenuItem, 产生检查标志（checkmark）去显示菜单项是否被选择的 CheckBoxMenuItem。

不同的系统使用不同的资源，对 Java 和 AWT 而言，我们必须在源代码中手工汇编所有的菜单。

628-630 页程序

在这个程序中，我避免了为每个菜单编写典型的冗长的 add()列表调用，因为那看起来像许多的无用的标志。取而代之的是，我安放菜单项到数组中，然后在一个 for 的循环中通过每个数组调用 add()简单地跳过。这样的话，增加和减少菜单项变得没那么讨厌了。

作为一个可选择的方法（我发现这很难令我满意，因为它需要更多的分配）CheckboxMenuItems 在数组的句柄中被创建是被称为安全创建；这对数组文件和其它的文件而言是真正的安全。

程序中创建了不是一个而是二个的菜单条来证明菜单条在程序运行时能被

交换激活。我们可以看到菜单条怎样组成菜单，每个菜单怎样组成菜单项（MenuItems），chenkboxMenuItems 或者其它的菜单（产生子菜单）。当菜单组合后，可以用 setMenuBar()方法安装到现在的程序中。值得注意的是当按钮被压下时，它将检查当前的菜单安装使用 getMenuBar()，然后安放其它的菜单条在它的位置上。

当测试是“open”（即开始）时，注意拼写和大写，如果开始时没有对象，Java 发出 no error（没有错误）的信号。这种字符串比较是一个明显的程序设计错误源。

校验和非校验的菜单项自动地运行，与之相关的 CheckBoxMenuItems 着实令人吃惊，这是因为一些原因它们不允许字符串匹配。（这似乎是自相矛盾的，尽管字符串匹配并不是一种很好的办法。）因此，我们可以匹配一个目标对象而不是它们的标签。当演示时，getState()方法用来显示状态。我们同样可以用 setState()改变 CheckboxMenuItem 的状态。

我们可能会认为一个菜单可以合理地置入超过一个的菜单条中。这看似合理，因为所有我们忽略的菜单条的 add()方法都是一个句柄。然而，如果我们试图这样做，这个结果将会变得非常的别扭，而远非我们所希望得到的结果。（很难知道这是一个编程中的错误或者说是他们试图使它以这种方法去运行所产生的。）这个例子同样向我们展示了为什么我们需要建立一个应用程序以替代程序片。（这是因为应用程序能支持菜单，而程序片是不能直接使用菜单的。）我们从帧处继承代替从程序片处继承。另外，我们为类建一个构建器以取代 init()安装事件。最后，我们创建一个 main()方法并且在我们建的新型对象里，调整它的大小，然后调用 show()。它与程序片只在很小的地方有不同之处，然而这时它已经是一个独立的设置窗口应用程序并且我们可以使用菜单。

13.15.2 对话框

对话框是一个从其它窗口弹出的窗口。它的目的是处理一些特殊的争议和它们的细节而不使原来的窗口陷入混乱之中。对话框大量在设置窗口的编程环境中使用，但就像前面提到的一样，鲜于在程序片中使用。

我们需要从对话类处继承以创建其它类型的窗口、像帧一样的对话框。和窗框不同，对话框不能拥有菜单条也不能改变光标，但除此之外它们十分的相似。一个对话框拥有布局管理器（默认的是 BorderLayout 布局管理器）和过载 action()等等，或用 handleEvent()去处理事件。我们会注意到 handleEvent()的一个重要差异：当 WINDOW_DESTORY 事件发生时，我们并不希望关闭正在运行的应用程序！

相反，我们可以使用对话窗口通过调用 dispace()释放资源。在下面的例子中，对话框是由定义在那儿作为类的 ToeButton 的特殊按钮组成的网格构成的（利用 GridLayout 布局管理器）。ToeButton 按钮围绕它自己画了一个帧，并且依赖它的状态：在空的中的“X”或者“O”。它从空白开始，然后依靠使用者的选择，转换成“X”或“O”。但是，当我们单击在按钮上时，它会在“X”和“O”之间来回交换。（这产生了一种类似填字游戏的感觉，当然比它更令人讨厌。）另外，这个对话框可以被设置为在主应用程序窗口中为很多的行和列变更号码。

632-634 页程序

ToeButton 类保留了一个句柄到它 ToeDialog 型的父类中。正如前面所述，ToeButton 和 ToeDialog 高度的结合因为一个 ToeButton 只能被一个 ToeDialog 所使用，但它却解决了一系列的问题，事实上这实在不是一个糟糕的解决方案因为没有另外的可以记录用户选择的对话类。当然我们可以使用其它的制造 ToeDialog.turn（ToeButton 的静态的一部分）方法。这种方法消除了它们的紧密联系，但却阻止了我们一次拥有多个 ToeDialog（无论如何，至少有一个正常地运行）。

paint()是一种与图形有关的方法：它围绕按钮画出矩形并画出“X”或“O”。这完全是冗长的计算，但却十分的直观。

一个鼠标单击被过载的 mouseDown()方法所俘获，最要紧的是检查是否有事件写在按钮上。如果没有，父窗口会被询问以找出谁选择了它并用来确定按钮的状态。值得注意的是按钮随后交回到父类中并且改变它的选择。如果按钮已经显示这为“X”和“O”，那么它们会被改变状态。我们能注意到本书第三章中描述的在这些计算中方便的使用的三个一组的 If-else。当一个按钮的状态改变后，按钮会被重画。

ToeDialog 的构建器十分的简单：它像我们所需要的一样增加一些按钮到 GridLayout 布局管理器中，然后调整每个按钮每边大小为 50 个像素（如果我们不调整窗口，那么它就不会显示出来）。注意 handleEvent() 正好为 WINDOW_DESTROY 调用 dispose()，因此整个应用程序不会被关闭。

ToeTest 设置整个应用程序以创建 TextField（为输入按钮网格的行和列）和“go”按钮。我们会领会 action()在这个程序中使用不太令人满意的“字符串匹配”技术来测试按钮的按下（请确定我们拼写和大写都是正确的!）。当按钮按下时，TextField 中的数据将被取出，并且，因为它们在字符串结构中，所以需要利用静态的 Integer.paresInt()方法来转变成中断。一旦对话类被建立，我们就必须调用 show()方法来显示和激活它。

我们会注意到 ToeDialog 对象赋值给一个对话句柄 d。这是一个上溯造型的例子，尽管它没有真正地产生重要的差异，因为所有的事件都是 show()调用的。但是，如果我们想调用 ToeDialog 中已经存在的一些方法，我们需要对 ToeDialog 句柄赋值，就不会在一个上溯中丢失信息。

1. 文件对话类

在一些操作系统中拥有许多的特殊内建对话框去处理选择的事件，例如：字库，颜色，打印机以及类似的事件。几乎所有的操作系统都支持打开和保存文件，但是，Java 的 FileDialog 包更容易使用。当然这会不再检测所有使用的程序片，因为程序片在本地磁盘上既不能读也不能写文件。（这会在新的浏览器中交换程序片的信任关系。）

下面的应用程序运用了两个文件对话类的窗体，一个是打开，一个是保存。大多数的代码到如今已为我们所熟悉，而所有这些有趣的活动发生在两个不同按钮单击事件的 action()方法中。

636-637 页程序

对一个“打开文件”对话框，我们使用构建器设置两个自变量；首先是父窗口句柄，其次是 FileDialog 标题条的标题。setFile()方法提供一个初始文件名－－

也许本地操作系统支持通配符，因此在这个例子中所有的.java 文件最开头会被显示出来。setDirectory()方法选择文件决定开始的目录（一般而言，操作系统允许用户改变目录）。

show()命令直到对话类关闭才返回。FileDialog 对象一直存在，因此我们可以从它那里读取数据。如果我们调用 getFile()并且它返回空，这意味着用户退出了对话类。文件名和调用 getDirectory()方法的结果都显示在 TextFields 里。

按钮的保存工作使用同样的方法，除了因为 FileDialog 而使用不同的构建器。这个构建器设置了三个自变量并且第三的一个自变量必须为 FileDialog.SAVE 或 FileDialog.OPEN。

13.16 新型 AWT

在 Java 1.1 中一个显著的改变就是完善了新 AWT 的创新。大多数的改变围绕在 Java 1.1 中使用的新事件模型：老的事件模型是糟糕的、笨拙的、非面向对象的，而新的事件模型可能是我所见过的最优秀的。难以理解一个如此糟糕的（老的 AWT）和一个如此优秀的（新的事件模型）程序语言居然出自同一个集团之手。新的考虑事件的方法看来中止了，因此争议不再变成障碍，从而轻易进入我们的意识里；相反，它是一个帮助我们设计系统的工具。它同样是 Java Beans 的精华，我们会在本章后面部分进入讲述。

新的方法设计对象做为“事件源”和“事件接收器”以代替老 AWT 的非面向对象串联的条件语句。正象我们将看到的内部类的用途是集成面向对象的原始状态的新事件。另外，事件现在被描绘为在一个类体系以取代单一的类并且我们可以创建自己的事件类型。

我们同样会发现，如果我们采用老的 AWT 编程，Java 1.1 版会产生一些看起来不合理的名字转换。例如，setsize()改成 resize()。当我们学习 Java Beans 时这会变得更加的合理，因为 Beans 使用一个独特的命名协议。名字必须被修改以在 Beans 中产生新的标准 AWT 组件。

剪贴板操作在 Java 1.1 版中也得到支持，尽管拖放操作“将在新版本中被支持”。我们可能访问桌面色彩组织，所以我们的 Java 可以同其余桌面保持一致。可以利用弹出式菜单，并且为图像和图形作了改进。也同样支持鼠标操作。还有简单的为打印的 API 以及简单地支持滚动。

13.16.1 新的事件模型

在新的事件模型的组件可以开始一个事件。每种类型的事件被一个个别的类所描绘。当事件开始后，它受理一个或更多事件指明“接收器”。因此，事件源和处理事件的地址可以被分离。

每个事件接收器都是执行特定的接收器类型接口的类对象。因此作为一个程序开发者，我们所要做的是创建接收器对象并且在被激活事件的组件中进行注册。event-firing 组件调用一个 addXXXListener()方法来完成注册，以描述 XXX 事件类型接受。我们可以容易地了解到以 addListened 名的方法通知我们任何的事件类型都可以被处理，如果我们试图接收事件我们会发现编译时我们的错误。Java Beans 同样使用这种 addListener 名的方法去判断那一个程序可以运行。

我们所有的事件逻辑将装入到一个接收器类中。当我们创建一个接收器类时唯一的一点限制是必须执行专用的接口。我们可以创建一个全局接收器类，这种情况在内部类中有助于被很好地使用，不仅仅是因为它们提供了一个理论上的接

收器类组到它们服务的 UI 或业务逻辑类中，但因为（正像我们将会在本章后面看到的）事实是一个内部类维持一个句柄到它的父对象，提供了一个很好的通过类和子系统边界的调用方法。

一个简单的例子将使这一切变得清晰明确。同时思考本章前部 Button2.java 例子与这个例子的差异。

639-640 页程序

我们可比较两种方法，老的代码在左面作为注解。在 init()方法里，只有一个改变就是增加了下面的两行：

b1.addActionListener(new B1());

b2.addActionListener(new B2());

按钮按下时，addActionListener()通知按钮对象被激活。B1 和 B2 类都是执行接口 ActionListener 的内部类。这个接口包括一个单一的方法 actionPerformed()（这意味着当事件激活时，这个动作将被执行）。注意 actionPreformed()方法不是一个普通事件，说得更恰当些是一个特殊类型的事件，ActionEvent。如果我们想提取特殊 ActionEvent 的信息，因此我们不需要故意去测试和下溯造型自变量。

