Chapter 6: Swing and Multimedia

With the availability of cheap computers that support sound and high-speed animation the concept of multimedia has received much attention in recent years. Many computers today allow you to play music CD's, movie clips, or even watch TV in a separate window while you work with another program. With the recent introduction of DVD disks – a version of a CD ROM disk that uses advanced compression and storage techniques – it has become possible to digitally store entire feature movies on a disk and play them back on a computer.

Java, as a modern programming language, also allows for adding some multimedia components to an application or applet, and third-party vendors are increasingly providing Java classes that allow using many of the standard multimedia formats to be used with Java classes. For example, Apple is working to release Java classes that allow embedding QuickTime and MPEG movies into a Java application.

In addition to supporting multimedia, it has become apparent that Java needed more sophisticated GUI elements that allow for more flexibility than the AWT introduced in chapter 4. SUN Microsystems decided to provide a more flexible and more capable version of the AWT, and since Java version 1.2 SUN has added a collection of packages ad classes called Swing to its JDK. One interesting feature of Swing classes is the ability for the programmer – and in fact the user – to choose an overall look and feel of an application that could be different from that of standard underlying operating system. Swing allows the user to change the appearance of a running program at any time to make it look either like a standard Windows program, a Macintosh program, or a Unix program, regardless of the operating system under which the program is executing. One can even choose a distinct "Java-like" appearance, which has been selected as the default look for Swing based programs and applets.

In this section we will introduce the more common Swing components and learn how to create programs and applets based on Swing classes instead of AWT classes. That will give our programs a much more professional look and we will have several very useful classes at our disposal that would be difficult to recreate using the AWT. We will also show how to convert a program using AWT classes into an equivalent one using Swing. In subsequent chapters, however, we will resort back to the AWT because of simplicity and downward compatibility. It is always possible and reasonably easy to convert any of the programs in the following chapters to equivalent Swing-based programs if a highly professional and customizable appearance is desired.

We will also introduce some common multimedia techniques such as loading and displaying images and animations and adding sound support to programs and applets. Finally, we will show an example of creating an application that uses sophisticated and resource-intensive drawing techniques such as "off-screen" drawing and "rubber-banding".

To make it easier to switch from AWT to Swing classes we will contrast corresponding classes from either package to highlight their differences. That means that this chapter relies on a reasonable understanding of the AWT (chapter 4), especially since the general event handling mechanism has not changed.[1]

[1] Swing uses most events from the `java.awt.event` package but introduces many more events and listeners in the `javax.swing.event` packages. We will introduce new events and listeners when discussing the classes using them. For a general discussion of events and listeners, refer to chapter 4.4.

This chapter is not an exhaustive discussion of Swing components in their full generality. Since Swing components are very flexible it would be beyond the scope of this book to describe everything that can be done using Swing. Instead, we will focus on the most useful Swing components and describe their most likely use so that the reader will get a quick working knowledge of Swing. More details are available using other resources such as the Java API for technical information on the fine points of Swing or books dedicated to Swing.

Quick View

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

6.1. Introduction to Swing
To Swing or not to Swing; Overview of Swing Classes; Converting from AWT to Swing

6.2. Basic Swing Classes
Pluggable Look-and-Feel: Customizing the User Interface; The Essentials: Frames, Applets, Buttons, and Icons; Surroundings: Borders, Panels, and Menus; States: Checkboxes, Radiobuttons, and Drop-down lists

6.3. Advanced Swing Classes
Scrolling, Splitting, and Tabbing; Models and Views: Lists; Swing Text Components; Holding Swinging Dialogs

6.4. Trees, Tables, and Graphics
Trees; Tables; Graphing and Painting in Swing

(*) 6.5. Images and Sounds
Loading and Filtering Images; Animation and Enhanced Image Loading; Loading and Displaying Images in Applets; Off-Screen Drawing and Double-Buffering; Working with Sounds

(**) 6.6. In Full Swing: Two Examples
SharkAttack Game; MandelBrot Set with RubberBand

(*) These sections are optional but recommended
(**) These sections are optional

6.1. Introduction to Swing

Swing, in short, consists of sophisticated and flexible GUI components written entirely in Java. They include everything from buttons and labels to trees, tables, and split panes, and they are meant to replace and enhance their older AWT counterparts. They also provide additional functionality not found previously. All Swing classes are written entirely in Java, which means that they will look and feel the same on all supported platforms. Older AWT components were based on native code written

for a particular operating system, which meant that the "write once, run everywhere" philosophy on which Java was based was not necessarily always true.[2]

Swing is actually part of a larger effort that Sun announced during their "Java One" conference in 1997 named Java Foundation Classes (JFC). After the initial excitement over Java died down somewhat, many programmers and especially software companies began to complain that AWT-based Java programs are not always "industrial-strength". While the general language philosophy for writing programs in Java was widely applauded, features such as enhanced graphics support, drag-and-drop, improved security mechanisms, and enhanced GUI elements were not found in the original releases of Java. Also, Microsoft began to develop their own enhancements of Java[3] that would work only on Windows platforms but would allow developers to create more robust and modern programs on that platform. Sun, with support from other companies such as Netscape, therefore developed the Java Foundation Classes to bring Java up to modern standards of application development and in fact to set new standards, particularly in the area of network programming.

The JFC that were eventually incorporated into the Java 1.2 release consist of several components:

- **Swing**: a wealth of flexible GUI components that could be configured on the fly by the programmer or even the user to have a particular "look and feel"
- **Accessibility**: a unified framework to support assistive support to users with disabilities in using Java programs (to support, for example, screen readers and Braille displays)
- **Java 2D**: enhanced text, graphics, and imaging components for two-dimensional graphics
- **Drag and Drop**: support data exchange between Java programs and in particular between Java and non-Java (native) programs

In this chapter we will discuss many of the new Swing components and give plenty of examples showing how to use them, but we will not cover Java 2D and Drag and Drop support. We will also not mention the Accessibility API for lack of time, but that particular feature is easily integrated into existing Swing-based programs.[4]

To Swing or not to Swing

Before you read through this – lengthy – chapter, a natural question is whether you really need to use Swing components and what the pros and cons are. For one thing, this chapter is not used in any of the subsequent chapters of this text so you do not need to read it before you can continue with another topic. In that sense, this chapter is completely optional.

However, programs created using Swing components look much more professional that AWT based ones and the additional elements that Swing provides give you a lot more design choices and options. It is easy, for example, to use text formatted in HTML in a Java program by using the Swing component `JEditorPane`. Doing something similar with AWT components would be quit time-consuming and difficult.

---

[2] In fact, the "*write once, run everywhere*" concept was occasionally rephrased as "*write once, debug everywhere*". Using Swing components, programs have a much better chance of behaving the same on different platforms.

[3] Microsoft called its improved classes the "Microsoft Foundation Classes" and they are integrated into Microsoft's Java Development program J++.

[4] Virtually all Swing classes implement the `Accessible` interface which requires implementation of a method `public AccessibleContext getAccessibleContext()`.

Swing does not necessarily make your programs easier to create but it generally makes them better looking, easier to run on other platforms, and more complete. Since Swing components are written entirely in Java, they are not restricted to a least common denominator implementation across platforms. Therefore, Swing components can generally be more functional than AWT components. Buttons, for example, do not necessarily have to be square; with Swing you could create round or even star-shaped buttons. Lists, as another example, can easily contain objects other than strings through a flexible View/Model approach. In short, we recommend that you use Swing to create all of your programs unless there is a good reason not to do so. It is straightforward to convert existing AWT programs into Swing programs, and you can find some guidelines below to make the conversion process simple.

There *is* one drawback when using Swing components: Swing has not yet found its way into the JVM provided by Netscape 4 or below, or Internet Explorer 5 or below. That means that applets created using Swing components will not run in these web browsers without help. However, a standard solution for this problem is to install a Java Runtime Plugin package on client computers, which is available for free from `www.javasoft.com`, and then ask that plugin to run Java programs. The plugin supports the latest JVM from Sun and includes Swing classes as well as all other JFC improvements. Sooner or later this disadvantage will disappear altogether, as all JVMs will eventually include Swing and the other JFC classes, so you might as well learn about Swing right now.

There are several other caveats you should know about before starting to program in Swing. The two most important issues are:

- You should not mix Swing and AWT GUI elements in one program. While in principle AWT and Swing components *can* coexist, there are subtle problems with that, and occasionally unexpected behavior will result. To avoid problems, use Swing components exclusively for GUI elements. At the very minimum, the top-level container must always be a Swing component such as a `JFrame` or a `JApplet`. You can, however, mix none-GUI AWT classes with Swing classes without problems
- Swing GUI components are not thread-safe whereas AWT components are. If you create programs where GUI elements are manipulated by threads, you may need to implement your own synchronization mechanisms (see chapter 5.3). Since most programs manipulate GUI elements only from event handlers, this issue does not apply. But if, for example, you use a thread to manipulate the items in a list as well as letting the user do the same, you must implement your own synchronization mechanism to avoid corrupting the integrity of the list.

The following table might help you decide whether to use AWT or Swing-based programming:

Objective	To Swing or Not to Swing
Create only stand-alone programs	use Swing
Create applets only for Intranet	use Swing and Java plugin
Create small applets for Internet	may not want to use Swing
Create large-scale, applet-based programs	use Swing and Java plugin
Create mixed stand-alone programs and applets	use Swing and Java plugin, if necessary
Have large applet collection	convert to Swing one by one, then provide access to both versions
Use commercial tools such as J++ to develop Java programs and applets	can not use Swing unless you upgrade the compiler and JVM when it becomes available

Table 6.1.1: When to use Swing

Overview of Swing Classes

Like everything in Java, the Swing classes are grouped into various packages, prefaced by `javax` instead of the usual `java`. Table 6.1.2 shows an overview of the most important Swing packages.

Package Name	Package Description
`javax.swing`	Collection of pure Java components that work the same on all platforms (if possible)
`javax.swing.border`	Classes and interface for drawing borders around Swing components
`javax.swing.event`	Additional events to support new Swing components
`javax.swing.table`	Classes and interfaces for dealing with `javax.swing.JTable`.
`javax.swing.text`	Provides classes and interfaces that deal with editable and non-editable text components.
`javax.swing.text.html`	Provides the class `HTMLEditorKit` and supporting classes for HTML text editors.
`javax.swing.text.rtf`	Provides a class `RTFEditorKit` for Rich-Text-Format text editors.
`javax.swing.tree`	Provides classes and interfaces for dealing with `javax.swing.JTree`.

Table 6.1.2: Selected Swing packages

Another important part of Swing is the `javax.accessibility` package that provides support to people with visual impairments so that a programmer can include accessibility options to support those users in working with Java programs.

Most applications using Swing will get by with importing classes in `javax.swing` and perhaps `javax.swing.event` or `javax.swing.border`. The other packages are generally used to provide support for more complicated classes from `javax.swing` such as `JTree` and `JTable`.

Since there are so many new features and possibilities that Swing offers we will use the AWT as a foundation to group classes in `javax.swing` into three categories:

Code-Compatible:
Components that have a different class name (usually the name from the AWT prefaced by the letter "`J`") but the same or very similar methods as before. All classes in this category have additional features that are not available in their AWT counterparts. See table 6.1.3 for a list of code-compatible classes.

Enhanced:
Components that emulate AWT classes but work differently to support the various enhancements they offer. See table 6.1.4 for a list of enhanced classes.

New:
Components that do not have direct equivalent classes in the AWT. See tables 6.1.5 and 6.1.6 for a list of new classes.

`JApplet`	`JButton`	`JCheckBox`	`JCheckBoxMenuItem`
`JDialog`	`JFrame`	`JLabel`	`JMenu`
`JMenuBar`	`JMenuItem`	`JPanel`	`JPopupMenu`
`JScrollBar`	`JScrollPane`	`JTextField`	

Table 6.1.3: `javax.swing.*` *classes code-compatible with AWT classes*

`JComboBox`	`JFileChooser`	`JList`	`JTextArea`
`JRadioButton`			

Table 6.1.4: `javax.swing.*` *Classes that are enhanced over comparable AWT classes*

Box	ButtonGroup	ImageIcon	JEditorPane
JInternalFrame	JOptionPane	JPasswordField	JPopupMenu.Separator
JProgressBar	JRadioButtonMenuItem	JSeparator	JSplitPane
JTabbedPane	JTable	JTextPane	JToggleButton
JToolBar	JToolTip	JTree	ToolTipManager
UIManager			

Table 6.1.5: GUI-related `javax.swing.*` *classes without equivalent in AWT*

DebugGraphics	DefaultButtonModel	DefaultCellEditor
DefaultComboBoxModel	DefaultFocusManager	DefaultListCellRenderer
DefaultListModel	DefaultListSelectionModel	DefaultSingleSelectionModel
GrayFilter	Timer	SizeRequirements
SwingConstants	SwingUtilities	

Table 6.1.6: Additional `javax.swing.*` *Classes that are not present in the AWT*

Before explaining how to work with many of these classes, here is a preview of the look and implied functionality of some of the enhanced and new Swing classes.

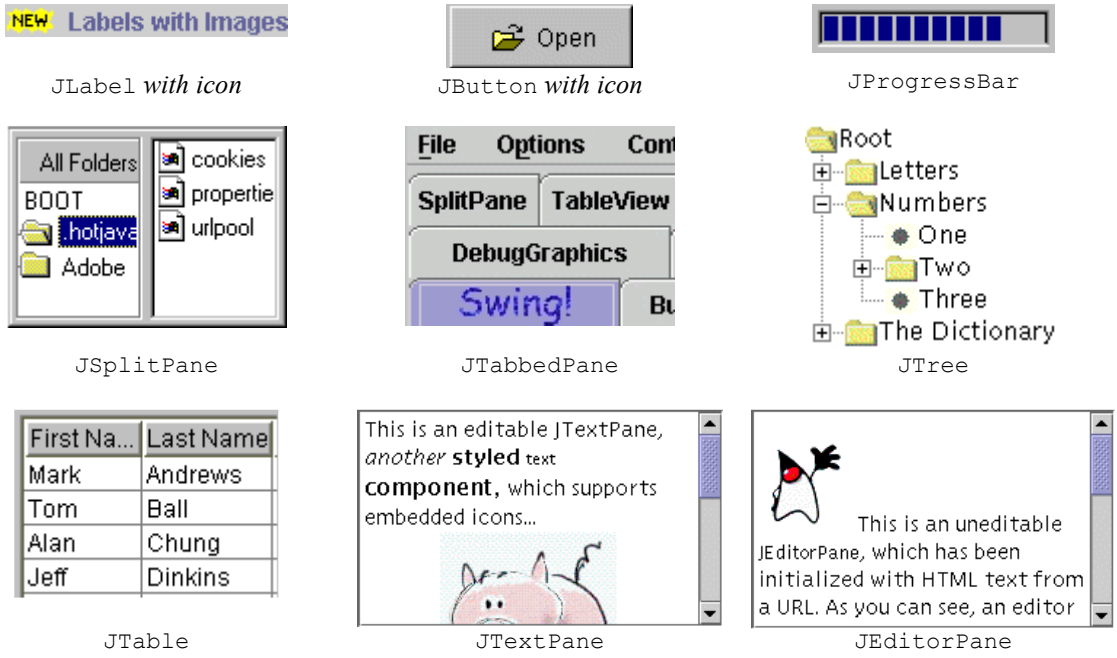


Figure 6.1.7: Look and Feel of some new and improved Swing GUI components

Converting from AWT to Swing

It is fairly easy to convert existing programs based on the AWT to equivalent programs using Swing. Typically, during that conversion process additional features can be added that make an 'old' AWT based program almost immediately into a better looking, easier to use Swing based program. This section will briefly cover the necessary steps to convert older program to Swing.

