# Chapter 4: Events, GUI Components, Graphics

In this chapter we will learn how to use object-oriented programming introduced in chapter three to create programs and applets with a graphical user interface, menus, buttons, input areas, radio buttons, painting, one or more windows, dialog boxes, and more. These components are defined in a Java package called the *Abstract Windows Toolkit*, or AWT for short. We introduce the AWT and explain how to use its classes to design professional-looking programs and applets.[1]

This chapter frequently refers to definitions of classes and packages as described in the Java *Application Programming Interface* (API). The Java API is a well-organized, detailed, and very useful collection of HTML documents that describes all predefined classes available in Java. It is automatically installed when you install the JDK and can be viewed with a web browser such as Netscape or Internet Explorer. You need the Java API for virtually every program you create and we assume that you can readily access it.

## Quick View

Here is a quick overview of the topics covered in this chapter.



(*) This section is optional.

[1] The AWT provides sufficiently many components to create full-fledged programs, but Sun integrated another package called Swing into the JDK since version 1.2. Swing is more flexible than the AWT, but it is also more difficult to use. We introduce Swing in chapter 6.

# 4.1. Packages and the Abstract Windows Toolkit (AWT)

Previously we created our programs from scratch. Our classes did not extend any *existing* class (except for the basic `Object` class that every Java class extends). That allowed us to create programs illustrating the principles of object-oriented programming, but we could not use windows, buttons, graphics, etc. Our programs were limited by the confines of text-based programs.

To create reasonably sophisticated programs we need to understand and use at least some of the many classes that the Java environment provides. We can still create and use class defined from scratch, but we will always incorporate existing classes into our applications from now on. Our programs will follow a fairly easy scheme:

- Study the available classes, their capabilities, and restrictions.
- Identify existing classes that are useful for a project.
- Use and/or extend those classes to customize them for a particular application.
- Override and overload existing methods to adjust them to our purposes.
- Add additional methods to enable the application to work properly.
- Design additional classes that can be used in one or more applications.

## Packages

Java's existing classes are grouped into packages with names like `java.net` (for classes related to network programming), `java.util` (for utility classes), or `java.awt` (for the *Abstract Windows Toolkit*)[2].

### Package

*Java includes a multitude of predefined classes that are grouped into packages according to their functionality (see table 4.02). To use a class from a package it must be imported using the syntax:*

```
import package.name.ClassName;
```

*to import one the class* `ClassName` *from* `package.name`*, or using wildcard characters*

```
import package.name.*;
```

*to import all classes from* `package.name`.[3] *Classes from the* `java.lang` *package are automatically imported without explicitly using the* `import` *keyword.*

### *Example 4.01: Extending a Button and String*

The package `java.awt` contains a class named `Button`, and the `java.lang` class contains a class named `String`. Create a small Java program that instantiates objects of type `Button` and `String`, respectively.

[2] Swing classes are prefaced with `javax` instead of `java`, as in `javax.swing.JButton`.
[3] Only one wildcard character per package can be used, but multiple `import` statements are possible. When using wildchard characters the compiler only imports classes that are necessary, i.e. referenced in the current class.

The program, as usual, consists of a standard `main` method, which instantiates two objects:[4]

```
public class PackageTest
{  // standard Main Method
   public static void main(String args[])
   {  Button button = new Button();
      String string = new String();
   }
}
```

But this class does not compile because the compiler can not find the class `Button` (see figure 4.01).

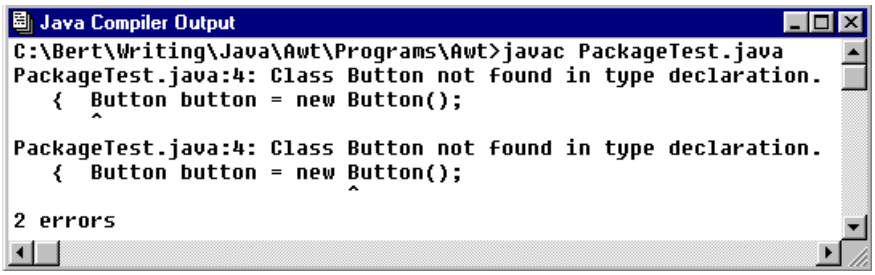



*Figure 4.01: Using a class without importing it*

The `String` class does not create a problem because it is part of the `java.lang` package, which is always available. To fix the error, we need to import the `Button` class from the `java.awt` package or use wildcards to let the compiler automatically search for the class(es) needed:

```
import java.awt.Button;

public class PackageTest
{  /* as before */}
```

or

```
import java.awt.*;

public class PackageTest
{  /* as before */}
```

In both cases the byte-code generated by the compiler is the same but the first version compiles faster because the compiler does not need to search for the correct class in the indicated package.

■

Table 4.02 shows a brief overview of the most important Java packages shipped with the JDK:

| Package Name | Package Description |
|---|---|
| `java.applet` | Used to declare an applet and to interface the applet with the web browser it is embedded in. Provides mechanisms to load images and sounds. |
| `java.awt` | Contains graphical user interface components such as buttons and graphics primitives. Most Java classes import classes from this package. |
| `java.awt.event` | Defines events, listeners, and adapters for the event-handling scheme that specifies how Java programs interact with a user. |
| `java.io` | For input and output classes, including file and keyboard input and output. |
| `java.lang` | Classes related to the core of the Java language such as threads and mathematical operations. Classes in this package are automatically available. |
| `java.net` | Classes for network-relating programming. |
| `java.sql` | Classes to interface Java programs to SQL database servers. |
| `java.util` | Utility classes for dates, time manipulations, vectors, lists, and more. |
| `javax.swing` | Swing classes are GUI components that are more functional and flexible than AWT classes. An understanding of AWT is necessary before using Swing. |

[4] Section 3.1 describes how to instantiate objects from classes.

*Table 4.02: Overview of important Java packages*

Other packages that will not play a prominent role in this text are:

```
java.awt.color, java.awt.datatransfer, java.awt.dnd, java.awt.event, java.awt.font,
java.awt.geom, java.awt.image, java.awt.print, java.beans, java.lang.ref,
java.lang.reflect, java.math, java.rmi, java.rmi.activation, java.rmi.dgc,
java.rmi.registry, java.rmi.server, java.security, java.security.acl, java.security.cert,
java.security.interfaces, java.security.spec, java.text, java.util.jar, java.util.zip,
javax.accessibility, javax.swing.border, javax.swing.colorchooser, javax.swing.event,
javax.swing.filechooser, javax.swing.plaf, javax.swing.table, javax.swing.text,
javax.swing.tree, javax.swing.undo[5]
```

## Abstract Windows Toolkit

Since most applications use classes from the java.awt package, we will start our investigation by describing the most useful classes in this package. Here is the definition of the AWT, which will be at the heart of virtually every application unless it uses Swing components as described in chapter 6.

### AWT – Abstract Windows Toolkit

> *The AWT – Abstract Windows Toolkit – is a Java package containing interfaces, classes, and exceptions for creating event-based, windowed Java programs and applets. The AWT contains standard Graphical User Interface (GUI) elements, drawing primitives, event-handling mechanisms, windows, dialogs, components, layout managers, interfaces, and exceptions (see tables 4.03 and 4.04)*

Tables 4.03 and 4.04 show the classes and interfaces in the java.awt packages as of JDK 1.2. Newer versions of the JDK may include additional interfaces or classes.

| | | | |
|---|---|---|---|
| ActiveEvent | Adjustable | Composite | CompositeContext |
| ItemSelectable | LayoutManager | LayoutManager2 | MenuContainer |
| Paint | PaintContext | PrintGraphics | Shape |
| Stroke | Transparency | | |

*Table 4.03: Interfaces in the* `java.awt package`

| | | | |
|---|---|---|---|
| AlphaComposite | AWTEvent | AWTEventMulticaster | AWTPermission |
| BasicStroke | BorderLayout | Button | Canvas |
| CardLayout | Checkbox | CheckboxGroup | CheckboxMenuItem |
| Choice | Color | Component | ComponentOrientation |
| Container | Cursor | Dialog | Dimension |
| Event | EventQueue | FileDialog | FlowLayout |
| Font | FontMetrics | Frame | GradientPaint |
| Graphics | Graphics2D | GraphicsConfigTemplate | GraphicsConfiguration |
| GraphicsDevice | GraphicsEnvironment | GridBagConstraints | GridBagLayout |
| GridLayout | Image | Insets | Label |
| List | MediaTracker | Menu | MenuBar |
| MenuComponent | MenuItem | MenuShortcut | Panel |
| Point | Polygon | PopupMenu | PrintJob |
| Rectangle | RenderingHints | RenderingHints.Key | Scrollbar |
| ScrollPane | SystemColor | TextArea | TextComponent |
| TextField | TexturePaint | Toolkit | Window |

*Table 4.04: List of classes in* `java.awt`

[5] This is not an exhaustive list. Every new release of the JDK adds more packages and classes to what makes up the Java standard. It is quite a task to know all packages and to keep up with every new release and bug-fixes.

This is a *lot* of information to digest and it is impossible to understand all of these classes immediately. We will concentrate on the most important ones:

**Basic windows classes:**
`Frame` and `Dialog`
**Basic GUI classes:**
`Button`, `Label`, `TextField`, `TextArea`, and `Checkbox`
**Basic event interfaces:**
`ActionListener`, `WindowListener`, `MouseListener`, and `KeyListener`
**Basic layout control:**
`FlowLayout`, `BorderLayout`, `GridLayout`, and `Panel`
**Basic drawing Support:**
`Graphics` and `Canvas`

In the remainder of this chapter, we explain these basic classes and show how they can be used. We mention a few additional classes but we will not cover *all* AWT classes.

**Software Engineering Tip:** The Java API that is part of the JDK is the authority for all Java packages and classes, describing the usage, capabilities, and restrictions of each and every available class. Make sure that you can easily access the API. You can, for example, bookmark it in your web browser, or even define it as the homepage of your web browser.

## 4.2. The Basic Java Program

We have already created Java programs in previous chapters. This time we want to create programs that run inside their own window instead of using standard output. At the heart of these programs is the `Frame` class.

### The Frame Hierarchy

We already know that a basic Java program must contain the standard `main` method to define the *default program entry point* (see section 1.1). To create programs that include "nice" input and output areas, Java programs instantiate at least one object of type `Frame`.

#### The Frame Class

*A* `Frame` *represents a window that can contain a title, a menu, GUI components such as text areas and buttons, and graphics. Its appearance depends on the underlying operating system*[6] *and includes a close box and resize feature. Components within a* `Frame` *are laid out by default using a* `BorderLayout` *(see section 4.5). Frames are invisible by default. If a* `Frame` *is no longer needed, its* `dispose` *method should be called. The AWT defines this class as follows:*

```
public class Frame extends Window implements MenuContainer
```

[6] Using the Swing class `JFrame` lets you to create other look-and-feels such as *Macintosh*, *Windows*, or *Unix* looks, independent of the platform your program is running on.

```
{  // constructors
   public Frame()
   public Frame(String title)
   // selected methods
   public String getTitle()
   public void setTitle(String title)
   public MenuBar getMenuBar()
   public void setMenuBar(MenuBar mb)
   public void dispose()
}
```

***Example 4.02: A simple Frame-based program***

Write a small program that uses the standard main method to instantiate a Frame object. Make sure you import the appropriate classes from the java.awt package. Execute the program and describe what happens.

The constructor of a `Frame` object is overloaded. We use the second version to provide a title as we instantiate the frame. We use the `import` statement to import the class from the AWT package:

```
import java.awt.Frame;

public class MyFrameTest
{  // standard main method
   public static void main(String args[])
   {  Frame myFrame = new Frame("My First Frame"); }
}
```

This program compiles fine, but nothing happens when it is executed.

■

The above definition states that a `Frame` is initially invisible but to see something on the screen we need to change its state from invisible to visible. As with every object, we can use methods to manipulate it, but none of the `Frame` methods indicate that they can change its visibility state. But the `Frame` class is a descendent of another class, `Window`, and hence inherits all *its* methods and fields.

## The Window Class

*A* `Window` *represents a top-level window without borders or menus. It can contain other components from the AWT which are by default laid out using a* `BorderLayout` *(see section 4.5). Initially, a window is not visible. The AWT defines this class as follows:*

```
public class Window extends Container
{  // constructor
   public Window(Frame parent)
   // selected methods
   public void pack()
   public void show()
   public void toFront()
   public void toBack()
   public void addWindowListener(WindowListener l)
   public void removeWindowListener(WindowListener l)
}
```

All methods of the `Window` class are available to objects of type `Frame` type. A `Window` in turn extends `Container`, so that all non-private methods of `Container` are also available to a `Window` or `Frame`:

## The Container Class

*A `Container` is a class that can contain and keep track of other AWT components. It provides a functionality that all subclass automatically share. The AWT defines this class as follows:*

```
public abstract class Container extends Component
{   // constructor
   protected Container()
   // selected methods
   public void add(Component comp)
   public void add(Component comp, Object constraints)
   public Insets getInsets()
   public void setLayout(LayoutManager mgr)
   public void validate()
   public Dimension getPreferredSize()
   public void paint(Graphics g)
}
```

### *Example 4.03: Making a Frame visible*

Write a small program that uses the standard `main` method to instantiate a `Frame` object and displays the frame.

We add one line to the code shown in example 4.02 to make the frame visible:

```
import java.awt.Frame;

public class MyFrameTest
{  // standard Main Method
   public static void main(String args[])
   {  Frame myFrame = new Frame("My First Frame");
      myFrame.show();
   }
}
```

When the program executes, a small window with a title appears (see figure 4.05) whose look is dictated by the underlying platform. The window can be resized as usual and contains a standard close box.


*Figure 4.05: A `Frame` in its original shape and after manual resizing*

But while the window does contain a standard close box, it *cannot* be closed by clicking on that box. To terminate the program, you need to press `CONTROL-C`.[7]

[7] You could also terminate the shell used to start the program. Under *Windows*, press `CONTROL-ALT-DELETE` to selectively `"End Task"`; under *Unix* use the `kill` command; on a *Macintosh*, press `SHIFT-OPTION-ESC`.

■

`Container` extends `Component` and we should define that class to understand the functionality it passes along to `Container`, `Window`, and `Frame`. Since most methods of `Component` are not directly useful to us we will not elaborate on that class. But it is important to understand that a `Frame` is part of a hierarchy of classes and inherits methods and functionality from all of its superclasses.

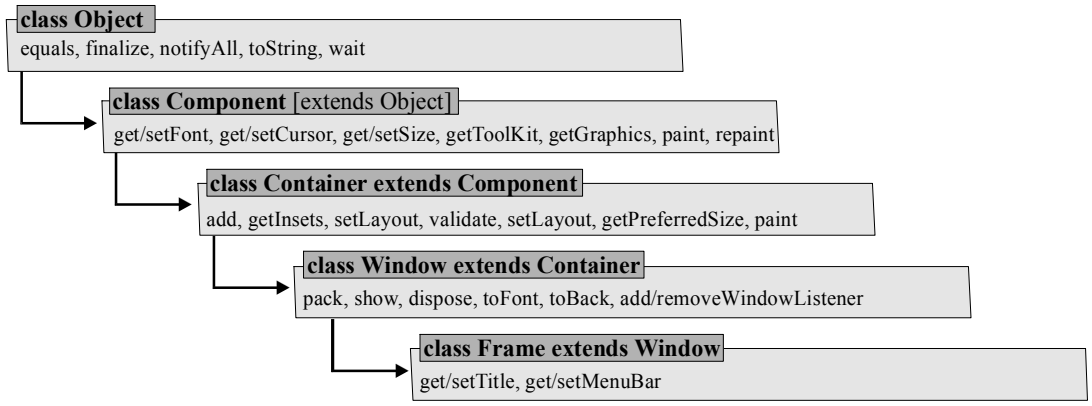



*Figure 4.06: Hierarchy of the* `Frame` *class*

We have seen two versions of a Java program: the first one executes, does nothing, and quits. The second one at least shows something on the screen, but no longer quits. Neither of them is a prototype for good programs.

> **Software Engineering Tip:** A basic GUI program should always show something on the screen to make the user aware that the program is executing, and it should contain at least one GUI element that the user can manipulate to close the program gracefully.

## Responding to Events

Most programs interact with the user through GUI elements such as buttons. When a user clicks a button, it generates an event that can be intercepted by special methods in your class that specify what action your program should perform when the button is clicked. Examples 4.04 and 4.05 illustrate the basic principles of event-driven programming, while a detailed description can be found in section 4.4. The simplest GUI element is a `Button`, so we use it to give the user the option of closing a program.

### *Example 4.04: Adding buttons to a Frame*

Write a program that shows a frame with a title on the screen. The frame should include a button labeled "Quit". When the user clicks that button the program should quit.

We already know how to display a frame on the screen and we can consult the AWT to find the appropriate class to represent a button.

## The Button Class

*A* `Button` *is a class that represents a GUI element that looks like a button in the underlying operating system. The AWT defines this class as follows:*

```
public class Button extends Component
{  // constructors
   public Button()
   public Button(String label)
   // selected methods
   public String getLabel()
   public void setLabel(String label)
   public void addActionListener(ActionListener l)
   public void removeActionListener(ActionListener l)
}
```

**Problem Analysis:** The program, or class, we want to create does not only need to *show* a `Frame`, it should *be* a `Frame`. Therefore we create a class that *extends* `Frame`. It should contain a button, so we add a field of type `Button` to the class.[8] Like most classes it has a constructor and it gets a standard `main` method to make it executable.

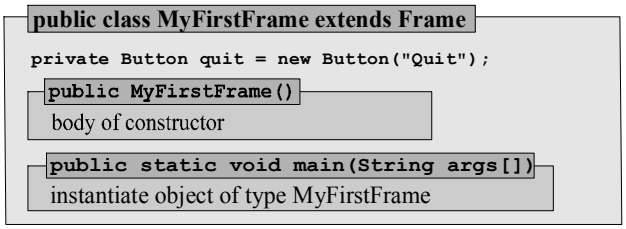



*Figure 4.07: Representation of* `MyFirstFrame` *class*

**Class Implementation:** The button is added to the frame using the `add` method (`Frame` extends `Container`) in the class constructor, which also calls the inherited `show` method to ensure that the frame becomes visible. The standard `main` method instantiates an object of its own type so that the constructor executes automatically to initialize the frame.

```
import java.awt.*;

public class MyFirstFrame extends Frame
{  // fields
   private Button quit = new Button("Quit");
   // constructor
   public MyFirstFrame()
   {  super("Test Window");
      add(quit);
      pack();
      show();
   }
   // standard main method
   public static void main(String args[])
   {  MyFirstFrame mft = new MyFirstFrame(); }
}
```

We have imported all classes from `java.awt` so that the compiler can decide which ones are needed. We added a call to the `pack` method, which computes and resizes the window so that all components

[8] Compare "is-a" and "has-a" phrases introduced in chapter 3.

fit comfortably. When the class executes, a small frame containing a button with the label "Quit" appears, as shown in figure 4.08, but when you click on that button, nothing happens.


*Figure 4.08: Frame with Quit button*

Just because a button appears and is labeled "Quit" does not mean that a click on that button produces the correct – or indeed any – action. We need to create the code that specifies the action to be taken. To do so requires three steps:

- Inform the JVM that our class intends to handle button events.
- Activate a button so that it can generate events when clicked.
- Implement a method that specifies how our program handles a button click.

Java provides the ActionListener interface to link action events generated by GUI components such as buttons to specific methods that decide how to react to them (see "*Interfaces*" in section 3.6). Recall that an interface specifies abstract methods that must be implemented in a class that wants to implement the interface.

## The ActionListener Interface

*An* ActionListener *is an interface from the* java.awt.event *package that defines a method that handles action events. The AWT defines this interface as follows:*

```
public interface ActionListener extends EventListener
{  // abstract method
   public abstract void actionPerformed(ActionEvent e)
}
```

Here is how to utilize the ActionListener interface to activate a button and specify what action it should cause. The details of Java's event processing scheme are described in section 4.4.

```
import java.awt.*;
import java.awt.event.*;

public class MyFirstFrame extends Frame
             implements ActionListener
{  private Button quit = new Button("Quit");

   public MyFirstFrame()
   {  super("Test Window");
      add(quit);
      pack();
      show();
      quit.addActionListener(this);

   }
   public void actionPerformed(ActionEvent e)
   {  dispose();
      System.exit(0);
   }
   public static void main(String args[])
   {  MyFirstFrame mf = new MyFirstFrame(); }
}
```

/* Import components, interfaces, and events */

/* Class of type Frame and ActionListener */

/* Define GUI components as fields */

/* Constructor to initialize frame */

/* Register button as action event source and tie it to event handler. Here it is this class itself. */

/* Specify what to do with action events */

/* Quits program, returns to operating system */

/* Make program executable */

When executing this class, a small window pops up, as before, containing a button labeled "Quit". Clicking on that button causes the program to quit, closing the window and cleaning up all resources it occupied.

■

**Software Engineering Tip:** A basic event-driven program with buttons can follow the framework:

- Import all classes from `java.awt.*` and `java.awt.event.*`
- Define your class so that it extends `Frame` and implements `ActionListener`
- Define GUI components such as buttons as fields
- Use the constructor to define the layout of the components, to activate them using `buttonName.addActionListenerName(this)`, and to make the frame visible using `pack` and `show`.
- Define a method `public void actionPerformed(ActionEvent ae)` to specify what action is to be taken when click on a button has taken place.

## Simple GUI Input and Output

Some GUI components such as buttons are active and can send out action events, others can be used for convenient input and output areas.

***Example 4.05: A complete Frame based program with buttons***

Create a program with two buttons and one output area. One of the buttons cause the program to quit, a click on the other button should result in some text appearing in the output area. Moreover, the displayed text should alternate between two versions every time the second button is clicked.

**Problem Analysis:** Using our standard "is-a", "has-a", and "does-a" terminology from chapter 3 we define the class skeleton as follows:

**"Is-a":** `Frame` and `ActionListener` (to handle events)

**"Has-a":** Two buttons and one "output area". Also needs a `boolean` variable to decide which of two strings to display

**"Does-a":** To be executable it needs a `main` method. To initialize it uses a constructor. To handle button clicks it uses `actionPerformed`.

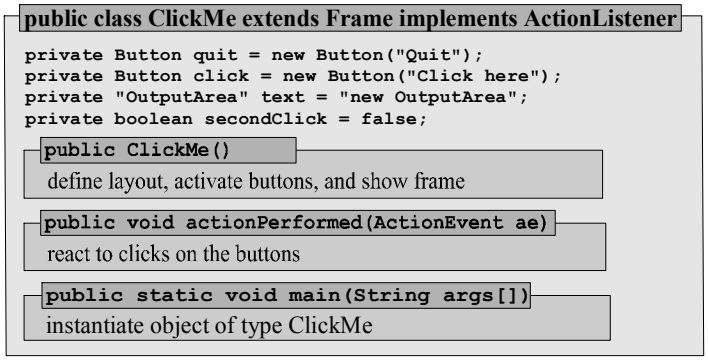



*Figure 4.09: Representation of* `ClickMe` *program*

**Class Implementation:** After inspecting the AWT to locate a class that could serve as our output area, we decide to use a `TextField`.

## The TextField Class

*A* `TextField` *is a component that provides room for one line of editable text. It can be used to get user input, or to display editable program output. The AWT defines this class as follows:*

```
public class TextField extends TextComponent
{  // constructors
   public TextField()
   public TextField(String text)
   public TextField(int columns)
   public TextField(String text, int columns)
   // selected methods
   public void addActionListener(ActionListener l)
   public void removeActionListener(ActionListener l)
}
```

Since `TextField` extends `TextComponent` we also need to know the methods of that class.

## The TextComponent Class

*A TextComponent is a class that allows setting, retrieving, and modifying text. The AWT defines this class as follows:*

```
public class TextComponent extends Component
{  // selected methods
   public void setText(String t)
   public String getText()
   public boolean isEditable()
   public void setEditable(boolean b)
   public String getSelectedText()
   public void select(int selStart, int selEnd)
   public void setCaretPosition(int position)
   public int getCaretPosition()
   public void addTextListener(TextListener l)
   public void removeTextListener(TextListener l)
}
```

We have collected the classes we need for our program (`Frame`, `Button`, and `TextField`), but two problems remain.