对编程者来说一个最好的事便是 actionPerformed()十分的简单易用。它是一个可以调用的方法。同老的 action()方法比较，老的方法我们必须指出发生了什么和适当的动作，同样，我们会担心调用基础类 action()的版本并且返回一个值去指明是否被处理。在新的事件模型中，我们知道所有事件测试推理自动进行，因此我们不必指出发生了什么；我们刚刚表示发生了什么，它就自动地完成了。如果我们还没有提出用新的方法覆盖老的方法，我们会很快提出。

13.16.2 事件和接收者类型

所有 AWT 组件都被改变成包含 addXXXListener()和 removeXXXListener()方法，因此特定的接收器类型可从每个组件中增加和删除。我们会注意到“XXX”在每个场合中同样表示自变量的方法，例如，addFooListener(FooListener fl)。下面这张表格总结了通过提供 addXXXListener()和 removeXXXListener()方法，从而支持那些特定事件的相关事件、接收器、方法以及组件。

事件，接收器接口及添加和删除方法 支持这个事件的组件

641-643 页表略

⑤：即使表面上如此，但实际上并没有 MouseMotiionEvent（鼠标运动事件）。单击和运动都合成到 MouseEvent 里，所以 MouseEvent 在表格中的这种另类行为并非一个错误。

可以看到，每种类型的组件只为特定类型的事件提供了支持。这有助于我们发现由每种组件支持的事件，如下表所示：

组件类型 支持的事件

643－644 页表略

一旦知道了一个特定的组件支持哪些事件，就不必再去寻找任何东西来响应那个事件。只需简单地：

(1) 取得事件类的名字，并删掉其中的“Event”字样。在剩下的部分加入“Listener”字样。这就是在我们的内部类里需要实现的接收器接口。

(2) 实现上面的接口，针对想要捕获的事件编写方法代码。例如，假设我们想捕获鼠标的移动，所以需要为 MouseMotiionListener 接口的 mouseMoved()方法编写代（当然还必须实现其他一些方法，但这里有捷径可循，马上就会讲到这个问题）。

(3) 为步骤 2 中的接收器类创建一个对象。随自己的组件和方法完成对它的注册，方法是在接收器的名字里加入一个前缀“add”。比如 addMouseMotionListener()。

下表是对接收器接口的一个总结：

接收器接口 接口中的方法

645 页表略

1. 用接收器适配器简化操作

在上面的表格中，我们可以注意到一些接收器接口只有唯一的一个方法。它们的执行是无轻重的，因为我们仅当需要书写特殊方法时才会执行它们。然而，接收器接口拥有多个方法，使用起来却不太友好。例如，我们必须一直运行某些事物，当我们创建一个应用程序时对帧提供一个 WindowListener，以便当我们得到 windowClosing()事件时可以调用 System.exit(0)以退出应用程序。但因为 WindowListener 是一个接口，我们必须执行其它所有的方法即使它们不运行任何事件。这真令人讨厌。

为了解决这个问题，每个拥有超过一个方法的接收器接口都可拥有适配器，它们的名我们可以在上面的表格中看到。每个适配器为每个接口方法提供默认的方法。（WindowAdapter 的默认方法不是 windowClosing()，而是 System.exit(0)方法。）此外我们所要做的就是从适配器处继承并过载唯一的需要变更的方法。例如，典型的 WindowListener 我们会像下面这样的使用。

646 页上程序

适配器的全部宗旨就是使接收器的创建变得更加简便。

但所谓的“适配器”也有一个缺点，而且较难发觉。假定我们象上面那样写一个 WindowAdapter：

646 页下程序

表面上一切正常，但实际没有任何效果。每个事件的编译和运行都很正常——只是关闭窗口不会退出程序。您注意到问题在哪里吗？在方法的名字里：是

WindowClosing()，而不是 windowClosing()。大小写的一个简单失误就会造成一个崭新的方法。但是，这并非我们关闭窗口时调用的方法，所以当然没有任何效果。

13.16.3 用 Java 1.1 AWT 制作窗口和程序片

我们经常都需要创建一个类，使其既可作为一个窗口调用，亦可作为一个程序片调用。为做到这一点，只需为程序片简单地加入一个 main()即可，令其在一个 Frame（帧）里构建程序片的一个实例。作为一个简单的示例，下面让我们来看看如何对 Button2New.java 作一番修改，使其能同时作为应用程序和程序片使用：

647-648 页程序

内部类 WL 和 main()方法是加入程序片的唯一两个元素，程序片剩余的部分则原封未动。事实上，我们通常将 WL 类和 main()方法做一结小的改进复制和粘贴到我们自己的程序片里（请记住创建内部类时通常需要一个外部类来处理它，形成它静态地消除这个需要）。我们可以看到在 main()方法里，程序片明确地初始化和开始，因为在这个例子里浏览器不能为我们有效地运行它。当然，这不会提供全部的浏览器调用 stop()和 destroy()的行为，但对大多数的情况而言它都是可接受的。如果它变成一个麻烦，我们可以：

(1) 使程序片句柄为一个静态类（以代替局部可变的 main()），然后：

(2) 在我们调用 System.exit()之前在 WindowAdapter.windowClosing()中调用 applet.stop()和 applet.destroy()。

注意最后一行：

aFrame.setVisible(true);

这是 Java 1.1 AWT 的一个改变。show()方法不再被支持，而 setVisible(true)则取代了 show()方法。当我们在本章后面部分学习 Java Beans 时，这些表面上易于改变的方法将会变得更加的合理。

这个例子同样被使用 TextField 修改而不是显示到控制台或浏览器状态行上。在开发程序时有一个限制条件就是程序片和应用程序我们都必须根据它们的运行情况选择输入和输出结构。

这里展示了 Java 1.1 AWT 的其它小的新功能。我们不再需要去使用有错误倾向的利用字符串指定 BorderLayout 定位的方法。当我们增加一个元素到 Java 1.1 版的 BorderLayout 中时，我们可以这样写：

aFrame.add(applet, BorderLayout.CENTER);

我们对位置规定一个 BorderLayout 的常数，以使它能在编译时被检验（而不是对老的结构悄悄地做不合适的事）。这是一个显著的改善，并且将在这本书的余下部分大量地使用。

2. 将窗口接收器变成匿名类

任何一个接收器类都可作为一个匿名类执行，但这一直有个意外，那就是我们可能需要在其它场合使用它们的功能。但是，窗口接收器在这里仅作为关闭应用程序窗口来使用，因此我们可以安全地制造一个匿名类。然后，main()中的下面这行代码：

aFrame.addWindowListener(new WL());

会变成：

649 页上程序

这有一个优点就是它不需要其它的类名。我们必须对自己判断是否它使代码变得易于理解或者更难。不过，对本书余下部分而言，匿名内部类将通常被使用在窗口接收器中。

3. 将程序片封装到 JAR 文件里

一个重要的 JAR 应用就是完善程序片的装载。在 Java 1.0 版中，人们倾向于试法将它们的代码填入到单个的程序片类里，因此客户只需要单个的服务器就可适合下载程序片代码。但这不仅使结果凌乱，难以阅读（当然维护也然）程序，但类文件一直不能压缩，因此下载从来没有快过。

JAR 文件将我们所有的被压缩的类文件打包到一个单个儿的文件中，再被浏览器下载。现在我们不需要创建一个糟糕的设计以最小化我们创建的类，并且用户将得到更快地下载速度。

仔细想想上面的例子，这个例子看起来像 Button2NewB，是一个单类，但事实上它包含三个内部类，因此共有四个。每当我们编译程序，我会用这行代码打包它到一个 JAR 文件：

jar cf Button2NewB.jar *.class

这是假定只有一个类文件在当前目录中，其中之一来自 Button2NewB.java（否则我们会得到特别的打包）。

现在我们可以创建一个使用新文件标签来指定 JAR 文件的 HTML 页，如下所示：

650 页上程序

与 HTML 文件中的程序片标记有关的其他任何内容都保持不变。

13.16.4 再研究一下以前的例子

为注意到一些利用新事件模型的例子和为学习程序从老到新事件模型改变的方法，下面的例子回到在本章第一部分利用事件模型来证明的一些争议。另外，每个程序包括程序片和应用程序现在都可以借助或不借助浏览器来运行。

1. 文本字段

这个例子同 TextField1.java 相似，但它增加了显然额外的行为：

650-652 页程序

当 TextField t1 的动作接收器被激活时，TextField t3 就是一个需要报告的场所。我们注意到仅当我们按下“enter”键时，动作接收器才会为“TextField”所激活。

TextField t1 附有几个接收器。T1 接收器从 t1 复制所有文字到 t2，强制所有字符串转换成大写。我们会发现这两个工作同是进行的，并且如果我们增加 T1K

接收器后我们再增加 T1 接收器，它就不那么重要：在文字字段内的所有的字符串将一直被强制变为大写。这看起来键盘事件一直在文字组件事件前被激活，并且如果我们需要保留 t2 的字符串原来输入时的样子，我们就必须做一些特别的工作。

T1K 有着其它的一些有趣的活动。我们必须测试 backspace（因为我们现在控制着每一个事件）并执行删除。caret 必须被明确地设置到字段的结尾；否则它不会像我们希望的运行。最后，为了防止原来的字符串被默认的机制所处理，事件必须利用为事件对象而存在的 consume()方法所“耗尽”。这会通知系统停止激活其余特殊事件的事件处理器。

这个例子同样无声地证明了设计内部类的带来的诸多优点。注意下面的内部类：

653 页中程序

t1 和 t2 不属于 T1 的一部分，并且到目前为止它们都是很容易理解的，没有任何的特殊限制。这是因为一个内部类的对象能自动地捕捉一个句柄到外部的创建它的对象那里，因此我们可以处理封装类对象的方法和内容。正像我们看到的，这十分方便（注释⑥）。

⑥：它也解决了“回调”的问题，不必为 Java 加入任何令人恼火的“方法指针”特性。

2. 文本区域

Java 1.1 版中 Text Area 最重要的改变就滚动条。对于 TextArea 的构建器而言，我们可以立即控制 TextArea 是否会拥有滚动条：水平的，垂直的，两者都有或者都没有。这个例子更正了前面 Java 1.0 版 TextArea1.java 程序片，演示了 Java 1.1 版的滚动条构建器：

653-655 页程序

我们发现只能在构造 TextArea 时能够控制滚动条。同样，即使 TE AR 没有滚动条，我们滚动光标也将被制止（可通过运行这个例子中验证这种行为）。

3. 复选框和单选钮

正如早先指出的那样，复选框和单选钮都是同一个类建立的。单选钮和复选框略有不同，它是复选框安置到 CheckboxGroup 中构成的。在其中任一种情况下，有趣的 ItemEvent 事件为我们创建一个 ItemListener 项目接收器。

当处理一组复选框或者单选钮时，我们有一个不错的选择。我们可以创建一个新的内部类去为每个复选框处理事件，或者创建一个内部类判断哪个复选框被单击并注册一个内部类单独的对象为每个复选对象。下面的例子演示了两种方法：

656-657 页程序

ILCheck 拥有当我们增加或者减少复选框时自动调整的优点。当然，我们对单选钮使用这种方法也同样的好。但是，它仅当我们的逻辑足以普遍的支持这种方法时才会被使用。如果声明一个确定的信号——我们将重复利用独立的接收器类，否则我们将结束一串条件语句。