We will group the AWT classes into two categories. The first consists of classes that can safely be mixed with Swing classes (see table 6.1.8). The second contains those classes that should be replaced by equivalent Swing classes (see table 6.1.9).

Category	Specific Classes
All existing layout managers	`BorderLayout`, `CardLayout`, `FlowLayout`, `GridBagLayout`, and `GridLayout`
All events, listeners, and adapters from the `java.awt.event` package	`ActionEvent`, `ActionListener`, `WindowEvent`, `WindowListener`, `WindowAdapter`, and so forth.
All non-GUI storage containers	`Dimension`, `Insets`, `Point`, `Polygon`, and `Rectangle`
Classes that provide access to system resources	`Color`, `Cursor`, `Font`, `FontMetrics`, `SystemColor`, and `Toolkit`
`Graphics` and `Image` related classes	`Graphics`, `Graphics2D`, `Image`, `MediaTracker`

Table 6.1.8: AWT Components that are safe with Swing

Note that Swing provides a `JSplitPane` (definition 6.3.5) and a `JTabbedPane` (definition 6.3.3) that can easily be used to replace some layouts and add extra functionality to your program.

AWT Component	Swing Component	Notes
`Applet`	`JApplet`	use `getContentPane().add` instead of `add` method; `JApplet` is part of `javax.swing` package
`Button`	`JButton`	Code compatible
`Canvas`	`JPanel` or `JLabel`	replace `paint` by `paintComponent`, `JPanel` and `JLabel` already have double buffering
`Checkbox`	`JCheckBox` or `JRadioButton`	code compatible (note the spelling difference)
`CheckboxGroup`	use `ButtonGroup` instead	`ButtonGroup` can group check boxes, radio buttons, and buttons
`CheckboxMenuItem`	`JCheckboxMenuItem` or `JRadioButtonMenuItem`	code compatible (note the spelling difference)
`Choice`	`JComboBox`	adding items is different
`Component`	`JComponent`	usually not used directly
`Dialog`	`JDialog` or `JOptionPane`	use `getContentPane().add` instead of add method
`FileDialog`	`JFileChooser`	differences
`Frame`	`JFrame`	use `getContentPane().add` instead of `add` method
`Label`	`JLabel`	code compatible
`List`	`JList`	lists need separate scroll pane and data model – different from AWT
`Menu`, `MenuBar`, `MenuItem`, `PopupMenu`	`JMenu`, `JMenuBar`, `JMenuItem`, `JPopupMenu`, also `JSeparator` and `JPopupMenu.Separator`	code-compatible, but separators are separate classes, not fields
`Panel`	`JPanel`	code compatible
`Scrollbar`	`JScrollBar` or `JSlider` or `JProgressBar`	depends on the class you are using
`ScrollPane`	`JScrollPane`	code compatible

TextArea	JTextArea	must add scrollbars manually, event listener is different
TextComponent	JTextComponent	not used directly
TextField	JTextField	code compatible
Window	JWindow	code compatible, not often used

Table 6.1.9: AWT classes and their Swing replacements

After replacing the AWT components by their corresponding Swing components, some immediate enhancements to your new program are possible, and you should take advantage of them:

- add borders and titled borders to some Swing components
- add tooltips to some swing components
- add images to buttons and labels
- easily check for right mouse click and double-click for use in popup menus
- add information and warning dialog boxes

While conversion to Swing is not automatic, it is not hard to do. Here is an outline of the steps necessary:

Definition 6.1.1: Rule of Thumb to convert AWT to Swing

To convert a program based on AWT components to Swing, follow these steps:

- *Make a backup copy of your source code and remove all class files*
- *Remove* `java.awt.*`, `java.applet.*` *or* `java.applet.Applet`, *leave* `import java.awt.event.*` *and add* `import javax.swing.*`
- *Change all AWT GUI components to their Swing counterparts as described in table 6.1.9. Generally that involves adding a "*`J`*" to the class name*
- *A* `List` *needs to be replaced by* `JList` *and associated with a model and a scroll pane*
- *A* `TextArea` *needs to be replaced with* `JTextArea` *and associated with a scroll pane; Any* `TextListener` *need to be replaced with* `DocumentListener`
- *Classes that do custom drawing and/or extend* `Canvas` *must be converted to* `JPanel`
- *For* `JFrame`, `JDialog`, *and* `JApplet` *replace code such as:*
  `setLayout(manager); add(component)`
  with
  `getContentPane().setLayout(manager); getContentPane().add(component)`
- *Import specific AWT classes that are safe for use with Swing one by one as described in table 6.1.8*

Then compile the new class using `java -deprication Source.java` *and use the Java API to resolve name differences and deprecated methods. Add simple improvements such as button images and tooltips, clean up your code, investigate using additional Swing components for added or improved functionality and test the new program.*

While we will introduce the essential Swing components in detail later, here is a brief example of how to convert a complete (but simple) program into an equivalent and slightly enhanced Swing-based program.

Example 6.1.2:

Convert the applet listed below into an equivalent one using Swing components. The applet is similar to that in example 4.33, with some simplifications to keep the code short. Add improvements if possible and easy.

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

public class MoveBoxInnerCanvas extends Applet implements ActionListener
{  private final int BOX_WIDTH = 30;
   private final int BOX_HEIGHT = 20;
   private final int INC = 4;
   private final Color COLOR = Color.red;
   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;
   private class MoveBoxCanvas extends Canvas
   {   public void paint(Graphics g)
      {  g.setColor(COLOR);
         g.fillRect(x, y, BOX_WIDTH, BOX_HEIGHT);
         g.setColor(Color.black);
         g.drawRect(0,0,getSize().width-1, getSize().height-1);
      }
   }
   private MoveBoxCanvas drawing = new MoveBoxCanvas();
   public void init()
   {  Panel buttons = new Panel();
      buttons.setLayout(new FlowLayout());
      buttons.add(up);    up.addActionListener(this);
      buttons.add(down);  down.addActionListener(this);
      buttons.add(left);  left.addActionListener(this);
      buttons.add(right); right.addActionListener(this);
      setLayout(new BorderLayout());
      add("South", buttons);
      add("Center", drawing);
   }
   public void actionPerformed(ActionEvent e)
   {  if (e.getSource() == up)
         y -= INC;
      else if (e.getSource() == down)
         y += INC;
      else if (e.getSource() == left)
         x -= INC;
      else if (e.getSource() == right)
         x += INC;
      drawing.repaint();
   }
}
```

The program, taken from example 4.33, lets the user move a box around the screen – not very exciting, but it will serve to illustrate our conversion process. Assuming that we made a backup copy of the program and removed all associated class files, we need to first modify the `import` statements. Following definition 6.1.1, we replace:

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

with

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

Next we change the class so that it extends `JApplet` instead of `Applet` and the inner class so that it extends `JPanel` instead of `Canvas`. We also change the `paint` method of the inner class to `paintComponent` and add a call to `super.paintComponent` as the first line (for more details on converting drawing code, see definition 6.4.13). We also change all `Button` classes to `JButton`, and `Panel` to `JPanel`. Finally, we replace the calls to the applet's `add` and `setLayout` methods by prefacing them with `getContentPane()`. Here is the new code:

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

public class MoveBoxInnerCanvas extends JApplet implements ActionListener
{  private final int BOX_WIDTH = 30;
   private final int BOX_HEIGHT = 20;
   private final int INC = 4;
   private final Color COLOR = Color.red;
   private JButton left = new JButton("Left");
   private JButton right = new JButton("Right");
   private JButton up = new JButton("Up");
   private JButton down = new JButton("Down");
   private int x = 50, y = 50;
   private class MoveBoxCanvas extends JPanel
   {   public void paintComponent(Graphics g)
      {  super.paintComponent(g);
         g.setColor(COLOR);
         g.fillRect(x, y, BOX_WIDTH, BOX_HEIGHT);
         g.setColor(Color.black);
         g.drawRect(0,0,getSize().width-1, getSize().height-1);
      }
   }
   private MoveBoxCanvas drawing = new MoveBoxCanvas();
   public void init()
   {  JPanel buttons = new JPanel(new FlowLayout());
      buttons.add(up);    up.addActionListener(this);
      buttons.add(down);  down.addActionListener(this);
      buttons.add(left);  left.addActionListener(this);
      buttons.add(right); right.addActionListener(this);
      getContentPane().setLayout(new BorderLayout());
      getContentPane().add("South", buttons);
      getContentPane().add("Center", drawing);
   }
   public void actionPerformed(ActionEvent ae)
   {  /* no changes */ }
}
```

Now we try to compile our new class (note that we instantiated the `JPanel` with the proper layout as input parameter). We will get a few error messages:

```
C:\temp>javac MoveBoxInnerCanvas.java
MoveBoxInnerCanvas.java:8: Class Color not found.
   private final Color COLOR = Color.red;
[... additional error messages referring to Color class not found ...]
MoveBoxInnerCanvas.java:15: Class Graphics not found.
   {   public void paintComponent(Graphics g)
[... additional error messages referring to Graphics class not found ...]
MoveBoxInnerCanvas.java:24: Class FlowLayout not found.
   {  JPanel buttons = new JPanel(new FlowLayout());
```

```
MoveBoxInnerCanvas.java:29: Class BorderLayout not found.
      getContentPane().setLayout(new BorderLayout());
```

Based on these error messages we add the AWT classes that are safe to use with Swing one by one to the import statement and recompile:

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.event.*;
import javax.swing.*;

public class MoveBoxInnerCanvas extends JApplet implements ActionListener
{  /* as above, no additional changes */ }
```

The class will compile perfectly and we have completed the conversion to Swing. But we can easily add a few improvements to our class. We add the following code to the `init` method to add borders, tooltips, and button icons:

```
public void init()
{  JPanel buttons = new JPanel(new FlowLayout());
   buttons.add(up);    up.addActionListener(this);
   buttons.add(down);  down.addActionListener(this);
   buttons.add(left);  left.addActionListener(this);
   buttons.add(right); right.addActionListener(this);
   drawing.setBorder(new TitledBorder("The Box Playground"));
   buttons.setBorder(new TitledBorder("Move the Box"));
   try
   {   URL base = getCodeBase();
       up.setIcon(new ImageIcon(new URL(base, "up.gif")));
       down.setIcon(new ImageIcon(new URL(base, "down.gif")));
       left.setIcon(new ImageIcon(new URL(base, "left.gif")));
       right.setIcon(new ImageIcon(new URL(base, "right.gif")));
   }
   catch(MalformedURLException murle)
   {  System.err.println("Error"); }
   up.setToolTipText("Box up");       down.setToolTipText("Box down");
   right.setToolTipText("Box right"); left.setToolTipText("Box left");
   getContentPane().setLayout(new BorderLayout());
   getContentPane().add("South", buttons);
   getContentPane().add("Center", drawing);
}
```

In order for the new code to compile we need to import `java.net.*` (for the `URL` classes) and `javax.swing.border.*` (for the titled borders). Of course we also need four image files in the same location as the class file. Comparing the new versus the old program will then look as follows:

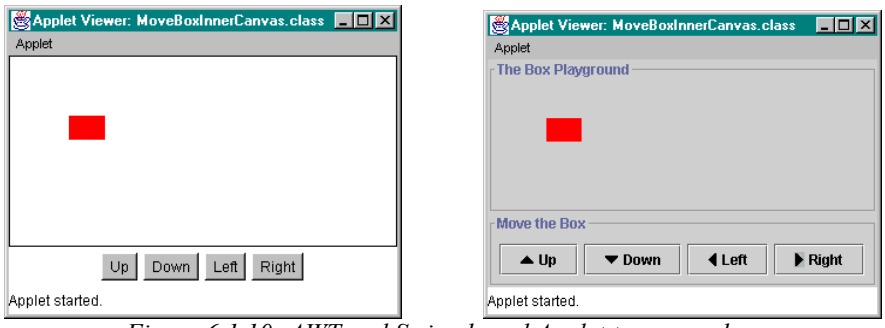


Figure 6.1.10: AWT and Swing based Applet to move a box

The new applet works exactly as before – which was not too exiting to begin with – but it uses borders and buttons with icons to improve its look. Each button will also show a tool tip when the mouse hovers over it for a brief period.

■

This example shows how to convert an easy program to Swing. More complicated programs may run into additional problems, but it should not be hard to overcome them. In the next few sections we will provide details about the new Swing components that should make you as familiar with Swing as chapter 4 made you with the AWT.

6.2. Basic Swing Classes

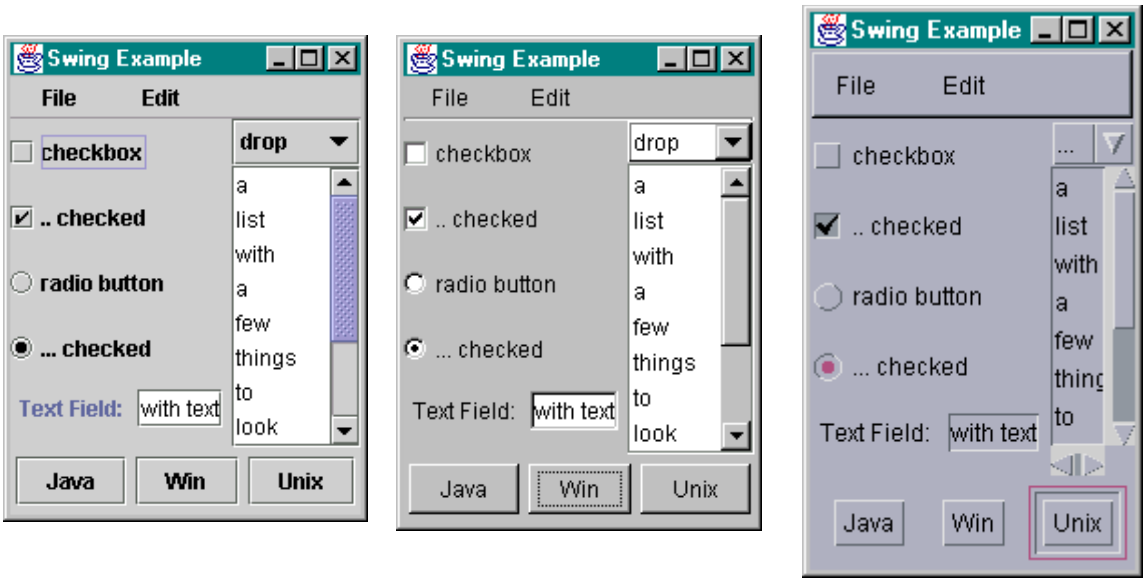
In this section we will introduce the more commonly used Swing classes and illustrate their usage. Most examples in this section will not necessarily be useful programs; they are intended to illustrate how to use the new and enhanced classes in principle. Once you completed this chapter you should return to the classes from chapter 4 and try to convert them to Swing components. We will start our discussion with the plugable look and feel that Swing provides.

Pluggable Look-and-Feel: Customizing the User Interface

Java allows you to choose a "look and feel" for your application or applet that determines how the various GUI elements will be rendered and how they behave when selected. You can select a "Windows", "Unix", "Java", or "Macintosh"[5] look for your program and you can even let the user select a different look while the program is running. In addition, programs can be created that will inherit the look of the underlying operating system. The default is set to a "Java" look but can be changed at any time.

This allows flexibility when deciding how your program should look overall yet lets you choose a look that is in accordance with the operating system you prefer.

[5] There are copyright problems associated with choosing different look and feels. In particular, Apple has not granted Sun the right to distribute a "Macintosh" look to be used on platforms other than Macintosh computers. Therefore, the Macintosh look and feel may not be available on non-Mac platforms.



Java Look-and-Feel *Windows Look-and-Feel* *Unix Look and Feel*

Figure 6.2.1: Common GUI Element in Java, Windows, and Unix Look-and-Feel

Two classes are responsible for defining the look-and-feel: `UIManager` and `SwingUtilities`:

Definition 6.2.1: The `UIManager` and `SwingUtilities` Classes

The `UIManager` *class keeps track of the current look and feel, while* `SwingUtilities` *provides several utility methods. In particular, the static method* `setLookAndFeel` *of the* `UIManager` *defines which "look-and-feel" the class uses, while the static method* `updateComponentTreeUI` *of* `SwingUtilities` *ensures that a currently active class updates its look-and-feel. The Java API defines theses classes as follows (only a few methods are listed here):*