- *We need to arrange two buttons and a text field in the constructor, but we do not know how to arrange GUI components.* We discuss the details of arranging components in section 4.5, for now we use the statement `setLayout(new FlowLayout())`, which arranges all components in one row.9
- *The* `actionPerformed` *method needs to react to clicks on the quit and click button, so we need to find the source of the event that* `actionPerformed` *intercepts.* We discuss the details of handling events in section 4.4, for now we use the method `getSource()` of the `ActionEvent` class, which returns a reference to the source of the event.10

Now we can create the complete class, following the outline described in example 4.04:

[9] The method `setLayout` is inherited from `Container`, which is a superclass of `Frame` several levels removed.
[10] The method `getSource` is defined in `EventObject`, a superclass of `ActionEvent` several levels removed.

```
import java.awt.*;
import java.awt.event.*;

public class ClickMe extends Frame implements ActionListener
{  // fields
   private Button quit = new Button("Quit");
   private Button click = new Button("Click here");
   private TextField text = new TextField(10);
   private boolean secondClick = false;
   // constructor
   public ClickMe()
   {  super("Click Example");
      setLayout(new FlowLayout());
      add(quit);
      add(click);
      click.addActionListener(this);
      quit.addActionListener(this);
      add(text);
      pack();
      show();
   }
   // Methods
   public void actionPerformed(ActionEvent e)
   {  if (e.getSource() == quit)
         System.exit(0);
      else if (e.getSource() == click)
      {  if (secondClick)
            text.setText("not again !");
         else
            text.setText("Uh, it tickles");
         secondClick = !secondClick;
      }
   }
   // standard main method
   public static void main(String args[])
   {  ClickMe myFrame = new ClickMe(); }
}
```

The code in `actionPerformed` uses `ae.getSource()` to inspect which of the two buttons was clicked. If it was the `quit` button, the program exits, otherwise it displays one of two versions of text and switches the state of the `boolean` variable `secondClick` to its opposite.


*Figure 4.10:* `ClickMe` *class, clicked once*


*Figure 4.11:* `ClickMe` *class, clicked twice*

■

## 4.3. The Basic Java Applet

One of the strengths of Java is that it allows you to create programs called *applets* that execute automatically inside a Java-enabled web browser such as Netscape Communicator of Microsoft Internet Explorer. Conceptually, an applet is different from a stand-alone program:

- A standalone program either extends no class or is based on the `Frame` class. Its default program entry point is the standard `main` method.
- An applet is a class that extends the `java.applet.Applet` class and does not need to contain the standard `main` method. In addition, the JVM for an applet is provided by the web browser[11], which has important security consequences.

It is usually easy to convert a `Frame`-based program to an applet and visa versa, and you can even create 'dual-purpose' classes that could be used as a standalone application or as an applet.

## Creating an `Applet`

As usual, before we can do any examples we need to define what we mean by an applet:

### The `Applet` Class

*An applet is a program that is embedded inside another application such as a web browser. An applet must extend the* `Applet` *class, which is part of the* `java.applet` *package.*[12]*. It has no constructor and does not need a standard* `main` *method. Execution is governed by the methods* `init`*, which executes once when an applet is loaded,* `start`*, which is called each time the applet is revisited, and* `stop`*, called whenever the web page with the applet is replaced by another page.*

```
public class Applet extends Panel
{  // constructor
   public Applet()
   // selected methods
   public String getParameter(String name)
   public AppletContext getAppletContext()
   public void showStatus(String msg)
   public void init()
   public void start()
   public void stop()
   public URL getCodeBase()
   public URL getDocumentBase()
}
```

When executing an applet inside a web page the JVM is provided by the web browser and could support a different version of Java than the one used to create the applet. For example, Netscape and Internet Explorer before version 4 support Java version 1.0, after version 4 they support Java version 1.1.

**Software Engineering Tip:** You can create a link to a Java web browser plugin developed by Sun so that a user can install it before loading your applet (see section 6.2). The software and installation instruction can be found at `www.javasoft.com/products/plugin`. Then you can be sure that your applet executes in the JVM supporting the latest version of Java. Alternatively you could create applets that adhere to Java 1.0 to ensure they can run on most browsers, but that will limit the functionality these applets can have.

[11] Sun has developed a plugin to provide a JVM for applets that is independent from the one build into the browser. It can support the latest version of Java, but must be installed before it can be used (compare section 6.2).
[12] AWT-based applets extend `java.applet.Applet`, Swing-based applets extend `javax.swing.JApplet` instead.

An applet contains methods that are discussed in chapters 6 (Swing and Multimedia), 7 (Files and Security), and 9 (Networking). At this point we need to understand that:

- The `init` method, called once when the applet is loaded, is comparable to a constructor (but cannot call a superclass constructor). An applet can override it to provide initialization code.
- The `start` method, called each time the applet is revisited, can be overridden to cause an action every time a user visits a page, such as starting an animation.
- The `stop` method is called when the web page with the applet is replaced by another page. It can cause an action every time a user leaves a web page, such as stopping an animation.
- An `Applet` extends `Panel`, not `Frame` or `Window`. Therefore an applet can not have a menu or a title, but it can instantiate frames which *can* have these features.[13].
- There are certain security mechanisms in place when the JVM executes an applet inside a web browser. For example, an applet is not allowed to call the `System.exit` method.

### *Example 4.06: A simple Applet with buttons*

In example 4.05 we created a standalone program with two buttons and an input field so that clicking one button quit the program, the other one caused some text to appear. Convert that program into an applet and execute it inside a web browser. Override the `start` and `stop` methods to display additional text.

**Software Engineering Tip:** To convert a stand-alone program to an applet, follow these steps:

- Instead of extending `Frame` your class must extend `Applet` (imported from `java.applet`).
- Rename the constructor to `public void init()`, since that method is the standard entry point for applets. Remove calls to the superclass constructor and to `show` and `pack`.
- Remove the standard `main` method that a stand-alone program must have.
- Remove all calls to the `System.exit` method, because applets are not allowed to call it.
- Add (override) the methods public `void start()` and `public void stop()` if necessary. The `stop` method can ensure that the applet does not occupy resources when it is not visible.

In our case we apply this tip to change the code from example 4.05 as indicated in figure 4.12.

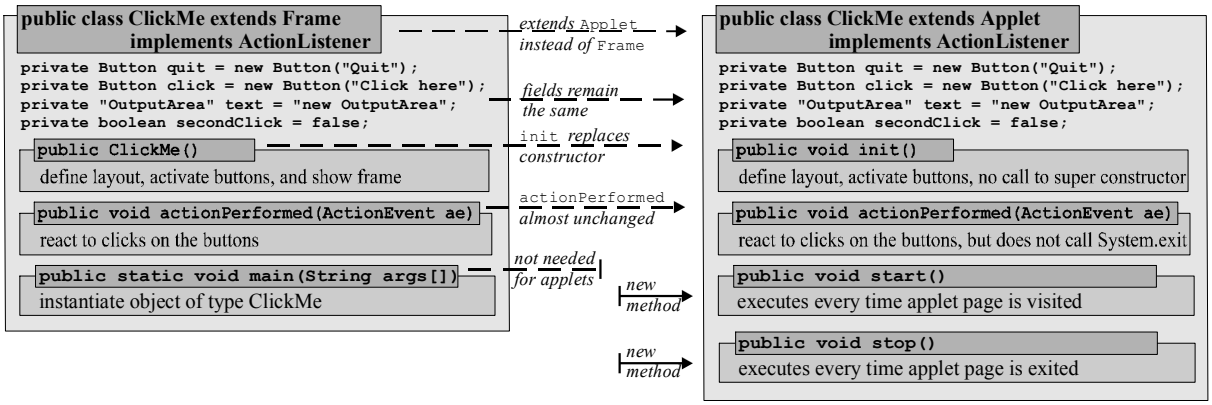



*Figure 4.12: Converting program* `ClickMe` *to applet* `ClickMe`

```
import java.applet.*;
import java.awt.*;
```

[13] Swing-based Applets extending `javax.swing.JApplet` *can* have menus and own dialog boxes (see chapter 6).

```
import java.awt.event.*;

public class ClickMe extends Applet implements ActionListener
{  // fields
   private Button quit = new Button("Quit");
   private Button click = new Button("Click here");
   private TextField text = new TextField(10);
   private boolean secondClick = false;
   // methods
   public void init()
   {  setLayout(new FlowLayout());
      add(quit);
      add(click);
      add(text);
      quit.addActionListener(this);
      click.addActionListener(this);
   }
   public void start()
   {  text.setText("Applet started"); }
   public void stop()
   {  text.setText("Applet stopped"); }
   public void actionPerformed(ActionEvent e)
   {  if (e.getSource() == quit)
         text.setText("Can not quit applets");
      else if (e.getSource() == click)
      {  if (secondClick)
            text.setText("not again !");
         else
            text.setText("Uh, it tickles");
         secondClick = !secondClick;
      }
   }
}
```

■

## Executing Applets via the APPLET Tag

To execute the applet, we need to embed it into an HTML document using a special HTML tag called the APPLET tag.[14]

### The Applet Tag

*The* `APPLET` *tag is part of the HTML formatting language for web documents and is used to embed an applet in a web page. It has a variety of options as indicated in table 4.14 and must have as a minimum the form:*

```
<APPLET CODE="ClassName.class" WIDTH="###" HEIGHT="###">
</APPLET>
```

*where* `ClassName.class` *is the name of the class file generated by the JDK*[15] *and the numbers for* `WIDTH` *and* `HEIGHT` *indicate the width and height that the web browser should make available to the applet. Java applets can not be resized.*

[14] To execute an applet using Sun's Java plugin requires a different HTML tag. See section 6.2 for details.
[15] Applets do not have to be located in the same directory nor on the same machine as the HTML code containing the `APPLET` tag but additional options for the applet tag may be necessary. See table XXX for details.

### *Example 4.07: Embedding an applet in an HTML file*

We have created and compiled an applet in example 4.06. Create an HTML file that embeds that applet into a web page and open it with a Java-enabled web browser.

The code we created was compiled and stored in a file named `ClickMe.class`. To embed it into a web page we need to create an HTML document that contains as a minimum the following text:

```
<HTML>
<APPLET CODE="ClickMe.class" WIDTH="300" HEIGHT="60">
</APPLET>
</HTML>
```

If you save this document as `ClickMe.html` the applet can be executed by opening `ClickMe.html` with Netscape Navigator or Microsoft Internet Explorer (see figure 4.13).

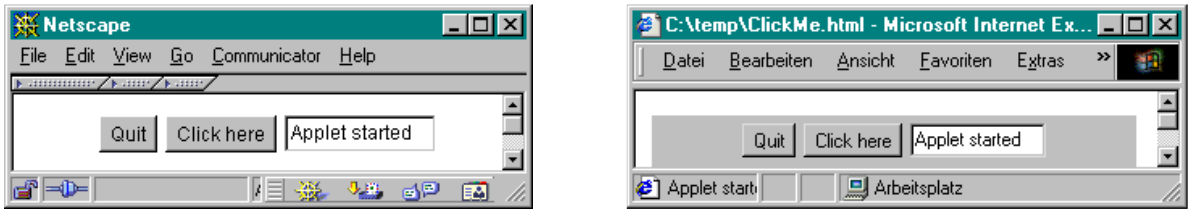



*Figure 4.13: Executing the `ClickMe` applet in different web browsers*[16]

■

**Software Engineering Tip:** The JDK includes a utility program named `appletviewer` to execute applets. Use that program to develop your applet because reloading a web page in a web browser may not load the newest class file. The appletviewer ignores all text except for the `APPLET` tag. To use it, type at the command prompt: `appletviewer filename.html`, where `filename.html` is the HTML document that includes the `APPLET` tag for your applet.

Before releasing your applet, test it using the most likely version(s) of web browser(s) that may be used to view your applet. There are subtle differences between the JVM's of different browsers and different platforms, even though Java applets *should* be platform and browser independent.

The APPLET tag can include several other options. Its general form is:

```
<APPLET ALT      = "alternateText"   ALIGN  = "alignment"
        ARCHIVE  = "archiveList"     CODE   = "ClassName.class"
        CODEBASE = "codebaseURL"     NAME   = "appletInstanceName"
        WIDTH    = "# of pixels"     HEIGHT = "# of pixels"
        VSPACE   = "pixels"          HSPACE = "pixels">
    <PARAM NAME = "name1" VALUE = "value1">
    <PARAM NAME = "name2" VALUE = "value2">
    . . .
    alternate text including HTML formatting tags
</APPLET>
```

[16] The applet can only execute in a web browser whose JVM supports the Java 1.1 specifications or better.

where the options are described in table 4.14. The most commonly used options, in addition to the required CODE, WIDTH, and HEIGHT ones are CODEBASE (to point to applet classes in other locations) and ARCHIVE (to speed up loading applets).

| Applet Parameter | Meaning |
|---|---|
| CODEBASE | The URL of the directory that contains the class file. If not specified, class file and HTML document must be in the same directory. |
| ARCHIVE | One or more archives containing classes and other resources that are preloaded for faster download times |
| CODE | The name of the file that contains the applet's class file. Its location is relative to the CODEBASE. |
| ALT | Text that should be displayed if the browser understands the APPLET tag but can't run Java applets. |
| NAME | A name for the applet to make it possible for applets on the same page to communicate with each other. |
| WIDTH, HEIGHT | The width and height (in pixels) of the applet display area, not counting any windows or dialogs that the applet brings up. |
| ALIGN | Values of this attribute are the same as for the IMG tag: left, right, top, texttop, middle, absmiddle, baseline, bottom, and absbottom. |
| VSPACE, HSPACE | The number of pixels above and below the applet (VSPACE) and on each side of the applet (HSPACE). |
| <PARAM NAME = "name" VALUE = "value"> | An applet-specific attribute. Applets access their attributes with the getParameter method. Any number of PARAM tags can be used. |

*Table 4.14: Possible parameters for the applet tag and their meaning*

**Software Engineering Tip:** Any text between <APPLET ...> and </APPLET> that is not part of the options in table 4.3.2 is ignored by Java-enabled browsers, while non-Java enabled browsers ignore the APPLET tag and display the alternative text. Always include text that explains the basic purpose of your applet so that browsers that are not Java aware will display that text. The text could, for example, include a link to download a Java-aware browser.

### *Example 4.08: Applets with PARAM tags*

Suppose we created a SlideShow applet and saved it on a machine named www.javamachine.edu. We want to embed that applet into a web document located on www.webmachine.edu and use it to load three images from www.javamachine.edu/Images. Create an HTML document containing the appropriate applet tag.

The applet and the enclosing HTML document are located on different machines so that we use the CODEBASE option to specify the base location of the applet. To locate the images we use parameter tags so that our applet has enough information to find them.

```
<HTML>
<H3>A Slide Show</H3>

<CENTER>
<APPLET CODEBASE="http://www.javamachine.edu"
        CODE="SlideShow.class"
        WIDTH="300" HEIGHT="300"
        ALT="Slide Show Applet">
   <PARAM NAME="ImageBase" VALUE="Images">
   <PARAM NAME="ImageNum" VALUE="3">
   <PARAM NAME="image1" VALUE="image1.gif">
   <PARAM NAME="image2" VALUE="image2.gif">
   <PARAM NAME="image3" VALUE="image3.gif">
```

```
<B>Your browser can not handle Java applets</B>
</APPLET>
</CENTER>
</HTML>
```

Regardless of where this file is saved, the applet is loaded from `www.javamachine.edu` and occupies a 300 by 300 pixel area in the web browser, centered inside the browser window. If the browser understands the applet tag but can not handle Java applets, it displays "`Slide Show Applet`". If the browser can not understand the applet tag, it shows "`Your browser can not handle Java applets`". The names of the parameter tags must have meaning to the way the particular applet is written and should be found in the documentation for the applet.

■

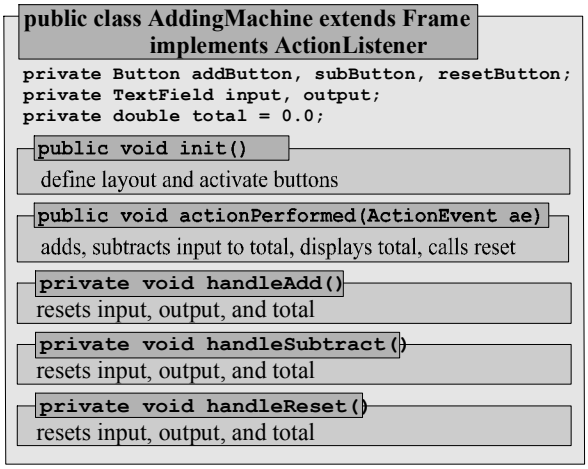
For most of this chapter we use the simple form of the applet tag, assuming that the HTML document and the Java applet are located in the same directory.

### *Example 4.09: The AddingMachine applet*

Create an applet that adds or subtracts numbers entered by a user and displays a running total. Use the appletviewer to test your applet, then execute it in a web browser that understands Java version 1.1 or better.

**Problem Analysis:** We need one button to cause addition and a second to cause subtraction. Numbers are entered in one text field and the total is displayed in another. We also add a button to reset the calculations. Therefore, our class:

"**Is-a**": `Applet` and `ActionListener`
"**Has-a**": three buttons, two text fields, and one running total
"**Does-a**" initialization through the constructor, adds and subtracts via respective methods, `reset` the calculations, and decides via the `actionPerformed` method which operation to perform.




**Class Implementation:** When a user clicks on the add (subtract) button, the number contained in `input` is added (subtracted) to `total` and the new `total` is displayed in `output`. When a user clicks on the reset button, `total` should be set to `0` and the `input` and `output` fields should be cleared. We layout all components in one row, using `setLayout(new FlowLayout())`, as in example 4.05.

The problem that we face is that data can be extracted from the `TextField` with the `getText` method, but it returns a `String`. To do calculations we need to convert the `String` to a number. In section 2.4 we introduced the `Console` class, which contains code to convert a `String` to a number (also compare *Wrapper Classes* in section 2.5). According to sections 2.4 and 2.5 the statement

`Double.valueOf(s.trim()).doubleValue()` converts a `String s` to a `double`, if possible.[17] Here is the complete code for our applet.[18]

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AddingMachine extends Applet implements ActionListener
{  private Button addButton = new Button("Add");
   private Button subButton = new Button("Subtract");
   private Button resetButton = new Button("Reset");
   private TextField input = new TextField(10);
   private TextField output = new TextField(10);
   private double total = 0;

   public void init()
   {  setLayout(new FlowLayout());
      add(addButton);
      add(subButton);
      add(input);
      add(resetButton);
      add(output);
      addButton.addActionListener(this);
      subButton.addActionListener(this);
      resetButton.addActionListener(this);
      handleReset();
   }
   private void handleAdd()
   {  total += Double.valueOf(input.getText().trim()).doubleValue();
      output.setText(String.valueOf(total));
   }
   private void handleSubtract()
   {  total -= Double.valueOf(input.getText().trim()).doubleValue();
      output.setText(String.valueOf(total));
   }
   private void handleReset()
   {  total = 0.0;
      input.setText("0.0");
      output.setText("0.0");
   }
   public void actionPerformed(ActionEvent e)
   {  if (e.getSource() == addButton)
         handleAdd();
      else if (e.getSource() == subButton)
         handleSubtract();
      else if (e.getSource() == resetButton)
         handleReset();
   }
}
```

Implementing "handler" methods that are called by `actionPerformed` lets you reuse their functionality. For example, we call `handleReset` in the constructor and in `actionPerformed`. To execute the compiled class file, we need an HTML document similar to the following:

[17] If the String does not represent a valid double, a `NumberFormatException` is thrown (see chapter 5).
[18] The `init` and `actionPerformed` methods must be `public` to override the inherited versions.

```
<HTML>
<APPLET CODE="AddingMachine.class"
        WIDTH="400"
        HEIGHT="100">
</APPLET>
</HTML>
```

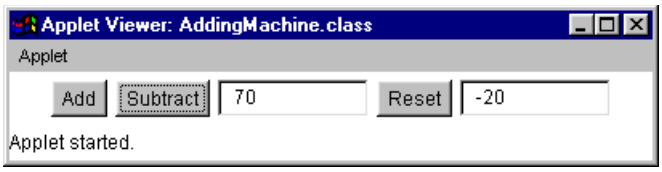



*Figure 4.15:* `AddingMachine` *applet*

■

**Software Engineering Tip:** A class that implements `ActionListener` must have an `actionPerformed` method. Keep that method simple. Use `if-else-if` statements to delegate all work to `private` "handler" methods such as `handleAdd` in the above example. If a handler methods is useful to other classes and cannot cause harm you could mark it `public`.

## 4.4. Event Handling

Programs with a graphical user interface have to overcome the problem that user input is not sequential. While non-GUI programs display a prompt, wait for user input via the keyboard, then continue, programs with a graphical user interface contain buttons, menus, input fields, etc. The user can, at any time, select any of those input options, or choose to quit the program, minimize or maximize it, or bring other windows to the front. Thus, GUI-based programs can not wait for user input, since they do not know from where that input will to come. Instead, they react to "events".

### The Basics of Events

At the heart of every GUI-based program is the *event*.

**Event**

*Events are pieces of information generated by visual components or user interface components that Java classes can react to by implementing special methods. All events are kept in a system-level event queue where they can be retrieved and acted upon by methods. There are two conceptual types of events:*

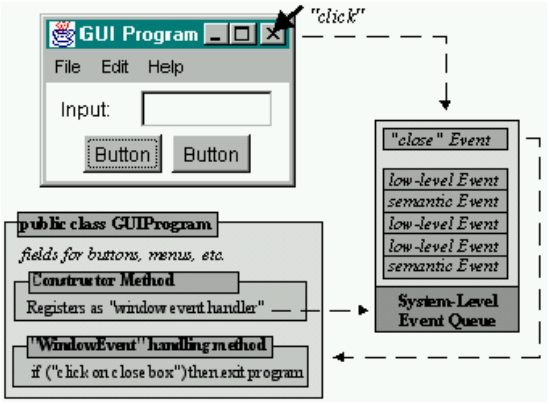
- *Low-level events represents a low-level input or window-system occurrence on a visual component on the screen. They are modeled by the classes* `ComponentEvent`, `FocusEvent`, `InputEvent`, `KeyEvent`, `MouseEvent`, `ContainerEvent`, *and* `WindowEvent`
- *Semantic events that include user-defined information from an interface component. They are represented by the classes* `ActionEvent`, `AdjustmentEvent`, `ItemEvent` *and* `TextEvent`.

This definition is rather abstract, but once we see Java's event-handling scheme in action it will turn out to be logical, easy, and flexible. We consider a simple example without programming first.

### *Example 4.10: How Frame based programs exit*

Imagine a program, whose frame contains several buttons and other components. When a user clicks on the standard close box provided by the window, the program closes. Describe, in words, how that could happen. Remember that the program does not know at what time the user chooses to select the window close box.

The program cannot simply wait until the user selects the close box. Instead, when the program starts, it notifies the operating system that it is willing to react to "window events" such as a click on the close box. It implements, but does not call directly, one or more methods that determine the action to be taken in case a window event takes place. Then the program goes about its usual business. When the user clicks the close box, a special event, perhaps named "window-closing", is inserted into the event queue. Since our program has notified the operating system that it intends to process window events, the event is passed along to our program where it is received by one of the special methods handling window events. That method executes and the program quits.




■

The next example illustrates how to implement the mechanism described in example 4.10.

### *Example 4.11: A program with a WindowListener*

Create a program consisting of a standard window. When the user clicks on the close box of that window, the program should exit. Use the following background information:

- A standard window in Java is represented by the `Frame` class.
- To handle "window events", a class must implement `WindowListener`. That interface contains the abstract methods `windowClosing`, `windowOpened`, `windowIconified`, `windowDeiconified`, `windowClosed`, `windowActivated`, and `windowDeactivated`, which take an input parameter of type `WindowEvent`.
- To register with the operating system that our program wants to handle window-related events, the `Frame` method `addWindowListener` is used.

Our program needs to extend `Frame` and implement the `WindowListener` interface. Thus, our class is defined as:

```
import java.awt.*;         // for Frame
import java.awt.event.*;   // for WindowListener and WindowEvent

public class EventTester extends Frame implements WindowListener
{ /* implementation */ }
```

The constructor needs to inform the Java Virtual Machine that `this` class wants to handle window events and make the frame visible:

```
public EventTester()
{  addWindowListener(this);
```

```
        show();
    }
```

Since our class implements `WindowListener`, it must implement *all* methods of that interface:

```
public void windowClosing(WindowEvent we)
{  System.exit(0); }
public void windowOpened(WindowEvent we)
{  System.out.println("Window opened"); }
public void windowIconified(WindowEvent we)
{  System.out.println("Window iconified"); }
public void windowDeiconified(WindowEvent we)
{  System.out.println("Window deiconified"); }
public void windowClosed(WindowEvent we)
{  System.out.println("Window closed"); }
public void windowActivated(WindowEvent we)
{  System.out.println("Window activated"); }
public void windowDeactivated(WindowEvent we)
{  System.out.println("Window deactivated"); }
```

We also need the standard `main` method to make our class executable:

```
public static void main(String args[])
{  EventTester et = new EventTester(); }
```

If we add these methods to the class `TestProgram`, we can compile and execute it to see the window event handling methods in action (see figure 4.16).

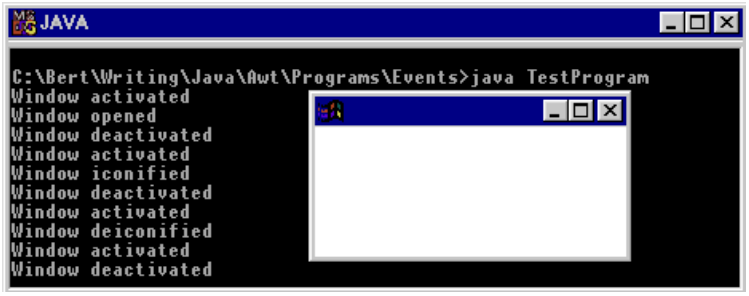



*Figure 4.16: Window events generated by a Frame*

■

If a class implements `WindowListener` it must provide implementations for all of the methods in that interface, even if it only needs `windowClosing`. The other methods can be left without method body, but they must be present. That introduces a lot of unnecessary code and Java provides a shortcut using inner classes and adapters. But before talking about shortcuts we need to understand in detail how Java processes events and how we can make our programs react to them.

## Java Events in Details

In Java, events are represented by a hierarchy of event classes, and interfaces describe the methods that react to specific event types. Classes that want to handle events implement one or more of the interfaces and implement their methods. The details of when which method is called is handled by the JVM.

## The Java Event-Handling Scheme

*Every event source can produce events specific to its type, which are entered into the system-level event queue. Components can register event listener interfaces with the Java Virtual Machine and must implement specific methods to react to events originating from the registered source. Only events for which a component has registered an event listener are passed to that listener. If an event source has registered several listeners, each one receives a replica of the original event.*

*All events are represented by subclasses of* `java.util.EventObject` *and inherit the method*

```
public Object getSource()
```

*The most common event types are part of the* `java.awt.event` *package and are summarized in table 4.17.*

Table 4.17 summarizes the most common event types and lists some of their methods.

| Event Type | Selected Event Methods |
|---|---|
| `ActionEvent` | `public String getActionCommand()` |
| `InputEvent` | `public boolean isShiftDown()`<br>`public boolean isControlDown()`<br>`public boolean isAltDown()`<br>`public long getWhen()`<br>`public void consume()` |
| `ItemEvent` | `public Object getItem()`<br>`public int getStateChange()` |
| `KeyEvent` | `public int getKeyCode()`<br>`public char getKeyChar()` |
| `MouseEvent` | `public int getX()`<br>`public int getY()`<br>`public Point getPoint()`<br>`public int getClickCount()` |
| `WindowEvent` | `public Window getWindow()` |
| `FocusEvent, ComponentEvent, ContainerEvent, and TextEvent` | no particularly important methods |

*Table 4.17: Event classes with selected methods*

Events are processed by methods defined in event listeners.

## Event Listeners

*Java provides event listeners that contain abstract methods to react to specific event types. The listeners, defined as interfaces, are found in the* `java.awt.event` *package and are summarized in table 4.18. Each listener can be attached to an event source via the* `addXXXListener(XXXListener l)` *method.*

Table 4.18 summarizes the most common event listener interfaces and lists their abstract methods.

| Interface | Contains Methods |
|---|---|
| `ActionListener` (S) | `actionPerformed` |
| `ItemListener` (S) | `itemStateChanged` |
| `WindowListener` (L) | `windowClosing, windowOpened, windowIconified, windowDeiconified, windowClosed, windowActivated, windowDeactivated` |
| `ComponentListener` (L) | `componentMoved, componentHidden, componentResized, componentShown` |

| | |
|---|---|
| AdjustmentListener (S) | adjustmentValueChanged |
| MouseMotionListener (L) | mouseDragged, mouseMoved |
| MouseListener (L) | mousePressed, mouseReleased, mouseEntered, mouseExited, mouseClicked |
| KeyListener (L) | keyPressed, keyReleased, keyTyped |
| FocusListener (L) | focusGained, focusLost |
| ContainerListener (L) | componentAdded, componentRemoved |
| TextListener (S) | textValueChanged |

*Table 4.18: Event listeners and their abstract methods [(L) low-level event, (S) semantic event]*

### *Example 4.12: Processing WindowEvent, MouseEvent, and KeyEvent*

Create a program that brings up a single frame, which processes events of type `WindowEvent`, `MouseEvent`, and `KeyEvent` by printing a simple identification string.

The program is similar to example 4.11 but our class needs to implement `MouseListener` and `KeyListener` in addition to `WindowListener`. It must override all methods of the implemented interfaces listed in table 4.18. We must register a listener for each event type we are prepared to handle, otherwise the corresponding events would not be passed to the appropriate listener methods:

```
import java.awt.*;
import java.awt.event.*;

public class MultipleEventTester extends Frame
             implements WindowListener, MouseListener, KeyListener
{  // constructor
   public MultipleEventTester()
   {  addWindowListener(this);
      addMouseListener(this);
      addKeyListener(this);
      setSize(200,200);
      show();
   }
   // ** Window-event handling Methods **
   public void windowClosing(WindowEvent we)
   {  System.exit(0); }
   public void windowOpened(WindowEvent we)
   {  System.out.println("Window opened "); }
   public void windowIconified(WindowEvent we)
   {  System.out.println("Window iconified " + we); }
   public void windowDeiconified(WindowEvent we)
   {  System.out.println("Window deiconified " + we); }
   public void windowClosed(WindowEvent we)
   {  System.out.println("Window closed " + we); }
   public void windowActivated(WindowEvent we)
   {  System.out.println("Window activated " + we); }
   public void windowDeactivated(WindowEvent we)
   {  System.out.println("Window deactivated " + we); }
   // ** Mouse-event handling Methods **
   public void mousePressed(MouseEvent me)
   {  System.out.println("Mouse pressed " + me); }
   public void mouseReleased(MouseEvent me)
   {  System.out.println("Mouse released " + me); }
   public void mouseEntered(MouseEvent me)
   {  System.out.println("Mouse entered " + me); }
   public void mouseExited(MouseEvent me)
   {  System.out.println("Mouse exited " + me); }
   public void mouseClicked(MouseEvent me)
   {  System.out.println("Mouse clicked " + me); }
   // ** Key-event handling Methods **
   public void keyPressed(KeyEvent ke)
```

```
    {  System.out.println("key pressed " + ke); }
    public void keyReleased(KeyEvent ke)
    {  System.out.println("key released " + ke); }
    public void keyTyped(KeyEvent ke)
    {  System.out.println("key typed " + ke); }
    // Standard Main Method
    public static void main(String args[])
    {  MultipleEventTester p = new MultipleEventTester(); }
}
```

When the program executes we can experiment with various events to see the messages displayed.

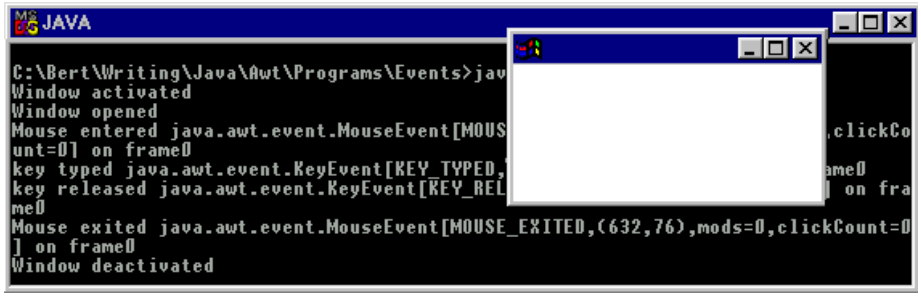



*Figure 4.19: Window, mouse, and key events generate by a* `Frame`

■

Events do not have to be handled inside the same class that displays and initializes a frame. It is often desirable to separate the code that presents GUI components and code that is responsible for actions to be taken.

### *Example 4.13: A program with a separate event handler class*

Create three classes: one is responsible for displaying a frame, the second handles key events by printing them out and exits the program when the user hits 'q', and the third handles window events and exits the program when the window's close box is clicked.

The first class extends `Frame`, the second implements `KeyListener`, and the third implements `WindowListener`. The Frame class registers the other classes as event listeners for the appropriate events.

```
import java.awt.*;
import java.awt.event.*;

public class SeperateListenersTest extends Frame
{  private KeyEventHandler keyListener = new KeyEventHandler();
   private WindowCloser windowListener = new WindowCloser();

   public SeperateListenersTest()
   {  addKeyListener(keyListener);
      addKeyListener(windowListener);
      setSize(200,200);
      setVisible(true);
   }
   public static void main(String args[])
   {  SeperateListenersTest p = new SeperateListenersTest(); }
}
```

The other classes implement `KeyListener` and `WindowListener`, respectively, and implement the methods to handle their event types (see table 4.17):

```
public class KeyEventHandler implements KeyListener
{  public void keyPressed(KeyEvent ke)
   {  if (ke.getKeyChar() == 'q')
         System.exit(0);
   }
   public void keyReleased(KeyEvent ke)
   { }
   public void keyTyped(KeyEvent ke)
   {  System.out.println("Key Listener: Key pressed: " + ke.getKeyChar());}
}

public class WindowCloser implements WindowListener
{  public void windowClosing(WindowEvent we)
   {  System.exit(0); }
   public void windowOpened(WindowEvent we)       { /* empty */ }
   public void windowIconified(WindowEvent we)    { /* empty */ }
   public void windowDeiconified(WindowEvent we)  { /* empty */ }
   public void windowClosed(WindowEvent we)       { /* empty */ }
   public void windowActivated(WindowEvent we)    { /* empty */ }
   public void windowDeactivated(WindowEvent we)  { /* empty */ }
}
```

When `SeperateListenersTest` executes, it displays each key typed. If a user hits 'q', the program exits because of the `KeyEventListener`, if a user clicks the window's close box it exits because of `WindowCloser`.

■

At this time we can improve the `AddingMachine` applet from example 4.09.

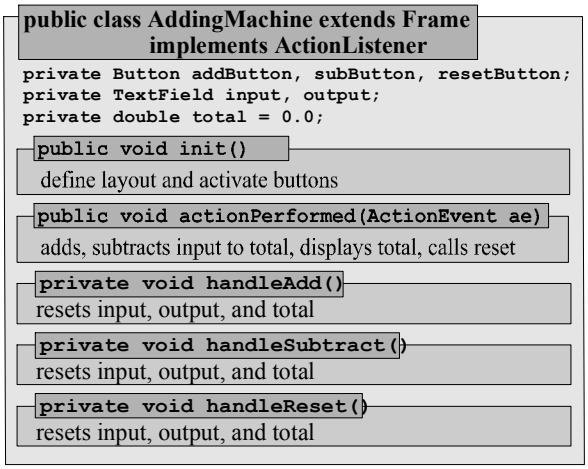
### *Example 4.14: Adding a KeyListener to the AddingMachine applet*

Consider the previous applet `AddingMachine` in example 4.09. Describe the listeners and event types used. Modify the program so that the user can press the `RETURN` key in addition to clicking the "Add" button to add a number.

In example 4.09 we created the `AddingMachine` applet with methods `handleAdd`, `handleSubtract`, and `handleReset`. These methods were marked private so that they cannot be called from other classes.

We want to create a separate class that reacts to key events. If the `RETURN` key is pressed, the class calls the `handleAdd` method of the `AddingMachine`. We modify `AddingMachine` by:

- registering our new key event handler for the input field of the applet
- changing the accessibility of the `handleAdd` method to `public`.




The key handler needs a reference to the `AddingMachine`, so we initialize a field master in the constructor that points to the current instance of an `AddingMachine`.

```
import java.awt.event.*;

public class AddingMachineKeyHandler implements KeyListener
{  private AddingMachine master = null;

   public AddingMachineKeyHandler(AddingMachine _master)
   {  master = _master; }
   public void keyPressed(KeyEvent ke)
   { if (ke.getKeyCode() == KeyEvent.VK_ENTER)
        master.handleAdd();
   }
   public void keyReleased(KeyEvent ke)
   {  /* empty */ }
   public void keyTyped(KeyEvent ke)
   {  /* empty */ }
}
```

The changes to `AddingMachine` are minimal and are marked in bold and italics:

```
public class AddingMachine extends Applet implements ActionListener
{  // fields as before
   public void init()
   {  /* as before plus the following line: /
      input.addKeyListener(new AddingMachineKeyHandler(this));
   }
   public void handleadd()
   {  /* implementation as before */ }
   /* all remaining code as before */
}
```

We add the `KeyListener` to the input `TextField`, because the only time the `ENTER` key should function like the "Add" button is when the user types a number and then hits `ENTER`.[19]

■

To summarizes, table 4.20 shows the available event classes, who can generate them, and what listeners specify the methods to handle them.

| Event | Generated By | Event Listeners |
|---|---|---|
| ActionEvent(S) | Button, List, MenuItem, TextField | ActionListener |
| ItemEvent(S) | Choice, Checkbox, CheckboxMenuItem, List | ItemListener |
| WindowEvent(L) | Dialog, Frame | WindowListener |
| ComponentEvent(L) | Dialog, Frame | ComponentListener |
| AdjustmentEvent(S) | Scrollbar, ScrollPane | AdjustmentListener |
| MouseEvent(L) | Canvas, Dialog, Frame, Panel, Window | MouseMotionListener |
| MouseEvent(L) | Canvas, Dialog, Frame, Panel, Window | MouseListener |
| KeyEvent(L) | Component | KeyListener |
| FocusEvent(L) | Component | FocusListener |
| ContainerEvent(L) | Container | ContainerListener |
| TextEvent(S) | TextComponent | TextListener |
| InputEvent(L) | Component | all component input-event listeners |

*Table 4.20: Java events and their listeners [(L) = low-level event, (S) = semantic event]*

[19] Static fields in the `KeyEvent` class represent the available key codes. For example, if the user presses `SHIFT-b` (a capital `'B'`), a `keyPressed` event with code `VK_SHIFT` is generated , then when the `'b'` is released a `keyReleased` event is generated with code `VK_B`, as well as a `keyTyped` event with value `'B'`. Check the Java API for details.

## Adapters, Event Listeners, and Inner Classes

When a class implements, say, `WindowListener`, it must implement 8 methods, even if we are only interested in one or two of them. Java lets you use 'short versions' of event listeners, using adapters and inner classes, to implement only the methods you need.

### Listener Adapters

> *Java provides several adapter classes that can serve as superclasses of event listeners. Extending an adapter class allows you to override the particular event handling method that you want to implement. All other methods have a default implementation.*
>
> *Java provides the adapter classes* `ComponentAdapter`, `ContainerAdapter`, `FocusAdapter`, `KeyAdapter`, `MouseAdapter`, `MoseMotionAdapter`, *and* `WindowAdapter`, *which can be extended to serve as event listeners.*

#### *Example 4.15: Handling events via adapter classes*

Use an adapter class as a listener for window events to enable a frame to quit when the standard close box is selected.

The class extends Frame but does not implement `WindowListener`. Instead it contains a listener of type `WindowCloser` as a field, which it registers using `addWindowListener`.

```
import java.awt.*;
import java.awt.event.*;

public class ProgramWithAdapterListener extends Frame
{  private WindowCloser listener = new WindowCloser();

   public ProgramWithAdapterListener()
   {  addWindowListener(listener);
      setSize(200,200);
      show();
   }
   public static void main(String args[])
   {  ProgramWithAdapterListener p = new ProgramWithAdapterListener(); }
}
```

The class `WindowCloser` extends `WindowAdapter` and overrides only the `windowClosing` method.

```
public class WindowCloser extends WindowAdapter
{  public void windowClosing(WindowEvent we)
   {  System.exit(0); }
}
```

When the program executes, it shows an empty window that can be closed by clicking its close box because the `windowClosing` method of the `WindowCloser` class will react to that event.

■

The `WindowCloser` class can be used by any frame-based program to activate its standard close box, but the class is not be appropriate if some action must be taken before closing a program. A more flexible approach uses inner classes to embed an event handler directly inside the class that needs it.

## Inner Classes

*An inner class is a class defined inside another class. It has access to all fields and methods of its enclosing class.[20] To define a named inner class, you declare a field of type* `class`*, including the body the inner class , using the syntax:*

```
[modifier] class EnclosingClass [extends Class] [implements Interface]
{  //fields
   [modifier] class InnerClassName [extends Class] [implements Interface]
   {  /* definition of inner class, including fields and methods */ }
   // remaining fields and methods of enclosing class
}
```

*To define an anonymous inner class use the keyword* `new` *together with a class type to extend and the definition of the class as input argument to a method:*

```
someMethod(new ClassType([constructorInputList])
           {  /* definition of inner class, including fields and methods */
           })
```

### *Example 4.16: Anonymous and named inner class adapters*

Redo example 4.15, first implementing the window-event listener as a named inner class, then as an anonymous inner class.

To use a named inner class we move the `WindowCloser` code inside `ProgramWithAdapterListener`. We can then add a new instance of it as the window listener for our program.

```
import java.awt.*;
import java.awt.event.*;

public class ProgramWithNamedListener extends Frame
{  // inner class defined as field of type class
   private class InnerWindowCloser extends WindowAdapter
   {  public void windowClosing(WindowEvent we)
      {  System.exit(0); }
   }
   // constructor
   public ProgramWithNamedListener()
   {  addWindowListener(new InnerWindowCloser());
      setSize(200,200);
      setVisible(true);
   }
   // Standard Main Method
   public static void main(String args[])
   {  ProgramWithNamedListener p = new ProgramWithNamedListener(); }
}
```

To use an anonymous inner class we use the definition of `WindowCloser`, which extends `WindowAdapter`, as input to the `addWindowListener` method, but without giving the inner class a name.

[20] The keyword `this` now refers to an instance of the inner class. To access an instance of the enclosing class, preface the keyword `this` by the name of the enclosing class. In Java 1.2 and below an inner class has to preface members of the enclosing class by the special keyword `this$0` before it can access them.

```
import java.awt.*;
import java.awt.event.*;

public class ProgramWithAnonymousListener extends Frame
{  // constructor
   public ProgramWithAnonymousListener()
   {  addWindowListener(new WindowAdapter()
                        {  public void windowClosing(WindowEvent we)
                           {  System.exit(0); }
                        });
      setSize(200,200);
      setVisible(true);
   }
   // Standard Main Method
   public static void main(String args[])
   {  ProgramWithAnonymousListener p = new ProgramWithAnonymousListener(); }
}
```

■

**Software Engineering Tip**: Inner classes are used to define "on-the-fly" event handling classes or classes that are "short" and useful only for one other class. Anonymous inner classes are easily defined but can not be reused within the enclosing class and can have no constructor. Named inner classes are more flexible, reusable inside the enclosing class, and provider cleaner code. Therefore use named inner classes instead of anonymous ones.

To clarify when an inner class accesses a member of its enclosing class you may want to preface it with `EnclosingClassName.this`:

```
public class EnclosingClass
{  private int x;                          // field of enclosing class
   private class InnerClassName
   {  int y;                               // field of inner class
      void innerMethod()
      {   EnclosingClass.this.x = 10;      // accessing enclosing class member
          y = 10;                          // accessing inner class member
      }
   }
}
```

## Generating your own Events

After learning how pre-defined components generate event we want to explore how to create our own events. We provide two examples of classes that generate standard events and place them properly into the event queue. Both classes can be reused and improved upon for other projects. The first example intercepts action events from buttons and converts them into its own events.

### *Example 4.17: A ColorSelector class*

Create a `ColorSelector` class that displays some of the standard colors in a row and allows you to choose one of them. Test the new class.

**Problem Analysis:** We want to generate a component that shows various colors and lets the user select one of them by clicking on the color. As soon as the component knows which color the user has

selected it should generate an event so that classes using `ColorSelector` can determine which color was selected and act accordingly. Our standard "is-a", "has-a", and "does-a" analysis shows the following:

| | |
|---|---|
| **"Is-a":** | Component that can contain and arrange "color selection components" |
| **"Has-a":** | Several "color selection components", one "selected color" |
| **"Does-a":** | Generates an event if a "color selection component" was clicked and inform listeners of that fact so that they can retrieval of "selected color" |

We need to find out which, if any, classes can be used to clarify the quotation marks in this description of our class, so we search the AWT for potentially useful classes. We find:

- A `Panel` class to contain and arrange components
- The `Color` class that represents a color

The `Panel` class (see section 4.5) contains a `setLayout` method to define its layout and an (overloaded) `add` method to add components, arranged in the specified layout. The `Color` class is defined as follows:

## The Color Class

*The `Color` class represents platform-independent RGB (red-green-blue) colors. It color can be referred to by static name or by specifying integer values between 0 and 255 for the red, green, and blue components of the new color. The AWT defines the `Color` class as follows:*

```
public class Color extends Object
{  // fields
   public static final Color black, blue, cyan, gray, green,
                             magenta, orange, pink, red,
                             white, yellow, lightGray, darkGray
   // selected constructor
   public Color(int red, int green, int blue)
}
```

For a "color selection component" we use a button that is drawn in the color it represents. Now we can clarify our class design:

- **Class type:** `Panel` and implements `ActionListener`
- **Class fields:** An array of buttons drawn in specific colors, and a field of type `Color`, storing the selected color
- **Class methods:** A constructor that initializes the buttons and arranges them in a row, a `getSelectedColor` method to return the selected color, and `actionPerformed` to intercept a button click, set the selected color to the color of the button clicked, generate a new event, and inform any listener that an event has occurred.

**Class Implementation:** Before we can proceed with the implementation we need to know how to generate an event, and how to inform listeners that an event has occurred. To generate an event we create an instance of an `ActionEvent`, which is after all a class.

## The `ActionEvent` Class

*An* `ActionEvent` *represents a semantic event indicating that an action has taken place. It contains the context for the event as a* `String` *and the source that produced the event as an* `Object`*. The AWT defines this class as follows:*

```
public class ActionEvent extends AWTEvent
{  // selected fields
   public static final int ACTION_PERFORMED
   // constructors
   public ActionEvent(Object source, int id, String command)
   // Methods
   public String getActionCommand()
   public int getModifiers()
}
```

To generate a new action event we could therefore use a statement such as:

```
new ActionEvent(eventSource, ActionEvent.ACTION_PERFORMED, "Description")
```

To inform a listener, we could use the following algorithm:

- Use a field `listener` of type `ActionListener` to store a listener reference
- Provide a method `addActionListener` to initialize the `listener` field
- Create a new action event using the above syntax and call the `performAction` method of the `listener`

But this algorithm is too simple, because the new event is only passed to *one* registered `listener`. Events should be passed to *all* listeners registered with a class, but our class can only register one `listener` at a time. Therefore, another approach is needed that keeps track of *all* registered listeners and passes the event to *each* of them in turn. There is a class to do exactly that:

## The `AWTEventMulticaster`

*The* `AWTEventMulticaster` *class manages a stable structure consisting of a chain of event listeners and dispatches events to all listeners. It can be used to add or remove listeners during the process of an event dispatch operation. The class implements the interfaces* `ActionListener`*,* `FocusListener`*,* `ItemListener`*,* `KeyListener`*,* `MouseListener`*,* `MouseMotionListener`*,* `WindowListener`*, and* `TextListener` *and contains all methods from these listeners as well as*

```
public static XXXListener add(XXXListener l, XXXListener newListener)
public static XXXListener remove(XXXListener l, XXXListener oldListener)
```

*where* `XXXListener` *is any of the implemented listeners. The* `add` *method appends* `newListener` *to* `l` *and returns the combined listener. The* `remove` *method removes* `oldListener` *from* `l` *and returns the shorted listener.*

This class should be used whenever we want to create a component that fires its own events.

**Software Engineering Tip:** To generate a component that fires its own action events use the following algorithm:

- Define a field `listener` of type `ActionEvent` and initialize it to `null`
- Implement the method `public void addActionListener(ActionListener newListener)` and use the static method `AWTEventMulticaster.add` to append `newListener` to `listener`.
- Implement the method `public void removeActionListener(ActionListener oldListener)` and use the static method `AWTEventMulticaster.remove` to remove `oldListener` from `listener`.
- Instantiate a new `ActionEvent` and call `listener.actionPerformed(newEvent)` if `listener` is not `null`.

Calling `listener.actionPerformed` causes the `AWTEventMulticaster` to pass the new event to *all* registered listeners and lets you add multiple action listeners to our component.

This algorithm solves the problem of generating events and informing any listeners so that we can implement our `ColorSelector` class as follows:

- We define an array of `Color` as a `static final` field, an array of `Button` (one per color) , a field `selectedColor` of type `Color`, and a field `listener` of type `ActionListener`.
- The constructor initializes the array of buttons, using the method `setBackground` to set the color for each button, and activates each button.
- We create the methods `addActionListener` and `removeActionListener` described above.
- The `actionPerformed` method is called automatically if one of the buttons is clicked. We set `selectedColor` to the color of the clicked button using `getBackground`, create a new event, and call all registered action listeners.

Here is the complete code:[21]

```
import java.awt.*;
import java.awt.event.*;

public class ColorSelector extends Panel implements ActionListener
{  private static Color COLORS[] =
      { Color.white,   Color.black,  Color.gray,    Color.lightGray,
        Color.red,     Color.pink,   Color.orange,  Color.yellow,
        Color.magenta, Color.green,  Color.cyan,    Color.blue };
   private Button buttons[] = new Button[COLORS.length];
   private Color selectedColor = Color.black;
   private ActionListener listener = null;

   public ColorSelector()
   {  setLayout(new FlowLayout());
      for (int i = 0; i < COLORS.length; i++)
      {   buttons[i] = new Button("  ");
          buttons[i].setBackground(COLORS[i]);
          buttons[i].addActionListener(this);
          add(buttons[i]);
      }
   }
   public void addActionListener(ActionListener newListener)
   {  listener = AWTEventMulticaster.add(listener, newListener); }
   public void removeActionListener(ActionListener oldListener)
   {  listener = AWTEventMulticaster.remove(listener, oldListener); }
   public void actionPerformed(ActionEvent e)
   {  if (listener != null)
      {   selectedColor = ((Button)e.getSource()).getBackground();
```

[21] This class is generally useful and should be documented using the `javadoc` tool.

```
            listener.actionPerformed(new ActionEvent(this,
                    ActionEvent.ACTION_PERFORMED, selectedColor.toString()));
        }
    }
    public Color getColor()
    {  return selectedColor; }
}
```

Here is a program to test our new class, using an inner class to close the program (see figure 4.21).

```
import java.awt.*;
import java.awt.event.*;

public class ColorSelectorTest extends Frame implements ActionListener
{  private ColorSelector colors = new ColorSelector();
   private class WindowCloser extends WindowAdapter
   {  public void windowClosing(WindowEvent we)
      {  System.exit(0); }
   }

   public ColorSelectorTest()
   {  super("ColorSelectorTest");
      add(colors);
      colors.addActionListener(this);
      addWindowListener(new WindowCloser());
      pack(); show();
   }
   public void actionPerformed(ActionEvent e)
   {  if (e.getSource() == colors)
      {  setBackground(colors.getColor());
         colors.setBackground(colors.getColor());
      }
   }
   public static void main(String args[])
   {  ColorSelectorTest t = new ColorSelectorTest(); }
}
```


*Figure 4.21:* `ColorSelectorTest` *with red (left) and blue (right) square pressed*

■

This example acts as an event translator. The `Button` fields generate action events with a specific signature that are translated into other events with a different signature. The same mechanism can be used to translate, say, mouse events to action events, or to generate new events.

### *Example 4.18: Generating events automatically*

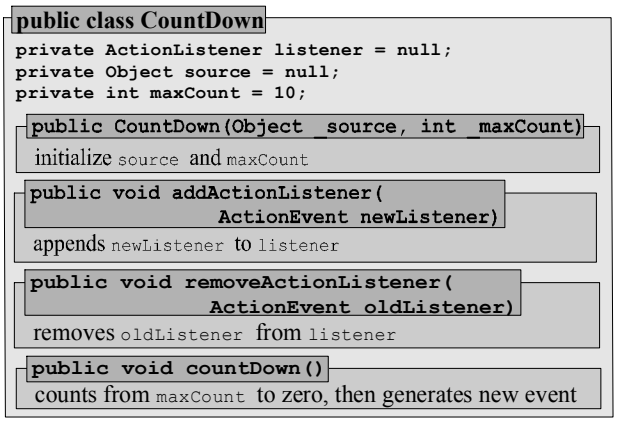
Create a mechanism so that components such as buttons can generate action events automatically as well as when clicked.

**Problem Analysis:** Usually buttons generate action events when a user clicks them. Now we want to create a class (or interface) so that *it* causes a button or other component to fire an action event without user interaction, while retaining the standard functionality of a button.

We create a class that counts down from a specified value to zero. When it reaches zero it generates an action event with same signature as a button event. For a program that uses buttons, events

generated when the user clicks the button and events generated by our class will be treated the same.

The constructor of the class initializes an integer `maxCount` from which to count down[22] and an Object `source` to serve as the source for the event. The class follows our software engineering tip about generating events, so it uses an `AWTEventMulticaster` and needs methods `addActionListener` and `removeActionListener`. Counting starts as soon as a method `startCounting` is called. When the method reaches zero, it generates an action event with `source` as source.




```
import java.awt.*;
import java.awt.event.*;

public class CountDown
{  ActionListener listener = null;
   Object source = null;
   int maxCount = 10;

   public CountDown(Object _source, int _maxCount)
   {  maxCount = _maxCount;
      source = _source;
   }
   public void addActionListener(ActionListener newListener)
   {  listener = AWTEventMulticaster.add(listener, newListener); }
   public void removeActionListener(ActionListener oldListener)
   {  listener = AWTEventMulticaster.remove(listener, oldListener); }
   public void startCounting()
   {  if (listener != null)
      {  for (int i = maxCount; i >= 0; i--)
            System.out.println("i: " + i);
         System.out.println("Done. Generating event now ...");
         listener.actionPerformed(new ActionEvent(source,
                           ActionEvent.ACTION_PERFORMED, "CountDown"));
      }
   }
}
```

To see this class in action, we create a stand-alone test class extending `Frame` with buttons `show` and `count` and one text field. The `show` button is used as input to a fourth field of type `CountDown`. The `actionPerformed` method will display a message if `show` is the event source, or start the countdown if `count` is the event source. After counting down the `CountDown` object generates a message that has as its source the `show` button, so that `actionPerformed` displays a message because it thinks that `show` has been clicked.

```
import java.awt.*;
import java.awt.event.*;

public class CountDownTester extends Frame implements ActionListener
```

[22] A more useful version of `CountDown` should wait for a specified number of seconds instead of simply counting down, which happens very quickly. We can create that improved version after learning about threads in chapter 5.

```
{  private Button start = new Button("Start");
   private Button show = new Button("Show");
   private TextField display = new TextField(25);
   private CountDown count = new CountDown(show, 25);
   private class WindowCloser extends WindowAdapter
   {  public void windowClosing(WindowEvent we)
      {  System.exit(0); }
   }

   public CountDownTester()
   {  super("CountDown Tester");
      setLayout(new FlowLayout());
      add(start);
      add(show);
      add(display);
      show.addActionListener(this);
      start.addActionListener(this);
      count.addActionListener(this);
      addWindowListener(new WindowCloser());
      pack(); show();
   }
   public void actionPerformed(ActionEvent e)
   {  if (e.getSource() == start)
         count.startCounting();
      else if (e.getSource() == show)
         display.setText("Event came from: " + e.getActionCommand());
   }
   public static void main(String args[])
   {  CountDownTester c = new CountDownTester(); }
}
```

Figure 4.22, left side, shows that clicking on `show` generates a message. When you click on `start` , the `CountDown` class counts down, then generates an event that is interpreted as an `ActionEvent` whose source is the `show` button, as displayed in figure 4.22 on the right.


*Figure 4.22:* `CountDownTester`*, with* `Show` *button clicked (left) and* `Start` *button clicked (right)*

The method `getActionCommand` reveals the true source of the event, but the message is displayed because `getSource` equals `show`. Therefore the `show` button can be activated either by clicking on it or by the `CountDown` object.

This idea of generating an event without user intervention can be used, for example, to associate a timer with a button: if a user does not do anything for a specified amount of time, the timer automatically "presses" a button.[23]

■

## 4.5. GUI Components and Layout Control

[23] Such a `CountDownTimer` can be created by combining this example with example 5.14 in section 5.2.

Up to this point all GUI components were arranged in a row, using a `FlowLayout`. In this section we discuss more layout options and introduce the GUI components `Label`, `List`, `TextArea`, `Choice`, and `Checkbox`. Layout control in Java is usually handled by layout managers.

## Layout Control, Panels, and Labels

Java programs run on a variety of operating systems, which have slightly different specifications of how a GUI component looks. When you pick a class such as a `Button` the JVM passes that request to the underlying operating system, which provides a "standard button" to the JVM, which in turn is represented by the `Button` class. Since a `Button` has different dimensions depending on the operating system on which your program executes, specifying its location in absolute coordinates does not work.[24] Instead Java uses layout managers to specify the layout of. A layout manager chooses the best possible positioning for components depending on its specifications, ensuring that the overall look, not the absolute coordinates, of your program remains the same on different system.

### LayoutManager

> *A layout manager determines the order, relative size, and relative location of the components inside a display area. A* `LayoutManager` *is defined by the AWT as an* `Interface`*, and Java provides five specific classes*[25] *that implement* `LayoutManager`[26]*:* `FlowLayout`*,* `GridLayout`*,* `BorderLayout`*,* `CardLayout`*, and* `GridBagLayout` *(see table 4.23).*
>
> *A layout manager queries the* `getPreferredSize` *method of a component for the size requirements that a component has and tries to locate components according to their requirements and its specifications.*

Table 4.23 describes the available layout managers.

| Layout Manager | Description |
|---|---|
| `FlowLayout` | Places components in a row so that each component receives as much space as needed. The row will be centered and "wrapped" if necessary. Components are added from left to right, using the `add` method. |
| `GridLayout` | Uses a table with regularly spaced rows and columns to position components. All cells in the table will be of equal size, determined by the largest component in a cell. Components are added to the table row by row, filling each row left to right, using the `add` method. |
| `BorderLayout` | Provides five areas labeled "`North`", "`South`", "`East`", "`West`" and "`Center`" to place components. The "`Center`" area receives all leftover space. Components are added using the `add` method, specifying the component to add and one of the strings "`East`", "`West`", "`North`", "`South`", or "`Center`". |
| `CardLayout` | Contains several 'cards'. Only one card is visible at a time and each card can be organized using its own layout manager. |
| `GridBagLayout` | Provides a table with irregularly spaced columns and rows for placing components. Each cell can have a different size. |

*Table 4.23: The five layout managers of the AWT*

[24] It is possible to specify absolute coordinates for components, but we use layout managers exclusively in this text.
[25] Swing adds another layout manager called `BoxLayout` to this group.
[26] You can also define your own layout managers, which we will not explore this in this text.

`GridBagLayout` may sound like the most flexible manager, but it is also the hardest one to use. `CardLayout` is another specialized layout manager and we will not need very often. The first three layout managers are used extensively in this text, and careful combinations of them can create almost any layout you want.

### *Example 4.19: FlowLayout, GridLayout, and BorderLayout in action*

Create an program containing five buttons that are arranged using a `FlowLayout`, a `GridLayout`, and a `BorderLayout` (in other words, create three different versions of that program). Determine how these layout managers work when the frame is resized.

Each program defines a layout manager, adds five buttons, and makes itself visible. To properly close the frame, we attach an instance of a `WindowCloser` as window event listener, defined as:

```
import java.awt.event.*;

public class WindowCloser extends WindowAdapter
{  public void windowClosing(WindowEvent we)
   {  System.exit(0); }
}
```

**FlowLayout:**

```
import java.awt.*;

public class ShowFlow extends Frame
{  public ShowFlow()
   {  super("FlowLayout example");
      setLayout(new FlowLayout());
      add(new Button("Button 1"));
      add(new Button("Button 2"));
      add(new Button("Button 3"));
      add(new Button("Button 4"));
      add(new Button("Button 5"));
      addWindowListener(new WindowCloser());
      pack(); show();
   }
   public static void main(String args[])
   {  ShowFlow fl = new ShowFlow(); }
}
```

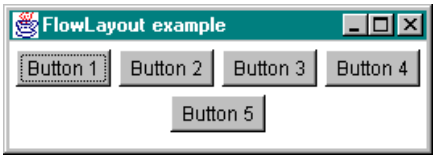



*Figure 4.24:* `FlowLayout` *in action*

The buttons are arranged in a row. If the window is larger than the width of the row, the row is centered. If the window is smaller, the row wraps around. If the window is too small, some buttons become invisible.

**GridLayout:**

```
import java.awt.*;

public class ShowGrid extends Frame
{  public ShowGrid()
   {  super("GridLayout example");
      setLayout(new GridLayout(2, 3));
      add(new Button("Button 1"));
      add(new Button("Button 2"));
      add(new Button("Button 3"));
      add(new Button("Button 4"));
      add(new Button("Button 5"));
      addWindowListener(new WindowCloser());
      pack(); show();
   }
   public static void main(String args[])
   {  ShowGrid gl = new ShowGrid(); }
}
```

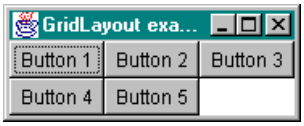



*Figure 4.25:* `GridLayout` *in action*

The buttons are arranged in a table with 2 rows and 3 columns. There are six cells and five buttons, so the cell in the bottom right remains empty. All buttons have the same size. If the window is resized, all buttons change by the same amount. If the window is too small, button labels are cut off, but all

buttons remain visible.

**BorderLayout:**

```
import java.awt.*;

public class ShowBorder extends Frame
{  public ShowBorder()
   {  super("BorderLayout example");
      setLayout(new BorderLayout());
      add("East", new Button("Button 1"));
      add("West", new Button("Button 2"));
      add("North", new Button("Button 3"));
      add("South", new Button("Button 4"));
      add("Center", new Button("Button 5"));
      addWindowListener(new WindowCloser());
      pack(); show();
   }
   public static void main(String args[])
   {  ShowBorder bl = new ShowBorder(); }
}
```

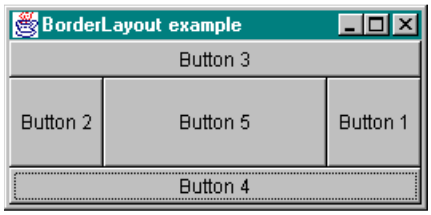



*Figure 4.26:* `BorderLayout` *in action*

4 buttons are arranged according to the direction on a compass, the 5th is in the center. If the window is resized, the outer buttons are at least as wide or high as necessary, with extra space going to the center component. If the window is too small, the center button gets smaller while the outer buttons receive the required space if possible.

■

These layout possibilities by themselves are not sufficient to produce an appearance that people are used to from 'standard' GUI programs. But with the help of the `Panel` class these layouts can be combined to create professional looks.

## The `Panel` Class

> *A* `Panel` *is a generic container to hold other components. A* `Panel` *can have its own layout manager, and different panels can be combined in a* `Frame`*,* `Applet`*, or other* `Panel` *using other layout managers. The* `Panel` *class extends* `Container` *and inherits its methods as defined previously. The methods* `setLayout` *and* `add` *are used most frequently.*

To create complex layouts, you instead create one or more panels, define their layout manager, add the components to each panels according its layout manager, and then arrange the panels according to another layout manager.

### *Example 4.20: Identifying layout possibilities*

Consider the screen shot of a Java program in figure 4.27. Identify the panels and their layouts (only `GridLayout`, `BorderLayout`, and `FlowLayout` were used).

Do not forget that the entire window also has a layout manager, arranging the individual panels[27].

[27] You will be asked to write a program to create this below in the exercises.

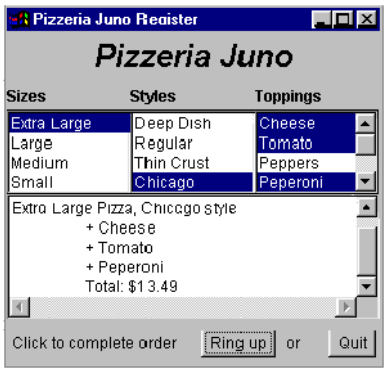



*Figure 4.27: Sample Layout*

This frame contains different "regions":

- the string "`Pizzeria Juno`"
- the labels `Sizes`, `Styles`, and `Toppings`, which are equally spaced, suggesting a `GridLayout`
- three lists containing the various choices, again of equal size, suggesting another `GridLayout`
- the output area containing the detailed order and its price
- the bottom part with two labels and two buttons arranged in a row, suggesting a `FlowLayout`

Figure 4.28 shows how these component panels are combined:

- The title (A) and the labels `Size`, `Styles`, and `Toppings` (B) are arranged in two rows of different size. That suggests a `BorderLayout` to combine them (D).
- Component (D) is arranged with the three lists (C), again having different sizes. Therefore, another `BorderLayout` combine those components (E).
- Now there are three components left: the "top" part (E), the middle part (F) and the bottom part (G). Those could be arranged via another `BorderLayout`, which finishes the layout:[28]

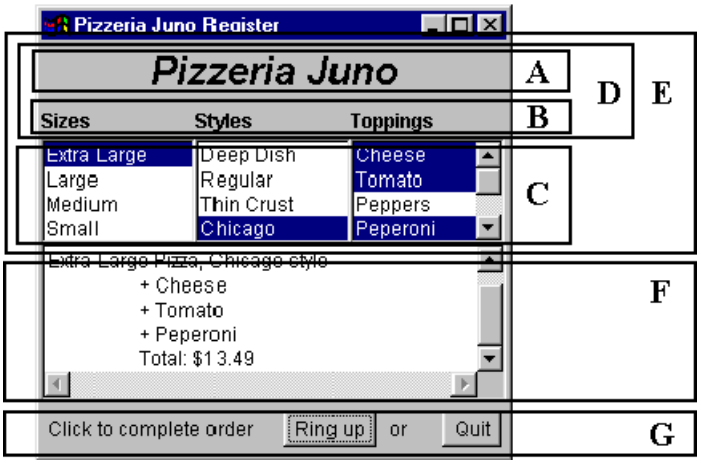



*Figure 4.28: Layout possibilities of a frame using panels*

**Components:**

- A =one Component
- B = `GridLayout`
- C = `GridLayout`
- D = `BorderLayout` of A and B
- E = `BorderLayout` of C and D
- F = one component
- G = `FlowLayout`

**Entire frame:**

- `BorderLayout` of E, F, and G

**Alternative:**

- E = `BorderLayout` of A, B, and C
- D not necessary

■

[28] There are other alternatives to achieve a this layout. If you resize the frame to see how the various components behave, it is possible to uniquely identify the layouts and panels used.

### *Example 4.21: Combining layouts using the Panel class*

Create an applet that contains four text fields in a two-by-two table and two buttons centered in a row at the top of the applet. Your applet does not need to do anything.

Recall that an applet uses the `init` method to layout its components, whereas frames use their constructor. The text fields are arranged according to a `GridLayout`, while the buttons use a `FlowLayout`. A third layout manager combines both components according to a `BorderLayout` with the button panel in the "`North`" and the text field panel in the "`Center`". The resulting look is shown in figure 4.29.

```
import java.applet.*;
import java.awt.*;

public class PanelExample extends Applet
{  public void init()
   {  Panel buttons = new Panel();
      buttons.setLayout(new FlowLayout());
      buttons.add(new Button("Okay"));
      buttons.add(new Button("Cancel"));

      Panel textGrid = new Panel();
      textGrid.setLayout(new GridLayout(2,2));
      for (int i = 0; i < 4; i++)
         textGrid.add(new TextField(4));

      setLayout(new BorderLayout());
      add("North", buttons);
      add("Center", textGrid);
   }
}
```

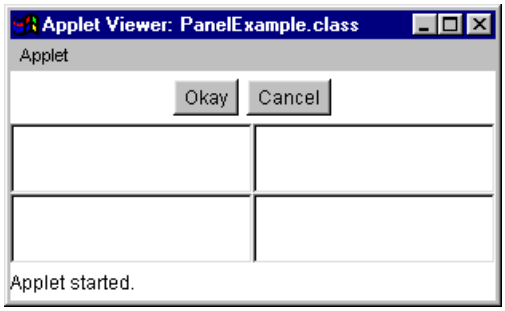



*Figure 4.29: Three layout managers combined via Panels*

■

Here is more useful example of combining layout managers for an appealing look.

### *Example 4.22: A redesigned AddingMachine applet*

Recall our `AddingMachine` applet created in example 4.09. Use as many panels and layout managers as necessary to produce an applet with a more appealing look.

In example 4.09 all components were arranged in one row. This time we choose an arrangement similar to the following:

- Buttons are arranged in a row, centered inside the window. We use a `FlowLayout` panel.
- Text fields can nicely be arranged in a regular grid, including an explanatory label in front of each field. Hence, we use a `GridLayout` panel.

- The panels are arranged so that the buttons are at the bottom, the `GridLayout` should occupy the rest. Therefore we pick a `BorderLayout`.

The new applet differs from its predecessor only in look, not in functionality. Therefore the only method to change is `init`, everything else remains the same.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AddingMachine extends Applet implements ActionListener
{  private Button addButton = new Button("Add");
   private Button subButton = new Button("Subtract");
   private Button resetButton = new Button("Reset");
   private TextField input = new TextField(10);
   private TextField output = new TextField(10);
   private int total = 0;

   public void init()
   {  Panel buttons = new Panel();
      buttons.setLayout(new FlowLayout());
      buttons.add(addButton);
      buttons.add(subButton);
      buttons.add(resetButton);

      Panel display = new Panel();
      display.setLayout(new GridLayout(2,2));
      display.add(new Label("Add/subtract:"));
      display.add(input);
      display.add(new Label("Result:"));
      display.add(output);

      setLayout(new BorderLayout());
      add("Center", display);
      add("South", buttons);

      addButton.addActionListener(this);
      subButton.addActionListener(this);
      resetButton.addActionListener(this);
      handleReset();
   }
   // The methods handleAdd, handleSubtract, handleReset, and actionPerformed
   // are unchanged.
}
```

The look produced by this layout is reasonable. Further improvements could consist of changing the font, adding a border, adding padding around various components, etc., which we will not pursue.