4. 下拉列表

下拉列表在 Java 1.1 版中当一个选择被改变时同样使用 ItemListener 去告知我们：

657-659 页程序

这个程序中没什么特别新颖的东西（除了 Java 1.1 版的 UI 类里少数几个值得关注的缺陷）。

5. 列表

我们消除了 Java 1.0 中 List 设计的一个缺陷，就是 List 不能像我们希望的那样工作：它会与单击在一个列表元素上发生冲突。

659-660 页程序

我们可以注意到在列表项中无需特别的逻辑需要去支持一个单击动作。我们正好像我们在其它地方所做的那样附加上一个接收器。

6. 菜单

为菜单处理事件看起来受益于 Java 1.1 版的事件模型，但 Java 生成菜单的方法常常麻烦并且需要一些手工编写代码。生成菜单的正确方法看起来像资源而不是一些代码。请牢牢记住编程工具会广泛地为我们处理创建的菜单，因此这可以减少我们的痛苦（只要它们会同样处理维护任务！）。另外，我们将发现菜单不支持并且将导致混乱的事件：菜单项使用 ActionListeners（动作接收器），但复选框菜单项使用 ItemListeners（项目接收器）。菜单对象同样能支持 ActionListeners（动作接收器），但通常不那么有用。一般来说，我们会附加接收器到每个菜单项或复选框菜单项，但下面的例子（对先前例子的修改）演示了一个联合捕捉多个菜单组件到一个单独的接收器类的方法。正像我们将看到的，它或许不值得为这而激烈地争论。

661-664 页程序

在我们开始初始化节（由注解“Initialization code:”后的右大括号指明）的前面部分的代码同先前（Java 1.0 版）版本相同。这里我们可以注意到项目接收器和动作接收器被附加在不同的菜单组件上。

Java 1.1 支持“菜单快捷键”，因此我们可以选择一个菜单项目利用键盘替代鼠标。这十分的简单；我们只要使用过载菜单项构建器设置第二个自变量为一个 MenuShortcut（菜单快捷键事件）对象即可。菜单快捷键构建器设置重要的方法，当它按下时不可思议地显示在菜单项上。上面的例子增加了 Control-E 到“Exit”

菜单项中。

我们同样会注意 setActionCommand()的使用。这看似一点陌生因为在各种情况下“action command”完全同菜单组件上的标签一样。为什么不正好使用标签代替可选择的字符串呢？这个难题是国际化的。如果我们重新用其它语言写这个程序，我们只需要改变菜单中的标签，并不审查代码中可能包含新错误的所有逻辑。因此使这对检查文字字符串联合菜单组件的代码而言变得简单容易，当菜单标签能改变时“动作指令”可以不作任何的改变。所有这些代码同“动作指令”一同工作，因此它不会受改变菜单标签的影响。注意在这个程序中，不是所有的菜单组件都被它们的动作指令所审查，因此这些组件都没有它们的动作指令集。

大多数的构建器同前面的一样，将几个调用的异常增加到接收器中。大量的工作发生在接收器里。在前面例子的 BL 中，菜单交替发生。在 ML 中，“寻找 ring”方法被作为动作事件（ActionEvent）的资源并对它进行造型送入菜单项，然后得到动作指令字符串，再通过它去贯穿串联组，当然条件是对它进行声明。这些大多数同前面的一样，但请注意如果“Exit”被选中，通过进入封装类对象的句柄（MenuNew.this）并创建一个 WINDOW_CLOSING 事件，一个新的窗口事件就被创建了。新的事件被分配到封装类对象的 dispatchEvent()方法，然后结束调用 windowsClosing()内部帧的窗口接收器（这个接收器作为一个内部类被创建在 main()里），似乎这是“正常”产生消息的方法。通过这种机制，我们可以在任何情况下迅速处理任何的信息，因此，它非常的强大。

FL 接收器是很简单尽管它能处理特殊菜单的所有不同的特色。如果我们的逻辑十分的简单明了，这种方法对我们就很有用处，但通常，我们使用这种方法时需要与 FooL，BarL 和 BazL 一道使用，它们每个都附加到一个单独的菜单组件上，因此必然无需测试逻辑，并且使我们正确地辨识出谁调用了接收器。这种方法产生了大量的类，内部代码趋向于变得小巧和处理起来简单、安全。

7. 对话框

在这个例子里直接重写了早期的 ToeTest.java 程序。在这个新的版本里，任何事件都被安放进一个内部类中。虽然这完全消除了需要记录产生的任何类的麻烦，作为 ToeTest.java 的一个例子，它能使内部类的概念变得不那遥远。在这点，内嵌类被嵌套达四层之深！我们需要的这种设计决定了内部类的优点是否值得增加更加复杂的事物。另外，当我们创建一个非静态的内部类时，我们将捆绑非静态类到它周围的类上。有时，单独的类可以更容易地被复用。

666-668 页程序

由于“静态”的东西只能位于类的外部一级，所以内部类不可能拥有静态数据或者静态内部类。

8. 文件对话框

这个例子是直接用新事件模型对 FileDialogTest.java 修改而来。

668-670,页程序

如果所有的改变是这样的容易那将有多棒，但至少它们已足够容易，并且我

们的代码已受益于这改进的可读性上。

13.16.5 动态绑定事件

新 AWT 事件模型给我们带来的一个好处就是灵活性。在老的模型中我们被迫为我们的程序动作艰难地编写代码。但新的模型我们可以用单一方法调用增加和删除事件动作。下面的例子证明了这一点：

670-672 页程序

这个例子采取的新手法包括：

(1) 在每个按钮上附着不少于一个的接收器。通常，组件把事件作为多造型处理，这意味着我们可以为单个事件注册许多接收器。当在特殊的组件中一个事件作为单一造型被处理时，我们会得到 TooManyListenersException（即太多接收器异常）。

(2) 程序执行期间，接收器动态地被从按钮 B2 中增加和删除。增加用我们前面见到过的方法完成，但每个组件同样有一个 removeXXXListener()（删除 XXX 接收器）方法来删除各种类型的接收器。

这种灵活性为我们的编程提供了更强大的能力。

我们注意到事件接收器不能保证在命令他们被增加时可被调用（虽然事实上大部分的执行工作都是用这种方法完成的）。

13.16.6 将事务逻辑与 UI 逻辑区分开

一般而言，我们需要设计我们的类如此以至于每一类做“一件事”。当涉及用户接口代码时就更显得尤为重要，因为它很容易地封装“您要做什么”和“怎样显示它”。这种有效的配合防止了代码的重复使用。更不用说它令人满意的从 GUI 中区分出我们的“事物逻辑”。使用这种方法，我们可以不仅仅更容易地重复使用事物逻辑，它同样可以更容易地重复使用 GUI。

其它的争议是“动作对象”存在的完成分离机器的多层次系统。动作主要的定位规则允许所有新事件修改后立刻生效，并且这是如此一个引人注目的设置系统的方法。但是这些动作对象可以被在一些不同的应用程序使用并且因此不会被一些特殊的显示模式所约束。它们会合理地执行动作操作并且没有多余的事件。

下面的例子演示了从 GUI 代码中多么地轻松的区分事物逻辑：

673-674 页程序

可以看到，事物逻辑是一个直接完成它的操作而不需要提示并且可以在 GUI 环境下使用的类。它正适合它的工作。区分动作记录了所有 UI 的详细资料，并且它只通过它的公共接口与事物逻辑交流。所有的操作围绕中心通过 UI 和事物逻辑对象来回获取信息。因此区分，轮流做它的工作。因为区分中只知道它同事物逻辑对象对话（也就是说，它没有高度的结合），它可以被强迫同其它类型的对象对话而没有更多的烦恼。

思考从事物逻辑中区分 UI 的条件，同样思考当我们调整传统的 Java 代码使它运行时，怎样使它更易存活。

13.16.7 推荐编码方法

内部类是新的事件模型，并且事实上旧的事件模型连同新库的特征都被它好的支持，依赖老式的编程方法无疑增加了一个新的混乱的因素。现在有更多不同的方法为我们编写讨厌的代码。凑巧的是，这种代码显现在本书中和程序样本中，并且甚至在文件和程序样本中同 SUN 公司区别开来。在这一节中，我们将看到一些关于我们会和不会运行新 AWT 的争执，并由向我们展示除了可以原谅的情况，我们可以随时使用接收器类去解决我们的事件处理需要来结束。因为这种方法同样是最简单和最清晰的方法，它将会对我们学习它构成有效的帮助。

在看到任何事以前，我们知道尽管 Java 1.1 向后兼容 Java 1.0（也就是说，我们可以在 1.1 中编译和运行 1.0 的程序），但我们并不能在同一个程序里混合事件模型。换言之，当我们试图集成老的代码到一个新的程序中时，我们不能使用老式的 action()方法在同一个程序中，因此我们必须决定是否对新程序使用老的，难以维护的方法或者升级老的代码。这不会有太多的竞争因为新的方法对老的方法而言是如此的优秀。

1. 准则：运行它的好方法

为了给我们一些事物来进行比较，这儿有一个程序例子演示向我们推荐的方法。到现在它会变得相当的熟悉和舒适。

675-676 页程序

这是颇有点微不足道的：每个按钮有它自己的印出一些事物到控制台的接收器。但请注意在整个程序中这不是一个条件语句，或者是一些表示“我想要知道怎样使事件发生”的语句。每块代码都与运行有关，而不是类型检验。也就是说，这是最好的编写我们的代码的方法；不仅仅是它更易使我们理解概念，至少是使我们更易阅读和维护。剪切和粘贴到新的程序是同样如此的容易。

2. 将主类作为接收器实现

第一个坏主意是一个通常的和推荐的方法。这使得主类（有代表性的是程序片或帧，但它能变成一些类）执行各种不同的接收器。下面是一个例子：

677-678 页程序

这样做的用途显示在下述三行里：

```
addWindowListener(this);
b1.addActionListener(this);
b2.addActionListener(this);
```

因为 Badidea1 执行动作接收器和窗中接收器，这些程序行当然可以接受，并且如果我们一直坚持设法使少量的类去减少服务器检索期间的程序片载入的作法，它看起来变成一个不错的主意。但是：

(1) Java 1.1 版支持 JAR 文件，因此所有我们的文件可以被放置到一个单一的压缩的 JAR 文件中，只需要一次服务器检索。我们不再需要为 Internet 效率而减

少类的数量。

(2) 上面的代码的组件更加的少，因此它难以抓住和粘贴。注意我们必须不仅要执行各种各样的接口为我们的主类，但在 actionPerformed()方法中，我们利用一串条件语句测试哪个动作被完成了。不仅仅是这个状态倒退，远离接收器模型，除此之外，我们不能简单地重复使用 actionPerformed()方法因为它是指定为这个特殊的应用程序使用的。将这个程序例子与 GoodIdea.java 进行比较，我们可以正好捕捉一个接收器类并粘贴它和最小的焦急到任何地方。另外我们可以为一个单独的事件注册多个接收器类，允许甚至更多的模块在每个接收器类在每个接收器中运行。

3. 方法的混合

第二个 bad idea 混合了两种方法：使用内嵌接收器类，但同样执行一个或更多的接收器接口以作为主类的一部分。这种方法无需在书中和文件中进行解释，而且我可以臆测到 Java 开发者认为他们必须为不同的目的而采取不同的方法。但我们却不必——在我们编程时，我们或许可能会倾向于使用内嵌接收器类。

679-680 页程序

因为 actionPerformed()动作完成方法同主类紧密地结合，所以难以复用代码。它的代码读起来同样是凌乱和令人厌烦的，远远超过了内部类方法。不合理的是，我们不得不在 Java 1.1 版中为事件使用那些老的思路。