```
public class UIManager extends Object implements Serializable
{  public static void setLookAndFeel(String className)
       throws ClassNotFoundException, InstantiationException,
              IllegalAccessException, UnsupportedLookAndFeelException
   public static String getSystemLookAndFeelClassName()
   public static String getCrossPlatformLookAndFeelClassName()
}
public class SwingUtilities extends Object implements SwingConstants
{  public static boolean isLeftMouseButton(MouseEvent anEvent)
   public static boolean isMiddleMouseButton(MouseEvent anEvent)
   public static boolean isRightMouseButton(MouseEvent anEvent)
   public static void updateComponentTreeUI(Component c)
}
```

The commonly used "look-and-feel" classes are represented by:

```
javax.swing.plaf.metal.MetalLookAndFeel
com.sun.java.swing.plaf.windows.WindowsLookAndFeel
com.sun.java.swing.plaf.motif.MotifLookAndFeel
```

Example 6.2.2:

Create some code segments that define various user interface looks for Swing classes:

- a "Java" look
- a "Unix" look (the standard X-Windows look on Unix is called "Motif")
- a look that is consistent with the operating system running the class
- a look that is similar across platforms

We can only show a code segment since we have not yet defined any actual Swing GUI components. However, from the above definition we can see that we need to use the method:

```
UIManager.setLookAndFeel(LookandFeelName)
```

to define a particular look, and

```
SwingUtilities.updateComponentTreeUI(ComponentToUpdate)
```

to update components in case they are already instantiated. Therefore, our code segments might be as follows (taking care of the various exceptions in one `catch` clause):

To define a "Java" look:

```
try
{  String theLook = "javax.swing.plaf.metal.MetalLookAndFeel";
   UIManager.setLookAndFeel(theLook);
}
catch (Exception e)
{  System.err.println("Exception: " + e); }
```

To define a "Unix" look:

```
try
{  String theLook = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
   UIManager.setLookAndFeel(theLook);
}
catch (Exception e)
{  System.err.println("Exception: " + e); }
```

To define a look that is consistant with the operating system running the class:

```
try
{  String theLook = UIManager.getSystemLookAndFeelClassName();
   UIManager.setLookAndFeel(theLook);
}
catch (Exception e)
{  System.err.println("Exception: " + e); }
```

To define a look that is similar across platforms:

```
try
{  String theLook = UIManager.getCrossPlatformLookAndFeelClassName();
   UIManager.setLookAndFeel(theLook);
}
catch (Exception e)
{  System.err.println("Exception: " + e); }
```

In addition, we also may need to force any components already instantiated to update themselves to the new look by calling:

```
SwingUtilities.updateComponentTreeUI(ComponentToUpdate);
```

where `ComponentToUpdate` is the top-level component that contains all Swing components that should receive the new look-and-feel. Since that method does not throw any exceptions, it can be placed anywhere after issuing a call to `setLookAndFeel`. In most cases components will also have slightly different size requirements after defining a new look so that the involved classes should be allowed to resize themselves. If the top-level component is a `JFrame` (similar to a `Frame`), you can achieve that by calling

```
pack();
```

■

In the next example we will see a complete program to define and update the look-and-feel of a Java application.

Note that `SwingUtilities` implements `SwingConstants` which contains several constants used by a variety of Swing components. That class is defined as follows:

Definition 6.2.3: The `SwingConstants` Class

A collection of constants used by several Swing components for positioning and orientation. The constants available from this class are:

```
CENTER, LEFT, RIGHT, TOP, BOTTOM
HORIZONTAL, VERTICAL
NORTH, EAST, SOUTH, WEST
NORTH_EAST, SOUTH_EAST, SOUTH_WEST, NORTH_WEST
```

All constants are `static` *and* `public`.

The Essentials: Frames, Applets, Buttons, and Icons

Now we are ready to introduce some of the basic Swing components so that we can create "real" examples using Swing instead of the AWT. We will start with the `JFrame` class. Just as the `Frame` class is the basis for all AWT-based standalone programs, the `JFrame` class is the basic component for standalong Swing programs. A `JFrame` can contain Swing elements such as buttons, labels, text fields and text areas. In fact, `JFrame` extends `Frame` so that all methods known to our familiar `Frame` class also apply to a `JFrame`.

Definition 6.2.4: The `JFrame` Class

The `JFrame` *class provides a "standard window" with close and resize boxes. It usually contains other Swing components such as buttons, labels, menus, etc. The class extends* `java.awt.Frame` *and the only incompatibility to a* `Frame` *is that components are not directly added via the* `add` *method but instead to an instance of the* `contentPane`. *A reference to that is obtained via*

`getContentPane` *which returns a* `Container`*. Therefore, to add a component to a specific layout of a* `JFrame` *you use*[6]*:*

```
getContentPane.setLayout(newLayout);
getContentPane.add(newComponent);
```

Swing menus are defined using the `void setJMenuBar(JMenuBar menubar)`*. You can also define how a* `JFrame` *behaves when its standard close box is clicked*[7] *using:*

```
public void setDefaultCloseOperation(int operation)
```

where operation could be one of:[8]

```
WindowConstants.DO_NOTHING_ON_CLOSE
WindowConstants.HIDE_ON_CLOSE
WindowConstants.DISPOSE_ON_CLOSE
```

Example 6.2.5:

Create a "standard framework" for a standalone application based on `JFrame` and make sure that the program has a "Java" look-and-feel. The program should close appropriately when its standard close box is clicked.

Since `JFrame` extends `Frame`, we create a standard application similar those created in chapter 4:

- the class extends `JFrame` (not `Frame`)
- the class implements `ActionListener` to intercept action events
- we provide a constructor with the standard `validate`, `pack`, and `setVisible` methods and an appropriate call to `setDefaultCloseOperation`
- we add the `actionPerformed` method
- we instantiate a new instance of our class in the standard `main` method

Before instantiating our class we define the "Java look" as defined previously. Here is the complete code:

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

public class BasicJFrame extends JFrame implements ActionListener
{  // any fields go here
   public BasicJFrame()
   {  super("Basic JFrame Program");
      setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
      // additional methods go here
      validate(); pack(); setVisible(true);
   }
   public void actionPerformed(ActionEvent ae)
   {  /* event-handling code goes here */ }
   public static void main(String args[])
```

[6] Instead of issuing multiple calls to `getContentPane` you may want to define a reference to it and then manipulate the `contentPane` using its reference variable.

[7] A `JFrame`, by default, has a "Java look", hides on close and has a border layout. You can of course add an appropriate `WindowListener` and set the standard closing behavior to `DO_NOTHING_ON_CLOSE`.

[8] Since JDK 1.3 another constant `JFrame.EXIT_ON_CLOSE` is available to exit the program immediately.

```
      { try
         {  String theLook = "javax.swing.plaf.metal.MetalLookAndFeel";
            UIManager.setLookAndFeel(theLook);
         }
         catch (Exception e)
         {  System.err.println("Exception: " + e); }
         BasicJFrame bjf = new BasicJFrame();
      }
   }
```

We do not need to call `SwingUtilities.updateComponentTreeUI` because we chose a look-and-feel *before* instantiating the class. The program should compile and execute fine, displaying a tiny window.

Figure 6.2.2: The `BasicJFrame` *program*

The window can be resized and iconized and will disappear when its standard close box is clicked. However, while the frame will disappear the program will not actually close. The behavior we defined was `DISPOSE_ON_CLOSE` but what we really need is to call `System.exit`. That can not be accomplished using the `setDefaultCloseOperation` method so we need to create an inner class extending `WindowAdapter` and attach it as a `WindowListener` to our `JFrame`. Since that procedure is exactly as it was for a `Frame`, we will leave it as an exercise (compare example 4.15).

■

Before we can create more involved examples illustrating the various classes available we need to have some Swing components to begin with. Perhaps the easiest ones are the code-compatible classes `JButton` and `JLabel`:

Definition 6.2.6: The `JButton` Class

A `JButton` *is an implementation of a standard "push" button. It behaves similar to its AWT counterpart*[9] *but can in addition to text accommodate an image or text and an image. The Java API defines it as follows:*

```
public class JButton extends AbstractButton implements Accessible
{  // constructors
   public JButton()
   public JButton(Icon icon)
   public JButton(String text)
   public JButton(String text, Icon icon)
   // selected methods (inherited from AbstractButton)
   public String getText()
   public void setText(String text)
   public void setIcon(Icon icon)
   public void setToolTipText(String text)
}
```

[9] The AWT `Button` class uses the methods `getLabel` and `setLabel` to manipulate the button's text. Those methods continue to work for a `JButton` (they are deprecated) but the more consistent methods `setText` and `getText` are better choices.

Note that a `JButton` has methods `setText` and `getText`. The `Button` class from the AWT contains methods `setLabel` and `getLabel` instead. Therefore, when converting a `Button` to a `JButton` you also need to change `set/getLabel` methods to `set/getText`.

Definition 6.2.7: The `JLabel` Class

A `JLabel` *is a display area for non-editable information. It behaves similar to its AWT counterpart*[10] *but can in addition to text accommodate an image or text and an image. The Java API defines it as follows*

```
public class JLabel extends JComponent
       implements SwingConstants, Accessible
{  // constructors
   public JLabel()
   public JLabel(String text)
   public JLabel(Icon image)
   public JLabel(String text, int horizontalAlignment)
   public JLabel(Icon image, int horizontalAlignment)
   public JLabel(String text, Icon icon, int horizontalAlignment)
   // selected methods
   public String getText()
   public void setText(String text)
   public void setIcon(Icon icon)
   public void setToolTipText(String text)
}
```

The alignment constants for the constructors come from `SwingConstants`.

Example 6.2.8

Create a program containing one centered label and three buttons. When you click on a button, the look-and-feel of your program should change to the one indicated on the button. Also, the program should exit when clicking on the standard close box.

We can use the basic framework of example 6.2.5, adding three buttons and one label as fields. Since no layout is specified, we will use a `FlowLayout` to arrange everything in one row. To ensure that the program will quit when necessary we define a named inner class and use it to handle a `windowClosing` event. Here is the code:

```
import java.awt.FlowLayout;
import java.awt.Container;
import java.awt.event.*;
import javax.swing.*;

public class JFrameWithButtons extends JFrame implements ActionListener
{  private JButton windLook = new JButton("Windows");
   private JButton unixLook = new JButton("Unix");
   private JButton javaLook = new JButton("Java");
   private JLabel label = new JLabel("Welcome to Swing",
                                     SwingConstants.CENTER);
   private class WindowCloser extends WindowAdapter
   {  public void windowClosing(WindowEvent we)
```

[10] A `JLabel` uses constants from `SwingConstants` to indicating the alignment or text and image, while `Label` uses constants from the `Label` class. When converting a `Label` to a `JLabel`, the class name indicating the alignment constant must therefore change from `Label` to `SwingConstants`.

```
      {  System.exit(0); }
   }
   public JFrameWithButtons()
   {  super("JFrame with Buttons");
      Container content = getContentPane();
      content.setLayout(new FlowLayout());
      content.add(label);
      content.add(windLook);  windLook.addActionListener(this);
      content.add(unixLook); unixLook.addActionListener(this);
      content.add(javaLook); javaLook.addActionListener(this);
      addWindowListener(new WindowCloser());
      validate(); pack(); setVisible(true);
   }
   public void actionPerformed(ActionEvent ae)
   {  String look = "javax.swing.plaf.metal.MetalLookAndFeel";
      if (ae.getSource() == javaLook)
         look = "javax.swing.plaf.metal.MetalLookAndFeel";
      else if (ae.getSource() == windLook)
         look = "com.sun.java.swing.plaf.windows.WindowsLookAndFeel";
      else if (ae.getSource() == unixLook)
         look = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
      try
      {  UIManager.setLookAndFeel(look);
         SwingUtilities.updateComponentTreeUI(this);
         pack();
      }
      catch (Exception e)
      {  System.err.println("Exception: " + e); }
   }
   public static void main(String args[])
   {  JFrameWithButtons jfb = new JFrameWithButtons(); }
}
```

When the program executes, the user can click the buttons to change the look-and-feel at any time. Figure 6.2.3 shows the different looks of our program. Note, in particular, that the three looks have slightly different size requirements, which are automatically adjusted by calling `pack`.

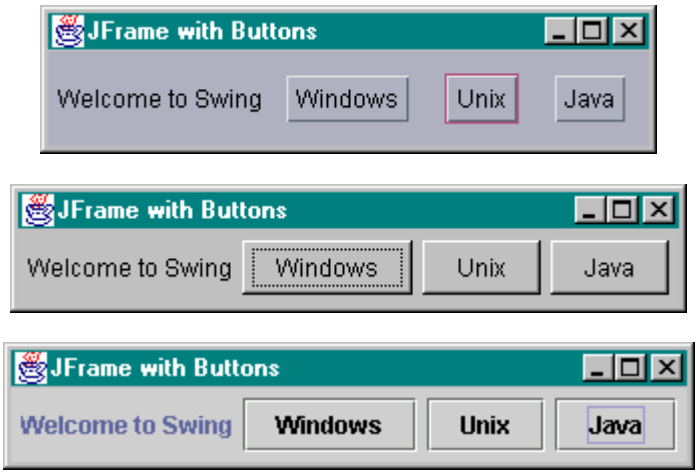


Figure 6.2.3: Three looks of `JFrameWithButtons` *(top: Unix, middle: Windows, bottom: Java)*

■

The functionality is exactly the same as for the corresponding AWT classes, but Swing allows you to easily add an icon to a button or label, or to exclusively use an icon instead of text. Icons are small images and are represented by the `ImageIcon` class.

Definition 6.2.9: The `ImageIcon` Class

The `ImageIcon` *class can load and represents an image that can be used as an icon in a* `JButton`*,* `JLabel`*, and other Swing components. The underlying image file must be either in* `GIF` *or* `JPEG` *format. An image is loaded using its file name or the URL*[11] *of its location. If an image is located in another directory, the Internet-standard forward slash ("/") must be used as separator. The Java API defines this class as follows:*

```
public class ImageIcon extends Object implements Icon, Serializable
{  // selected constructors
   public ImageIcon(String filename)
   public ImageIcon(URL location)
   // selected methods
   public int getIconWidth()
   public int getIconHeight()
}
```

We will later discuss URL's in more details (see definition 6.5.8) as well as explain why it is a useful feature to load image icon files from a URL instead of using a simple file name. For now, using file names will serve us just fine.