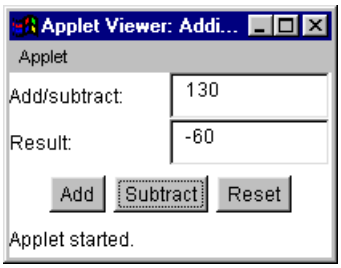



*Figure 4.30:* `AddingMachine` *with attractive layout*

■

**Software Engineering Tip:** It is important to create programs that have an appealing, uncluttered, and standard look. If extra code with extra panels are necessary to improve the look of your program, always go the extra mile. Your program is often judged by its first impression. There are recommended standards that define what a "standard windows look" is. For example:

- Buttons are arranged in a row and centered, with an `OK` button on the left and a `Cancel` button on the right.
- If a menu is present (see section 4.6), it consists at least of a `File`, `Edit`, and `Help` menu. The `File` menu contains at least an option to exit the program, labeled "`Exit`". The `Edit` menu contains the options `Cut`, `Copy`, and `Paste`, and the `Help` menu contains an `About` choice.
- XXX one more at least.

Additional information about program design can be found at: `www.XXX.org`.

We have used a new GUI component of type `Label` as a class field, which is defined as follows:

## The `Label` Class

*A* `Label` *is a one-line, non-editable text component used to provide brief, descriptive messages be arranged by a layout manager. Labels can be modified by methods but not directly by a user. The AWT defines this class as follows:*

```
public class Label extends Component
{  // fields
   public static final int LEFT, CENTER, RIGHT
   // constructors
   public Label()
   public Label(String text)
   public Label(String text, int alignment)
   // selected methods
   public String getText()
   public void setText(String text)
}
```

Labels are sometimes used without defining them as a named field as in example 4.22, which does not allow you to change the text once it is defined. It is usually better to define labels as fields so that they can be easily modified. This is especially important if you want to rewrite your program to support a language other than English.[29]

**Software Engineering Tip**: Any text that is visible to the user such as labels, error messages, prompts, etc., should be defined as constant fields. That way text can be easily modified as your program is developed.

Background colors, fonts, and other visual components should also be defined as constants to quickly change the overall look of your program. Consider using a separate class named `Constants` that contains all constants, strings, colors, formatting tools, and fonts used in your program as `static final` fields.[30]

[29] Java provides extensive support for internationalizing programs via a `ResourceBundle` class.
[30] Java provides a `Properties` class to define persistent properties such as fonts, text, colors, etc.

As an extended example using buttons, text fields, layout managers, and labels that goes beyond our simple `AddingMachine`, let's design a calculator.

### *Example 4.23: A standard calculator applet*

Create an applet that simulates a calculator with the standard operations plus, minus, times, and divide. There should also be a Reset button to clear the display area and all pending computations.

**Problem Analysis:** A standard calculator such as the one shown in figure 4.31 contains numeric and operations buttons arranged in a grid, a display area at the top, and reset buttons that are slightly more prominent that the other buttons. It performs three distinct tasks:

- *Clicking a number button*: The number clicked is appended to the number in the display field, or replaces it if it is currently `0`.
- *Clicking an operator button:* Display result of pending computation, if any, store current number and operator, and wait for next number to define new pending calculation.
- *Clicking the reset button* `CE`: The display and all pending calculations are cleared.

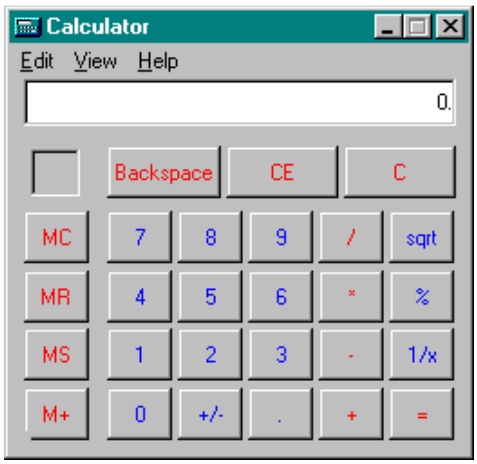



*Figure 4.31: The Windows Calculator*

To understand the second step, we use an available calculator and perform the following sequence:

| Keys pressed | Shown in calculator display |
|---|---|
| enter `1`: | the digit `1` replaces the number currently displayed, no operation is pending |
| enter `+`: | nothing happens, operation `1 + ?` is pending |
| enter `2`: | the digit `2` replaces the number currently displayed |
| enter `*`: | `1 + 2` is evaluated and the resulting `3` is displayed, operation `3 * ?` is pending |
| enter `4` | the digit `4` replaces the number currently displayed |
| enter `=`: | `3 * 4` is evaluated and the resulting `12` is displayed, no operation is pending |
| enter `-` | nothing happens, operation `12 - ?` is pending |
| enter `9` | the digits `9` replace the number currently displayed |
| enter `=` | `12 - 9` is evaluated and the resulting `3` is displayed, no operation is pending |

**Class Implementation:** The class is an applet, so its framework is as usual:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Calculator extends Applet implements ActionListener
{  // define private fields here for the GUI components
   // add other fields for the inner workings of the applet.
```

```
   public void init()
   { /* implementation */ }
   // add action-handling methods as needed
   public void actionPerformed(ActionEvent e)
   { /* implementation */ }
   // add private utility methods as needed
}
```

We complete the applet in two stages: first we create the appropriate look, then we create the required functionality.

**1. Creating the Look:** The layout defined in the `init` method should approximate that of a standard calculator, i.e. a `TextField` is on top, the number buttons are arranged in a grid, and the operations button are on the right side. The special reset key is arranged together with the `TextField`, and we put a `Label` in front of the `TextField`. Our first problem is that there are a lot of buttons so that our standard approach of separately named buttons makes the code unnecessarily long. Instead, we will use an array for the number and operations buttons. We define following fields:

```
private final String[] KEYS = {"7", "8", "9", "/",
                               "4", "5", "6", "*",
                               "1", "2", "3", "-",
                               ".", "0", "=", "+"};
private Button keys[] = new Button[KEYS.length];
private Button reset = new Button(" CE ");
private TextField display = new TextField("0");
```

We define a utility method `setup` to handle the layout in a `for` loop, because the buttons are defined as an array.

```
private void setup()
{  Panel calcKeys = new Panel();
   calcKeys.setLayout(new GridLayout(4, 4));
   for (int i = 0; i < KEYS.length; i++)
   {  keys[i] = new Button(KEYS[i]);
      calcKeys.add(keys[i]);
   }

   Panel top = new Panel();
   top.setLayout(new BorderLayout());
   top.add("Center", display);
   top.add("East", reset);

   setLayout(new BorderLayout());
   add("North", top);
   add("Center", calcKeys);
}
```

The `init` method calls setup and activates the buttons so that they can produce action events:

```
public void init()
{  setup();
   for (int i = 0; i < KEYS.length; i++)
      keys[i].addActionListener(this);
   reset.addActionListener(this);
   display.setEditable(false);
}
```

**2. Creating the Functionality:** In our problem analysis we determined four distinct tasks for our applet, handling a number button, an operator button, and the reset button. Therefore the `actionPerformed` method separates button clicks into four subtasks, represented by "handler" methods to be implemented later.

```
public void actionPerformed(ActionEvent e)
{  Object target = e.getSource();
   String label  = e.getActionCommand();
   if (target == reset)
      handleReset();
   else if ("0123456789.".indexOf(label) >= 0)
      handleNumber(label);
   else
      handleOperator(label);
}
private void handleNumber(String label)
{ /* implementation */ }
private void handleOperator(String label)
{ /* implementation */ }
private void handleReset()
{ /* implementation */ }
```

We used two methods of an `ActionEvent` to identify the source of an action event:

- `getSource`: to determine which buttons has been pressed
- `getActionCommand`: to determine the text of the button that created the action event

Now we need to implement the four handler methods, which can be treated separately.

**Handling a number:** Numbers (and periods) are handled according to the algorithm:

- if the number pressed is the first digit, the previous display is replaced by that digit
- if the number pressed is not the first digit, it is appended to the existing digits on the display
- there can be only one decimal point

Thus, we add a field to decide whether the digit entered is the first or not to our class:

```
private boolean firstDigit = true;
```

Then the code for handling a number button is as follows:

```
private void handleNumber(String key)
{  if (firstDigit)
      display.setText(key);
   else if ((key.equals(".")) && (display.getText().indexOf(".") < 0))
      display.setText(display.getText() + ".");
   else if (!key.equals("."))
      display.setText(display.getText() + key);
   firstDigit = false;
}
```

where the `indexOf` method is used to decide whether the number already contains a period.

**Handling an operator**. All of our operators are binary operators, requiring two numbers to operate on. The sequence of events is:

- you enter one number

- you enter an operator (+, -, *, or /)
- you enter the second number
- you enter another operator: the *previous* operator is applied to the numbers, the result is displayed, and the calculator waits for the next number to apply the current operator
- you enter the equals sign: the result of any pending calculation is displayed

Thus, we need two fields: one field to store the 'previous' number entered, and another field to store the 'previous' operator entered:

```
private double number = 0.0;
private String operator = "=";
```

With those fields in place, our code to handle an operator is as follows:

```
private void handleOperator(String key)
{  if (operator.equals("+"))
      number += getNumberFromDisplay();
   else if (operator.equals("-"))
      number -= getNumberFromDisplay();
   else if (operator.equals("*"))
      number *= getNumberFromDisplay();
   else if (operator.equals("/"))
      number /= getNumberFromDisplay();
   else if (operator.equals("="))
      number = getNumberFromDisplay();
   display.setText(String.valueOf(number));
   operator = key;
   firstDigit = true;
}
```

Note that the *current* operator is stored in the input parameter `key`. while the `if` statements refer to *previous* operator stored in the `operator` field. Once the calculation is complete, `operator` is set to `key` to be used for the next calculation. The utility method `getNumberFromDisplay` to convert the String contained in the display area to a `double` value (compare the `Console` class in section 2.4 and the *Wrapper Classes* in section 2.5, as well as example 4.09) is defined as follows:

```
private double getNumberFromDisplay ()
{  return Double.valueOf(display.getText()).doubleValue(); }
```

**Handling a reset click:** We return all fields to their original state and clear the display area:

```
private void handleReset()
{  display.setText("0");
   firstDigit = true;
   operator = "=";
}
```

That's it, all we have to do is put all the pieces together and we have a fully functioning calculator supporting the basic calculations that can be embedded in any web page.

In the exercises you are asked to expend this applet to include a backspace key, a "plus/minus" key, and a variety of scientific functions such as sine and cosine.

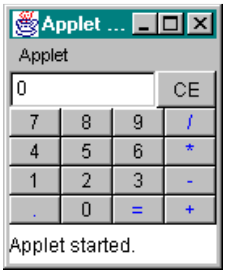



*Figure 4.32:* `Calculator` *applet*

■

Now it is time to introduce additional GUI components that we can arrange in subsequent examples.

## `List` and `TextArea`

`List` and `TextArea` components are 'smart' in the sense that they know how to handle actions such as scrolling or editing on their own. Here is the definition of the `List` class.

### The `List` Class

*A `List` is a sequential arrangement of string items with a vertical scrollbar that scrolls up and down automatically when requested by the user[31]. A `List` can produce action events when a user double-clicks on an item, and item events if other actions such as selections happen. Lists can allow single selections or multiple selections. The `List` class has a substantial number of useful methods, listed in table 4.33.*

A list that contains more items that can be visible automatically has a vertical scrollbar and clicking the scrollbar scrolls it up or down. The details are handled exclusively by the `List` class.

| The List Class | |
|---|---|
| Definition: | `extends Component implements ItemSelectable` |
| Constructors: | `public List()`<br>`public List(int rows)`<br>`public List(int rows, boolean multipleMode)` |
| `Selected Methods:` | `public void add(String item)`<br>`public void add(String item, int index)`<br>`public void replaceItem(String newValue,int index)`<br>`public void remove(int position)`<br>`public void removeAll()`<br>`public int getItemCount()`<br>`public String getItem(int index)`<br>`public String getSelectedItem()`<br>`public String[] getSelectItems()`<br>`public int getSelectedIndex()`<br>`public int[] getSelectedIndexes()`<br>`public void select(int index)`<br>`public void addItemListener(ItemListener l)`<br>`public void removeItemListener(ItemListener l)`<br>`public void addActionListener(ActionListener l)`<br>`public void removeActionListener(ActionListener l)` |

*Table 4.33: Definition and methods of `List`, as defined in the AWT*

### *Example 4.24: A ToDo program using List and TextField*

Create a "To-Do" program that lets you enter tasks (things you need to do) into a list. You should be able to add and remove tasks and change their priority by moving them up or down.

**Problem Analysis:** We need to enter new tasks into a task list, so we need a text field and a list for existing tasks. We have to add and remove tasks, so we need two buttons to cause these actions. We

[31] The Swing equivalent of `java.awt.List` is `javax.swing.JList`. That class is more flexible because it can keep track of objects of type different from `String`. For details, please check definition 6.3.1.

also have to move tasks up or down in the task list so we need two additional buttons. Thus, our class skeleton is as follows:

**"Is-a":** extends `Frame` implements `ActionListener`
**"Has-a":** one `TextField`, one `List`, and 4 `Button` fields, plus two labels to guide the user
**"Does-a":** The method `actionPerformed` intercepts button clicks, and the standard `main` method makes the program executable. Other methods are `handleAdd`, `handleDel`, `handleDecPriority`, and `handleIncPriority`.

**Class Implementation:** As before we create this program by first implementing the layout, then the functionality. We follow the standard framework for programs, including a named inner class to close the program:

```
import java.awt.*;
import java.awt.event.*;

public class TaskList extends Frame implements ActionListener
{  private Button add = new Button("Add");
   private Button del = new Button("Delete");
   private Button up  = new Button(" + ");
   private Button down = new Button(" - ");
   private List list = new List();
   private TextField taskInput = new TextField();
   private Label priorityLabel = new Label("Change Priorities");
   private Label taskLabel = new Label("Task:");
   private class WindowCloser extends WindowAdapter
   {  public void windowClosing(WindowEvent we)
      {  System.exit(0); }
   }

   public TaskList()
   {  /* constructor to define layout and activate buttons */ }
   public void actionPerformed(ActionEvent ae)
   {  /* reacts to button clicks */ }
   public static void main(String args[])
   {  TaskList tl = new TaskList(); }
}
```

**Step 1: Creating the Layout:** To create the layout we define a private `setup` method.

- We place the add and delete buttons in one row at the bottom of the applet.
- The text field, together with a label, goes at the top of the applet.
- The list of existing tasks is in the center.
- We place the buttons to change priorities between the list and the input area.

```
private void setup()
{  Panel buttons = new Panel();
   buttons.setLayout(new FlowLayout());
   buttons.add(add); buttons.add(del);

   Panel priorities = new Panel();
   priorities.setLayout(new FlowLayout());
   priorities.add(up); priorities.add(priorityLabel); priorities.add(down);

   Panel input = new Panel();
   input.setLayout(new BorderLayout());
   input.add("West", taskLabel); input.add("Center", taskInput);

   Panel top = new Panel();
```

```
        top.setLayout(new GridLayout(2,1));
        top.add(input); top.add(priorities);

        setLayout(new BorderLayout());
        add("Center", list); add("South", buttons); add("North", top);
    }
```

The setup method is called in the constructor, which also activates the buttons, attaches the window listener, and makes the frame visible. The resulting layout is shown in figure 4.34.

```
public TaskList()
{   super("Task List");
    setup();
    add.addActionListener(this);
    del.addActionListener(this);
    up.addActionListener(this);
    down.addActionListener(this);
    addWindowListener(new WindowCloser());
    pack();
    show();
}
```

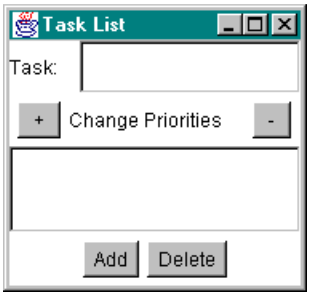



*Figure 4.34: Layout of* `TaskList` *program*

**2. Creating the functionality:** Now we need to decide what should happen and when:

- If a user clicks on the add button, a task entered in the text field should be added to the list as a new task. The new task should be added to the end of the list.
- If a user clicks on delete, the highlighted task, if any, should be removed from the list.
- If a user clicks on the up or down buttons, the highlighted task, if any, should be moved up or down in the task list, if possible.

The `actionPerformed` method delegates the action to 4 "handler" methods. It checks if the conditions to call a handler method are satisfied.

```
public void actionPerformed(ActionEvent ae)
{   if ((ae.getSource() == add) && (!taskInput.getText().equals("")))
       handleAdd(taskInput.getText().trim());
    else if ((ae.getSource() == del) && (list.getSelectedIndex() >= 0))
       handleDel(list.getSelectedIndex());
    else if ((ae.getSource() == up) && (list.getSelectedIndex() > 0))
       handleIncPriority(list.getSelectedIndex());
    else if ((ae.getSource() == down) && (list.getSelectedIndex() >= 0))
       handleDecPriority(list.getSelectedIndex());
    taskInput.requestFocus();
}
```

At the end of handling an action event we call `taskInput.requestFocus`, which places the cursor into the `TextField` so that the user can enter a new task without having to position the cursor manually. It remains to implement the handler methods, which use several of the methods from the `List` class to manipulate the items in the list.

```
private void handleAdd(String newTask)
{   list.add(newTask);
    list.select(list.getItemCount()-1);
    taskInput.setText("");
}
```

```
private void handleDel(int pos)
{   list.remove(pos);
    list.select(pos);
}
```

```
private void handleIncPriority(int pos)
{  String item = list.getItem(pos);
   list.remove(pos);
   list.add(item, pos-1);
   list.select(pos-1);
}
```

```
private void handleDecPriority(int pos)
{  if (pos < list.getItemCount()-1)
   {  String item = list.getItem(pos);
      list.remove(pos);
      list.add(item, pos+1);
      list.select(pos+1);
   }
}
```

To add a convenient editing feature to our program, we active the list so that double-clicking on a list item will copy it into the input `TextField`. You can then remove the old entry and add a new, modified version. To accomplish this we activate the list by adding one line to the end of the constructor:

```
list.addActionListener(this);
```

In `actionPerformed`, we add another `else if` statement to the last `if`:

```
else if (ae.getSource() == list)
   taskInput.setText(list.getSelectedItem());
```

Figure 4.34 shows the resulting program with several to-do items.

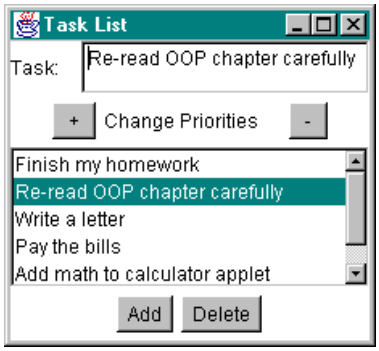



*Figure 4.35: `TaskList` applet*

■

The final basic GUI element we need to explore is a `TextArea`, which can contain several lines of editable or non-editable text, including horizontal and vertical scroll bars.

## The `TextArea` Class

*A `TextArea` provides space for editable or non-editable text and can be used to input or output multiple lines of text. It contains horizontal and vertical scrollbars and handles scrolling on its own. Special control characters can be used for formatting, such as `\n` to advances the insertion point to the beginning of a new line and `\t` to inserts a tab character. `TextArea` extends `TextComponent` and inherits methods and fields from that class. The AWT defines this class as follows:*

```
public class TextArea extends TextComponent
{  // constructors
   public TextArea()
   public TextArea(String text)
```

```
        public TextArea(int rows, int columns)
        public TextArea(String text, int rows, int cols)
        // selected methods
        public void append(String str)
        public void insert(String str, int pos)
        public void replaceRange(String str, int start, int end)
    }
```

Here is a simple example to understand the basic functionality of a `TextArea`.

***Example 4.25: Using the TextArea class***

Create a program with a `TextField`, two `Buttons`, and a `TextArea`. When the user clicks on the first buttons, the text from the `TextField` is added to the `TextArea`. A click on the second button adds the text but precedes it by a `newline` character.

Our class extends `Frame`, implements `ActionListener`, and needs four fields: two buttons, a `TextField`, and a `TextArea`. We arrange the buttons and the `TextField` at the top and give the `TextArea` the remaining space.

```
import java.awt.*;
import java.awt.event.*;

public class TextAreaTest extends Frame implements ActionListener
{  private TextField input = new TextField();
   private TextArea  output = new TextArea();
   private Button add = new Button("Add");
   private Button addLn = new Button("Add + Return");
   private class WindowCloser extends WindowAdapter
   {  public void windowClosing(WindowEvent we)
      {  System.exit(0); }
   }

   public TextAreaTest()
   {  super("TextAreaTest");
      setup();
      add.addActionListener(this);
      addLn.addActionListener(this);
      addWindowListener(new WindowCloser());
      pack(); show();
   }
   public void actionPerformed(ActionEvent e)
   {  if (e.getSource() == add)
         output.append(input.getText());
      else if (e.getSource() == addLn)
         output.append("\n"+input.getText());
   }
   private void setup()
   {  Panel top = new Panel();
      top.setLayout(new BorderLayout());
      top.add("West", add); top.add("Center",input); top.add("East", addLn);

      setLayout(new BorderLayout());
      add("North", top); add("Center", output);
   }
   public static void main(String args[])
   {  TextAreaTest tat = new TextAreaTest(); }
}
```

While this is perhaps not the most exciting program, it does show a text area in action and how to add text either within a line or in a new line (see figure 4.36).

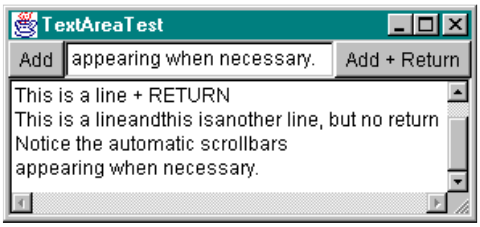



*Figure 4.36: Simple example with* `TextArea`

■

To utilize a `TextArea` for something useful, we enhance our Calculator program from example 4.23.

### *Example 4.26: Adding a TextArea to the Calculator applet*

Create a program that simulates a calculator as in example 4.23, but this version should display a log of all calculations performed, suitably formatted.

**Problem Analysis:** In example 4.23 we created a working calculator, but it shows only one number, either the one entered or the result of a computation. We want to enhance it to see the sequence of operations it performs. For example, if we enter `10 + 10 * 3 =`, which results in `60` because the calculator performs operations in order, not in order of precedence, we want to see a log of these calculations, something like:

```
    10
+   10
*    2
=   60[32]
```

The calculations should be entered into a `TextArea` so that we can scroll up and down any time. We need to determine how to add the `TextArea`, and how to display the appropriate results in it.

**Class Implementation:** The structure of the Calculator applet from example 4.23 is shown in figure 4.37. We need to determine how to add the `TextArea`, and how to make it work.

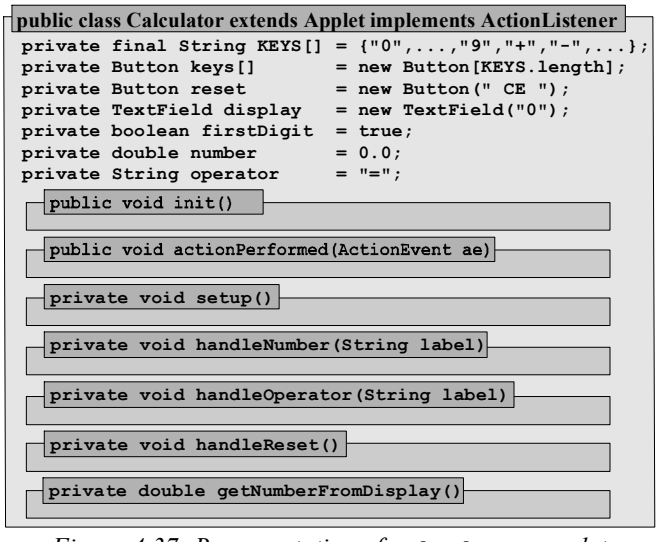



*Figure 4.37: Representation of* `Calculator` *applet*

**Adding the `TextArea`:** To add the new component to the layout is easy. We defined a new field using:

```
private TextArea log
            = new TextArea();
```

and add it to the applet by appending the line

```
add("South", log);
```

[32] Mathematically 10 + 10 * 2 is 30, not 60, but as a log it does represent the keys we pressed. It is understood that our calculator performs calculations in order. In the exercises you are asked to create a mathematically correct log.

to the end of the `setup` method.

**Enabling the `TextArea`:** The log should change whenever an operation button is pressed, while pressing numeric buttons appends digits to the display only. Therefore we need to modify the `handleOperator` method.

If we enter the keys `10 + 20 * 3 =` then `handleOperator` is called when the + button is clicked, at which time 10 shows in the display. The number 10 and the operation + should be entered into the log. The next time `handleOperator` is called is when the * button is pressed and 20 is displayed. The number 20 should be entered into the log, together with the * operation. Finally, `handleOperator` is called when = is pressed and the operator together with the final result should be entered into the log. Therefore:

- If an operator button is pressed, add the number currently displayed to the log, and in the next line show the value of the key pressed.
- If the = button is pressed, show the result of the computation and add an empty line

Therefore we modify `handleOperator` as follows (shown in bold and italics):

```
private void handleOperator(String key)
{  if (operator.equals("+"))
      number += getNumberFromDisplay();
   else if (operator.equals("-"))
      number -= getNumberFromDisplay();
   else if (operator.equals("*"))
      number *= getNumberFromDisplay();
   else if (operator.equals("/"))
      number /= getNumberFromDisplay();
   else if (operator.equals("="))
      number = getNumberFromDisplay();
   log.append(String.valueOf(getNumberFromDisplay()) + "\n" + key + "\t");
   if (key.equals("="))
      log.append(String.valueOf(number)+"\n\n\t");
   display.setText(String.valueOf(number));
   operator = key;
   firstDigit = true;
}
```

We also redesign the `reset` operations so that it clears the log by replacing the text in the `TextArea` with a single tab character, because a new number will be entered without an operator next.