4. 继承一个组件

创建一个新类型的组件时，在运行事件的老方法中，我们会经常看到不同的地方发生了变化。这里有一个程序例子来演示这种新的工作方法：

680-685 页程序

这个程序例子同样证明了各种各样的发现和显示关于它们的信息的事件。这种显示是一种集中显示信息的方法。一组字符串去获取关于每种类型的事件的信息，并且 show()方法对任何图像对象都设置了一个句柄，我们采用并直接地写在外观代码上。这种设计是有意的被某种事件重复使用。

激活面板代表了这种新型的组件。它是一个底部有一个按钮的彩色的面板，并且它由利用接收器类为每一个单独的事件来引发捕捉所有发生在它之上的事件，除了那些在激活面板过载的老式的 processEvent()方法（注意它应该同样调用 super.processEvent()）。利用这种方法的唯一理由是它捕捉发生的每一个事件，因此我们可以观察持续发生的每一事件。processEvent()方法没有更多的展示代表每个事件的字符串，否则它会不得不使用一串条件语句去寻找事件。在其它方面，内嵌接收类早已清晰地知道被发现的事件。（假定我们注册它们到组件，我们不需要任何的控件的逻辑，这将成为我们的目的。）因此，它们不会去检查任何事件；这些事件正好做它们的原材料。

每个接收器修改显示字符串和它的指定事件，并且调用重画方法 repaint()因此将显示这个字符串。我们同样能注意到一个通常能消除闪烁的秘诀：

public void update(Graphics g) {

paint(g);
}

我们不会始终需要过载 update()，但如果我们写下一些闪烁的程序，并运行它。默认的最新版本的清除背景然后调用 paint()方法重新画出一些图画。这个清除动作通常会产生闪烁，但是不必要的，因为 paint()重画了整个的外观。

我们可以看到许多的接收器——但是，对接收器输入检查指令，但我们却不能接收任何组件不支持的事件。（不像 BadTechnuque.java 那样我们能时时刻刻看到）。

试验这个程序是十分的有教育意义的，因为我们学习了许多的关于在 Java 中事件发生的方法。一则它展示了大多数开窗口的系统中设计上的瑕疵：它相当的难以去单击和释放鼠标，除非移动它，并且当我们实际上正试图用鼠标单击在某物体上时开窗口的会常常认为我们是在拖动。一个解决这个问题的方案是使用 mousePressed() 鼠标按下方法和 mouseReleased() 鼠标释放方法去代替 mouseClicked()鼠标单击方法，然后判断是否去调用我们自己的以时间和 4 个像素的鼠标滞后作用的“mouseReallyClicked()真实的鼠标单击”方法。

5. 蹩脚的组件继承

另一种做法是调用 enableEvent()方法，并将与希望控制的事件对应的模型传递给它（许多参考书中都曾提及这种做法）。这样做会造成那些事件被发送至老式方法（尽管它们对 Java 1.1 来说是新的），并采用象 processFocusEvent()这样的名字。也必须要记住调用基础类版本。下面是它看起来的样子。

686-692 页程序

的确，它能够工作。但却实在太蹩脚，而且很难编写、阅读、调试、维护以及再生。既然如此，为什么还不使用内部接收器类呢？

13.17 Java 1.1 用户接口 API

Java 1.1 版同样增加了一些重要的新功能，包括焦点遍历，桌面色彩访问，打印“沙箱内”及早期的剪贴板支持。

焦点遍历十分的简单，因为它显然存在于 AWT 库里的组件并且我们不必为使它工作而去做任何事。如果我们制造我们自己组件并且想使它们去处理焦点遍历，我们过载 isFocusTraversable()以使它返回真值。如果我们想在一个鼠标单击上捕捉键盘焦点，我们可以捕捉鼠标按下事件并且调用 requestFocus()需求焦点方法。

13.17.1 桌面颜色

利用桌面颜色，我们可知道当前用户桌面都有哪些颜色选择。这样一来，就可在必要的时候通过自己的程序来运用那些颜色。颜色都会得以自动初始化，并置于 SystemColor 的 static 成员中，所以要做的唯一事情就是读取自己感兴趣的成员。各种名字的意义是不言而喻的：desktop，activeCaption， activeCaptionText，activeCaptionBorder ， inactiveCaption ， inactiveCaptionText ， inactiveCaptionBorder，window，windowBorder，windowText，menu，menuText，text， textText， textHighlight， textHighlightText，textInactiveText，control，

controlText， controlHighlight， controlLtHighlight， controlShadow， controlDkShadow， scrollbar， info（用于帮助）以及 infoText（用于帮助文字）。

13.17.2 打印

非常不幸，打印时没有多少事情是可以自动进行的。相反，为完成打印，我们必须经历大量机械的、非 OO（面向对象）的步骤。但打印一个图形化的组件时，可能多少有点儿自动化的意思：默认情况下，print()方法会调用 paint()来完成自己的工作。大多数时候这都已经足够了，但假如还想做一些特别的事情，就必须知道页面的几何尺寸。

下面这个例子同时演示了文字和图形的打印，以及打印图形时可以采取的不同方法。此外，它也对打印支持进行了测试：

693-698 页程序

这个程序允许我们从一个选择列表框中选择字体（并且我们会注意到很多有用的字体在 Java 1.1 版中一直受到严格的限制，我们没有任何可以利用的优秀字体安装在我们的机器上）。它使用这些字体去打出粗体，斜体和不同大小的文字。另外，一个新型组件调用过的绘图被创建，以用来示范图形。当打印图形时，绘图拥有的 ring 将显示在屏幕上和打印在纸上，并且这三个衍生类 Plot1，Plot2，Plot3 用不同的方法去完成任务以便我们可以看到我们选择的事物。同样，我们也能在一个绘图中改变一些 ring——这很有趣，因为它证明了 Java 1.1 版中打印的脆弱。在我的系统里，当 ring 计数显示“too high”（究竟这是什么意思？）时，打印机给出错误信息并且不能正确地工作，而当计数给出“low enough”信息时，打印机又能工作得很好。我们也会注意到，当打印到看起来实际大小不相符的纸时页面的大小便产生了。这些特点可能被装入到将来发行的 Java 中，我们可以使用这个程序来测试它。

这个程序为促进重复使用，不论何时都可以封装功能到内部类中。例如，不论何时我想开始打印工作（不论图形或文字），我必须创建一个 PrintJob 打印工作对象，该对象拥有它自己的连同页面宽度和高度的图形对象。创建的 PrintJob 打印工作对象和提取的页面尺寸一起被封装进 PrintData class 打印类中。

1. 打印文字

打印文字的概念简单明了：我们选择一种字体和大小，决定字符串在页面上存在的位置，并且使用 Graphics.drawSrting()方法在页面上画出字符串就行了。这意味着，不管怎样我们必须精确地计算每行字符串在页面上存在的位置并确定字符串不会超出页面底部或者同其它行冲突。如果我们想进行字处理，我们将进行的工作与我们很相配。ChangeFont 封装进少量从一种字体到其它的字体的变更方法并自动地创建一个新字体对象和我们想要的字体，款式（粗体和斜体——目前还不支持下划线、空心等）以及点阵大小。它同样会简单地计算字符串的宽度和高度。当我们按下“Print text”按钮时，TBL 接收器被激活。我们可以注意到它通过反复创建 ChangeFont 对象和调用 drawString()来在计算出的位置打印出字符串。注意是否这些计算产生预期的结果。（我使用的版本没有出错。）

2. 打印图形

按下“Print graphics”按钮时，GBL 接收器会被激活。我们需要打印时，创建的 PrintData 对象初始化，然后我们简单地为这个组件调用 print()打印方法。为强制打印，我们必须为图形对象调用 dispose()处理方法，并且为 PrintData 对象调用 end()结束方法（或改变为为 PrintJob 调用 end()结束方法。）

这种工作在绘图对象中继续。我们可以看到基础类绘图是很简单的——它扩展画布并且包括一个中断调用 ring 来指明多少个集中的 ring 需要画在这个特殊的画布上。这三个衍生类展示了可达到一个目的的不同的方法：画在屏幕上和打印的页面上。

Plot1 采用最简单的编程方法：忽略绘画和打印的不同，并且过载 paint()绘画方法。使用这种工作方法的原因是默认的 print()打印方法简单地改变工作方法转而调用 Paint()。但是，我们会注意到输出的尺寸依赖于屏幕上画布的大小，因为宽度和高度都是在调用 Canvas.getSize()方法时决定是，所以这是合理的。如果我们图像的尺寸一值都是固定不变的，其它的情况都可接受。当画出的外观的大小如此的重要时，我们必须深入了解的尺寸大小的重要性。不凑巧的是，就像我们将在 Plot2 中看到的一样，这种方法变得很棘手。因为一些我们不知道的好的理由，我们不能简单地要求图形对象以它自己的大小画出外观。这将使整个的处理工作变得十分的优良。相反，如果我们打印而不是绘画，我们必须利用 RTTI instanceof 关键字（在本书 11 章中有相应描述）来测试 PrintGrapics，然后下溯造型并调用这独特的 PrintGraphics 方法：getPrintJob()方法。现在我们拥有 PrintJob 的句柄并且我们可以发现纸张的高度和宽度。这是一种 hacky 的方法，但也许这对它来说是合理的理由。（在其它方面，到如今我们看到一些其它的库设计，因此，我们可能会得到设计者们的想法。）

我们可以注意到 Plot2 中的 paint()绘画方法对打印和绘图的可能性进行审查。但是因为当打印时 Print()方法将被调用，那么为什么不使用那种方法呢？这种方法同样也在 Plot3 中也被使用，并且它消除了对 instanceof 使用的需求，因为在 Print()方法中我们可以假设我们能对一个 PrintGraphics 对象造型。这样也不坏。这种情况被放置公共绘画代码到一个分离的 doGraphics()方法的办法所改进。

2. 在程序片内运行帧

如果我们想在一个程序片中打印会怎以样呢？很好，为了打印任何事物我们必须通过工具组件对象的 getPrintJob()方法拥有一个 PrintJob 对象，设置唯一的一个帧对象而不是一个程序片对象。于是它似乎可能从一个应用程序中打印，而不是从一个程序片中打印。但是，它变为我们可以从一个程序片中创建一个帧（相反的到目前为止，我在程序片或应用程序例子中所做的，都可以生成程序片并安放帧。）。这是一个很有用的技术，因为它允许我们在程序片中使用一些应用程序（只要它们不妨碍程序片的安全）。但是，当应用程序窗口在程序片中出现时，我们会注意到 WEB 浏览器插入一些警告在它上面，其中一些产生“Warning:Applet Window.（警告：程序片窗口）”的字样。

我们会看到这种技术十分直接的安放一个帧到程序片中。唯一的事是当用户关闭它时我们必须增加帧的代码（代替调用 System.exit()）：

700-701 页程序

伴随 Java 1.1 版的打印支持功能而来的是一些混乱。一些宣传似乎声明我们

能在一个程序片中打印。但 Java 的安全系统包含了一个特点，可停止一个正在初始化打印工作的程序片，初始化程序片需要通过一个 Web 浏览器或程序片浏览器来进行。在写作这本书时，这看起来像留下了一个未定的争议。当我在 WEB 浏览器中运行这个程序时，printdemo（打印样本）窗口正好出现，但它却根本不能从浏览器中打印。

13.17.3 剪贴板

Java 1.1 对系统剪贴板提供有限的操作支持（在 Java.awt.datatransfer package 里）。我们可以将字符串作这文字对象复制到剪贴板中，并且我们可以从剪贴板中粘贴文字到字符中对角中。当然，剪贴板被设计来容纳各种类型的数据，存在于剪贴板上的数据通过程序运行剪切和粘贴进入到程序中。虽然剪切板目前只支持字符串数据，Java 的剪切板 API 通过“特色”概念提供了良好的可扩展性。当数据从剪贴板中出来时，它拥有一个相关的特色集，这个特色集可以被修改（例如，一个图形可以被表示成一些字符串或者一幅图像）并且我们会注意到如果特殊的剪贴板数据支持这种特色，我们会对此十分的感兴趣。