Example 6.2.10:

Redo example 6.2.8, but add an icon to each button as well as to the label. Also use the `setToolTipText` method to define popup tips that appear when the mouse hovers over one of the three buttons.

Of course we first need some images that can serve as icons. There are a wide variety of icons and other images available for free download from the Internet, and there are several programs that can capture parts of a screen into an image file that can serve as an icon[12]. Let's assume we have, somehow, obtained the following four image files:

`new.gif`		`windows.gif`	
`xwin.gif`		`java.gif`	

Suppose further that these image files are stored in a folder called `Icons` which is located in the same directory as the Java class file `JFrameWithButtons` below. Then we modify the code of the original class as follows (with the modified pieces in bold and italics):

```
import java.awt.FlowLayout;
import java.awt.Container;
import java.awt.event.*;
import javax.swing.*;

public class JFrameWithButtons extends JFrame implements ActionListener
{  JButton windLook = new JButton("Windows",
                                  new ImageIcon("Icons/windows.gif"));
   JButton unixLook = new JButton("Unix",
                                  new ImageIcon("Icons/xwin.gif"));
   JButton javaLook = new JButton("Java",
                                  new ImageIcon("Icons/java.gif"));
   JLabel label = new JLabel("Welcome to Swing",
```

[11] See definition 6.5.8 for a brief discussion of a Uniform Resource Locator (URL) and the corresponding `URL` class.
[12] Of course you have to be careful not to violate copyright laws. There are free icon collections available on the Internet that can be used without problems.

```
                                    new ImageIcon("Icons/new.gif"),
                                    SwingConstants.CENTER);
    private class WindowCloser extends WindowAdapter
    {   public void windowClosing(WindowEvent we)
        {  System.exit(0); }
    }
    public JFrameWithButtons()
    { // everything as before, as well as the lines:
       windLook.setToolTipText("Windows Look-and-Feel");
       unixLook.setToolTipText("Motif Look-and-Feel");
       javaLook.setToolTipText("Java Look-and-Feel");
    }
    public void actionPerformed(ActionEvent ae)
    { /* no change */ }
    public static void main(String args[])
    { /* no change */ }
}
```

That's all that is needed to get a much better look for our simple program as illustrated in the figure below:

Figure 6.2.4: `JButton` *and* `JLabel` *with attached icons and pop-up tool tip*

■

Of course Swing also allows you to create applets just as easily as standalone programs. It provides the `JApplet` class for that purpose, an enhanced version of `java.applet.Applet`.

Definition 6.2.11: The `JApplet` Class

The `JApplet` *class provides support for a "standard applet", i.e. a program that runs inside a web browser. It usually contains other Swing components such as buttons, labels, menus, etc.. The class is part of* `javax.swing` *and extends* `java.applet.Applet`*. The only incompatibility to* `Applet` *is that components are not directly added to* `JApplet` *via the* `add` *method but instead to an instance of the* `contentPane`*. A reference to that is obtained via* `getContentPane` *which returns a* `Container`*. Therefore, to add a component to a specific layout of a* `JApplet` *you use*[13]*:*

```
getContentPane.setLayout(newLayout)
getContentPane.add(newComponent)
```

In contrast to the `Applet` *class,* `JApplet` *also directly supports menus that are set via:*

```
public void setJMenuBar(JMenuBar menuBar)
```

Swing applets can also bring up non-modal dialogs[14] *because there is a* `JDialog` *constructor that does not require any input.*

[13] Instead of issuing multiple calls to `getContentPane` you may want to a reference to it and then manipulate the `contentPane` using its reference variable.
[14] See definition 6.3.26 and 6.3.29 for an introduction to Swing dialogs.

Example 6.2.12:

Convert the class `JFrameWithButtons` from example 6.2.8 into an applet (a `JApplet`, to be specific). Make sure to use the original version of the class *not* including icons.

If the class had been a `Frame` and we wanted to convert it into an `Applet` we would change the constructor to `public void init()`, remove the reference(s) to `pack` and `setVisible`, and make sure not to call `System.exit`. We could also remove the standard `main` method, which is not automatically called for applets, but that is not necessary. It is no different converting a `JFrame` to `JApplet`, so here is the code:

```
import java.awt.FlowLayout;
import java.awt.Container;
import java.awt.event.*;
import javax.swing.*;

public class JAppletWithButtons extends JApplet implements ActionListener
{  JButton windLook = new JButton("Windows");
   JButton unixLook = new JButton("Unix");
   JButton javaLook = new JButton("Java");
   JLabel label = new JLabel("Welcome to Swing", SwingConstants.CENTER);
   public void init()
   {  Container content = getContentPane();
      content.setLayout(new FlowLayout());
      content.add(label);
      content.add(windLook);  windLook.addActionListener(this);
      content.add(unixLook);  unixLook.addActionListener(this);
      content.add(javaLook);  javaLook.addActionListener(this);
   }
   public void actionPerformed(ActionEvent ae)
   {  String look = "javax.swing.plaf.metal.MetalLookAndFeel";
      if (ae.getSource() == javaLook)
         look = "javax.swing.plaf.metal.MetalLookAndFeel";
      else if (ae.getSource() == windLook)
         look = "com.sun.java.swing.plaf.windows.WindowsLookAndFeel";
      else if (ae.getSource() == unixLook)
         look = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
      try
      {  UIManager.setLookAndFeel(look); }
      catch (Exception e)
      {  System.err.println("Exception: " + e); }
      SwingUtilities.updateComponentTreeUI(this);
   }
}
```

We could add tool tips just as we did previously but loading icon images will *not* work. Note, in particular that we did not need to import any class from the "old" `java.applet` package.

■

To run the applet, make sure to create an appropriate HTML document, such as:

```
<HTML>
<APPLET CODE="JAppletWithButtons.class"
        WIDTH="400"
        HEIGHT="80">
</APPLET>
</HTML>
```

and test it using the `appletviewer` program provided with the JDK, as usual.

While dynamically changing the look-and-feel of the applet works perfectly fine, different looks have different size constraints. Since an applet's size is fixed via the HEIGHT and WIDTH parameters that could create a problem. A simple solution is to use a JApplet class to pop up a JFrame, and work with a JFrame as usual (avoiding of course calling System.exit)

Note: One difficulty distributing applets using Swing is that older versions of web browsers do not support Swing and are therefore unable to run *anything* that contains one or more Swing components. In particular, the above example will run perfectly fine using the appletviewer program but will not run at all with Netscape versions 4 or below or Internet Explorer versions 5 or below (try it and watch the Java Console for error messages). While this is indeed a serious downside of Swing, a solution is to not embed a JApplet in a web page via the standard <APPLET> … </APPLET> tags but using the <EMBED> … </EMBED> tags for Netscape or <OBJECT> … </OBJECT> for Internet Explorer. Using these latter tags a web browser will execute an applet using the Java Runtime environment (JRE) that can be downloaded for free from www.javasoft.com.

The good news is that the JRE module from SUN provides a truly standard implementation of the JVM, including full standardized support for Swing. Thus you can be sure that your applets will execute *exactly* as you intended. The bad news is that the JRE is *required* to run and users need to download it before being able to execute any Swing-based applets. In some sense this defeats the entire idea of Java applets, i.e. components that can execute in a web browser *without* downloading any additional software. In time this problem will likely disappear, as more web browsers will fully support Swing. At this time the <EMBED> … </EMBED> or <OJBECT> … </OBJECT> tags provide the only solution to use Swing with virtually any web browser version, but it requires the user to install the JRE first. Moreover, Netscape only understands the EMBED tag, while Internet Explorer only knows about OBJECT. With a little bit of programming effort using JavaScript, however, it is possible to check a client's web browser version and then use either the APPLET, EMBED, or OBJECT tag, depending on which one is supported by the browser. In the rule of thumb below we show how to use the right combination of EMBED and OBJECT tags to ensure that an applet will work using the JRE from either Netscape or Internet Explorer. It is rather complicated, but if it is done once most of the information can be "cut-and-pasted" to other HTML documents with little adjustment.

Definition 6.2.13: Rule of Thumb for Embedding Swing Applets in a Web Browser

Applets with Swing components can be embedded into web pages using the <EMBED> … </EMBED> *or* <OBJECT> … </OBJECT> *tags and a suitable Java Runtime Environment (JRE) such as the one provided by Sun [15]. This requires that a JRE be installed on the computer that is loading the web page. To embed an applet into a web page that is viewed either with Netscape or with Internet Explorer, use the following HTML code:*

```
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
   width ="###" height ="###"
   [codebase="full_url_to_download_the_JRE"]>
   <PARAM NAME="CODE" VALUE="AppletCode.class">
   [<PARAM NAME="CODEBASE" VALUE="url">]
   [<PARAM NAME="ARCHIVE" VALUE ="File.jar">]
   [<PARAM NAME="more_params" VALUE="more_values">]
  <COMMENT>
    <EMBED type="application/x-java-applet;version=#.#"
      width = "###" height="###"
      java_CODE = "AppletCode.class"
```

[15] Sun offers an "Applet tag conversion" program that will convert an <APPLET> tag into the appriate <EMBED> and/or <OBJECT> tags. More information about the Java plugin including download information for the plugin and applet converter is available at http://www.javasoft.com/products/plugin

```
        [java_CODEBASE = "url"] [java_ARCHIVE = "File.jar"]
        [more_params = "more_values"]
        [pluginspage="full_url_to_download_the_JRE"]>
      </EMBED>
    </COMMENT>
  </OBJECT>
```

If the JRE is present and installed properly on the client computer, it will load and execute the applet. Otherwise a suitable message will offer the user a chance to download the JRE before continuing from the URL specified in `codebase` *or* `pluginspage`.[16]

Example 6.2.14:

Modify the applet in example 6.2.12 so that the text appearing on the buttons is provided via appropriate parameter tags in an HTML document. Make sure to use the `EMBED/OBJECT` tags instead of the `APPLET` tags and download and install the latest version of the JRE. Test your application with Netscape and Internet Explorer. Report what happens if you specify a version higher than that for the currently installed JRE package in the `EMBED` tag.

There are two parts to the problem: first we need to modify our code to read the appropriate parameter tag and second we need to provide a different HTML page using `EMBED/OBJECT` instead of `APPLET`. We will use three parameters named `button_win`, `button_xwin`, `button_java` with appropriate values.

To solve the first problem we note that `JApplet` extends `Applet` so that the usual method `getParameter` can be used to read applet parameter values. The new applet is listed below with changes appearing in bold and italics:

```
public class JAppletWithButtons extends JApplet implements ActionListener
{  JButton windLook = new JButton("Windows");
   JButton unixLook = new JButton("Unix");
   JButton javaLook = new JButton("Java");
   JLabel label = new JLabel("Welcome to Swing", SwingConstants.CENTER);
   public void init()
   {  if (getParameter("button_win") != null)
         windLook.setText(getParameter("button_win"));
      if (getParameter("button_xwin") != null)
         unixLook.setText(getParameter("button_xwin"));
      if (getParameter("button_java") != null)
         javaLook.setText(getParameter("button_java"));
      Container content = getContentPane();
      content.setLayout(new FlowLayout());
      content.add(label);
      content.add(windLook);  windLook.addActionListener(this);
      content.add(unixLook); unixLook.addActionListener(this);
      content.add(javaLook); javaLook.addActionListener(this);
   }
   public void actionPerformed(ActionEvent ae)
   {  /* no changes from before */ }
```

[16] Using the `EMBED` to activate a Java Applet via the JRE will render the "Java Console" option of Netscape and Internet Explorer useless unless it is enabled via the plugin control panel. Also note that location of the matching angular brackets, which is slightly different for `EMBED` and `OBJECT` tags.

Note that we use the `setText` method for `JButton` instead of (deprecated) `setLabel` for `Button`. To solve the second problem, first recall how the appropriate `APPLET` tags with these parameters would look like:

```
<HTML>
<APPLET CODE="JAppletWithButtons.class"
        WIDTH="550"
        HEIGHT="80">
   <PARAM NAME="button_win" VALUE="Windows Look">
   <PARAM NAME="button_xwin" VALUE="Unix Look">
   <PARAM NAME="button_java" VALUE="Java Look">
</APPLET>
</HTML>
```

This HTML page would work fine for the `appletviewer` but not when loaded into Netscape 4 or Internet Explorer 5. Instead, we use the following HTML code:

```
<HTML>
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
     width="550" height="80"
     codebase="http://www.javasoft.com/products/plugin">
     <PARAM NAME="CODE" VALUE="JAppletWithButtons.class">
     <PARAM NAME="button_win"  VALUE="Windows Look ">
     <PARAM NAME="button_xwin" VALUE ="Unix Look ">
     <PARAM NAME="button_java" VALUE="Java Look ">
  <COMMENT>
    <EMBED type="application/x-java-applet;version=1.2"
       width="550" height="80"
       java_CODE="JAppletWithButtons.class"
       button_win = "Windows Look"
       button_xwin = "Unix Look"
       button_java = "Java Look"
       pluginspage = "http://www.javasoft.com/products/plugin">
    </EMBED>
  </COMMENT>
</OBJECT>
</HTML>
```

If the JRE is correctly installed and its version compatible with the requested version 1.2, the applet will appear at the expected position and will act almost exactly like a "true" applet[17].

To simulate what will happen if the JRE is not installed or has a lower version than requested, we can change the line in the HTML document

```
<EMBED type="application/x-java-applet;version=1.2"
```

to

```
<EMBED type="application/x-java-applet;version=1.3"
```

assuming that the JRE we have installed is version 1.2. When this modified HTML page is loaded into Netscape 4 the following dialog box will appear:

[17] Loading time is slower, since the JRE plugin must be loaded first, and it in turn loads the specified applet. Output directed to standard output will not appear in the browser's Java Console unless that option has been enabled via the plugin control panel.

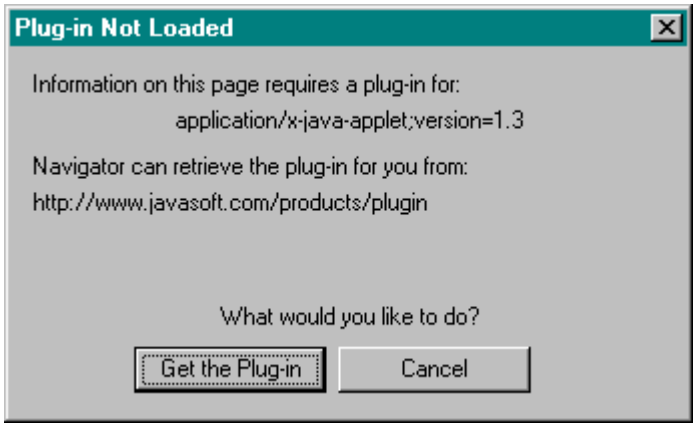


Figure 6.2.5: Dialog box to download a plugin that is not currently installed

If you click on "`Get the Plug-in`" the URL `http://www.javasoft.com/products/plugin` will load and you can download the appropriate plugin from there. After installing it, you can reload the original HTML document and the new plugin should handle the Java applet just fine. Internet Explorer will behave similar.

■

Now that we know how to create either standalone programs or applets using Swing components we can introduce additional Swing components.

Surroundings: Borders, Panels, and Menus

At this point we have seen that there is not that much difference between AWT and Swing components. Swing classes simply provide more features and have therefore more methods that you need to know but in principle they work just as the AWT components (at least at the level we are using them). We can therefore move quickly to introduce additional Swing components, starting with borders[18].