```
private void handleReset()
{  display.setText("0");
   log.setText("\t");
   firstDigit = true;
   operator = "=";
}
```

That's it – almost. When you test the enhanced calculator applet you will notice that the very first number is not aligned correctly and that the log area can be edited by the user. Therefore, we add two lines to the end of the `init` method (shown in bold and italics):

```
public void init()
{  setup();
   for (int i = 0; i < KEYS.length; i++)
      keys[i].addActionListener(this);
```

```
        reset.addActionListener(this);
        display.setEditable(false);
        log.setEditable(false);
        log.setText("\t");
    }
```

A sample run of our enhanced calculator applet is shown in figure 4.38.

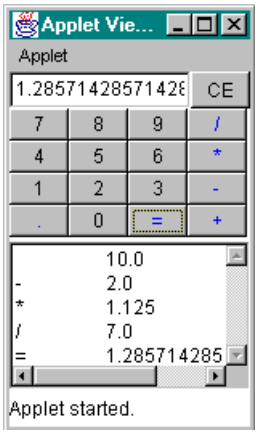



*Figure 4.38: Enhanced calculator with log area*

■

## 4.6. Menus and Dialogs

So far all programs and applets used only one window. That is sufficient for many applications but larger programs usually need to open multiple windows, dialog boxes, deal with various fonts, and use a menu bar. This section explains how to attach menus to frames and how to use dialog windows.

### The `Menu`, `MenuBar`, and `MenuItem` Classes

*A menu represents a list of menu items that becomes visible if a user clicks on it.. Selecting a menu item can cause an action event.*[33] *Menus can be combined into a menu bar attached to the top of a* `Frame`*. The most common classes to create menu items, menus, and menu bars are (compare table 4.39):*

- *A* **`MenuBar`** *is a container for menus that is attached to a* `Frame` *via the method* `setMenuBar` *of a* `Frame`*.A* `MenuBar` *can contain any number of* `Menu` *objects, added via its* `add` *method.*
- *A* **`Menu`** *is a collection of menu items with a common title. The title shows up as clickable text in a* `MenuBar` *and clicking on it reveals the collection of menu items. A* `Menu` *can contain* `MenuItem` *and* `Menu` *objects that are added using its* `add` *method.*
- *A* **`MenuItem`** *represents an individual choice in a menu and can register an* `ActionListener` *to generate* `ActionEvents`.

Table 4.39 summarizes the constructors and methods for `MenuBar`, `Menu`, and `MenuItem`.

| Class | Constructors and Selected Methods |
|---|---|

[33] Menu choices can be selected using the mouse or via keyboard shortcuts. Refer to the Java API for details.

| | |
|---|---|
| public class MenuBar<br>extends MenuComponent<br>implements MenuContainer | public MenuBar()<br>public Menu add(Menu m)<br>public void remove(MenuComponent mc) |
| public class Menu<br>extends MenuItem<br>implements MenuContainer | public Menu()<br>public Menu(String label)<br>public MenuItem add(MenuItem mi)<br>public void addSeparator()<br>public void remove(MenuComponent mc) |
| public class MenuItem<br>extends MenuComponent | public MenuItem()<br>public MenuItem(String label)<br>public String getLabel()<br>public void setLabel(String label)<br>public boolean isEnabled()<br>public void setEnabled(boolean b)<br>public void addActionListener(ActionListener l)<br>public void removeActionListener(ActionListener l) |
| public class CheckboxMenuItem<br>extends MenuItem<br>implements ItemSelectable | public CheckboxMenuItem(String label)<br>CheckboxMenuItem(String label, boolean state)<br>public boolean getState()<br>public void setState(boolean newState) |

*Table 4.39: Menu-related classes, constructors, and selected methods*

### *Example 4.27: A program with a standard menu bar*

Create a stand-alone program with a `MenuBar` containing the a `File` menu with menu items `New`, `Open`, and `Exit`, and a `Edit` menu with the usual menu items. Disable all menu choices except `File | Exit`.

The usual items in an `Edit` menu are `Cut`, `Copy`, and `Paste`. All menu items are represented as individual fields:

```
import java.awt.*;
import java.awt.event.*;

public class MenuTest extends Frame implements ActionListener
{  private MenuItem fileNew    = new MenuItem("New");
   private MenuItem fileOpen   = new MenuItem("Open");
   private MenuItem fileExit   = new MenuItem("Exit");
   private MenuItem editCut    = new MenuItem("Cut");
   private MenuItem editCopy   = new MenuItem("Copy");
   private MenuItem editPaste  = new MenuItem("Paste");

   public MenuTest()
   {  super("Menu Test Program");
      Menu file = new Menu("File");
      file.add(fileNew);    fileNew.setEnabled(false);
      file.add(fileOpen);   fileOpen.setEnabled(false);
      file.addSeparator();
      file.add(fileExit);   fileExit.setEnabled(true);

      Menu edit = new Menu("Edit");
      edit.add(editCut);    editCut.setEnabled(false);
      edit.add(editCopy);   editCopy.setEnabled(false);
      edit.add(editPaste);  editPaste.setEnabled(false);

      MenuBar bar = new MenuBar();
      bar.add(file);
      bar.add(edit);
      setMenuBar(bar);

      fileExit.addActionListener(this);
      setSize(100, 100);
```

```
        show();
    }
    public void actionPerformed(ActionEvent e)
    {   if (e.getSource() == fileExit)
            System.exit(0);
    }
    public static void main(String args[])
    {   MenuTest f = new MenuTest(); }
}
```

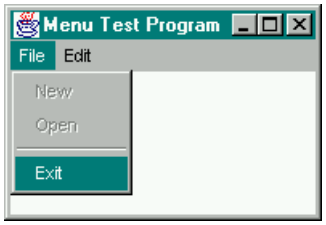



*Figure 4.40:* `MenuTest` *program, with* `File | Exit` *selected*

■

Menus work similar to buttons, since both components generate action events, and they are a standard part of almost every program. In addition to menus, many programs also need dialogs, which are windows owned by a parent window that are used to display informative messages, confirmation messages, or to obtain user input.

## Dialog

*A* `Dialog` *is a window that is owned by another* `Frame` *or* `Dialog`*. It is initially invisible and is disposed of when its parent window is disposed. There are two types of* `Dialog` *windows (see Dialog constructors in table 4.41):*

- *Modal dialog: A dialog that blocks all input to its parent and must be closed or made invisible before the parent can receive input again.*
- *Non-modal dialog: A dialog that does not block input and can coexist with its parent window. Unless specified otherwise, a* `Dialog` *is non-modal.*

A `Dialog` is instantiated using one of the constructors shown in table 4.41. It extends `Window` and therefore inherits all methods from `Window` such as `pack` and `show`.

| Constructor | Meaning |
|---|---|
| `public Dialog(Type parent)` | A new (non-modal) dialog with parent `parent`. The `Type` of parent can be `Frame` or `Dialog`. |
| `public Dialog(Type parent, boolean modal)` | A new modal or non-modal dialog with parent `parent`. The `Type` of parent can be `Frame` or `Dialog`. |
| `public Dialog(Type parent, String title)` | A new (non-modal) dialog with parent `parent` and title `title`. The `Type` of parent can be `Frame` or `Dialog`. |
| `public Dialog(Type parent, String title, boolean modal)` | A new modal or non-modal dialog with title `title` and parent `parent`. The `Type` of parent can be `Frame` or `Dialog`. |

*Table 4.41: Constructors of the* `Dialog` *class*

### *Example 4.28: Adding a simple dialog to a program*

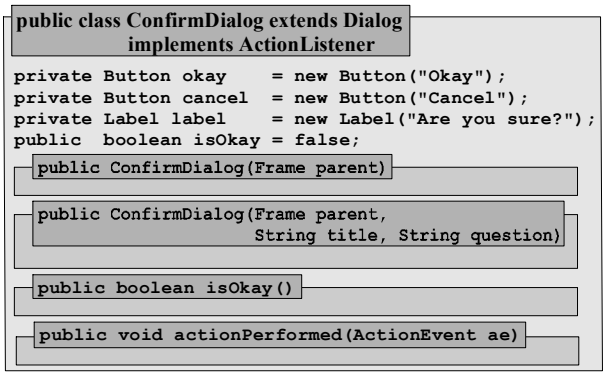
Add a `Dialog` to the program in example 4.27 so that the user must confirm whether to exit the program before the program exits.

**Problem Analysis:** We need to create a dialog that appears when the user wants to exit the program and asks a question similar to "Do you really want to exit?". It should offer two choices, such as "Okay" and "Cancel". Only if the user clicks "OK" should the program exit.

Programs that ask the user for confirmation before performing an action are common. For example, before deleting a file a program could ask "Do you really want to delete this file", or before disconnecting a dialup connect to the Internet a program could ask "Do you really want to disconnect?". Therefore, instead of providing a solution for our concrete example we create a flexible ConfirmDialog class that can be adjusted to different situations.

The ConfirmDialog class should be modal, because it requires the user to make a decision before the program that brought up the dialog can continue. It should be constructed with a flexible title and text for the question it asks so that it can accommodate different situations, and it needs a way to inform the parent which of the two options was chosen by the user.

**Class Implementation:** Our program needs a Label to contain the question it asks, and two buttons to represent Okay and Cancel. It gets a public boolean field isOkay that contains true if the Okay button was clicked or false otherwise. In either case the method should also remove the dialog so that the parent program can continue. The constructor sets the label and title for the dialog, and we provide an additional constructor to create a default confirmation dialog.




Since a dialog extends Window it has a close box. We add a window listener so that clicking on that close box is equivalent to clicking the Cancel button.

```
import java.awt.*;
import java.awt.event.*;

public class ConfirmDialog extends Dialog implements ActionListener
{  private Button okay = new Button("Okay");
   private Button cancel = new Button("Cancel");
   private Label  label = new Label("Are you sure?", Label.CENTER);
   public  boolean isOkay = false;
   private class WindowCloser extends WindowAdapter
   {  public void windowClosing(WindowEvent we)
      {  ConfirmDialog.this.isOkay = false;
         ConfirmDialog.this.hide();
      }
   }

   public ConfirmDialog(Frame parent)
   {  this(parent, "Please confirm", "Are you sure?"); }
   public ConfirmDialog(Frame parent, String title, String question)
   {  super(parent, title, true);
      label.setText(question);
      setup();
      okay.addActionListener(this);
      cancel.addActionListener(this);
```

```
        addWindowListener(new WindowCloser());
        setResizable(false);
        pack(); show();
    }
    private void setup()
    {   Panel buttons = new Panel();
        buttons.setLayout(new FlowLayout());
        buttons.add(okay); buttons.add(cancel);
        setLayout(new BorderLayout());
        add("Center", label);  add("South", buttons);
    }
    public void actionPerformed(ActionEvent ae)
    {   isOkay = (ae.getSource() == okay);
        hide();
    }
}
```

To modify the `MenuTest` program from example 4.27 we need to ensure that we show an appropriate `ConfirmDialog` when the `Exit` menu item is selected and that the program only exists when the `Okay` button is chosen. The changes are minimal and are shown in bold and italics (see figure 4.42).

```
public class MenuTest extends Frame implements ActionListener
{  // all fields and methods remain unchanged, except
   public void actionPerformed(ActionEvent e)
   {  if (e.getSource() == fileExit)
      {  ConfirmDialog exit = new ConfirmDialog(this, "Confirm Exit",
                                                "Do you really want to exit?");
         if (exit.isOkay)
            System.exit(0);
      }
   }
}
```

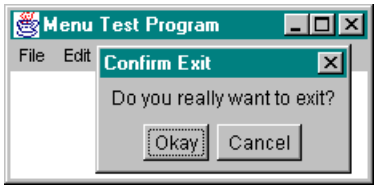



*Figure 4.42:* `MenuTest` *with dialog after selecting* `File | Exit`

■

**Software Engineering Tip:** By convention dialogs should be centered inside the parent window, but by default a `Dialog` appears in the upper left corner of the screen. To properly center a dialog, insert the following statements between `pack()` and `show()`:

```
int x = parent.getLocation().x + (parent.getSize().width-getSize().width)/2;
int y = parent.getLocation().y + (parent.getSize().height-getSize().height)/2;
x = Math.min(getToolkit().getScreenSize().width - getSize().width, x);
y = Math.min(getToolkit().getScreenSize().height - getSize().height, x);
setLocation(Math.max(0, x), Math.max(0, y));
```

The next example uses a non-modal dialog to obtain user input and shows how to call back to the parent class to use its methods.

***Example 4.29: Computing compound interest using a complete GUI program***

In example 2.09, and example 2.15 we computed compound interest for a base amount, using yearly and monthly compounding. Create a GUI program to compute the compound interest for various compounding periods.

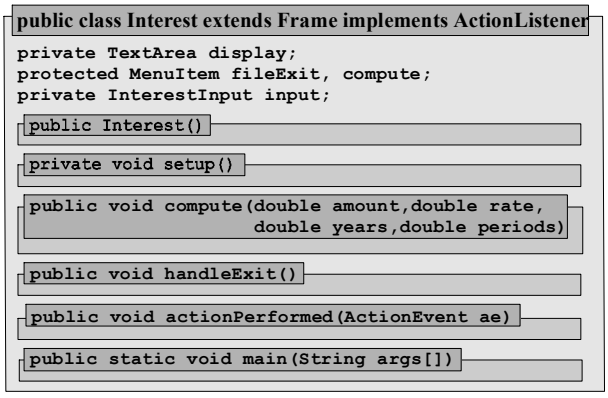
**Problem Analysis:** The formula for the amount of money if `x` dollars are compounded `p` times per year at a rate of `i`% for `n` years is:

$$amount = x \cdot \left(1 + \frac{i}{p \cdot 100}\right)^{p \cdot n}$$

Each parameter in this formula should be definable by the user. We want to use a non-modal dialog box to let the user enter values for the starting amount `x`, the interest rate `i`, and the number of years `n`. As compounding periods we want to use yearly, monthly, and daily compounding and we want to use menus to select them. The output of a program should appear in a text area. Instead of buttons we use menus to navigate the program.

**Class Implementation:** Based on our analysis we know we need at least two classes, an executable class with a menu and text area and class for the input dialog with fields for the parameters.

*1. The Executable Class:* The class `Interest` contains a menu with options `File | Exit` to exit the program and a `Compute | Compute` menu to bring up the input dialog and perform the computation. The class offers a `public compute` method, which does the actual computation, and a `handleExit` method using the `ConfirmDialog` from example 4.28 to exit the program. The `compute` method is called from the input dialog, not this class.




We add a formatting tool as described in section 1.5 to format the amounts as dollar values:

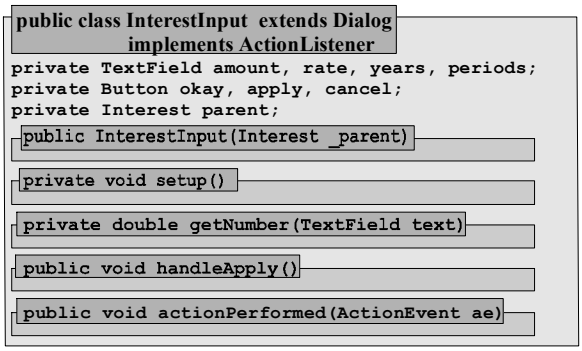
```
import java.text.*;
import java.awt.*;
import java.awt.event.*;

public class Interest extends Frame implements ActionListener
{  private static final DecimalFormat DOLLARS = new DecimalFormat("#,###.00");
   private TextArea display = new TextArea(10, 30);
   protected MenuItem fileExit = new MenuItem("Exit");
   protected MenuItem compute = new MenuItem("Compute");
   private InterestInput input = null;
   private class WindowCloser extends WindowAdapter
   {  public void windowClosing(WindowEvent we)
      {  Interest.this.handleExit(); }
   }

   public Interest()
   {  super("Interest Computer");
      setup();
      input = new InterestInput(this);
      fileExit.addActionListener(this);
```

```
        compute.addActionListener(this);
        addWindowListener(new WindowCloser());
        display.setEditable(false);
        pack(); show();
    }
    private void setup()
    {  Menu fileMenu = new Menu("File");
        fileMenu.add(fileExit);
        Menu computeMenu = new Menu("Compute");
        computeMenu.add(compute);
        MenuBar menuBar = new MenuBar();
        menuBar.add(fileMenu);
        menuBar.add(computeMenu);
        setMenuBar(menuBar);
        setLayout(new BorderLayout());
        add("Center", display);
    }
    public void compute(double amount, double rate,
                        double years,  double periods)
    {  display.append("\n$" +amount +" at " +rate +"% for " +years +" years\n");
        display.append("Compounded " + periods + " times per year:\n\n");
        for (int year = 1; year <= years; year++)
        {  double money = amount * Math.pow(1 + rate/periods/100, periods*year);
            display.append("Year " + year + ": $" + DOLLARS.format(money) + "\n");
        }
    }
    public void handleExit()
    {  ConfirmDialog exit = new ConfirmDialog(this,"Confirm","Really exit?");
        if (exit.isOkay)
            System.exit(0);
    }
    public void actionPerformed(ActionEvent ae)
    {  if (ae.getSource() == fileExit)
            handleExit();
        else if (ae.getSource() == compute)
            input.show();
    }
    public static void main(String args[])
    {  Interest interest = new Interest(); }
}
```

*2. The Input Dialog Class:* The class `InterestInput` contains text fields to enter the base amount, interest rate, number of years, and compounding period and an `Okay`, `Apply`, and `Cancel` button. Clicking on `Okay` calls `compute` of the parent class to perform the computation and closes the dialog. `Apply` performs the computations without closing the dialog, while `Cancel` closes the dialog without any computations. The content of the text fields are converted to double using the `private` method `getNumber`.




```
import java.awt.*;
import java.awt.event.*;

public class InterestInput extends Dialog implements ActionListener
```

```
{  private TextField amount = new TextField("1000.00");
   private TextField rate = new TextField("6.5");
   private TextField years = new TextField("5");
   private TextField periods = new TextField("12");
   private Button okay = new Button("Okay");
   private Button cancel = new Button("Cancel");
   private Button apply = new Button("Apply");
   private Interest parent = null;

   public InterestInput(Interest _parent)
   {  super(_parent, "Interest Data");
      parent = _parent;
      setup();
      okay.addActionListener(this);
      cancel.addActionListener(this);
      apply.addActionListener(this);
      pack();
   }
   private void setup()
   {  Panel panel = new Panel();
      panel.setLayout(new GridLayout(4, 2));
      panel.add(new Label("Amount ", Label.RIGHT));        panel.add(amount);
      panel.add(new Label("Interest rate ", Label.RIGHT)); panel.add(rate);
      panel.add(new Label("Years ", Label.RIGHT));         panel.add(years);
      panel.add(new Label("Periods ", Label.RIGHT));       panel.add(periods);
      Panel buttons = new Panel();
      buttons.setLayout(new FlowLayout());
      buttons.add(okay); buttons.add(apply); buttons.add(cancel);
      setLayout(new BorderLayout());
      add("South", buttons); add("Center", panel);
   }
   private double getNumber(TextField text)
   {  return Double.valueOf(text.getText().trim()).doubleValue(); }
   private void handleApply()
   {  parent.compute(getNumber(amount), getNumber(rate),
                     getNumber(years),  getNumber(periods));
   }
   public void actionPerformed(ActionEvent ae)
   {  if (ae.getSource() == cancel)
         hide();
      else if (ae.getSource() == apply)
         handleApply();
      else if (ae.getSource() == okay)
      {  handleApply();
         hide();
      }
   }
}
```

Figure 4.43 shows the Interest program in action. The non-modal dialog window can be visible and both `Interest` and `InterestInput` can receive events. Computation starts when `Apply` is clicked.

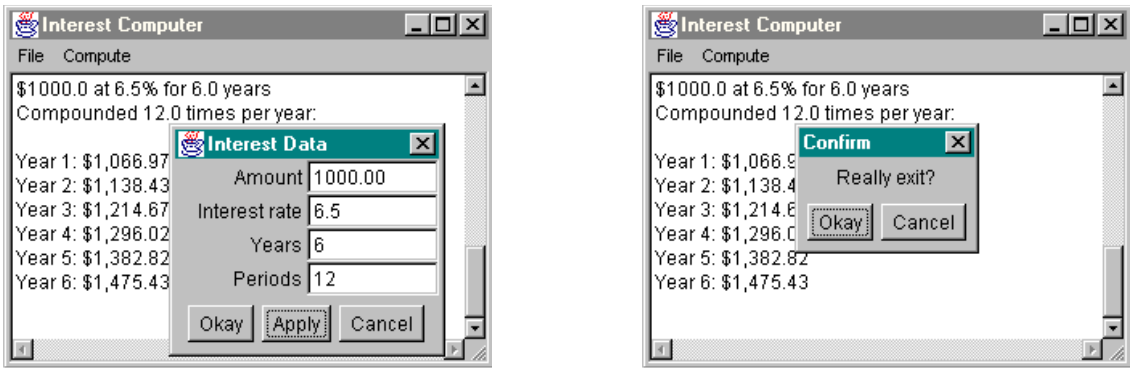



*Figure 4.43: The* `Interest` *program displaying a computation (left) and trying to exit (right)*

■

## 4.7. Drawing and Painting

The last concept we want to cover is how to create customized graphics and drawing, which is surprisingly easy. Any class that extends `java.awt.Component` can draw lines, rectangles, circles, etc using the `Graphics` class provided by the method