下面的程序简单地对 TextArea 中的字符串数据进行剪切，复制，粘贴的操作做了示范。我们将注意到的是我们需要按照剪切、复制和粘贴的顺序进行工作。但如果我们看见一些其它程序中的 TextField 或者 TextArea，我们会发现它们同样也自动地支持剪贴板的操作顺序。程序中简单地增加了剪贴板的程序化控制，如果我们想用它来捕捉剪贴板上的文字到一些非文字组件中就可以使用这种技术。

701-703 页程序

创建和增加菜单及 TextArea 到如今似乎已变成一种单调的活动。这与通过工具组件创建的剪贴板字段 clipbd 有很大的区别。

所有的动作都安置在接收器中。CopyL 和 Cupl 接收器同样除了最后的 CutL 线以外删除被复制的线。特殊的两条线是 StringSelection 对象从字符串从创建并调用 StringSelection 的 setContents()方法。说得更准确些，就是放一个字符串到剪切板上。

在 PasteL 中，数据被剪贴板利用 getContents()进行分解。任何返回的对象都是可移动的匿名的，并且我们并不真正地知道它里面包含了什么。有一种发现的方法是调用 getTransferDateFlavors()，返回一个 DataFlavor 对象数组，表明特殊对象支持这种特点。我们同样能要求它通过我们感兴趣的特点直接地使用 IsDataFlavorSupported()。但是在这里使用一种大胆的方法：调用 getTransferData（ ）方法，假设里面的内容支持字符串特色，并且它不是个被分类在异常处理器中的难题 。

在将来，我们希望更多的数据特色能够被支持。

13.18 可视编程和 Beans

迄今为止，我们已看到 Java 对创建可重复使用的代码片工作而言是多么的有价值。“最大限度地可重复使用”的代码单元拥有类，因为它包含一个紧密结合在一起的单元特性（字段）和单元动作（方法），它们可以直接经过混合或通过继承被重复使用。

继承和多形态性是面向对象编程的精华，但在大多数情况下当我们创建一个应用程序时，我们真正最想要的恰恰是我们最需要的组件。我们希望在我们的设计中设置这些部件就像电子工程师在电路板上创造集成电路块一样（在使用 Java 的情况下，就是放到 WEB 页面上）。这似乎会成为加快这种“模块集合”编制程序方法的发展。

“可视化编程”最早的成功——非常的成功——要归功于微软公司的 Visual Basic（VB，可视化 Basic 语言），接下来的第二代是 Borland 公司 Delphi（一种客户/服务器数据库应用程序开发工具，也是 Java Beans 设计的主要灵感）。这些编程工具的组件的像征就是可视化，这是不容置疑的，因为它们通常展示一些类型的可视化组件，例如：一个按惯或一个 TextField。事实上，可视化通常表现为组件可以非常精确地访问运行中程序。因此可视化编程方法的一部分包含从一个调色盘从拖放一个组件并将它放置到我们的窗体中。应用程序创建工具像我们所做的一样编写程序代码，该代码将导致正在运行的程序中的组件被创建。

简单地拖放组件到一个窗体中通常不足以构成一个完整的程序。一般情况下，我们需要改变组件的特性，例如组件的色彩，组件的文字，组件连结的数据库，等等。特性可以参照属性在编程时进行修改。我们可以在应用程序构建工具中巧妙处置我们组件的属性，并且当我们创建程序时，构建数据被保存下来，所以当该程序被启动时，数据能被重新恢复。

到如今，我们可能习惯于使用对象的多个特性，这也是一个动作集合。在设计时，可视化组件的动作可由事件部分地代表，意味着“任何事件都可以发生在组件上”。通常，由我们决定想发生的事件，当一个事件发生时，对所发生的事件连接代码。

这是关键性的部分：应用程序构建工具可以动态地询问组件（利用映象）以发现组件支持的事件和属件。一旦它知道它们的状态，应用程序构建工具就可以显示组件的属性并允许我们修改它们的属性（当我们构建程序时，保存它们的状态），并且也显示这些事件。一般而言，我们做一些事件像双击一个事件以及应用程序构建工具创建一个代码并连接到事件上。当事件发生时，我们不得不编写执行代码。应用程序构建工具累计为我们做了大量的工作。结果我们可以注意到程序看起来像它所假定的那样运行，并且依赖应用程序构建工具去为我们管理连接的详细资料。可视化的编程工具如此成功的原因是它们明显加快构建的应用程序的处理过程——当然，用户接口作为应用程序的一部分同样的好。

13.18.1 什么是 Bean

在经细节处理后，一个组件在类中被独特的具体化，真正地成为一块代码。关键的争议在于应用程序构建工具发现组件的属性和事件能力。为了创建一个 VB 组件，程序开发者不得不编写正确的同时也是复杂烦琐的代码片，接下来由某些协议去展现它们的事件和属性。Delphi 是第二代的可视化编程工具并且这种开发语言主动地围绕可视化编程来设计因此它更容易去创建一个可视化组件。但是，Java 带来了可视化的创作组件做为 Java Beans 最高级的“装备”，因为一个 Bean 就是一个类。我们不必再为制造任何的 Bean 而编写一些特殊的代码或者使用特殊的编程语言。事实上，我们唯一需要做的是略微地修改我们对我们方法命名的办法。方法名通知应用程序构建工具是否是一个属性，一个事件或是一个普通的方法。

在 Java 的文件中，命名规则被错误地曲解为“设计范式”。这十分的不幸，

因为设计范式（参见第 16 章）惹来不少的麻烦。命名规则不是设计范式，它是相当的简单：

(1) 因为属性被命名为 xxx，我们代表性的创建两个方法：getXxx()和 setXxx()。注意 get 或 set 后的第一个字母小写以产生属性名。“get”和“set”方法产生同样类型的自变量。“set”和“get”的属性名和类型名之间没有关系。

(2) 对于布尔逻辑型属性，我们可以使用上面的“get”和“set”方法，但我们也可以用“is”代替“ get”。

(3) Bean 的普通方法不适合上面的命名规则，但它们是公用的。

4.对于事件，我们使用“listener（接收器）”方法。这种方法完全同我们看到过的方法相同：(addFooBarListener(FooBarListener) 和 removeFooBarListener(FooBarListener)方法用来处理 FooBar 事件。大多数时候内建的事件和接收器会满足我们的需要，但我们可以创建自己的事件和接收器接口。

上面的第一点回答了一个关于我们可能注意到的从 Java 1.0 到 Java 1.1 的改变的问题：一些方法的名字太过于短小，显然改写名字毫无意义。现在我们可以看到为了制造 Bean 中的特殊的组件，大多数的这些修改不得不适合于“get”和“set”命名规则。

现在，我们已经可以利用上面的这些指导方针去创建一个简单的 Bean：

706-707 页程序

首先，我们可看到 Bean 就是一个类。通常，所有我们的字段会被作为专用，并且可以接近的唯一办法是通过方法。紧接着的是命名规则，属性是 jump，color，jumper，spots（注意这些修改是在第一个字母在属性名的情况下进行的）。虽然内部确定的名字同最早的三个例子的属性名一样，在 jumper 中我们可以看到属性名不会强迫我们使用任何特殊的内部可变的名字（或者，真的拥有一些内部的可变的属性名）。

Bean 事件的句柄是 ActionEvent 和 KeyEvent，这是根据有关接收器的“add”和“remove”命名方法得出的。最后我们可以注意到普通的方法 croak()一直是 Bean 的一部分，仅仅是因为它是一个公共的方法，而不是因为它符合一些命名规则。

13.18.2 用 Introspector 提取 BeanInfo

当我们拖放一个 Bean 的调色板并将它放入到窗体中时，一个 Bean 的最关键的部分的规则发生了。应用程序构建工具必须可以创建 Bean（如果它是默认的构建器的话，它就可以做）然后，在此范围外访问 Bean 的源代码，提取所有的必要的信息以创立属性表和事件处理器。

解决方案的一部分在 11 章结尾部分已经显现出来：Java 1.1 版的映象允许一个匿名类的所有方法被发现。这完美地解决了 Bean 的难题而无需我们使用一些特殊的语言关键字像在其它的可视化编程语言中所需要的那样。事实上，一个主要的原因是映象增加到 Java 1.1 版中以支持 Beans（尽管映象同样支持对象串联和远程方法调用）。因为我们可能希望应用程序构建工具的开发者将不得不映象每个 Bean 并且通过它们的方法搜索以找到 Bean 的属性和事件。

这当然是可能的，但是 Java 的研制者们希望为每个使用它的用户提供一个标

准的接口，而不仅仅是使 Bean 更为简单易用，不过他们也同样提供了一个创建更复杂的 Bean 的标准方法。这个接口就是 Introspector 类，在这个类中最重要的方法静态的 getBeanInfo()。我们通过一个类处理这个方法并且 getBeanInfo()方法全面地对类进行查询，返回一个我们可以进行详细研究以发现其属性、方法和事件的 BeanInfo 对象。

通常我们不会留意这样的一些事物——我们可能会使用我们大多数的现成的 Bean，并且我们不需要了解所有的在底层运行的技术细节。我们会简单地拖放我们的 Bean 到我们窗体中，然后配置它们的属性并且为事件编写处理器。无论如何它都是一个有趣的并且是有教育意义的使用 Introspector 来显示关于 Bean 信息的练习，好啦，闲话少说，这里有一个工具请运行它（我们可以在 forgbean 子目录中找到它）：

709-711 页程序

BeanDumper.dump()是一个可以做任何工作的方法。首先它试图创建一个 BeanInfo 对象，如果成功地调用 BeanInfo 的方法，就产生关于属性、方法和事件的信息。在 Introspector.getBeanInfo()中，我们会注意到有一个另外的自变量。由它来通知 Introspector 访问继承体系的地点。在这种情况下，它在分析所有对象方法前停下，因为我们对看到那些并不感兴趣。

因为属性，getPropertyDescriptors()返回一组的属性描述符号。对于每个描述符号我们可以调用 getPropertyType()方法彻底的通过属性方法发现类的对象。这时，我们可以用 getName()方法得到每个属性的假名（从方法名中提取），getname()方法用 getReadMethod()和 getWriteMethod()完成读和写的操作。最后的两个方法返回一个可以真正地用来调用在对象上调用相应的方法方法对象（这是映象的一部分）。对于公共方法（包括属性方法），getMethodDescriptors(）返回一组方法描述字符。每一个我们都可以得到相当的方法对象并可以显示出它们的名字。

对于事件而言，getEventSetDescriptors()返回一组事件描述字符。它们中的每一个都可以被查询以找出接收器的类，接收器类的方法以及增加和删除接收器的方法。BeanDumper 程序打印出所有的这些信息。

如果我们调用 BeanDumper 在 Frog 类中，就像这样：

java BeanDumper frogbean.Frog

它的输出结果如下（已删除这儿不需要的额外细节）：

712-713 页程序

这个结果揭示出了 Introspector 在从我们的 Bean 产生一个 BeanInfo 对象时看到的大部分内容。我们可注意到属性的类型和它们的名字是相互独立的。请注意小写的属性名。（当属性名开头在一行中有超过不止的大写字母，这一次程序就不会被执行。）并且请记住我们在这里所见到的方法名（例如读和与方法）真正地从一个可以被用来在对象中调用相关方法的方法对象中产生。

通用方法列表包含了不相关的事件或者属性，例如 croak()。列表中所有的方法都是我们可以有计划的为 Bean 调用，并且应用程序构建工具可以选择列出所有的方法，当我们调用方法时，减轻我们的任务。