Definition 6.2.15: The Border Classes

Almost every Swing component can be enhanced by placing it inside one of several predefined borders via the inherited `setBorder` *method. The classes representing different border styles are:*

```
BevelBorder, EtchedBorder, LineBorder, MatteBorder, SoftBevelBorder
```

In addition, a `CompoundBorder` *can be used to combine two borders, an* `EmptyBorder` *can be used to add spacing, and a* `TitledBorder` *can be used add a title to a border.*

The border classes are contained in the `javax.swing.border` *package and are set by using the* `setBorder` *method inherited from* `JComponent`*. The various borders are instantiated using one of the constructors in tables 6.2.6 and 6.2.7*

[18] The `Border` class actually has no equivalent class from the AWT, but it provides every Swing component with the ability to put a border around itself using the method `setBorder` inherited from `JComponent`.

Class	Methods
BevelBorder	public BevelBorder(int bevelType) public BevelBorder(int bevelType, Color highlight, Color shadow) public static final int RAISED, LOWERED
EtchedBorder	public EtchedBorder() public EtchedBorder(int etchType) public EtchedBorder(int etchType, Color highlight, Color shadow) public static final int RAISED, LOWERED
LineBorder	public LineBorder(Color color) public LineBorder(Color color, int thickness)
MatteBorder	public MatteBorder(Icon tileIcon) public MatteBorder(int top, int left, int bottom, int right, Color color) public MatteBorder(int top, int left, int bottom, int right, Icon tileIcon)
SoftBevelBorder	public SoftBevelBorder(int bevelType) public SoftBevelBorder(int bevelType, Color highlight, Color shadow) Types from BevelBorder

Table 6.2.6: A Selection of the available `Border` *constructors*

Class	Methods
TitledBorder	public TitledBorder(String title) public TitledBorder(Border border) public TitledBorder(Border border, String title)
CompoundBorder	public CompoundBorder(Border outside ,Border inside)
EmptyBorder	public EmptyBorder(int top, int left, int bottom, int right)

Table 6.2.7: `Border` *combination classes*

Example 6.2.16:

Create a program with labels surrounded by each type of border. Also add a title to one of the borders, create a compound border, and add extra spacing to one border.

The code does not need much explanation, it should speak for itself:

```
import java.awt.GridLayout;
import java.awt.Color;
import javax.swing.*;
import javax.swing.border.*;

public class BorderOverview extends JFrame
{  public BorderOverview()
   {  super("Overview of Borders");
      ImageIcon icon = new ImageIcon("Icons/zigzag.gif");
      JLabel labels[] = new JLabel[10];
      labels[0] = new JLabel("Label with a BevelBorder (RAISED)");
      labels[0].setBorder(new BevelBorder(BevelBorder.RAISED));
      labels[1] = new JLabel("Label with a BevelBorder (LOWERED)");
      labels[1].setBorder(new BevelBorder(BevelBorder.LOWERED));
      labels[2] = new JLabel("Label with a EtchedBorder (RAISED)");
      labels[2].setBorder(new EtchedBorder(EtchedBorder.RAISED));
      labels[3] = new JLabel("Label with a EtchedBorder (LOWERED)");
      labels[3].setBorder(new EtchedBorder(EtchedBorder.LOWERED));
      labels[4] = new JLabel("Label with a LineBorder");
      labels[4].setBorder(new LineBorder(Color.red));
      labels[5] = new JLabel("Label with a MatteBorder");
      labels[5].setBorder(new MatteBorder(5, 5, 5, 5, icon));
```

```
        labels[6] = new JLabel("Label with a SoftBevelBorder (LOWERED)");
        labels[6].setBorder(new SoftBevelBorder(BevelBorder.LOWERED));
        labels[7] = new JLabel("Label with a Titled LineBorder");
        labels[7].setBorder(new TitledBorder(
                            new LineBorder(Color.gray, 2),"Title Border"));
        labels[8] = new JLabel("Compound border (Matte and Etched)");
        labels[8].setBorder(new CompoundBorder(
                            new EtchedBorder(EtchedBorder.RAISED),
                            new MatteBorder(5, 5, 5, 5, icon)));
        labels[9] = new JLabel("Border with extra spacing");
        labels[9].setBorder(new CompoundBorder(
                            new EtchedBorder(EtchedBorder.RAISED),
                            new EmptyBorder(8, 8, 8, 8)));

        getContentPane().setLayout(new GridLayout(5, 2, 5, 5));
        for (int i = 0; i < labels.length; i++)
           getContentPane().add(labels[i]);
        validate(); pack(); setVisible(true);
    }
    public static void main(String args[])
    {  BorderOverview bo = new BorderOverview(); }
}
```

Figure 6.2.8 shows how these various borders look like.

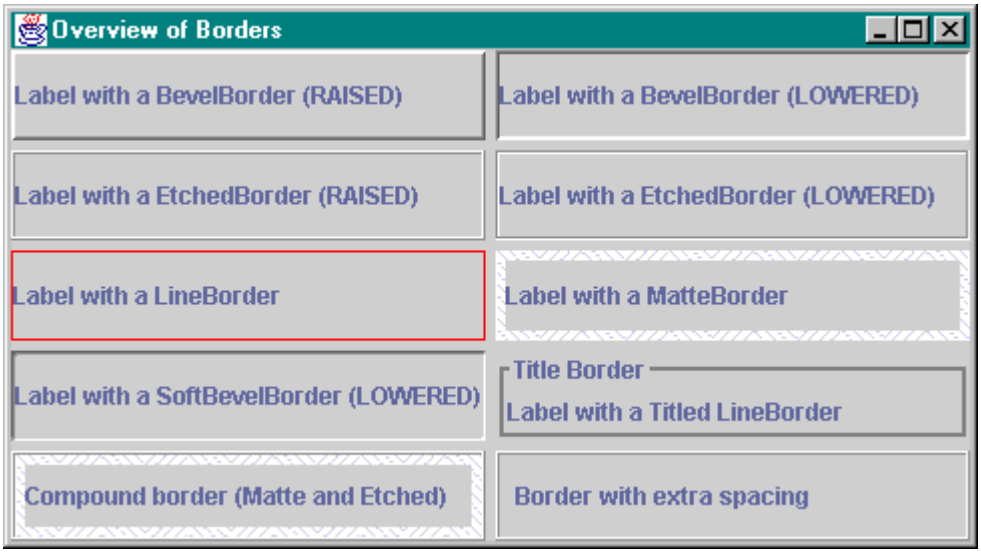


Figure 6.2.8: Appearance of various `Border` *classes*

■

To introduce the remaining components, we need to first find a class that can contain other components for more sophisticated layout schemes.

Definition 6.2.17: The `JPanel` Class

> *The* `JPanel` *class represents a generic container, somewhat similar to the AWT* `Panel` *class. Its main purpose is to group other components via the familiar layout managers from the AWT* [19].

[19] Swing actually provides several additional layout managers but we will not introduce them here. Please refer to the Java API for details.

```
public class JPanel extends JComponent implements Accessible
{  // selected constructors
   public JPanel()
   public JPanel(LayoutManager layout)
   // selected methods inherited from JComponent
   protected void paintComponent(Graphics g)
   protected void repaint()
   public void setToolTipText(String text)
   public void setBorder(Border border)
   // selected methods inherited from Container
   public void setLayout(LayoutManager mgr)
}
```

The `JPanel` *class is also used to contain Graphics, replacing the* `java.awt.Canvas` *class (see definition 6.4.13). The Java API defines it as follows:*

This class is used exactly like its AWT partner class `Panel`, so we do not need a simple example. Note, however, we will see in definition 6.4.13 that this class also serves to replace the `Canvas` class. Here is a somewhat more interesting example using the `JPanel` class.

Example 6.2.18:

Create a class `JPanelBox` that extends `JPanel` and places an etched border with a title around it. The class should have two constructors:

- The first constructor takes a `Component` and a `String` as input, places the component in the center of the panel, and puts an etched border with the string as title around it.
- The second constructor takes a `LayoutManager` and a `String` as input, sets the layout of the panel to the input layout manager, and puts an etched border with the string as title around it.

The class will be used in several other examples, so make sure to test the class extensively (you should also provide appropriate documentation comments as an exercise).

Adding a border around Swing components is not hard and requires only a few lines, as we have seen. But since borders are such an easy means to improve the look of a program, we would like to reduce a "few lines" to a single line. That will be the purpose of this class. In addition, whenever we use the `JPanelBox` we will always use the same type of border, giving all our applications a unified look. If we change the border type in `JPanelBox` all borders for programs using that class would also change.

Our class needs two constructors, both of which will need to add a titled border and an etched border to a component. We will use a utility method `setTitledBorder` to actually define the border, calling that method in both constructors. The rest of the class is straightforward:

```
import java.awt.Component;
import java.awt.BorderLayout;
import java.awt.LayoutManager;
import javax.swing.*;
import javax.swing.border.*;

public class JPanelBox extends JPanel
{  public JPanelBox(Component component, String title)
   {  super(new BorderLayout());
```

```
      add("Center", component);
      setTitledBorder(title);
   }
   public JPanelBox(LayoutManager layout, String title)
   {  super(layout);
      setTitledBorder(title);
   }
   public void setTitledBorder(String title)
   {  EtchedBorder etched = new EtchedBorder(EtchedBorder.LOWERED);
      TitledBorder titled = new TitledBorder(etched, title);
      super.setBorder(titled);
   }
}
```

This class can be used in two ways (since there are two constructors). The first constructor can be used to put a border around any component, while the second constructor could be thought of as an improved `JPanel` class. As an example, we will add a title to a single button, then use an instance of a `JPanelBox` when we would normally use a `JPanel`:

```
import java.awt.FlowLayout;
import java.awt.GridLayout;
import javax.swing.*;

public class JPanelBoxTest extends JFrame
{  private JButton titledButton = new JButton("Button with a Title");
   private JButton quit = new JButton("Quit");
   private JButton test = new JButton("Test");
   public JPanelBoxTest()
   {  super("JPanelBox Test");
      JPanelBox buttons = new JPanelBox(new FlowLayout(), "Action");
      buttons.add(quit);
      buttons.add(test);
      getContentPane().setLayout(new GridLayout(2,1));
      getContentPane().add(new JPanelBox(titledButton, "A Title"));
      getContentPane().add(buttons);
      validate(); pack(); setVisible(true);
   }
   public static void main(String args[])
   {  JPanelBoxTest jpbt = new JPanelBoxTest(); }
}
```

The `buttons` instance of `JPanelBox` replaces `JPanel` with a better looking panel, while using an anonymous `JPanelBox` adds a title to an existing component, in this case a `JButton`. The appearance of the program (which of course does nothing at all) is as follows:

Figure 6.2.9: The `JPanelBox` Class in Action

■

We will use this `JPanelBox` class in several future examples. Next we will explain how to use menus and popup menus which works both in standalone programs as well as for applets extending `JApplet`.

Definition 6.2.19: The `JMenu` Group of Classes

Swing provides several classes to create standard menus. They are:

```
JMenuItem, JCheckBoxMenuItem, JRadioButtonMenuItem, JSeparator,
JMenu, JMenuBar
```

Menu items are constructed using their intended title as parameter and added to instances of a `JMenu` *via the* `add` *method.* `JMenus` *in turn are added to an instance of* `JMenuBar`*, which can be attached to a* `JFrame` *or* `JApplet` *via the* `setJMenuBar` *method. The classes* `JMenuItem`*,* `JCheckBoxMenuItem`*, and* `JRadioButtonMenuItem` *can also receive an* `ImageIcon` *as second input parameter to its constructor, similar to* `JButton` *and* `JLabel`*. A* `JMenuItem` *can be activated via* `addActionListener(actionListener)`.

Example 6.2.20:

Create a program with several menu items and separators. There should be at least two menu choices, and each type of menu item should be represented. When the user selects a particular item, an appropriate message should appear and the state of the menu should change if appropriate. Some menu items should include icons and there should be at least one submenu within a menu.

Aside from the possibility of including icons, there is little difference between using the `JMenu` Swing classes and the `Menu` classes from the AWT (although their internal implementations are very much different). Therefore, we will define those menu items that need to generate action events as fields, define the necessary `JMenu` and `JMenuBar` items and add everything to its right place.

We will create a `File` and a `Tools` menu. `File` contains `Open`, `Save`, and `Exit`, divided by a separator. The `Tools` menu will contain a `JMenuCheckBoxItem` and a submenu, consisting of two `JMenuRadioButtonItems`. We add appropriate icons to `File | Open` and `File | Save`, assuming that the image files are located in an `Icons` directory. Finally, we allow all menu items to generate action events, capture them using a standard `actionPerformed` method, and display their results in a label that we add to our frame. Here is the complete class, extending `JFrame`:

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

public class MenuOverview extends JFrame implements ActionListener
{  JMenuItem fileOpen = new JMenuItem("Open",
                                      new ImageIcon("Icons/open.gif"));
   JMenuItem fileSave = new JMenuItem("Save",
                                      new ImageIcon("Icons/save.gif"));
   JMenuItem fileExit = new JMenuItem("Exit");
   JCheckBoxMenuItem toolView = new JCheckBoxMenuItem("View Tools");
   JRadioButtonMenuItem toolConverts =
                    new JRadioButtonMenuItem("Convert Hyperlinks");
   JRadioButtonMenuItem toolDisplays =
                    new JRadioButtonMenuItem("Display Hyperlinks", true);
   JLabel status = new JLabel("This is a status bar to show the events");
   public MenuOverview()
```

```
    {  super("Menu Overview");
       JMenu file = new JMenu("File");
       JMenu tools = new JMenu("Tools");
       file.add(fileOpen);   fileOpen.addActionListener(this);
       file.add(fileSave);   fileSave.addActionListener(this);
       file.add(new JSeparator());
       file.add(fileExit);  fileExit.addActionListener(this);
       tools.add(toolView); toolView.addActionListener(this);
       JMenu toolLinks = new JMenu("Links ...");
       toolLinks.add(toolConverts); toolConverts.addActionListener(this);
       toolLinks.add(toolDisplays); toolDisplays.addActionListener(this);
       tools.add(toolLinks);
       JMenuBar menu = new JMenuBar(); menu.add(file); menu.add(tools);
       getContentPane().add(status);
       setJMenuBar(menu);
       validate(); pack(); setVisible(true);
    }
    public void actionPerformed(ActionEvent ae)
    {  if (ae.getSource() == fileExit)
          System.exit(0);
       else
          status.setText("Command: " + ae.getActionCommand());
    }
    public static void main(String args[])
    {  MenuOverview mo = new MenuOverview(); }
}
```

When the program executes, it will look as follows:

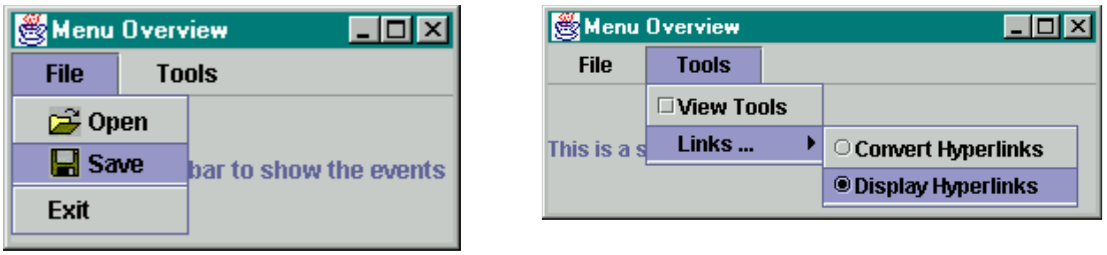


Figure 6.2.10: Menus with icons, check boxes, and radio buttons

■

In addition to standard menus appearing at the top of a frame, Java also supports variable popup menus that can be made to appear at any time. Popup menus are most often used as context-sensitive menus that appear when the user clicks the right mouse button over a component. The popup menu choices are usually dependent on the particular component they are attached to and provide options that are specific to that component.

Definition 6.2.21: Popup Menus

Java provides support for popup menus via the `JPopupMenu` *class. Popup menus can popup over any* `Component` *when the user clicks the right mouse button (or at other occasions). The* `JPopupMenu` *class can contain a* `JMenuItem`, `JCheckboxMenuItem`, `JRadioButtonMenuItem`, *or* `JPopupMenu.Separator`, *including icons attached to those objects. The Java API defines this class as follows:*