```
public void paint(Graphics g)
```

Since `Frame` and `Applet` extend `Component`, you can override the inherited `paint` method to draw.[34]

### The `Graphics` Class

> *The* `Graphics` *class is used to draw inside visible components or off-screen images. A* `Graphics` *object contains all information necessary to accomplish the drawing, including the drawing coordinates, color, font, and clipping regions*[35]*. Drawing takes place in a two-dimensional integer coordinate system whose origin is the upper left-hand corner of the component. The x axis increases horizontally towards the right, the y axis vertically and down (see figure 4.43).*
>
> `Graphics` *objects can not be instantiated and must be obtained from existing* `Graphics` *objects or using the* `getGraphics` *method of a* `Component`*. The* `Graphics` *class contains a multitude of methods as shown in tables 4.45 and 4.46.*

[34] The preferred method for custom drawings is to create a separate class that extends `Canvas` and overrides the `paint` method, as outlined in the *Canvas* class, section 4.7, and example 4.32.
[35] Java version 1.2 and above includes a `Graphics2D` class that provides more control over drawings (see Java API).

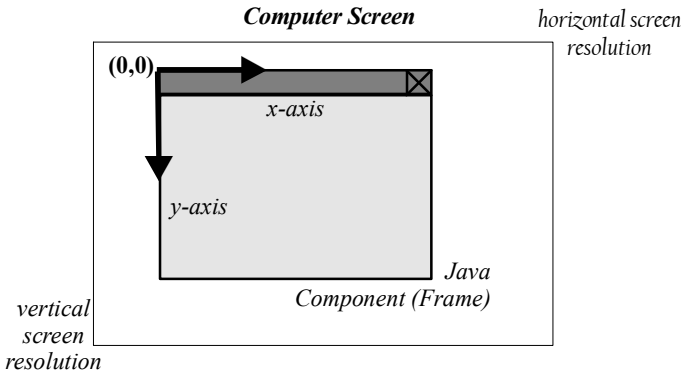



*Figure 4.44: Coordinate system of a component inside the computer screen*

The Graphics class can manipulate the current drawing color and font using the get/set methods shown in table 4.45.

| Get/Set Methods of the Graphics class | |
|---|---|
| `public abstract Color getColor()` | gets current drawing color |
| `public abstract void setColor(Color c)` | sets current drawing color |
| `public abstract Font getFont()` | gets current drawing font |
| `public abstract void setFont(Font font)` | sets current drawing font |
| `public FontMetrics getFontMetrics()` | gets information about font size |

*Table 4.45: Set/get methods of the* `Graphics` *class*

To draw, the methods of the `Graphics` class shown in table 4.46 can be used.

| Drawing Methods of the Graphics class | |
|---|---|
| `drawLine(int x1, int y1, int x2, int y2)` | a line from `(x1,y1)` to `(x2, y2)` |
| `drawRect(int x, int y, int width, int height)` | a rectangle starting at `(x,y)` with `width` and `height`. |
| `draw3DRect(int x, int y, int width, int height, boolean raised)` | 3 dimensional rectangle, either raised or lowered |
| `drawOval(int x, int y, int width, int height)` | draws oval inside rectangle at `(x,y)` with `width` and `height` |
| `drawArc(int x, int y, int width, int height, int start, int arc)` | arc inside rectangle with angle from `start` to `start+arc` |
| `drawPolyline(int x[], int y[], int nPoints)` | connects `(x[0], y[0])`, `(x[1], y[1])`, ... |
| `drawPolygon(Polygon p)` | draws a closed polygon |
| `drawString(String str, int x, int y)` | draws `str` starting at `(x,y)` |
| `clearRect(int x, int y, int width, int height)` | fills rectangle with background color |
| `drawImage(Image img, int x, int y, ImageObserver observer)` | draws image img starting at `(x,y)` and notifies observer about progress |
| `drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)` | as before but automatically scales image to width and height |

*Table 4.46: Drawing methods of the* `Graphics` *class*

The methods `draw3DRect`, `drawArc`, `drawOval`, `drawPolygon`, and `drawRect` are also available as `fill3DRect`, `fillArc`, `fillOval`, `fillPolygon`, and `fillRect` to draw an outline and fill it with the current drawing color.

## Simple Drawings

Since we can not instantiate a `Graphics` object, we must use an existing one to draw. The `paint` method provides such an object as input parameter. If we override that method, we have access to a Graphics object to accomplish the drawing for us.[36] Here is a simple example:

***Example 4.30: Drawing simple graphics primitives***

Create an program that draws an oval, a rectangle, a filled circle, a string, a filled 3D rectangle, and an arc, each in a different color. The filled circle and arc should also have a black border around them.

We create a class extending `Frame` and override the inherited `paint` method. To change colors we use the `setColor` method of a `Graphics` object and the `Color` constants introduced in section 4.4.[37]

```
import java.awt.*;
import java.awt.event.*;

public class GraphicsTest extends Frame
{  private class WindowCloser extends WindowAdapter
   {  public void windowClosing(WindowEvent we)
      {  System.exit(0); }
   }
   public GraphicsTest()
   {  super("Graphics Test");
      setSize(400, 400); show();
      addWindowListener(new WindowCloser());
   }
   public void paint(Graphics g)
   {  g.setColor(Color.black);  g.drawOval(10, 40, 30, 40);
      g.setColor(Color.red);    g.drawRect(40, 50, 50, 60);
      g.setColor(Color.blue);   g.drawString("Hello World", 80, 40);
      g.setColor(Color.green);  g.fillOval(120, 100, 40, 40);
      g.setColor(Color.pink);   g.fillArc(30, 120, 70, 70, 60, 120);
      g.setColor(Color.gray);   g.fill3DRect(90, 120,20, 20, true);
      // now drawing black borders around previous filled objects
      g.setColor(Color.black);
      g.drawOval(120, 100, 40, 40); g.drawArc(30, 120, 70, 70, 60, 120);
   }
   public static void main(String args[])
   {  GraphicsTest gt = new GraphicsTest(); }
}
```

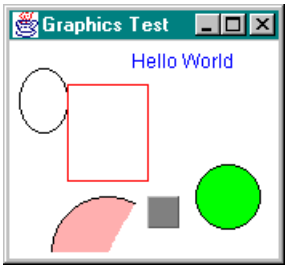



*Figure 4.47: Simple drawings*

■

Graphics can be combined with standard GUI elements, as in the next example.

[36] The `Graphics` class lacks methods, for example, to change the thickness of a line or the pattern of a filled object. The `Graphics2D` class provide such possibilities and is a good alternative to `Graphics`. You can typecast the `Graphics` object `g` as a `Graphics2D` objects in the first line of the `paint` method.

[37] Java also provides access to system-dependent color schemes using the `SystemColor` class (see Java API).

***Example 4.31: The MoveBox program***

Create a program that draws a filled rectangle of a specific size. Add buttons to move the rectangle right, left, up, and down.

**Problem Analysis:** The program extends `Frame`, as usual, and defines a row of buttons that is added to the bottom of the window. Drawing takes place inside the `paint` method, but to make the rectangle movable it is drawn at variable coordinates `x` and `y` defined as fields. These fields can be changed by methods `moveRight`, `moveUp`, etc., which are activated by clicking on the various buttons.

**Class Implementation:** The outline of the class is clear: fields to define the buttons to move the rectangle, integer `x` and `y` fields to store the coordinates of the rectangle, and methods `moveUp`, `moveDown`, `moveLeft`, and `moveRight`. Of course we need to activate the buttons and an `actionPerformed` method to call the various move methods. Here is our first attempt:

```
import java.awt.*;
import java.awt.event.*;

public class MoveBox extends Frame implements ActionListener
{  private final int WIDTH = 30, HEIGHT = 20, INC = 4;
   private Button left  = new Button("Left");
   private Button right = new Button("Right");
   private Button up    = new Button("Up");
   private Button down  = new Button("Down");
   private int x = 50, y = 50;

   public MoveBox()
   {  super("Moving Box");
      setup();
      left.addActionListener(this); right.addActionListener(this);
      up.addActionListener(this);   down.addActionListener(this);
      setSize(400, 400); show();
   }
   private void setup()
   {  Panel buttons = new Panel();
      buttons.setLayout(new FlowLayout());
      buttons.add(up); buttons.add(down);
      buttons.add(left); buttons.add(right);
      setLayout(new BorderLayout());
      add("South", buttons);
   }
   public void paint(Graphics g)
   {  g.fillRect(x, y, WIDTH, HEIGHT); }
   public void moveUp()
   {  y -= INC; }
   public void moveDown()
   {  y += INC; }
   public void moveLeft()
   {  x -= INC; }
   public void moveRight()
   {  x += INC; }
   public void actionPerformed(ActionEvent e)
   {  if (e.getSource() == up)
         moveUp();
      else if (e.getSource() == down)
         moveDown();
      else if (e.getSource() == left)
         moveLeft();
      else if (e.getSource() == right)
```

```
            moveRight();
        }
        public static void main(String args[])
        {  MoveBox mb = new MoveBox(); }
    }
```

The program compiles and draws the initial rectangle, but clicking on the buttons does *not* appear to move the rectangle. It only moves after you click on a button *and resize the window*. That happens because the paint method must be called to draw the box at its new location, and paint is called when constructing the frame and after resizing the window, but not after pressing a button. We need to call the inherited method `repaint` after reacting to a button click, which then in turn calls `paint`. Here is the new `actionPerformed` method, with one added line at the end:

```
public void actionPerformed(ActionEvent e)
{  if (e.getSource() == up)
      moveUp();
   else if (e.getSource() == down)
      moveDown();
   else if (e.getSource() == left)
      moveLeft();
   else if (e.getSource() == right)
      moveRight();
   repaint();
}
```

Now the rectangle moves after every button click without resizing the window.

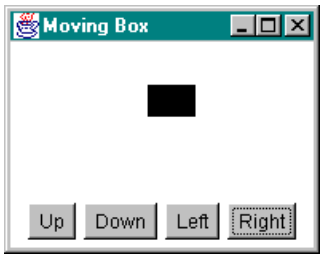



*Figure 4.48:* `MoveBox` *program*

■

The above example illustrates that `paint` is called automatically when the window needs updating and the `repaint` method can be used to force a call to `paint`.

**Software Engineering Tip:** The `Component` class contains 3 methods responsible for drawing:

- `public void repaint()`: Schedules a call to the `update` method.
- `public void update(Graphics g)`: Clears the drawing area and calls the `paint` method.
- `public void paint(Graphics g)`: Renders the `Graphics` object `g`.

To draw, you override `paint` to contain your drawing code. If you need to update your drawing, call `repaint`. The `repaint` method is also called automatically whenever the window needs updating (for example, if the window is resized). It is not usually overridden. You can override the `update` method to customize how the drawing area is cleared (for example to define a background image), but then you must include an explicit call to `paint`.

Instead of overriding the paint method of a frame (or applet), it is more convenient to use a special component called `Canvas` to contain the drawing code.

## Canvas and Fonts

If drawing takes place inside a frame's or applet's `paint` method it can be overlaid by other GUI elements, because the coordinate system for all components, including the `Graphics` object of the `paint` method, starts in the upper left-hand corner of the window. A frame, for example, contains a title bar with a standard close box, which covers a small band at the top of the drawing area. Any drawing with coordinates inside that area is covered by the window bar. The solution is to not draw directly into a frame (or applet) but instead in a `Canvas`, which has its own coordinate system and can be positioned with a layout manager.

### Canvas

> *A* `Canvas` *is a component with its own coordinate system that can be positioned inside other components using a layout manager.* `Canvas` *extends* `Component` *and inherits the methods* `public void paint(Graphics g)`, `public void update(Graphics g)`, *and* `public void repaint()`. *The* `repaint` *method is called automatically when the* `Canvas` *needs updating and can also be called explicitly. It calls* `update`*, which clears the screen and calls* `paint`*. The* `paint` *method can be overridden to contain drawing code.*

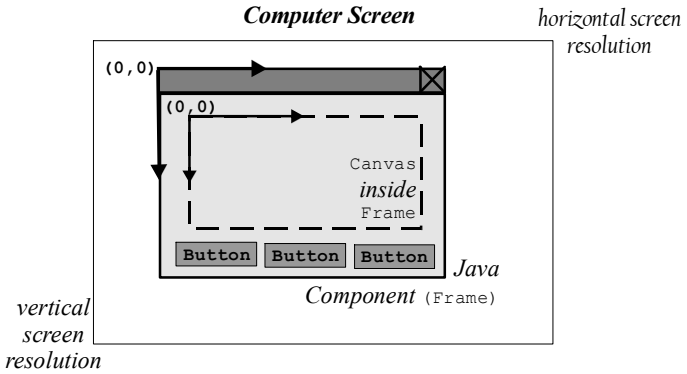



*Figure 4.49: A* `Canvas` *inside a* `Frame`*, both with their own coordinate systems*

### *Example 4.32: Positioning graphics with and without a Canvas*

Create an applet containing a single button in a `Panel` with a `FlowLayout`. Position the panel "`North`" inside the apple. Override the `paint` method of the applet to draw a String every 12 pixels, top to bottom. Describe the problem that you see, if any. Then use a `Canvas` to position the drawing in the center of the applet.

The `paint` method of the applet contains a loop from 12 to the height of the applet in steps of 12. Each time it draws a string at the current value of the looping variable.

```
import java.applet.*;
import java.awt.*;

public class DrawNoCanvas extends Applet
{  private Button draw = new Button("Draw");

   public void init()
```

```
   {  Panel buttons = new Panel();
      buttons.setLayout(new FlowLayout());
      buttons.add(draw);
      setLayout(new BorderLayout());
      add("North", buttons);
   }
   public void paint(Graphics g)
   {  for (int i = 12; i < getSize().height; i+=12)
         g.drawString("y location: " + i, 10, i);
   }
}
```

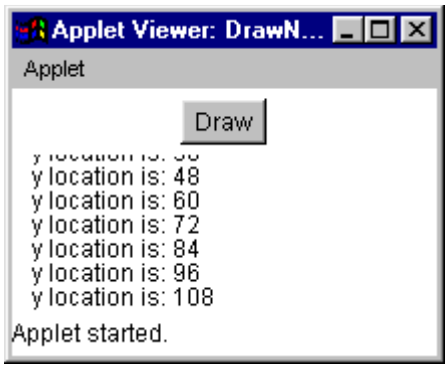



*Figure 4.50: Drawing without a* `Canvas`

Figure 4.50 shows that the `buttons` panel has cut-off the first few strings drawn. Therefore we redo the class by moving the drawing code into a new class extending `Canvas`. The `Canvas` is then added to the applet as a field and positioned by a layout manager. The strings are no longer cut-off by another component.

```
import java.applet.*;
import java.awt.*;

public class DrawWithCanvas extends Applet
{  private Button draw = new Button("Draw");
   private DrawCanvas drawing = new DrawCanvas();

   public void init()
   {  Panel buttons = new Panel();
      buttons.setLayout(new FlowLayout());
      buttons.add(draw);
      setLayout(new BorderLayout());
      add("North", buttons);
      add("Center", drawing);
   }
}

public class DrawCanvas extends Canvas
{  public void paint(Graphics g)
   {  for (int i = 12; i < getSize().height; i+=12)
         g.drawString("y location: " + i, 10, i);
   }
}
```

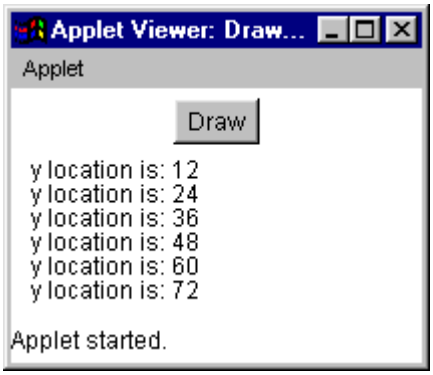



*Figure 4.51: Drawing with a* `Canvas`

■

**Software Engineering Tip:** To create graphics, create a class that extends `Canvas` and overrides the `paint` method. Add that class as a field to another class such as a frame or applet and position it using a layout manager. Separate the tasks that the classes need to accomplish:

- The frame (or applet) acts as controller, positions all components, handles user input, and delegates work to other objects.
- The canvas handles the drawing. Its most important methods are `paint`, which is overridden, and `repaint`, which can be called by the controller when necessary. Additional methods can be defined to manipulate the drawing.

### *Example 4.33: Adding a Canvas to the MoveBox program*

In example 4.31 we created a `MoveBox` program to move a rectangle around the screen. Rewrite `MoveBox` so that the drawing is contained in a `Canvas` and the rectangle 'wraps' around, i.e. if moved too far to the left it reappears on the right, etc.

Previously our program had the structure shown in figure 4.52. The fields `x` and `y` and the methods `moveUp`, `moveDown`, `moveLeft`, and `moveRight` were responsible for manipulating the drawing. The `actionPerformed` method reacted to button clicks by calling one of these methods, followed by a call to `repaint` to update the drawing.

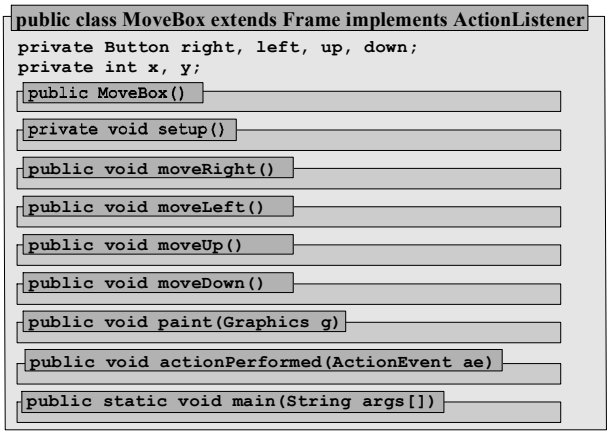



*Figure 4.52: Representation of* `MoveBox`

To separate tasks, we move the fields and methods responsible for drawing to a new class extending `Canvas` and call its methods from the `Frame` class to react to user input. We also expand the various "move" methods to wrap the rectangle to the other side when necessary.

The class responsible for drawing looks as follows:

```
import java.awt.*;

public class MoveBoxCanvas extends Canvas
{  private final int WIDTH = 30, HEIGHT = 20, INC = 4;
   private int x = 50, y = 50;

   public void paint(Graphics g)
   {  g.fillRect(x, y, WIDTH, HEIGHT);
      g.drawRect(0, 0, getSize().width-1, getSize().height-1);
   }
   public void moveUp()
   {  if (y > 0)
         y -= INC;
      else
         y = getSize().height - INC;
   }
   public void moveDown()
   {  if (y < getSize().height - INC)
         y += INC;
      else
         y = 0;
   }
   public void moveLeft()
   {  if (x > 0)
         x -= INC;
      else
         x = getSize().width - INC;
   }
   public void moveRight()
   {  if (x < getSize().width - INC)
         x += INC;
      else
```

```
            x = 0;
      }
   }
```

Each of the "move" methods checks whether the rectangle is close to a border (the `getSize` method returns the size of the canvas). If so, we change coordinates to make the rectangle appear on the opposite side. The modified `paint` method draws the rectangle and adds a border to the canvas.

The controlling class extends `Frame` as before and uses `MoveBoxCanvas` as a field. It defines the layout, adds the canvas, and calls the methods of the canvas to manipulate the appearance of the drawing in the `actionPerformed` method.

```
import java.awt.*;
import java.awt.event.*;

public class MoveBoxWithCanvas extends Frame implements ActionListener
{  private Button left  = new Button("Left");
   private Button right = new Button("Right");
   private Button up    = new Button("Up");
   private Button down  = new Button("Down");
   public MoveBoxCanvas drawing = new MoveBoxCanvas();

   public MoveBoxWithCanvas()
   {  super("Moving Box");
      setup();
      left.addActionListener(this); right.addActionListener(this);
      up.addActionListener(this);   down.addActionListener(this);
      setSize(400, 400); show();
   }
   private void setup()
   {  Panel buttons = new Panel();
      buttons.setLayout(new FlowLayout());
      buttons.add(up);   buttons.add(down);
      buttons.add(left); buttons.add(right);
      setLayout(new BorderLayout());
      add("South", buttons);
      add("Center", drawing);
   }
   public void actionPerformed(ActionEvent e)
   {  if (e.getSource() == up)
         drawing.moveUp();
      else if (e.getSource() == down)
         drawing.moveDown();
      else if (e.getSource() == left)
         drawing.moveLeft();
      else if (e.getSource() == right)
         drawing.moveRight();
      drawing.repaint();
   }
   public static void main(String args[])
   {  MoveBoxWithCanvas mb = new MoveBoxWithCanvas(); }
}
```

■

A drawing contained in a `Canvas` becomes a reusable object that can be used in other projects.

### *Example 4.34: A reusable StopSign class*

Create a `StopSign` class that can be used to add a stop symbol to other classes. Test the class by adding a stop symbol to the `ConfirmDialog` created in example 4.28.

**Problem Analysis:** A stop sign looks like a regular octagon (a polygon with 8 corners), filled in red (think of a street sign), as shown in figure 4.53. Its corner points have coordinates:

```
(10,0), (20,0), (30, 10), (30, 20),
(20,30), (10,30), (0, 20), and (0, 10)
```

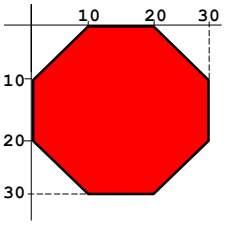



*Figure 4.53: A regular octagon as a stop sign*

**Class Implementation:** The `StopSign` class contains graphics, so it extends `Canvas`. We define the `x` and `y` coordinates of the octagon as constant arrays of integers, which are used to initialize a `Polygon` with those coordinates.[38] Another constant defines the color of the rectangle. The `paint` method draws the octagon by using the `fillPolygon` and `drawPolygon` methods of the `Graphics` class. No GUI components or layout managers are used, which means that the class does not know its own preferred size. Therefore we override the method `getPreferredSize` that is part of a `Component` and is called automatically when a layout manager wants to position the component. It returns the preferred dimension that our class wants to have.

```
import java.awt.*;

public class StopSign extends Canvas
{  private final static int[] X = {10, 20, 30, 30, 20, 10,  0,  0};
   private final static int[] Y = { 0,  0, 10, 20, 30, 30, 20, 10};
   private final static Polygon P = new Polygon(X, Y, X.length);
   private final static Color FILL_COLOR = new Color(200, 0, 0);

   public StopSign ()
   {  super();
      setBackground(Color.lightGray);
   }
   public Dimension getPreferredSize()
   {  return new Dimension(32, 32); }
   public void paint(Graphics g)
   {  g.setColor(FILL_COLOR);
      g.fillPolygon(P);
      g.setColor(Color.black);
      g.drawPolygon(P);
   }
}
```

The class is easy to use since it handles all drawing completely on its own. It can be added to the `ConfirmDialog` class as a field and positioned by a layout manager just as other GUI components, as long as it is saved in the same directory as the class using it. The modifications to the `ConfirmDialog` class from example 4.28 are shown in bold and italics below.

```
public class ConfirmDialog extends Dialog
             implements ActionListener
{  // fields as before as well as:
   private StopSign stop = new StopSign();
```

[38] A `Polygon` is a class contained in the `java.awt` package, together with other classes representing geometric objects or properties such as `Rectangle`, `Point` and `Dimension`. For details, check the Java API.

```
    // everything but setup as before:
    private void setup()
    {  // as before but append the line:
       add("West", stop);
    }
    public void actionPerformed(ActionEvent ae)
    {  /* as before, no change */ }
}
```


*Figure 4.54:* `ConfirmDialog` *with* `StopSign`

■

**Software Engineering Tip:** A class that extends `Component` contains an inherited `getPreferredSize` method, which is called by a layout manager to obtain the preferred size of the component. If the class itself uses layout managers to add components, the `getPreferredSize` method is automatically adjusted so that all components can fit. To explicitly specify the size of a component, override the method

```
public Dimension getPreferredSize()
```

Classes that extend `Canvas` should always override the `getPreferredSize` method.

To conclude our discussions about graphics and drawing, we briefly mention font support. Fonts are highly system-dependent and Java provides methods to obtain a list of fonts installed on a particular system. Since a detailed discussion about fonts takes too long, we restrict ourselves to deal with simple logical fonts only.

## Font

*The* `Font` *class represents fonts, which are divided into physical, system-dependent fonts and logical, system independent fonts. The JVM automatically substitutes appropriate physical fonts for the logical font names* `Dialog`, `DialogInput`, `Monospaced`, `Serif`, *and* `SansSerif`. *To construct a* `Font` *object, use the syntax*

```
Font myFont = new Font(String name, int style, int size)
```

*where* `name` *is a logical font name,* `style` *is a combination of* `Font.PLAIN`, `Font.BOLD`, *or* `Font.ITALIC`, *and* `size` *is the point size of the font. Fonts can be defined for every class that extends* `Component` *as well as for the* `Graphics` *class using the* `setFont` *method.*[39]

### *Example 4.35: Displaying available logical fonts*

Create a program that shows the available logical fonts in various styles and sizes.

The program extends `Frame`, defines a final array of logical font names, and overrides the `paint` method to draw some text in these logical fonts. The output is shown in figure 4.55.