最后，我们可以看到事件在接收器中完全地分析研究它的方法、增加和减少

接收器的方法。基本上，一旦我们拥有 BeanInfo，我们就可以找出对 Bean 来说任何重要的事物。我们同样可以为 Bean 调用方法，即使我们除了对象外没有任何其它的信息（此外，这也是映象的特点）。

13.18.3 一个更复杂的 Bean

接下的程序例子稍微复杂一些，尽管这没有什么价值。这个程序是一张不论鼠标何时移动都围绕它画一个小圆的 弧 5 蔽颐前聪率蟊昙　保躲谄聊恢醒胂允疽桓蛰帧奥 ang!”，并且一个动作接收器被激活。画布。当按下鼠标键时，我们可以改变的属性是圆的大小，除此之外还有被显示文字的色彩，大小，内容。BangBean 同样拥有它自己的 addActionListener()和 removeActionListener()方法，因此我们可以附上自己的当用户单击在 BangBean 上时会被激活的接收器。这样，我们将能够确认可支持的属性和事件：

714-717 页程序

最重要的是我们会注意到 BangBean 执行了这种串联化的接口。这意味着应用程序构建工具可以在程序设计者调整完属性值后利用串联为 BangBean 贮藏所有的信息。当 Bean 作为运行的应用程序的一部分被创建时，那些被贮藏的属性被重新恢复，因此我们可以正确地得到我们的设计。

我们能看到通常同 Bean 一起运行的所有的字段都是专用的——允许只能通过方法来访问，通常利用“属性”结构。

当我们注视着 addActionListener()的签名时，我们会注意到它可以产生出一个 TooManyListenerException（太多接收器异常）。这个异常指明它是一个单一的类型的，意味着当事件发生时，它只能通知一个接收器。一般情况下，我们会使用具有多种类型的事件，以便一个事件通知多个的接收器。但是，那样会陷入直到下一章我们才能准备好的结局中，因此这些内容会被重新回顾（下一个标题是“Java Beans 的重新回顾”）。单一类型的事件回避了这个难题。

当我们按下鼠标键时，文字被安入 BangBean 中间，并且如果动作接收器字段存在，它的 actionPerformed()方法就被调用，创建一个新的 ActionEvent 对象在处理过程中。无论何时鼠标移动，它的新座标将被捕捉，并且画布会被重画（像我们所看到的抹去一些画布上的文字）。

main()方法增加了允许我们从命令行中测试程序的功能。当一个 Bean 在一个开发环境中，main()方法不会被使用，但拥有它是绝对有益的，因为它提供了快捷的测试能力。无论何时一个 ActionEvent 发生，main()方法都将创建了一个帧并安置了一个 BangBean 在它里面，还在 BangBean 中附上了一个简单的动作接收器以打印到控制台。当然，一般来说应用程序构建工具将创建大多数的 Bean 的代码。当我们通过 BeanDumper 或者安放 BangBean 到一个可激活 Bean 的开发环境中去运行 BangBean 时，我们会注意到会有很多额外的属性和动作明显超过了上面的代码。那是因为 BangBean 从画布中继承，并且画布就是一个 Bean，因此我们看到它的属性和事件同样的合适。

13.18.4 Bean 的封装

在我们可以安放一个 Bean 到一个可激活 Bean 的可视化构建工具中前，它必须被放入到标准的 Bean 容器里，也就是包含 Bean 类和一个表示“这是一个 Bean”

的清单文件的 JAR（Java ARchive，Java 文件）文件中。清单文件是一个简单的紧随事件结构的文本文件。对于 BangBean 而言，清单文件就像下面这样：

```
Manifest-Version: 1.0

Name: bangbean/BangBean.class
Java-Bean: True
```

其中，第一行指出清单文件结构的版本，这是 SUN 公司在很久以前公布的版本。第二行（空行忽略）对文件命名为 BangBean.class。第三行表示“这个文件是一个 Bean”。没有第三行，程序构建工具不会将类作为一个 Bean 来认可。

唯一难以处理的部分是我们必须肯定“Name:”字段中的路径是正确的。如果我们回顾 BangBean.java，我们会看到它在 package bangbean（因为存放类路径的子目录称为“bangbean”）中，并且这个名字在清单文件中必须包括封装的信息。另外，我们必须安放清单文件在我们封装路径的根目录上，在这个例子中意味着安放文件在 bangbean 子目录中。这之后，我们必须从同一目录中调用 Jar 来作为清单文件，如下所示：

```
jar cfm BangBean.jar BangBean.mf bangbean
```

这个例子假定我们想产生一个名为 BangBean.jar 的文件并且我们将清单放到一个称为 BangBean.mf 文件中。

我们可能会想“当我编译 BangBean.java 时，产生的其它类会怎么样呢？”哦，它们会在 bangbean 子目录中被中止，并且我们会注意到上面 jar 命令行的最后一个自变量就是 bangbean 子目录。当我们给 jar 子目录名时，它封装整个的子目录到 jar 文件中（在这个例子中，包括 BangBean.java 的源代码文件——对于我们自己的 Bean 我们可能不会去选择包含源代码文件。）另外，如果我们改变主意，解开打包的 JAR 文件，我们会发现我们清单文件并不在里面，但 jar 创建了它自己的清单文件（部分根据我们的文件），称为 MAINFEST.MF 并且安放它到 META-INF 子目录中（代表“meta-information”）。如果我们打开这个清单文件，我们同样会注意到 jar 为每个文件加入数字签名信息，其结构如下：

```
Digest-Algorithms: SHA MD5
SHA-Digest: pDpEAG9NaeCx8aFtqPI4udSX/O0=
MD5-Digest: O4NcS1hE3Smnzlp2hj6qeg==
```

一般来说，我们不必担心这些，如果我们要做一些修改，可以修改我们的原始的清单文件并且重新调用 jar 以为我们的 Bean 创建了一个新的 JAR 文件。我们同样也可以简单地通过增加其它的 Bean 的信息到我们清单文件来增加它们到 JAR 文件中。

值得注意的是我们或许需要安放每个 Bean 到它自己的子目录中，因为当我们创建一个 JAR 文件时，分配 JAR 应用目录名并且 JAR 放置子目录中的任何文件到 JAR 文件中。我们可以看到 Frog 和 BangBean 都在它们自己的子目录中。

一旦我们将我们的 Bean 正确地放入一个 JAR 文件中，我们就可以携带它到一个可以激活 Bean 的编程环境中使用。使用这种方法，我们可以从一种工具到另一种工具间交替变换，但 SUN 公司为 Java Beans 提供了免费高效的测试工具在它们的“Bean Development Kit，Bean 开发工具”（BDK）称为“beanbox”。（我们可以从 www.javasoft.com 处下载。）在我们启动 beanbox 前，放置我们的 Bean

到 beanbox 中，复制 JAR 文件到 BDK 的“jars”子目录中。

13.18.5 更复杂的 Bean 支持

我们可以看到创建一个 Bean 显然多么的简单。在程序设计中我们几乎不受到任何的限制。Java Bean 的设计提供了一个简单的输入点，这样可以提高到更复杂的层次上。这些高层次的问题超出了这本书所要讨论的范围，但它们会在此做简要的介绍。我们可以在 http://java.sun.com/beans 上找到更多的详细资料。

我们增加更加复杂的程序和它的属性到一个位置。上面的例子显示一个独特的属性，当然它也可能代表一个数组的属性。这称为索引属性。我们简单地提供一个相应的方法（再者有一个方法名的命名规则）并且 Introspector 认可索引属性，因此我们的应用程序构建工具相应的处理。

属性可以被捆绑，这意味着它们将通过 PropertyChangeEvent 通知其它的对象。其它的对象可以随后根据对 Bean 的改变选择修改它们自己。

属性可以被束缚，这意味着其它的对象可以在一个属性的改变不能被接受时，拒绝它。其它的对象利用一个 PropertyChangeEvent 来通知，并且它们产生一个 ProptertyVetoException 去阻止修改的发生，并恢复为原来的值。

我们同样能够改变我们的 Bean 在设计时的被描绘成的方法：

(1) 我们可以为我们特殊的 Bean 提供一个定制的属性表。这个普通的属性表将被所有的 Bean 所使用，但当我们的 Bean 被选择时，它会自动地调用这张属性表。

(2) 我们可以为一个特殊的属性创建一个定制的编辑器，因此普通的属性表被使用，但当我们指定的属性被调用时，编辑器会自动地被调用。

(3)我们可以为我们的 Bean 提供一个定制的 BeanInfo 类，产生的信息不同于由 Introspector 默认产生的。

(4) 它同样可能在所有的 FeatureDescriptors 中改变“expert”的开关模式，以辨别基本特征和更复杂的特征。

13.18.6 Bean 更多的知识

另外有关的争议是 Bean 不能被编址。无论何时我们创建一个 Bean，都希望它会在一个多线程的环境中运行。这意味着我们必须理解线程的出口，我们将在下一章中介绍。我们会发现有一段称为“Java Beans 的回顾”的节会注意到这个问题和它的解决方案。

13.19 Swing 入门（注释⑦）

通过这一章的学习，当我们的工作方法在 AWT 中发生了巨大的改变后（如果可以回忆起很久以前，当 Java 第一次面世时 SUN 公司曾声明 Java 是一种“稳定，牢固”的编程语言），可能一直有 Java 还不十分的成熟的感觉。的确，现在 Java 拥有一个不错的事件模型以及一个优秀的组件复用设计——JavaBeans。但 GUI 组件看起来还相当的原始，笨拙以及相当的抽象。

⑦：写作本节时，Swing 库显然已被 Sun“固定”下来了，所以只要你下载并安装了 Swing 库，就应该能正确地编译和运行这里的代码，不会出现任何问题（应该能编译 Sun 配套提供的演示程序，以检测安装是否正确）。若遇到任何麻烦，请访问 http://www.BruceEckel.com，了解最近的更新情况。

而这就是 Swing 将要占领的领域。Swing 库在 Java 1.1 之后面世，因此我们可以自然而然地假设它是 Java 1.2 的一部分。可是，它是设计为作为一个补充在 Java 1.1 版中工作的。这样，我们就不必为了享用好的 UI 组件库而等待我们的平台去支持 Java 1.2 版了。如果 Swing 库不是我们的用户的 Java 1.1 版所支持的一部分，并且产生一些意外，那他就可能真正的需要去下载 Swing 库了。

Swing 包含所有我们缺乏的组件，在整个本章余下的部分中：我们期望领会现代化的 UI，来自按钮的任何事件包括到树状和网格结构中的图片。它是一个大库，但在某些方面它为任务被设计得相应的复杂——如果任何事都是简单的，我们不必编写更多的代码但同样设法运行我们的代码逐渐地变得更加的复杂。这意味着一个容易的入口，如果我们需要它我们得到它的强大力量。

Swing 相当的深奥，这一节不会去试图让读者理解，但会介绍它的能力和 Swing 简单地使我们着手使用库。请注意我们有意识的使用这一切变得简单。如果我们需要运行更多的，这时 Swing 能或许能给我们所想要的，如果我们愿意深入地研究，可以从 SUN 公司的在线文档中获取更多的资料。

13.19.1 Swing 有哪些优点

当我们开始使用 Swing 库时，会注意到它在技术上向前迈出了巨大的一步。Swing 组件是 Bean，因此他们可以支持 Bean 的任何开发环境中使用。Swing 提供了一个完全的 UI 组件集合。因为速度的关系，所有的组件都很小巧的（没有“重量级”组件被使用），Swing 为了轻便在 Java 中整个被编写。