```
public class JPopupMenu extends JComponent
                        implements Accessible, MenuElement
{  // constructors
```

```
   public JPopupMenu()
   public JPopupMenu(String label)
   // selected methods
   public void show(Component invoker, int x, int y)
}
```

To make a popup menu appear, attach an appropriate `MouseListener` *to the component that will own the menu and call the* `show` *method. To check for a mouse click with the right mouse button, use* `SwingUtilities.isRightMouseButton`.[20]

Example 6.2.22:

Create a program containing two components. Each component should have its own popup menu that appears when the user presses the right mouse button over that component. At least one of the menu items should cause some action.

We will make the two components owning the popup menus of type `JPanel`, each containing some label. One popup menu will contain an `Open`, `Save`, and `Exit` menu choice, where the `Exit` choice will cause the program to quit. The second popup menu will contain two other menu items. To activate the popup menus we will define two inner classes extending `MouseAdapter`. Each will be responsible for its own popup menu. One will be attached to the first panel, the other to the second. Here is the code:

```
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class PopupMenuTest extends JFrame implements ActionListener
{  private JPopupMenu file = new JPopupMenu();
   private JMenuItem  fileOpen = new JMenuItem("Open");
   private JMenuItem  fileSave = new JMenuItem("Save");
   private JMenuItem  fileExit = new JMenuItem("Exit");
   private JPopupMenu options = new JPopupMenu();
   private JMenuItem optionsSpell  = new JMenuItem("Spell Check");
   private JMenuItem optionsSecret = new JMenuItem ("Show Codes");
   private class FilePopupListener extends MouseAdapter
   {  private JComponent owner;
      public FilePopupListener(JComponent _owner)
      {  owner = _owner; }
      public void mousePressed(MouseEvent me)
      {  if (SwingUtilities.isRightMouseButton(me))
            PopupMenuTest.this.file.show(owner, me.getX(), me.getY());
      }
   }
   private class OptionsPopupListener extends MouseAdapter
   {  private JComponent owner;
      public OptionsPopupListener(JComponent _owner)
      {  owner = _owner; }
      public void mousePressed(MouseEvent me)
      {  if (SwingUtilities.isRightMouseButton(me))
            PopupMenuTest.this.options.show(owner, me.getX(), me.getY());
      }
   }
   public PopupMenuTest()
```

[20] To fine-tune the behavior of a popup menu, you can also attach a `PopupMenuListener` from `javax.swing.event` to the component that will own the menu. For details, check the Java API.

```
    {  super("Popup Menu Test");
       JPanel top = new JPanel(new FlowLayout());
       top.add(new JLabel("This is one particular panel"));
       top.setBorder(new EtchedBorder(EtchedBorder.LOWERED));
       JPanel bottom = new JPanel(new FlowLayout());
       bottom.add(new JLabel("This is another panel"));
       bottom.setBorder(new EtchedBorder(EtchedBorder.LOWERED));
       file.add(fileOpen); file.add(fileSave);
       file.add(new JPopupMenu.Separator());
       file.add(fileExit); fileExit.addActionListener(this);
       options.add(optionsSpell);
       options.add(optionsSecret);
       top.addMouseListener(new FilePopupListener(top));
       bottom.addMouseListener(new OptionsPopupListener(bottom));
       getContentPane().setLayout(new GridLayout(2,1));
       getContentPane().add(top); getContentPane().add(bottom);
       validate(); pack(); setVisible(true);
    }
    public void actionPerformed(ActionEvent ae)
    {  System.exit(0); }
    public static void main(String args[])
    {  PopupMenuTest pmt = new PopupMenuTest(); }
}
```

Note that the mouse adapters know which `JComponent` owns the popup menu. That ensures that the menu will appear at the correct (`x`,`y`) coordinates within that component. Note that the inner class can access the enclosing `private` class members `file` and `options` without problems.[21] When the class executes you can click the right mouse button over either panel to see the two popup menus as in the next figure.

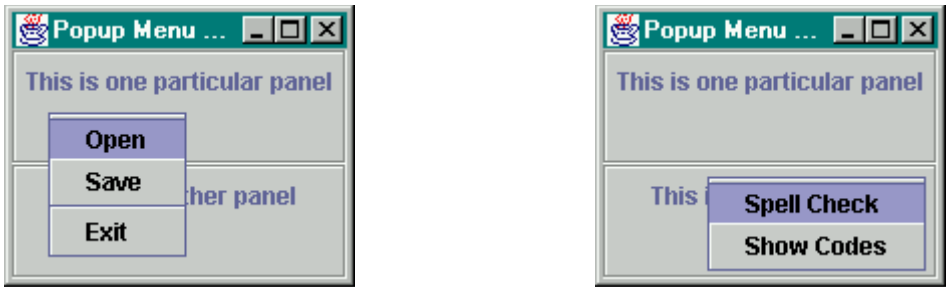


Figure 6.2.11: Popup menus in action

■

States: Checkboxes, Radiobuttons, and Drop-down lists

The next group of Swing components represents various "choices". Swing supports `JCheckBox`, `JRadioButton`, and `JComboBox`, as well as `ButtonGroup` to group choices into mutually exclusive categories.

[21] In an exercise you will be asked to change this class so that it achieves the same functionality with only one inner class. Note that in Java 1.2 and below you have to preface `file` and `options` with the keyword `this$0`.

Definition 6.2.23: The `JCheckBox`, `JRadioButton`, and `ButtonGroup` Classes

`JCheckBox` *and* `JRadioButton` *are classes that support toggles that can be in one of two states, "on" or "off". Check boxes are used to provide independent on/off switches, while radio buttons are usually grouped together using* `ButtonGroup` *to create mutually exclusive categories, i.e. exactly one choice in a category is active at any given time. Both classes have similar constructors:*

```
public class XXX extends JToggleButton implements Accessible
{  // constructors
   public XXX(Icon icon)
   public XXX(Icon icon, boolean selected)
   public XXX(String text)
   public XXX(String text, boolean selected)
   public XXX(String text, Icon icon)
   public XXX(String text, Icon icon, boolean selected)
}
```

where `XXX` *is either* `JCheckBox` *or* `JRadioButton`*. The state of either class can be determined using* `isSelected` *and can be set via* `setSelected` *or* `doClick`*. Both classes can be grouped into mutually exclusive categories by adding them to an instance of* `ButtonGroup` *using* `add`*, and both classes generate action events if activated via* `addActionListener`*.*

Example 6.2.24:

Create a program that simulates a standard Windows `Print` dialog. It should have mutually exclusive options of printing all pages, the current page, or the current selection, and options to specify whether to include a header, line numbers, or a file name on each page. The various options should appear inside panels with a titled border and the program should contain and `OK` and a `Cancel` button. Clicking on either button should terminate the program, and selecting one of the options should print an appropriate string to standard output. The entire frame should have a "Windows Look-and-Feel".

We clearly need at least two panels, one to contain the mutually exclusive options and another to contain the remaining choices. The first panel should contain radio buttons that are grouped together so that only one choice can be active. The second panel should contain checkboxes that can be independently turned on and off. We choose a titled etched border for both panels. Finally, there should be a third panel, containing the `OK` and `Cancel` buttons at the bottom of the frame. The entire program will extend `JFrame`. Here is the code:

```
import java.awt.GridLayout;
import java.awt.FlowLayout;
import java.awt.BorderLayout;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class PrintDialog extends JFrame implements ActionListener
{  private JButton okay = new JButton("OK");
   private JButton cancel = new JButton("Cancel");
   private JRadioButton all = new JRadioButton("All Pages");
   private JRadioButton current = new JRadioButton("Current Page");
   private JRadioButton selected = new JRadioButton("Selected Pages");
   private JCheckBox numbers = new JCheckBox("Include Line Numbers");
   private JCheckBox headers = new JCheckBox("Header on all pages");
   private JCheckBox fileName = new JCheckBox("File name on each page");
   public PrintDialog()
```

```
   {  super("Print Dialog");
      ButtonGroup pageOptions = new ButtonGroup();
      pageOptions.add(all);
      pageOptions.add(current);
      pageOptions.add(selected);
      all.setSelected(true);
      JPanel pagePanel = new JPanel(new GridLayout(3,1));
      pagePanel.add(all);          all.addActionListener(this);
      pagePanel.add(current);      current.addActionListener(this);
      pagePanel.add(selected);     selected.addActionListener(this);
      pagePanel.setBorder(new TitledBorder("Print Range:"));
      JPanel optionsPanel = new JPanel(new GridLayout(3,1));
      optionsPanel.add(numbers);   numbers.addActionListener(this);
      optionsPanel.add(headers);   headers.addActionListener(this);
      optionsPanel.add(fileName);  fileName.addActionListener(this);
      optionsPanel.setBorder(new TitledBorder("Format Options:"));
      JPanel buttons = new JPanel(new FlowLayout());
      buttons.add(okay);           okay.addActionListener(this);
      buttons.add(cancel);         cancel.addActionListener(this);
      getContentPane().setLayout(new BorderLayout());
      getContentPane().add("West", pagePanel);
      getContentPane().add("East", optionsPanel);
      getContentPane().add("South", buttons);
      validate(); pack(); setVisible(true);
   }
   public void actionPerformed(ActionEvent ae)
   {  if ((ae.getSource() == okay) || (ae.getSource() == cancel))
         System.exit(0);
      else
         System.out.println("Command: " + ae.getActionCommand());
   }
   public static void main(String args[])
   {  try
      {  String lf = "com.sun.java.swing.plaf.windows.WindowsLookAndFeel";
         UIManager.setLookAndFeel(lf);
      }
      catch (Exception e)
      {  System.err.println("Exception: " + e); }
      PrintDialog pd = new PrintDialog();
   }
}
```

The program will produce the following screen:

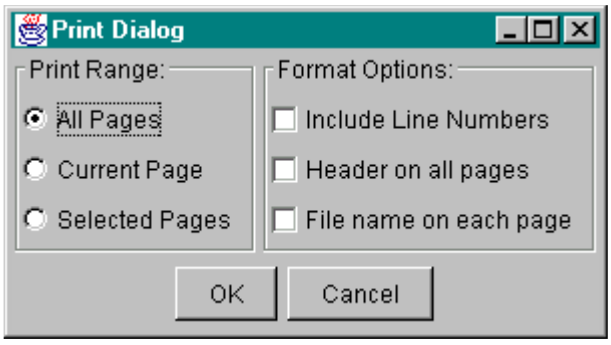


Figure 6.2.12: The `PrintDialog` *class in action*

Exactly one of the "Print Range" radio buttons can be selected at any time, while any combination of the "Format Options" check boxes is valid. Each time an option is selected or changed an appropriate action event is generated.

A similar screen is generated using the (default) Java or Unix look-and-feel.

■

Another type of choice is provided via the `JComboBox` class. It presents a "drop-down" list of choices from which the user can select one. If enabled, the class can also allow the user to type in their own option.

Definition 6.2.25: The `JComboBox` Class

This class represents a combination of a text field and drop-down list that lets the user either type in a value or select it from a drop-down list[22]. The editing capability is disabled by default but can be enabled so that the user can type their own choice. The Java API defines the class as follows:

```
public class JComboBox extends JComponent
       implements ItemSelectable, ListDataListener,
                  ActionListener, Accessible
{  // selected constructors
   public JComboBox()
   public JComboBox(Object[] items)
   // selected methods
   public void setSelectedItem(Object anObject)
   public Object getSelectedItem()
   public void setSelectedIndex(int anIndex)
   public int getSelectedIndex()
   public void addItem(Object anObject)
   public void setEnabled(boolean flag)
   public void setEditable(boolean flag)
}
```

This class is comparable to `java.awt.Choice`*, but the AWT* `Choice` *class does not provide for editing capabilities. Options to* `java.awt.Choice` *are added via* `add`*, while options to a* `JComboBox` *are added via* `addItem`*.*

Example 6.2.26:

Enhance previous example 6.2.24 of a print dialog to include a drop-down list with available (fictitious) printers. The editing capabilities of the combo box should be disabled in this case. When the user clicks on `Okay` or `Cancel`, the currently selected printer should be displayed on standard output.

We need to add a combo box with several fictitious printers as options and ensure that the combo box is not editable. To provide a nicer look, we add that combo box together with a label in a separate panel with its own border. Here is the new `PrintDialog` class, with the modified code in bold and italics:

```
public class PrintDialog extends JFrame implements ActionListener
{  // fields as before, together with the new field
   private JComboBox printers = new JComboBox();
   public PrintDialog()
   {  super("Print Dialog");
      printers.addItem("HP DeskJet 690C");
      printers.addItem("HP DeskJet 870C Series");
      printers.addItem("HP LaserJet 5");
```

[22] As usual with Swing components, the drop-down list could contain icons as well as strings. To acomplish that, a different rendering model must be defined for the class. We will briefly show an example of using a rendering model in the MandelBrot example at the end of this chapter. Items can also by removed from the list by using a model; please refer to the examples on lists and the Java API for details.