```
import java.awt.*;
import java.awt.event.*;
```

[39] To specify, for example, the font of a `Button b`, use `b.setFont(new Font("Dialog", Font.BOLD, 12));`

```
public class FontExample extends Frame
{  private static final String[] FONTS =
       {"Dialog", "DialogInput", "Monospaced", "Serif", "SansSerif"};
   private static final String TEXT = "A logical font example";

   public FontExample()
   {  super("Font Examples"); setSize(300, 300); show(); }
   public void paint(Graphics g)
   {  for (int i = 0; i < FONTS.length; i++)
      {   g.setFont(new Font(FONTS[i], Font.PLAIN, 12));
          g.drawString(FONTS[i] + " (plain): " + TEXT, 10, 20*i + 40);
      }
      for (int i = 0; i < FONTS.length; i++)
      {  g.setFont(new Font(FONTS[i], Font.BOLD + Font.ITALIC, 14));
         g.drawString(FONTS[i] + "(bold, italics): " + TEXT, 10, 20*i + 180);
      }
   }
   public static void main(String args[])
   {  FontExample fe = new FontExample(); }
}
```

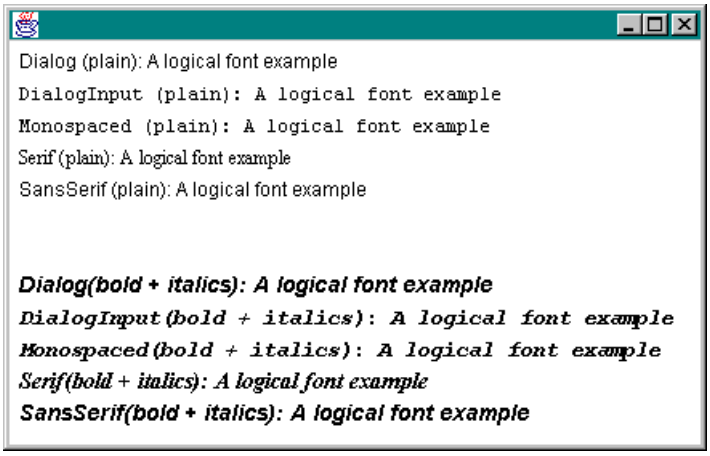



*Figure 4.55: Program displaying strings in different fonts and styles*

■

Additional information such as loading images and off-screen drawing is available in chapter 6.

# Case Study: An AddressBook Program

This section, which is optional, pulls together all topics we introduced in this chapter to create an complete, GUI-based, object oriented address book program to store and view different types of addresses. Our program uses frames, menus, fonts, dialogs, listener adapters, inheritance, polymorphism, and some previously created reusable classes.

### *Example 4.36: A complete address book program*

Create a complete address book program that:

- can handle two types of addresses (a public address containing a first name, last name, and email address, and a private one to also store a phone number)
- can add, delete, and edit addresses

- uses menus instead of buttons to perform the various actions
- uses dialogs to add and edit an address

The program does not need to save, retrieve, or print addresses.

**Problem Analysis:** To identify useful classes for this program we first describe how we envision it to function:

- When the program starts, a window appears that contains a menu, a list with names in the address book, and a display area to display a complete address.
- The user chooses from the menu which type of new address to insert into the address book.
- After the user has provided the information for an address, it is added to the list of existing addresses.
- A user can select a name to see the corresponding address, or to edit or remove the selected address.

This functionality suggests several classes:

- An `AddressBook` class extending `Frame` to represent the complete program.
- An `AddressDialog` class extending `Dialog` where the user can add or modify information for an address.
- An `AddressDisplay` class to show a complete address in a nice format. To create an appealing format, this class extends `Canvas`.

In addition, we clearly need two classes representing the two types of addresses and a flexible structure where objects can be inserted and retrieved at any time. The `List` class created in example 3.34 is just such a structure so we reuse it for our current project.

The AddressDialog and AddressDisplay classes each must handle two types of addresses, which can be accomplished using polymorphism.

Table 4.56 summarizes the classes we decided to use, together with our standard "is-a" and "has-a" terminology to help determine their fields and methods.

| Class | "is-a" | "has-a" | "does-a": |
|---|---|---|---|
| `Address` | `Object` | `firstName, lastName, email` | `getAddress, setAddress` |
| `PrivateAddress` | `Address` | as above, plus `phone` | as above |
| `AddressDialog` | `Dialog` | `Buttons, TextFields, Address` | `getData` |
| `AddressDisplay` | `Canvas` | `Address` | `setText` |
| `AddressBook` | `Frame` | `Menu, List, AddressDialog, AddressDisplay` | `handleQuit, handleAdd, handleDel, handleSort, handleShow, main` |

*Table 4.56: Overview of classes making up the* `AddressBook` *program*

Figure 4.57 attempts to show how the various classes work together.

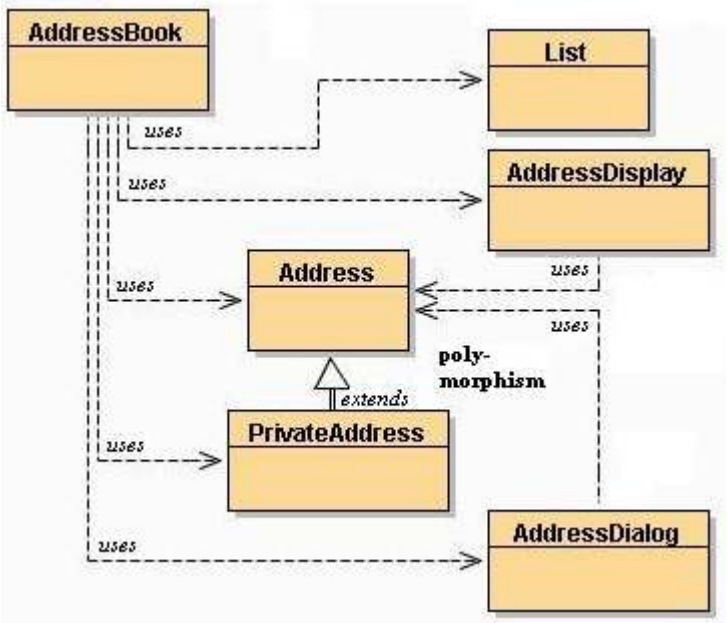



*Figure 4.57: Classes and methods for* `AddressBook` *program*

**Class Implementation**: The easiest classes to implement are `Address` and `PrivateAddress` (see figure 4.58). To ensure that polymorphism can work, the methods `setAddress` and `getAddress` use arrays of `String` as input/output parameters so that they can work with a flexible amount of information. We added a method `getLabels` that returns an array of `String`, representing the labels for the data fields of the class, and we override the inherited `toString` method to return the "name" portion of an address.

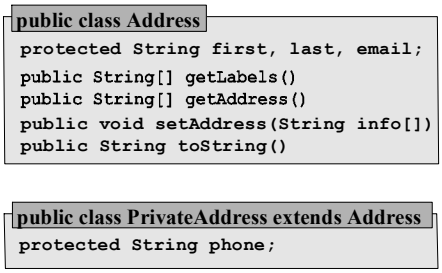



*Figure 4.58: Address and PrivateAddress classes*

```
public class Address
{  protected String firstName = "", lastName = ""; email = "";

   public String[] getLabels()
   {  String s[] = {"First Name:", "Last Name:", "Email:"};
      return s;
   }
   public String[] getAddress()
   {  String s[] = {firstName, lastName, email};
      return s;
   }
   public void setAddress(String info[])
   {  firstName = info[0];
      lastName = info[1];
      email = info[2];
   }
   public String toString()
   {   return lastName + ", " + firstName; }
}
```

`PrivateAddress` inherits the methods from `Address`, but overrides them with its own versions. Since the methods have the same header, polymorphism can be used to automatically choose the correct version.

```
public class PrivateAddress extends Address
{  protected String phone = "";

   public String[] getLabels()
   {  String s[] = {"First Name:", "Last Name:", "Email:", "Phone:"};
      return s;
   }
   public String[] getAddress()
   {  String s[] = {firstName, lastName, email, phone};
      return s;
   }
   public void setAddress(String info[])
   {  firstName = info[0];
      lastName = info[1];
      email = info[2];
      phone = info[3];
   }
}
```

To see how polymorphism works, we create the `AddressDisplay` class next. It receives an `Address` through the `setText` method and uses the `getLabels` and `getAddress` methods of that `Address` to extract the information to display. The `paint` method then properly formats the address, doing some simple calculations to determine the location for the text. The `computeSize` method computes the preferred dimensions of the canvas based on the information stored in the address and the fonts chosen and is called by the inherited `setSize` method every time a new address is set via `setText`.

Note that if the input to `setText` is of type Address, the `getAddress` method of the address returns an array of String of size three. If the input is of type `PrivateAddress`, the array returned has length four, via polymorphism.

**public class AddressDisplay extends Canvas**

```
private static Font FONT_LARGE, FONT_PLAIN;
private static FontMetrics METRICS_LARGE;
private static FontMetrics METRICS_PLAIN;
private Address address;

public AddressDisplay()
public void setText(Address _address)
private Dimension computeSize()
public void paint(Graphics g)
```

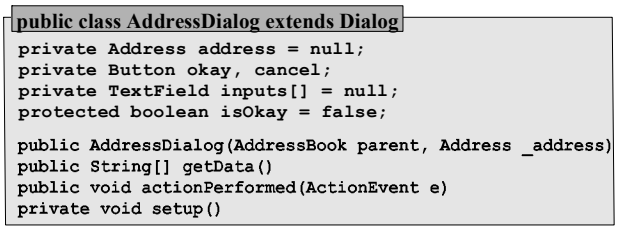
```
import java.awt.*;

public class AddressDisplay extends Canvas
{  private static Font FONT_LARGE = new Font("Serif", Font.BOLD, 16);
   private static Font FONT_PLAIN = new Font("Serif", Font.PLAIN, 12);
   private static FontMetrics METRICS_LARGE, METRICS_PLAIN;
   private Address address = null;

   public AddressDisplay()
   {  super();
      METRICS_LARGE = getFontMetrics(FONT_LARGE);
      METRICS_PLAIN = getFontMetrics(FONT_PLAIN);
   }
   public void setText(Address _address)
   {  address = _address;
      setSize(computeSize());
   }
   private Dimension computeSize()
   {  int max = METRICS_LARGE.stringWidth(address.toString());
      for (int i = 2; i < address.getLabels().length; i++)
         {  String text = address.getLabels()[i]+address.getAddress()[i];
```

```
            if (max < METRICS_PLAIN.stringWidth(text))
               max = METRICS_PLAIN.stringWidth(text);
         }
         return new Dimension(max + 20, address.getLabels().length * 20);
   }
   public void paint(Graphics g)
   {  if (address != null)
      {  g.setFont(FONT_LARGE);
         g.setColor(Color.blue);
         g.drawString(address.toString(), 5, 20);
         for (int i = 2; i < address.getLabels().length; i++)
         {  String text = address.getLabels()[i]+address.getAddress()[i];
            g.setFont(FONT_PLAIN);
            g.setColor(Color.black);
            g.drawString(text, 10, 20*i);
         }
      }
   }
}
```

Polymorphism is also used for the `AddressDialog` class, which is a modal dialog. The constructor receives an `Address` as input, which is used to initialize an array of text fields and labels. If a true `Address` is used, there are *three* text fields and labels, if a `PrivateAddress` is used, *four* fields and labels are created and laid out.




The protected field `isOkay` is set to true if the `okay` button was pressed so that the parent class can use `getData` to retrieve the information from the dialog.

```
import java.awt.*;
import java.awt.event.*;

public class AddressDialog extends Dialog implements ActionListener
{  private Address address = null;
   private Button okay = new Button("Okay");
   private Button cancel = new Button("Cancel");
   private TextField inputs[] = null;
   protected boolean isOkay = false;
   private class WindowCloser extends WindowAdapter
   {  public void windowClosing(WindowEvent we)
      {  AddressDialog.this.hide(); }
   }

   public AddressDialog(AddressBook parent, Address _address)
   {  super(parent, "Address Information", true);
      address = _address;
      setup();
      okay.addActionListener(this);
      cancel.addActionListener(this);
      addWindowListener(new WindowCloser());
      pack(); show();
   }
   public String[] getData()
   {  String inputStrings[] = new String[inputs.length];
      for (int i = 0; i < inputs.length; i++)
```

```
            inputStrings[i] = inputs[i].getText().trim();
         return inputStrings;
      }
      public void actionPerformed(ActionEvent e)
      {  if (e.getSource() == okay)
            isOkay = true;
         this.hide();
      }
      private void setup()
      {  Panel data = new Panel();
         data.setLayout(new GridLayout(address.getLabels().length, 2));
         inputs = new TextField[address.getLabels().length];
         for (int i = 0; i < address.getLabels().length; i++)
         {  inputs[i] = new TextField(address.getAddress()[i],20);
            data.add(new Label(address.getLabels()[i], Label.RIGHT));
            data.add(inputs[i]);
         }
         Panel buttons = new Panel();
         buttons.setLayout(new FlowLayout());
         buttons.add(okay); buttons.add(cancel);
         setLayout(new BorderLayout());
         add("Center", data);
         add("South", buttons);
      }
   }
```

With these classes in place we can now create the main `AddressBook` class. It uses of the `List` class created in example 3.34, which must be saved to the directory containing our current program. A `List` object becomes a field for `AddressBook`, as well as a standard `List` object from the `java.awt` package. The `java.awt.List` object contains a clickable list of names, where each name corresponds to an `Address` stored in the `List` at the same position. The `handleAdd`, `handleEdit`, and `handleDel` must make sure that the two lists are always in sync, which is simplified by the fact that the `List` class from example 3.34 adds new objects at the end, just as the add method for `java.awt.List`.

```
public class AddressBook  extends Frame
        implements ActionListener, ItemListener

   private List book;
   private AddressDialog dialog;
   private awt.List addresses
   private MenuItem fileExit, toolAddPrivate,
                    toolAddPublic, toolDel, toolShow;
   private AddressDisplay display;
   private AddressDialog dialog;
   private Address address;

   public AddressBook()
   public void itemStateChanged(ItemEvent ie)
   public void actionPerformed ActionEvent ae)
   private void handleEdit()
   private void handleAdd(Address newAddress)
   private void handleDel()
   private void handleShow()
   private void setup()
   public static void main(String args[])
```

To add scrolling capabilities to the canvas displaying an address, we add the `AddressDisplay` object to a `ScrollPane`, which in turn is added to the `AddressBook`. The `ScrollPane` will automatically provide scroll bars and handle scrolling on its own, provided that the dimensions of `AddressDisplay` are adjusted for every new address to display.

```
import java.awt.*;
import java.awt.event.*;

public class AddressBook extends Frame
       implements ActionListener, ItemListener
{  private List book = new List();
   private java.awt.List addresses = new java.awt.List();
   private MenuItem fileExit = new MenuItem("Exit");
   private MenuItem toolAddPrivate = new MenuItem("Private Address");
```

```
private MenuItem toolAddPublic  = new MenuItem("Public Address");
private MenuItem toolDel = new MenuItem("Delete Address");
private MenuItem toolShow = new MenuItem("Show Address");
private AddressDisplay display = new AddressDisplay();
private AddressDialog dialog = null;
private Address address = null;
private class WindowCloser extends WindowAdapter
{  public void windowClosing(WindowEvent we)
   {  System.exit(0); }
}

public AddressBook()
{  super("Address Book");
   setup();
   fileExit.addActionListener(this);
   toolAddPrivate.addActionListener(this);
   toolAddPublic.addActionListener(this);
   toolDel.addActionListener(this);
   toolShow.addActionListener(this);
   addresses.addActionListener(this);
   addresses.addItemListener(this);
   addWindowListener(new WindowCloser());
   pack(); show();
}
public void itemStateChanged(ItemEvent e)
{  if (addresses.getSelectedIndex() >= 0)
      handleShow();
}
public void actionPerformed(ActionEvent e)
{  Object target = e.getSource();
   if (target == fileExit)
      System.exit(0);
   else if (target == toolDel)
      handleDel();
   else if (target == toolAddPrivate)
      handleAdd(new PrivateAddress());
   else if (target == toolAddPublic)
      handleAdd(new Address());
   else if ( ((target == toolShow) || (target == addresses)) &&
             (addresses.getSelectedIndex() >= 0))
      handleEdit();
}
private void handleEdit()
{  int pos = addresses.getSelectedIndex();
   address = (Address)book.get(pos);
   dialog = new AddressDialog(this, address);
   if (dialog.isOkay)
   {  address.setAddress(dialog.getData());
      addresses.replaceItem(address.toString(), pos);
      display.setText(address);
      validate();
   }
}
private void handleAdd(Address newAddress)
{  dialog = new AddressDialog(this, newAddress);
   if (dialog.isOkay)
   {  newAddress.setAddress(dialog.getData());
      addresses.add(newAddress.toString());
      book.add(newAddress);
   }
}
private void handleDel()
{  if (addresses.getSelectedIndex() >= 0)
```

```
            {  book.delete(addresses.getSelectedIndex());
               addresses.remove(addresses.getSelectedIndex());
            }
         }
         private void handleShow()
         {  address = (Address)book.get(addresses.getSelectedIndex());
            display.setText(address);
            validate();
         }
         private void setup()
         {  Menu file = new Menu("File");
            file.add(fileExit);
            Menu edit = new Menu("Edit");
            Menu toolAdd = new Menu("Add Address");
            toolAdd.add(toolAddPrivate); toolAdd.add(toolAddPublic);
            Menu tools = new Menu("Tools");
            tools.add(toolAdd);
            tools.addSeparator();
            tools.add(toolDel); tools.add(toolShow);
            MenuBar bar = new MenuBar();
            bar.add(file);  bar.add(edit);  bar.add(tools);
            setMenuBar(bar);
            ScrollPane scroller = new ScrollPane();
            scroller.add(display);
            setLayout(new GridLayout(1,2));
            add(addresses); add(scroller);
         }
         public static void main(String args[])
         {  AddressBook book = new AddressBook(); }
      }
```

Figure 4.59 shows a few screen shots of our complete program in action.

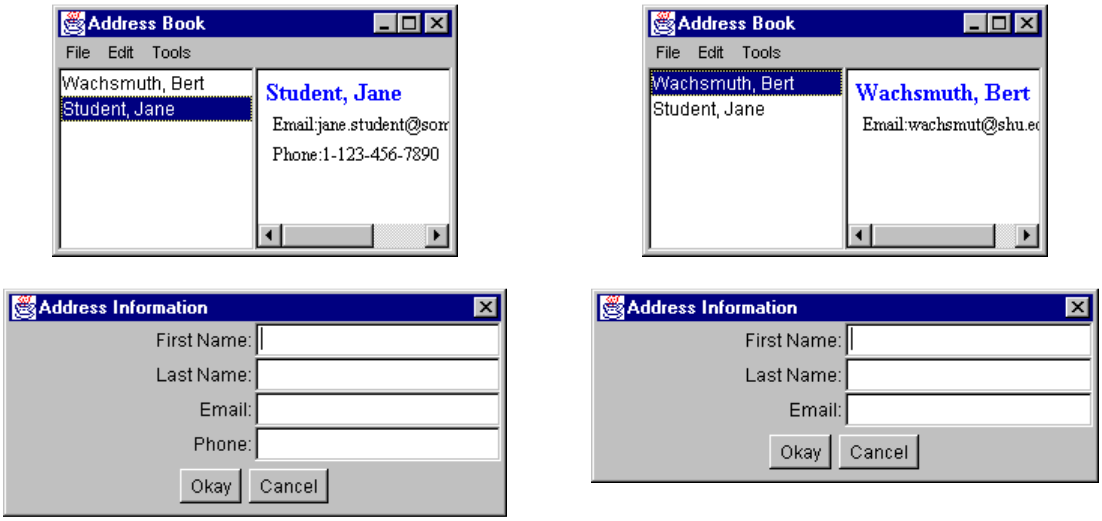



*Figure 4.59: The AddressBook program in action*

# Chapter Summary

In this chapter we introduced the following concepts and terminology:

**Package**
See section 4.1, example 4.01 (*Extending a Button and String*)
**AWT – Abstract Windows Toolkit**
**The Frame Class**
See section 4.2, example 4.02 (*A simple Frame-based program*)
**The Window Class, The Container Class**
See section 4.2, examples 4.03 (*Making a Frame visible*) and 4.04 (*Adding buttons to a Frame*)
**The Button Class, The ActionListener Interface**
See section 4.2, examples 4.04 (*Adding buttons to a Frame* […]) and 4.05 (*A complete Frame based program with buttons*)
**The TextField Class, The TextComponent Class**
See section 4.2, example 4.05 (*A complete Frame based program with buttons* […])
**The Applet Class**
See section 4.3, example 4.06 (*A simple Applet with buttons*)
**The Applet Tag**
See section 4.3, examples 4.07 (*Embedding an applet in an HTML file*), 4.08 (*Applets with PARAM tags*), and 4.09 (*The AddingMachine applet*)
**Event**
See section 4.4, examples 4.10 (*How Frame based programs exit*) and 4.11 (*A program with a WindowListener*)
**The Java Event-Handling Scheme, Event Listeners**
See section 4.4, examples 4.12 (*Processing WindowEvent, MouseEvent, and KeyEvent*), 4.13 (*A program with a separate event handler class*), and 4.14 (*Adding a KeyListener to the AddingMachine applet*)
**Listener Adapters**
See section 4.4, example 4.15 (*Handling events via adapter classes*)
**Inner Classes**
See section 4.4, examples 4.16 (*Anonymous and named inner class adapters*) and 4.17 (*A ColorSelector class*)
**The Color Class**
See section 4.4, example 4.17 (*A ColorSelector class* […])
**The ActionEvent Class**
See section 4.4, example 4.17 (*A ColorSelector class* […])
**The AWTEventMulticaster**
See section 4.4, examples 4.17 (*A ColorSelector class* […]) and 4.18 (*Generating events automatically*)
**LayoutManager**
See section 4.5, example 4.19 (*FlowLayout, GridLayout, and BorderLayout in action*)
**The Panel Class**
See section 4.5, examples 4.20 (*Identifying layout possibilities*), 4.21 (*Combining layouts using the Panel class*), and 4.22 (*A redesigned AddingMachine applet*)
**The Label Class**
See section 4.5, example 4.23 (*A standard calculator applet*)
**The List Class**
See section 4.5, example 4.24 (*A ToDo program using List and TextField*)
**The TextArea Class**
See section 4.5, examples 4.25 (*Using the TextArea class*) and 4.26 (*Adding a TextArea to the Calculator applet*)
**The Menu, MenuBar, and MenuItem Classes**
See section 4.6, example 4.27 (*A program with a standard menu bar*)
**Dialog**

See section 4.6, examples 4.28 (*Adding a simple dialog to a program*) and 4.29 (*Computing compound interest using a complete GUI program*)

**The Graphics Class**

See section 4.7, examples 4.30 (*Drawing simple graphics primitives*) and 4.31 (*The MoveBox program*)

**Canvas**

See section 4.7, examples 4.32 (*Positioning graphics with and without a Canvas*), 4.33 (*Adding a Canvas to the MoveBox program*), and 4.34 (*A reusable StopSign class*)

**Font**

See section 4.7, example 4.35 (*Displaying available logical fonts*)

**Case Study: An AddressBook Program**