最重要的是我们会希望 Swing 被称为“正交使用”；一旦我们采用了这种关于库的普遍的办法我们就可以在任何地方应用它们。这主要是因为 Bean 的命名规则，大多数的时候在我编写这些程序例子时我可以猜到方法名并且第一次就将它拼写正确而无需查找任何事物。这无疑是优秀库设计的品质证明。另外，我们可以广泛地插入组件到其它的组件中并且事件会正常地工作。

键盘操作是自动被支持的——我们可以使用 Swing 应用程序而不需要鼠标，但我们不得不做一些额外的编程工作（老的 AWT 中需要一些可怕的代码以支持键盘操作）。滚动被毫不费力地支持——我们简单地将我们的组件到一个 JScrollPane 中，同样我们再增加它到我们的窗体中即可。其它的特征，例如工具提示条只需要一行单独的代码就可执行。

Swing 同样支持一些被称为“可插入外观和效果”的事物，这就是说 UI 的外观可以在不同的平台和不同的操作系统上被动态地改变以符合用户的期望。它甚至可以创造我们自己的外观和效果。

13.19.2 方便的转换

如果我们长期艰苦不懈地利用 Java 1.1 版构建我们的 UI，我们并不需要扔掉它改变到 Swing 阵营中来。幸运的是，库被设计得允许容易地修改——在很多情况下我们可以简单地放一个“J”到我们老 AWT 组件的每个类名前面即可。下面这个例子拥有我们所熟悉的特色：

722-723 页程序

这是一个新的输入语句，但此外任何事物除了增加了一些“J”外，看起都像

这 Java 1.1 版的 AWT。同样，我们不恰当的用 add()方法增加到 Swing JFrame 中，除此之外我们必须像上面看到的一样先准备一些“content pane”。我们可以容易地得到 Swing 一个简单的改变所带来的好处。

因为程序中的封装语句，我们不得不调用像下面所写的一样调用这个程序：

java c13.swing.JbuttonDemo

在这一节里出现的所有的程序都将需要一个相同的窗体来运行它们。

13.19.3 显示框架

尽管程序片和应用程序都可以变得很重要，但如果在任何地方都使用它们就会变得混乱和毫无用处。这一节余下部分取代它们的是一个 Swing 程序例子的显示框架：

724 页程序

那些想显示它们自己的类将从 JPanel 处继承并且随后为它们自己增加一些可视化的组件。最后，它们创建一个包含下面这一行程序的 main()：

Show.inFrame(new MyClass(), 500, 300);

最后的两个自变量是显示的宽度和高度。

注意 JFrame 的标题是用 RTTI 产生的。

13.19.4 工具提示

几乎所有我们利用来创建我们用户接口的来自于 JComponent 的类都包含一个称为 setToolTipText(string）的方法。因此，几乎任何我们所需要表示的（对于一个对象 jc 来说就是一些来自 JComponent 的类）都可以安放在窗体中：

jc.setToolTipText("My tip");

并且当鼠标停在 JComponent 上一个超过预先设置的一个时间，一个包含我们的文字的小框就会从鼠标下弹出。

13.19.5 边框

JComponent 同样包括一个称为 setBorder()的方法，该方法允许我们安放一些各种各样有趣的边框到一些可见的组件上。下面的程序例子利用一个创建 JPanel 并安放边框到每个例子中的被称为 showBorder()的方法，示范了一些有用的不同的边框。同样，它也使用 RTTI 来找我们使用的边框名（剔除所有的路径信息），然后将边框名放到面板中间的 JLable 里：

725-726 页程序

这一节中大多数程序例子都使用 TitledBorder，但我们可以注意到其余的边框也同样易于使用。能创建我们自己的边框并安放它们到按钮、标签等等内——任何来自 JComponent 的东西。

13.19.6 按钮

Swing 增加了一些不同类型的按钮，并且它同样可以修改选择组件的结构：所有的按钮、复选框、单选钮，甚至从 AbstractButton 处继承的菜单项（这是因

为菜单项一般被包含在其中，它可能会被改进命名为“AbstractChooser”或者相同的什么名字)。我们会注意使用菜单项的简便，下面的例子展示了不同类型的可用的按钮：

726-727 页程序

JButton 看起来像 AWT 按钮，但它没有更多可运行的功能（像我们后面将看到的如加入图像等）。在 com.sun.java.swing.basic 里，有一个更合适的 BasicArrowButton 按钮，但怎样测试它呢？有两种类型的“指针”恰好请求箭头按钮使用：Spinner 修改一个中断值，并且 StringSpinner 通过一个字符串数组来移动（当它到达数组底部时，甚至会自动地封装)。ActionListeners 附着在箭头按钮上展示它使用的这些相关指针：因为它们是 Bean，我们将期待利用方法名，正好捕捉并设置它们的值。

当我们运行这个程序例子时，我们会发现触发按钮保持它最新状态，开或时关。但复选框和单选钮每一个动作都相同，选中或没选中（它们从 JToggleButton 处继承)。

13.19.7 按钮组

如果我们想单选钮保持“异或”状态，我们必须增加它们到一个按钮组中，这几乎同老 AWT 中的方法相同但更加的灵活。在下面将要证明的程序例子是，一些 AbstruactButton 能被增加到一个 ButtonGroup 中。

为避免重复一些代码，这个程序利用映射来生不同类型的按钮组。这会在 makeBPanel 中看到，makeBPanel 创建了一个按钮组和一个 JPanel，并且为数组中的每个 String 就是 makeBPanel 的第二个自变量增加一个类对象，由它的第一个自变量进行声明：

729-730 页程序

边框标题由类名剔除了所有的路径信息而来。AbstractButton 初始化为一个 JButton，JButtonr 的标签发生“失效”，因此如果我们忽略这个异常信息，我们会在屏幕上一直看到这个问题。getConstructor() 方法产生了一个通过 getConstructor()方法安放自变量数组类型到类数组的构建器对象，然后所有我们要做的就是调用 newInstance()，通过它一个数组对象包含我们当前的自变量——在这种例子中，就是 ids 数组中的字符串。

这样增加了一些更复杂的内容到这个简单的程序中。为了使“异或”行为拥有按钮，我们创建一个按钮组并增加每个按钮到我们所需的组中。当我们运行这个程序时，我们会注意到所有的按钮除了 JButton 都会向我们展示“异或”行为。

13.19.8 图标

我们可在一个 JLable 或从 AbstractButton 处继承的任何事物中使用一个图标（包括 JButton，JCheckbox，JradioButton 及不同类型的 JMenuItem)。利用 JLables 的图标十分的简单容易（我们会在随后的一个程序例子中看到)。下面的程序例子探索了我们可以利用按钮的图标和它们的衍生物的其它所有方法。

我们可以使用任何我们需要的 GIF 文件，但在这个例子中使用的这个 GIF 文

件是这本书编码发行的一部分，可以在 www.BruceEckel.com 处下载来使用。为了打开一个文件和随之带来的图像，简单地创建一个图标并分配它文件名。从那时起，我们可以在程序中使用这个产生的图标。

730-732 页程序

一个图标可以在许多的构建器中使用，但我们可以使用 setIcon()方法增加或更换图标。这个例子同样展示了当事件发生在 JButton（或者一些 AbstractButton）上时，为什么它可以设置各种各样的显示图标：当 JButton 被按下时，当它被失效时，或者“滚过”时（鼠标从它上面移动过但并不击它）。我们会注意到那给了按钮一种动画的感觉。

注意工具提示条也同样增加到按钮中。

13.19.9 菜单

菜单在 Swing 中做了重要的改进并且更加的灵活——例如，我们可以在几乎程序中任何地方使用他们，包括在面板和程序片中。语法同它们在老的 AWT 中是一样的，并且这样使出现在老 AWT 的在新的 Swing 也出现了：我们必须为我们的菜单艰难地编写代码，并且有一些不再作为资源支持菜单（其它事件中的一些将使它们更易转换成其它的编程语言）。另外，菜单代码相当的冗长，有时还有一些混乱。下面的方法是放置所有的关于每个菜单的信息到对象的二维数组里（这种方法可以放置我们想处理的任何事物到数组里），这种方法在解决这个问题方面领先了一步。这个二维数组被菜单所创建，因此它首先表示出菜单名，并在剩余的列中表示菜单项和它们的特性。我们会注意到数组列不必保持一致——只要我们的代码知道将发生的一切事件，每一列都可以完全不同。

732-736 页程序

这个程序的目的是允许程序设计者简单地创建表格来描述每个菜单，而不是输入代码行来建立菜单。每个菜单都产生一个菜单，表格中的第一列包含菜单名和键盘快捷键。其余的列包含每个菜单项的数据：字符串存在在菜单项中的位置，菜单的类型，它的快捷键，当菜单项被选中时被激活的动作接收器及菜单是否被激活等信息。如果列开始处是空的，它将被作为一个分隔符来处理。

为了预防浪费和冗长的多个 Boolean 创建的对象和类型标志，以下的这些在类开始时就作为 static final 被创建：bT 和 bF 描述 Booleans 和哑类 MType 的不同对象描述标准的菜单项（mi），复选框菜单项（cb），和单选钮菜单项（rb）。请记住一组 Object 可以拥有单一的 Object 句柄，并且不再是原来的值。

这个程序例子同样展示了 JLables 和 JMenuItems（和它们的衍生事物）如何处理图标的。一个图标经由它的构建器置放进 JLable 中并当对应的菜单项被选中时被改变。

菜单条数组控制处理所有在文件菜单清单中列出的，我们想显示在菜单条上的文件菜单。我们通过这个数组去使用 createMenuBar()，将数组分类成单独的菜单数据数组，再通过每个单独的数组去创建菜单。这种方法依次使用菜单数据的每一行并以该数据创建 JMenu，然后为菜单数据中剩下的每一行调用 createMenuItem()方法。最后，createMenuItem()方法分析菜单数据的每一行并且

判断菜单类型和它的属性，再适当地创建菜单项。终于，像我们在菜单构建器中看到的一样，从表示 createMenuBar(menuBar)的表格中创建菜单，而所有的事物都是采用递归方法处理的。

这个程序不能建立串联的菜单，但我们拥有足够的知识，如果我们需要的话，随时都能增加多级菜单进去。

13.19.10 弹出式菜单

JPopupMenu 的执行看起来有一些别扭：我们必须调用 enableEvents()方法并选择鼠标事件代替利用事件接收器。它可能增加一个鼠标接收器但 MouseEvent 从 isPopupTrigger()处不会返回真值——它不知道将激活一个弹出菜单。另外，当我们尝试接收器方法时，它的行为令人不可思议，这或许是鼠标单击活动引起的。在下面的程序例子里一些事件产生了这种弹出行为：

737-738 页程序

相同的 ActionListener 被加入每个 JMenuItem 中，使其能从菜单标签中取出文字，并将文字插入 JTextField。

13.19.11 列表框和组合框

列表框和组合框在 Swing 中工作就像它们在老的 AWT 中工作一样，但如果我们需要它，它们同样被增加功能。另外，它也更加的方便易用。例如，JList 中有一个显示 String 数组的构建器（奇怪的是同样的功能在 JComboBox 中无效!）。下面的例子显示了它们基本的用法。

738-739 页程序

最开始的时候，似乎有点儿古怪的一种情况是 JLists 居然不能自动提供滚动特性——即使那也许正是我们一直所期望的。增加对滚动的支持变得十分容易，就像上面示范的一样——简单地将 JList 封装到 JScrollPane 即可，所有的细节都自动地为我们照料到了。