```
            printers.addItem("Networked Office Printer");
            printers.addItem("Networked Computer Lab Printer");
            JPanel printerPanel = new JPanel(new FlowLayout(FlowLayout.LEFT));
            printerPanel.setBorder(new TitledBorder("Printer Selection"));
            printerPanel.add(new JLabel("Name: "));
            printerPanel.add(printers);
            // remaining code as before, with one line added at the end:
          getContentPane().add("North", printerPanel);
            validate(); pack(); setVisible(true);
        }
        public void actionPerformed(ActionEvent ae)
        {  if ((ae.getSource() == okay) || (ae.getSource() == cancel))
           {  System.out.println("Printer: " + printers.getSelectedItem());
              System.exit(0);
           }
           else
              System.out.println("Command: " + ae.getActionCommand());
        }
        public static void main(String args[])
        {  /* no change from before */ }
    }
```

This "new and improved" print dialog looks as follows:

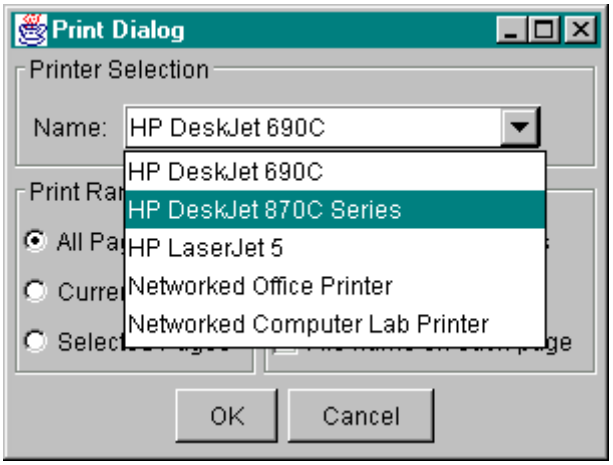


Figure 6.2.13: The new `PrintDialog` *class with a combo box for the printer name in Windows look*

■

All Swing components introduced so far are reasonably easy to use because they are closely related to their corresponding AWT classes. Next we will leave familiar territory and introduce classes that sound familiar but are used differently than their AWT namesakes, as well as additional classes that do not have a counterpart in the AWT. Note in particular that we have not yet mentioned lists and text areas. As it will turn out, those classes are different from their cousins in `java.awt`.

6.3. Advanced Swing Classes

We will start this section by defining three types of panes: a scrolling pane, a tabbing pane, and a split pane. The scroll pane, in particular, is frequently used to add scrolling capabilities to other

classes, while tab and split panes are a welcome addition to the `JPanel` class to allow for more sophisticated layouts.

Scrolling, Tabbing, and Splitting

The first and simplest to use class is `JScrollPane`. It adds horizontal and vertical scrolling capabilities to other Swing components and is entirely simple to use.

Definition 6.3.1: The `JScrollPane` Class

This class represents a container that can manage vertical and horizontal scrollbars around other components. It is typically used to provide scrolling capabilities for images, lists, text areas, tables, and trees. In addition to adding scrollbars, it can also add a border to a component via `setViewPortBorder`*, as well as be the recipient of a border via* `setBorder`*. The Java API defines this class as follows:*

```
public class JScrollPane extends JComponent
                         implements ScrollPaneConstants, Accessible
{  // selected constructor
   public JScrollPane(Component view)
   // selected methods
   public void setViewportBorder(Border viewportBorder)
   public void setBorder(Border border)
   public void setPreferredSize(Dimension newSize)
}
```

Example 6.3.2:

Create a program that loads a large image as an image icon into a `JLabel` using a `JScrollPane` to add scrolling capabilities to the image.

We have already seen that a `JLabel` can contain an image instead of or in addition to a string. There is no reason why that image has to be particularly small. But if the image is too large it makes the label too large to be positioned by a layout manager unless there are scrollbars attached to the image. That's where the `JScrollPane` class comes in.

Our class will extend `JFrame`, of course. We load the image into an `ImageIcon`, then set that icon as input to a `JLabel`. Instead of adding the `JLabel` directly to the frame, however, we insert it into an instance of a `JScrollPane`. Then we adjust the size of the scroll pane and add *it* to the frame instead of the label. To allow our program to quit gracefully, we also add a `Quit` button. Here is the code:

```
import java.awt.FlowLayout;
import java.awt.Dimension;
import java.awt.event.*;
import javax.swing.*;

public class ScrollingImage extends JFrame implements ActionListener
{  private JLabel imageLabel = new JLabel();
   private JButton quit = new JButton("Quit");
   private String fileName = "cat.jpg";
   public ScrollingImage()
   {  super("Scrolling Image");
```

```
        ImageIcon icon = new ImageIcon(fileName);
        imageLabel.setIcon(icon);
        JScrollPane scrollingLabel = new JScrollPane(imageLabel);
        scrollingLabel.setPreferredSize(new Dimension(300,150));
        JPanel buttons = new JPanel(new FlowLayout());
        buttons.add(quit);
        getContentPane().add("Center", scrollingLabel);
        getContentPane().add("South", buttons);
        quit.addActionListener(this);
        validate(); pack(); setVisible(true);
    }
    public void actionPerformed(ActionEvent ae)
    {  System.exit(0); }
    public static void main(String args[])
    {  ScrollingImage si = new ScrollingImage(); }
}
```

Note that we specifically set the dimensions of our scroll pane to 300 by 150 via `setPreferredSize`. If we did not do that, the layout manager would try to use the full dimensions of the image to position the label. Assuming that the image file named `cat.jpg` is present in the same directory as our class, the program will display a scrollable image similar to the following:

Figure 6.3.1: Resulting scrollable image of the `ScrollingImage` *class*

■

The second class represents a tabbed pane. It is used to group together various other components so that the user can click on a tab to bring the components located on that tab to the foreground. This class is very convenient to break up large numbers of GUI components into manageable groups or to relegate less frequently used components to an unobtrusive location.

Definition 6.3.3: The `JTabbedPane` Class

This class represents a component that lets the user switch between groups of components by clicking on a tab with a title or icon[23]*. The components in a tabbed pane typically consist of instances of* `JPanel` *containing in turn groups of other components. The Java API defines this class as follows*

```
public class JTabbedPane extends JComponent
       implements Serializable, Accessible, SwingConstants
```

[23] This class is somewhat comparable to `java.awt.CardLayout`, but in that class the additional groups are invisible whereas in a `JTabbedPane` the tabs are always visible. Note that `CardLayout` is not covered in chapter 4.

```
{  // selected constructors
   public JTabbedPane()
   // selected methods
   public void addTab(String title, Component component)
   public void addTab(String title, Icon icon, Component component)
   public void addTab(String title, Icon icon,
                      Component component, String toolTip)
   public boolean isEnabledAt(int index)
   public void setEnabledAt(int index, boolean enabled)
   public void setBackgroundAt(int index, Color background)
   public void setForegroundAt(int index, Color foreground)
}
```

Example 6.3.4:

Modify example 6.3.2 so that the scrolling image is added to one tab of a `JTabbedPane` and information about that image to a second.

The modifications are simple: instead of adding the scrolling image directly to the frame we first create an instance of `JTabbedPane`, then add the image label as its first tab component. As the second tab we add a panel containing the file name and image dimensions of the image so that the user can choose to see either the image or the information about the image. The code is almost as before, with the modified parts in bold and italics:

```
public class ScrollingTabImage extends JFrame implements ActionListener
{  private JLabel imageLabel = new JLabel();
   private JButton quit = new JButton("Quit");
   private String fileName = "cat.jpg";
   public ScrollingTabImage ()
   {  super("Scrolling Image");
      ImageIcon icon = new ImageIcon(fileName);
      imageLabel.setIcon(icon);
      JScrollPane scrollingLabel = new JScrollPane(imageLabel);
      scrollingLabel.setPreferredSize(new Dimension(300,150));
      JPanel info = new JPanel(new GridLayout(3,1));
      info.add(new JLabel("Image Name: " + fileName));
      info.add(new JLabel("Image Width: " + icon.getIconWidth()));
      info.add(new JLabel("Image Height: " + icon.getIconHeight()));
      JTabbedPane tabs = new JTabbedPane();
      tabs.addTab("Image", scrollingLabel);
      tabs.addTab("Info", info);
      JPanel buttons = new JPanel(new FlowLayout());
      buttons.add(quit);
      getContentPane().add("Center", tabs);
      getContentPane().add("South", buttons);
      quit.addActionListener(this);
      validate(); pack(); setVisible(true);
   }
   public void actionPerformed(ActionEvent ae)
   {  System.exit(0); }
   public static void main(String args[])
   { ScrollingTabImage sti = new ScrollingTabImage(); }
}
```

The tabs produced by this program will look as follows:

Figure 6.3.2: Both tabs of the `ScrollingTabImage` *class*

■

The last class in this section is `JSplitPane`. It allows the you to arrange two components next to each other, either horizontally or vertically, so that the divider between the two components can be adjusted at runtime by dragging it with the mouse.

Definition 6.3.5: The `JSplitPane` Class

This class is used to provide a user-adjustable view of two groups of components. The two components can be arranged vertically or horizontally and interactively resized. The orientation is specified using the constants `HORIZONTAL_SPLIT` *or* `VERTICAL_SPLIT` *of the* `JSplitPane` *class. The class can optionally be set to allow for "one-touch" expansion of either component, and to continuously repaint components during resizing. The Java API defines this class as follows:*

```
   public class JSplitPane extends JComponent implements Accessible
   {  // selected constructors
      public JSplitPane(int newOrientation,
                        Component component1, Component component2)
// selected methods
      public void setDividerSize(int newSize)
      public void setDividerLocation(int location)
      public void setOneTouchExpandable(boolean newValue)
      public void setContinuousLayout(boolean newContinuousLayout)
   }
```

Example 6.3.6:

In example 6.3.2 we created a `ScrollingImage` class that we later modified in the `ScrollingTabImage` class in example 6.3.4. This time, modify `ScrollingImage` again so that the image is situated on the right of a divider and information about the image on the left.

The modifications are similar to the previous one: instead of adding the scrolling image directly to the frame we first create a panel containing information about the image, then add the scrolling image and the information panel to an instance of `JSplitPane`, instantiated with horizontal orientation. We will let the class specify the location of the divider, but we will set its size to 10 points and turn on one-touch expansion and automatic repainting. The code is similar to before, with the modified parts in bold and italics:

```
public class ScrollingSplitImage extends JFrame implements ActionListener
{  private JLabel imageLabel = new JLabel();
   private JButton quit = new JButton("Quit");
   private String fileName = "cat.jpg";
   public ScrollingSplitImage ()
   {  super("Scrolling Image");
      ImageIcon icon = new ImageIcon(fileName);
      imageLabel.setIcon(icon);
      JScrollPane scrollingLabel = new JScrollPane(imageLabel);
      scrollingLabel.setPreferredSize(new Dimension(300,150));
      JPanel info = new JPanel(new GridLayout(3,1));
      info.add(new JLabel("Image Name:   " + fileName));
      info.add(new JLabel("Image Width:  " + icon.getIconWidth()));
      info.add(new JLabel("Image Height: " + icon.getIconHeight()));
      JSplitPane split = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                                        scrollingLabel, info);
      split.setDividerSize(10);
      split.setOneTouchExpandable(true);
      split.setContinuousLayout(true);
      JPanel buttons = new JPanel(new FlowLayout());
      buttons.add(quit);
      getContentPane().add("Center", split);
      getContentPane().add("South", buttons);
      quit.addActionListener(this);
      validate(); pack(); setVisible(true);
   }
   public void actionPerformed(ActionEvent ae)
   {  System.exit(0); }
   public static void main(String args[])
   { ScrollingSplitImage ssi = new ScrollingSplitImage(); }
}
```

When the program executes, it will arrange the scrolling image and the information pane next to each other with a 10 point divider in the middle. The divider contains special hotspots that allow the user to expand one of the components with a single click. We have also set the split plane so that both components will be continuously repainted as the user drags the divider around. None of this is reflected in the screen shot in figure 6.3.3 but it will hopefully convey the general idea.

Figure 6.3.3: Screenshot of `ScrollingSplitImage` *(divider can be moved interactively)*

■

Models and Views

Now we are ready to explain a common design element of many Swing components called the Model/View split. The model/view structure is, in fact, what makes Swing so flexible. The basic idea is to separate the data storage from the visual representation of the data. The "old" `java.awt.List` class, for example, did not split data storage and data representation and one class was used to *store* data (of type String) and to *represent* it visually to the user, including scrollbars and everything. That made the `List` class easy to use but not very flexible. You could not, for example, use it to store anything but String values. In the Swing `JList` class, on the other hand, data is stored separately in a `DefaultListModel`, which can efficiently store any type of object. This data is then visually represented to the user by a `JList` class associated with the `DefaultListModel`. The data representation could be based on strings only or could include strings and icons, or even graphics objects, as needed. This approach makes Swing components very flexible, but more complicated to use than AWT classes.

Definition 6.3.7: The Model/View Structure

Many advanced Swing components employ a model/view split of representing data. The Model is a class used to store data while the View is a class used to visually represent data to the user. Typical models used in Swing components are:

```
Document, DefaultListModel, DefaultTableModel, DefaultTreeModel
```

Typical views in Swing are represented by the classes:

```
JTextField, JTextArea, JEditorPane, JTextPane, JList, JTable, JTree
```

In most cases, a `JScrollPane` *can be used to provide scrolling capabilities to views. Typically, a model is instantiated and data is added to the model. An appropriate view class is then instantiated and tied to the data model. The user then manipulates the data in the model via the visual representation provided by the view. An appropriate event handler can be attached to the view or model to react to user actions or state changes. For details, please refer to table 6.3.4.*

This model/view structure in Swing does not always follow the same pattern, but the general idea is adhered to sufficiently closely so that it is important to understand it before introducing specific classes.

View	Default Model	Listener	Event
JTextField	PlainDocument	DocumentListener	DocumentEvent
JTextArea	PlainDocument	DocumentListener	DocumentEvent
JEditorPane	PlainDocument	DocumentListener	DocumentEvent
JTextPane	PlainDocument	DocumentListener	DocumentEvent
JComboBox	DefaultComboBoxModel	ItemListener	ItemEvent
JList	DefaultListModel	ListSelectionListener	ListSelectionEvent
JTable	DefaultTableModel	ListSelectionListener	ListSelectionEvent
JTree	DefaultTreeModel	TreeSelectionListener	TreeSelectionEvent

Table 6.3.4: Typical Models/Views and Associated Listeners and Events

As our first example of a class that uses this model/view split, we will discuss the `JList` class. As is typical, a `JList` does not contain data directly, nor does it support scrolling. Instead, data that your list should represent must be explicitly tied to the list via the appropriate model and scrolling

capabilities can be added using `JScrollPane`. This increases the complexity of `JList` over `java.awt.List` but it also increases its flexibility. You can, for example, add icons in addition to text to a list, or even complex drawings, and you can link a `JList` to existing data of types different from `String`. One big downside of this added flexibility is that a `JList` is not code-compatible to an AWT `List`.

We need to introduce three classes before we can work effectively with Swing lists: the view (`JList`), a possible model (`DefaultListModel`), and a listener (`ListSelectionListener`), so here we go.

Definition 6.3.8: The `JList` Class

This class presents a view onto sequential data and allows the user to select one or more of these data objects. Scrolling capabilities can be added using a `JScrollPane` *instance. A* `JList` *can be created to provide functionality similar to* `java.awt.List` *but it is not code-compatibly to the AWT* `List` *class. In particular, data is not added to a* `JList` *but to the attached model. The Java API defines this class as follows:*

```
public class JList extends JComponent
       implements Scrollable, Accessible
{  // selected constructors
   public JList(ListModel dataModel)
   // selected methods
   public Object getSelectedValue()
   public Object[] getSelectedValues()
   public int getSelectedIndex()
   public int[] getSelectedIndices()
   public void setSelectedIndex(int index)
   public void setSelectedIndices(int[] indices)
   public void clearSelection()
   public void setSelectionMode(int mode)
   public void setModel(ListModel dataModel)
   public void ensureIndexIsVisible(int index)
   public void addListSelectionListener(ListSelectionListener l)
}
```

By default, a `JList` *allows selecting multiple items, but that behavior can be modified using* `setSelectionMode(int mode)`*, where mode can be one of* `SINGLE_SELECTION`*,* `SINGLE_INTERVAL_SELECTION`*, or* `MULTIPLE_INTERVAL_SELECTION` *from the* `ListSelectionModel` *class.*

In other words, `JList` deals exclusively with the selecting process in a list but it does not actually store any values nor does it provide support for scrolling. The next class is an example of a model that can store data and associate it with a `JList`:

Definition 6.3.9: The `DefaultListModel` Class

This class can store sequential data suitable for representation in a `JList`*. It stores data dynamically, using only as much memory as necessary at any given time.*[24] *The Java API defines this class as follows:*

[24] We will explore dynamic data structures in detail in chapter 8. One of the structures that will be introduced there is a Vector, which is actually the underlying structure for DefaultListModel.