13.19.12 滑杆和进度指示条

滑杆用户能用一个滑块的来回移动来输入数据，在很多情况下显得很直观（如声音控制）。进程条从“空”到“满”显示相关数据的状态，因此用户得到了一个状态的透视。我最喜爱的有关这的程序例子简单地将滑动块同进程条挂接起来，所以当我们移动滑动块时，进程条也相应的改变：

739-740 页程序

JProgressBar 十分简单，但 JSlider 却有许多选项，例如方法、大或小的记号标签。注意增加一个带标题的边框是多么的容易。

13.19.13 树

使用一个 JTree 可以简单地像下面这样表示：

```
add(new JTree(
new Object[] {"this", "that", "other"}));
```

这个程序显示了一个原始的树状物。树状物的 API 是非常巨大的，可是——当然是在 Swing 中的巨大。它表明我们可以做有关树状物的任何事，但更复杂的任务可能需要不少的研究和试验。幸运的是，在库中提供了一个妥协："默认的"树状物组件，通常那是我们所需要的。因此大多数的时间我们可以利用这些组件，并且只在特殊的情况下我们需要更深入的研究和理解。

下面的例子使用了"默认"的树状物组件在一个程序片中显示一个树状物。当我们按下按钮时，一个新的子树就被增加到当前选中的结点下（如果没有结点被选中，就用根结节）：

740-742 页程序

最重要的类就是分支，它是一个工具，用来获取一个字符串数组并为第一个字符串建立一个 DefaultMutableTreeNode 作为根，其余在数组中的字符串作为叶。然后 node()方法被调用以产生"分支"的根。树状物类包括一个来自被制造的分支的二维字符串数组，以及用来统计数组的一个静态中断 i。DefaultMutableTreeNode 对象控制这个结节，但在屏幕上表示的是被 JTree 和它的相关（DefaultTreeModel）模式所控制。注意当 JTree 被增加到程序片时，它被封装到 JScrollPane 中——这就是它全部提供的自动滚动。

JTree 通过它自己的模型来控制。当我们修改这个模型时，模型产生一个事件，导致 JTree 对可以看见的树状物完成任何必要的升级。在 init()中，模型由调用 getModel()方法所捕捉。当按钮被按下时，一个新的分支被创建了。然后，当前选择的组件被找到（如果没有选择就是根）并且模型的 insertNodeInto()方法做所有的改变树状物和导致它升级的工作。

大多数的时候，就像上面的例子一样，程序将给我们在树状物中所需要的一切。不过，树状物拥有力量去做我们能够想像到的任何事——在上面的例子中我们到处都可看到"default（默认）"字样，我们可以取代我们自己的类来获取不同的动作。但请注意：几乎所有这些类都有一个具大的接口，因此我们可以花一些时间努力去理解这些错综复杂的树状物。

13.19.14 表格

和树状物一样，表格在 Swing 相当的庞大和强大。它们最初有意被设计成以 Java 数据库连结（JDBC，在 15 章有介绍）为媒介的"网格"数据库接口，并且因此它们拥有的巨大的灵活性，使我们不再感到复杂。无疑，这是足以成为成熟的电子数据表的基础条件而且可能为整本书提供很好的根据。但是，如果我们理解这个的基础条件，它同样可能创建相关的简单的 Jtable。

JTable 控制数据的显示方式，但 TableModel 控制它自己的数据。因此在我们创建 JTable 前，应先创建一个 TableModel。我们可以全部地执行 TableModel 接口，但它通常从 helper 类的 AbstractTableModel 处简单地继承：

743-744 页程序

DateModel 包括一组数据，但我们同样能从其它的地方得到数据，例如从数

据库中。构建器增加了一个 TableModelListener 用来在每次表格被改变后打印数组。剩下的方法都遵循 Bean 的命名规则，并且当 JTable 需要在 DateModel 中显示信息时调用。AbstractTableModel 提供了默认的 setValueAt()和 isCellEditable()方法以防止修改这些数据，因此如果我们想修改这些数据，就必须过载这些方法。

一旦我们拥有一个 TableModel，我们只需要将它分配给 JTable 构建器即可。所有有关显示，编辑和更新的详细资料将为我们处理。注意这个程序例子同样将 JTable 放置在 JScrollPane 中，这是因为 JScrollPane 需要一个特殊的 JTable 方法。

13.19.15 卡片式对话框

在本章的前部，向我们介绍了老式的 CardLayout，并且注意到我们怎样去管理我们所有的卡片开关。有趣的是，有人现在认为这是一种不错的设计。幸运的是，Swing 用 JTabbedPane 对它进行了修补，由 JTabbedPane 来处理这些卡片，开关和其它的任何事物。对比 CardLayout 和 JTabbedPane，我们会发现惊人的差异。

下面的程序例子十分的有趣，因为它利用了前面例子的设计。它们都是做为 JPanel 的衍生物来构建的，因此这个程序将安放前面的每个例子到它自己在 JTabbedPane 的窗格中。我们会看到利用 RTTI 制造的程序十分的小巧精致：

745-746 页程序

再者，我们可以注意到使用的数组构造式样：第一个元素是被置放在卡片上的 String，第二个元素是将被显示在对应窗格上 JPanel 类。在 Tabbed()构建器里，我们可以看到两个重要的 JTabbedPane 方法被使用：addTab()插入一个新的窗格，setSelectedIndex()选择一个窗格并从它开始。(一个在中间被选中的窗格证明我们不必从第一个窗格开始)。

当我们调用 addTab()方法时，我们为它提供卡片的 String 和一些组件（也就是说，一个 AWT 组件，而不是一个来自 AWT 的 JComponent)。这个组件会被显示在窗格中。一旦我们这样做了，自然而然的就不需要更多管理了——JTabbedPane 会为我们处理其它的任何事。

makePanel()方法获取我们想创建的类 Class 对象和用 newInstance()去创建并造型为 JPanel（当然，假定那些类是必须从 JPanel 继承才能增加的类，除非在这一节中为程序例子的结构所使用)。它增加了一个包括类名并返回结果的 TitledBorder，以作为一个 JPanel 在 addTab()被使用。

当我们运行程序时，我们会发现如果卡片太多，填满了一行，JTabbedPane 自动地将它们堆积起来。

13.19.16 Swing 消息框

开窗的环境通常包含一个标准的信息框集，允许我们很快传递消息给用户或者从用户那里捕捉消息。在 Swing 里，这些信息窗被包含在 JOptionPane 里的。我们有一些不同的可能实现的事件（有一些十分复杂)，但有一点，我们必须尽可能的利用 static JOptionPane.showMessageDialog() 和 JOptionPane.showConfirmDialog()方法，调用消息对话框和确认对话框。

13.19.17 Swing 更多的知识

这一节意味着唯一向我们介绍的是 Swing 的强大力量和我们的着手处，因此我们能注意到通过库，我们会感觉到我们的方法何等的简单。到目前为止，我们已看到的可能足够满足我们 UI 设计需要的一部分。不过，这里有许多有关 Swing 额外的情况——它有意成为一全功能的 UI 设计工具箱。如果我们没有发现我们所需要的，请到 SUN 公司的在线文件中去查找，并搜索 WEB。这个方法几乎可以完成我们能想到的任何事。

本节中没有涉及的一些要点：

■ 更多特殊的组件，例如 JColorChooser,JFileChooser,JPasswordField,JHTMLPane（完成简单的 HTML 格式化和显示）以及 JTextPane（一个支持格式化，字处理和图像的文字编辑器）。它们都非常易用。

■Swing 的新的事件类型。在一些方法中，它们看起来像违例：类型非常的重要，名字可以被用来表示除了它们自己之外的任何事物。

■新的布局管理：Springs & Struts 以及 BoxLayout

■分裂控制：一个间隔物式的分裂条，允许我们动态地处理其它组件的位置。

■JLayeredPane 和 JInternalFrame 被一起用来在当前帧中创建子帧，以产生多文件接口（MDI）应用程序。

■可插入的外观和效果，因此我们可以编写单个的程序可以像期望的那样动态地适合不同的平台和操作系统。

■自定义光标。

■JToolbar API 提供的可拖动的浮动工具条。

■双缓存和为平整屏幕重新画线的自动重画批次。

■内建“取消”支持。

■拖放支持。

13.20 总结

对于 AWT 而言，Java 1.1 到 Java 1.2 最大的改变就是 Java 中所有的库。Java 1.0 版的 AWT 曾作为目前见过的最糟糕的一个设计被彻底地批评，并且当它允许我们在创建小巧精致的程序时，产生的 GUI“在所有的平台上都同样的平庸”。它与在特殊平台上本地应用程序开发工具相比也是受到限制的，笨拙的并且也是不友好的。当 Java 1.1 版纳入新的事件模型和 Java Beans 时，平台被设置——现在它可以被拖放到可视化的应用程序构建工具中，创建 GUI 组件。另外，事件模型的设计和 Bean 无疑对轻松的编程和可维护的代码都非常的在意（这些在 Java 1.0 AWT 中不那么的明显）。但直至 GUI 组件－JFC/Swing 类－显示工作结束它才这样。对于 Swing 组件而言，交叉平台 GUI 编程可以变成一种有教育意义的经验。

现在，唯一的情况是缺乏应用程序构建工具，并且这就是真正的变革的存在之处。微软的 Visual Basic 和 Visual C++需要它们的应用程序构建工具，同样的是 Borland 的 Delphi 和 C++构建器。如果我们需要应用程序构建工具变得更好，我们不得不交叉我们的指针并且希望自动授权机会给我们所需要的。Java 是一个开放的环境，因此不但考虑到同其它的应用程序构建环境竞争，而且 Java 还促进它们的发展。这些工具被认真地使用，它们必须支持 Java Beans。这意味着一个平等的应用领域：如果一个更好的应用程序构建工具出现，我们不需要去约束它就可以使用——我们可以采用并移动到新的工具上工作即可，这会提高我们的

工作效率。这种竞争的环境对应用程序构建工具来说从未出现过，这种竞争能真正提高程序设计者的工作效率。

13.21 练习

(1)创建一个有文字字段和三个按钮的程序片。当我们按下每个按钮时，使不同的文字显示在文字段中。

(2)增加一个复选框到练习 1 创建的程序中，捕捉事件，并插入不同的文字到文字字段中。

(3)创建一个程序片并增加所有导致 action()被调用的组件，然后捕捉他们的事件并在文字字段中为每个组件显示一个特定的消息。

(4)增加可以被 handleEvent()方法测试事件的组件到练习 3 中。过载 handleEvent()并在文字字段中为每个组件显示特定的消息。

(5)创建一个有一个按钮和一个 TextField 的程序片。编写一个 handleEvent()，以便如果按钮有焦点，输入字符到将显示的 TextField 中。

(6)创建一个应用程序并将本章所有的组件增加主要的帧，包括菜单和对话框。

(7)修改 TextNew.java，以便字母在 t2 中保持输入时的样子，取代自动变成大写。

(8)修改 CardLayout1.java 以便它使用 Java 1.1 的事件模型。

(9)增加 Frog.class 到本章出现的清单文件中并运行 jar 以创建一个包括 Frog 和 BangBean 的 JAR 文件。现在从 SUN 公司处下载并安装 BDK 或者使用我们自己的可激活 Bean 的程序构建工具并增加 JAR 文件到我们的环境中，因此我们可以测试两个 Bean。

(10)创建我们自己的包括两个属性：一个布尔值为“on”，另一个为整型“level”，称为 Valve 的 Java Bean。创建一个清单文件，利用 jar 打包我们的 Bean，然后读入它到 beanbox 或到我们自己的激活程序构建工具里，因此我们可以测试它。

(11)修改 Menus.java，以便它处理多级菜单。这要假设读者已经熟悉了 HTML 的基础知识。但那些东西并不难理解，而且有一些书和资料可供参考。