```
   public class DefaultListModel extends AbstractListModel
   {  // constructor
      public DefaultListModel()
      // selected methods
      public int getSize()
      public void add(int index, Object element)
      public void addElement(Object obj)
      public Object remove(int index)
      public void clear()
      public Object get(int index)
      public Object set(int index, Object element)
      public boolean isEmpty()
}
```

Finally, the `ListSelectionListener` interface specifies how to react to changes in the selection of a `JList`:

Definition 6.3.10: The `ListSelectionListener` and `ListSelectionEvent` Classes

The `ListSelectionListener` *is notified when a selection in a list changes, and the* `ListSelectionEvent` *characterizes the change in the selection. Both classes are part of the* `javax.swing.event` *package. The Java API defines these classes as follows:*

```
   public abstract interface ListSelectionListener extends EventListener
   {  // method to be implemented
      public void valueChanged(ListSelectionEvent e)
   }
   public class ListSelectionEvent extends EventObject
   {  // methods
      public int getFirstIndex()
      public int getLastIndex()
      public boolean getValueIsAdjusting()
      public Object getSource() // inherited from EventObject
   }
```

The `getValueIsAdjusting` *method returns* `false` *if the user has finished making a selection from the list.*

Now we are ready to show an example that explains how these three classes work together to provide all the functionality of the old `java.awt.List` class and much more.

Example 6.3.11:

In example 6.3.2 we created a program that could display a single image (we later modified that program to use a tabbed plane and a split plane). Modify that program accordingly so that it can show one of several image files from a list. The image should change any time the user selects a different file name.

First, let's quickly get the layout for this example out of the way: we will show a list of file names on the left side of a split pane and the image corresponding to the selected name on the right. The image, of course, should be scrollable if necessary.

That said, here is our idea for creating our program (assuming, of course, that we have appropriate image files available).

- we use a field of type `DefaultListModel` to hold the names of the image files
- we use a field of type `JList` and connect it to the `DefaultListModel`
- we use a field of type `JLabel` to display the image as an image icon

- we add the file names to the list data model
- we set the list selection mode to `SINGLE_SELECTION`
- we define instances of a `JScrollPane` to add scrolling capabilities to the list as well as to the `JLabel`
- we define a `JSplitPane` and add the scrollable list as its left component and the scrollable image label as its right
- we add the `JSplitPane` as the center component of a `JFrame`
- we add a `ListSelectionListener` to the list to react to the user making a selection, and a named inner class of type `WindowAdapter` to handle closing of the frame

The file names of the images are given in an array of strings and are assumed to be located in a directory named `Images` at the same level where our class file is located. Here is the code:

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

public class ImageViewer extends JFrame implements ListSelectionListener
{  private String FILES[] = {"cat.jpg", "river.jpg", "swan.gif"};
   private DefaultListModel data = new DefaultListModel();
   private JList list = new JList(data);
   private JLabel image = null;
   private class WindowCloser extends WindowAdapter
   {  public void windowClosing(WindowEvent we)
      {  System.exit(0); }
   }
   public ImageViewer()
   {  super("Image Viewer");
      for (int i = 0; i < FILES.length; i++)
         data.addElement("Images/" + FILES[i]);
      image = new JLabel(new ImageIcon(FILES[0]), SwingConstants.CENTER);
      list.setSelectedIndex(0);
      list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
      JScrollPane scrollList = new JScrollPane(list);
      JScrollPane scrollImage = new JScrollPane(image);
      JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                                            scrollList, scrollImage);
      getContentPane().add("Center", splitPane);
      list.addListSelectionListener(this);
      addWindowListener(new WindowCloser());
      validate(); pack(); setVisible(true);
   }
   public void valueChanged(ListSelectionEvent lse)
   {  if (!lse.getValueIsAdjusting())
        image.setIcon(new ImageIcon(list.getSelectedValue().toString()));
   }
   public static void main(String args[])
   {  ImageViewer iv = new ImageViewer(); }
}
```

The only thing we added to the previous outline of our class is that we initially load the first image and select it in the list. We also make sure that the image appears centered inside the `JLabel` by using `SwingConstants.CENTER`.

The interesting part of the code is that our class implements the `ListSelectionListener` and therefore must implement the `valueChanged` method. In that method we first ensure that the user has indeed finished making a selection by calling `getValueIsAdjusting` before loading a new image into the label.

This short program can indeed display any image from the list, assuming of course that the image exists and is of valid format (GIF or JPEG).[25]

■

Before coming up with additional examples using lists, we will introduce some of the text handling components of Swing, namely `JTextField`, `JTextArea`, and `JEditorPane`.

Swing Text Components

The text components in Swing are greatly improved over their AWT counterpart. As all sophisticated Swing elements they use a view/model split to allow for much greater flexibility. For example, Swing provides a text component that can easily represent formatted text. In fact, that component understands standard HTML 3.2 formatting tags and can display formatted text – even including images – accordingly.

Since all Swing text components extend `JTextComponent` we will first introduce that class:

Definition 6.3.12: The `JTextComponent` Class

This class is the base class for all Swing text components. It is, for the most part, code-compatible with `java.awt.TextComponent`*, but provides added features. The class is part of the* `javax.swing.text` *package and is hardly ever used directly. The Java API defines it as follows:*

```
public abstract class JTextComponent extends JComponent
                                     implements Scrollable, Accessible
{  // selected methods
   public void setText(String t)
   public String getText()
   public void cut()
   public void copy()
   public void paste()
   public void setEnabled(boolean b)
   public void setEditable(boolean b)
   public String getSelectedText()
   public void replaceSelection(String content)
   public int getSelectionStart()
   public int getSelectionEnd()
   public void select(int selectionStart, int selectionEnd)
   public void selectAll()
   public void setDocument(Document doc)
   public Document getDocument()
}
```

[25] A natual improvement of this program would automatically put all image files in a particular directory into the list of image names. That is possible as soon as the `JFileChooser` class is introduced.

The easiest text component based on `JTextComponent` is `JTextField`, which is very similar to `java.awt.TextField`:

Definition 6.3.13: The `JTextField` Class

This class 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. It is mostly code-compatible with `java.awt.TextField`*, but provides several enhancements. The Java API defines this class as follows:*

```
public class JTextField extends JTextComponent
                        implements SwingConstants
{  // selected constructors
   public JTextField()
   public JTextField(String text)
   public JTextField(int columns)
   public JTextField(String text, int columns)
   public JTextField(Document doc, String text, int columns)
   //  selected methods
   public void setHorizontalAlignment(int alignment)
   public void addActionListener(ActionListener l)
}
```

The horizontal alignment is set using the `JTextField` *constants* `LEFT` *(the default),* `CENTER`*, or* `RIGHT`*. The default model used to store the text is a* `PlainDocument`*. The class can generate an action event when the* `ENTER` *key is pressed inside the field.*

For sophisticated use of text field and other text components we should next discuss the `PlainDocument` model that stores the text displayed in the text field. However, that level of detail would go beyond the scope of this chapter so we will refer you to the Java API for additional details. Instead, let's create a somewhat useful program that uses lists and text fields together and illustrates a list's capabilities to store objects other than strings.

Example 6.3.14:

Create a simple program that stores names together with email addresses. The data should be stored in a type `Address` consisting of a `protected` name and an email address as well as a `toString` method. The program should allow the user to enter new email addresses, keep a list of addresses entered, allow the user to delete a selected address from the list, and view the addresses entered. Only the names should appear in the list of addresses, but the associated email address should appear when the user selects a name from the list.

This example will illustrate the basic use of Swing text fields, which is mostly code-compatible to the `TextField` class from the AWT, as well as the fact that a `JList` can provide access to data of type different from `String`. First, we need to create the `Address` class as outlined:

```
public class Address
{  protected String name;
   protected String email;
   public Address(String _name, String _email)
   {  name = _name; email = _email; }
   public String toString()
   {  return name; }
}
```

The main program will work as follows: we obviously need text fields to let the user enter information and a list to provide access to names already entered. As described in definition 6.3.8 and 6.3.9 we will also need a data model that is associated with the list to store data of type `Address`. Finally, we need buttons to add and delete addresses. That already gives us the basic outline for our class:

```
public class EmailListing extends JFrame
       implements ActionListener, ListSelectionListener
{  private DefaultListModel data = new DefaultListModel();
   private JList list = new JList(data);
   private JTextField name = new JTextField();
   private JTextField email = new JTextField();
   private JButton add = new JButton("Add");
   private JButton del = new JButton("Delete");
   private class WindowCloser extends WindowAdapter
   {  public void windowClosing(WindowEvent we)
      {  System.exit(0); }
   }
   public EmailListing()
   {  /* defines the basic layout of the GUI components */ }
   public void handleDelete()
   {  /* handles deletion of an address selected in the list */ }
   public void handleAdd()
   {  /* handles the addition of a new address */ }
   public void actionPerformed(ActionEvent ae)
   {  /* react to action events by calling appropriate handlers */ }
   public void valueChanged(ListSelectionEvent lse)
   { /* displays a 'full' address corresponding to the selected name */ }
   public static void main(String args[])
   {  EmailListing el = new EmailListing(); }
}
```

Next we have to implement the various methods. The constructor is the longest. We'll add the two text fields at the top of our frame, the list in the center, and the two buttons at the bottom. To make our layout look more interesting, we will use the `JPanelBox` class defined in example 6.2.18 to add a titled border to the various components (you could use `setBorder` if you do not want to use the `JPanelBox` class). Here is the constructor:

```
public EmailListing()
{  super("Email List");
   JPanel buttons = new JPanel(new FlowLayout());
   buttons.add(add);
   buttons.add(del);
   JPanel input = new JPanel(new GridLayout(2,2));
   input.add(new JLabel("Name:  ")); input.add(name);
   input.add(new JLabel("Email: ")); input.add(email);
   JScrollPane scrollList = new JScrollPane(list);
   getContentPane().add("North", new JPanelBox(input, "Address"));
   getContentPane().add("Center", new JPanelBox(scrollList, "List"));
   getContentPane().add("South", buttons);
   list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
   list.addListSelectionListener(this);
   add.addActionListener(this);
   del.addActionListener(this);
   email.addActionListener(this);
   addWindowListener(new WindowCloser());
   validate(); pack(); setVisible(true);
}
```

Note that we are adding scrolling capabilities to the list using a `JScrollPane`. We also activate the email text field so that it can generate action events when the `ENTER` key is pressed. Next, we need to implement the two "handling" methods. The `handleDelete` method checks whether an entry in the list is currently selected, and if so removes that entry from the model. Note that we must ask the *list* whether an entry is selected but delete that entry from the data *model*, not the list.

```
public void handleDelete()
{  if (list.getSelectedIndex() >= 0)
      data.remove(list.getSelectedIndex());
}
```

The `handleAdd` method first checks if there is any text in the `name` field. If so, it generates a new address and adds it to the data model. Since data is attached to list, the list will automatically update to display the additional entry. After adding the new address we erase the entries from the text fields and put the cursor back into the `name` field so that the user can quickly add another address.

```
public void handleAdd()
{  if (!name.equals(""))
   {  data.addElement(new Address(name.getText(), email.getText()));
      name.setText("");
      email.setText("");
      name.requestFocus();
   }
}
```

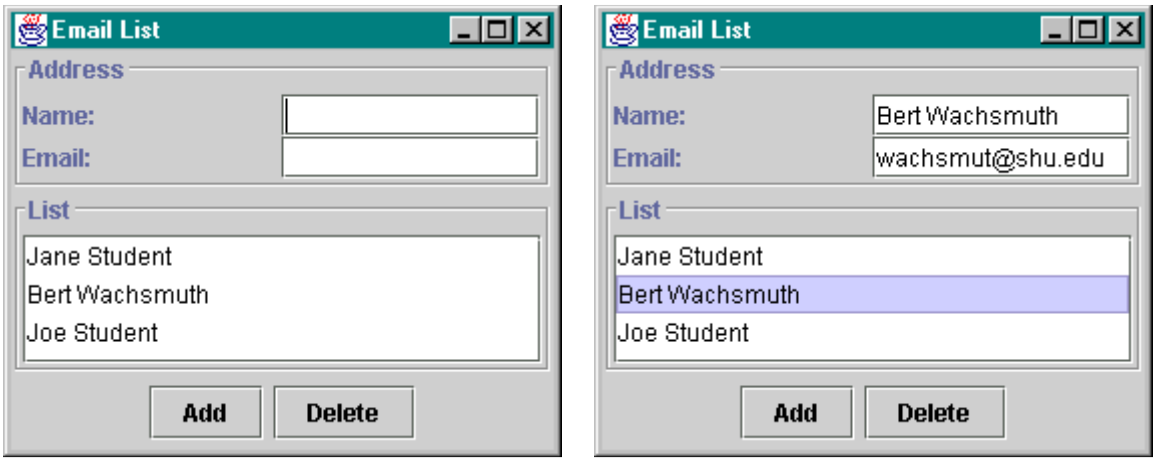
The `actionPerformed` method simply calls on these two handlers to perform the appropriate action: when the `del` button is pressed, call `handleDelete`, when the `add` button is pressed or when the user presses `ENTER` in the email field, call `handleAdd`:

```
public void actionPerformed(ActionEvent ae)
{  if (ae.getSource() == del)
      handleDelete();
   else if ((ae.getSource() == add) || (ae.getSource() == email))
      handleAdd();
}
```

The method `valueChanged` must be defined because our class implements `ListSelectionListener`. After the user has settled on a selection from the list, we get the `name` and `email` field of the corresponding address from the data model and display it in the text fields. We can retrieve a complete `Address` type from data but we must typecast it to ensure that the object really is of type `Address`.

```
public void valueChanged(ListSelectionEvent lse)
{  if (!lse.getValueIsAdjusting())
   {  Address address = (Address)data.get(list.getSelectedIndex());
      name.setText(address.name);
      email.setText(address.email);
   }
}
```

The `main` method, as outlined, simply creates an instance of `EmailListing` to get the program going. When everything is put together and the necessary classes are imported, the program will look as follows (don't forget to put the `Address` and `JPanelBox` classes in the same directory as this program):



`EmailListing` *with several addresses added* `EmailListing` *with existing address highlighted*

Figure 6.3.5: The `EmailListing` *class in action*

■

To make sure we also understand how multiple selections work in a `JList`, we will do one more example focusing on text fields and lists, this time using two lists (and hence two data models, one for each list).

Example 6.3.15:

Create a program to manage hiring and firing employees. You should be able to add names to a list of current employees. If you click on a name in that list, the name should move to a second list, containing people put "on probation". You should be able to fire people by selecting their names from the probation list, or restore them to regular employee status by moving them back to the employee list via buttons. The probation list should allow for multiple selections at a time. Separate the code responsible for the layout and the code for event handling into two different classes.

This time we are supposed to split our program into two classes. The first class we will call `EmployeeLists` and it will extend `JFrame`. It will focus on the layout of the various components. The second class called `EmployeeEventHandler` will contain all event-handling code.

The first class will get two lists with attached data models, one for a list of employees and the other for a list of people on probation. We also need a text field for the user to add new employee names. Finally there will be three buttons (`hire`, `fire`, and `restore`) to provide the user with the necessary action elements. The layout is simple, using again our `JPanelBox` class defined in definition 6.2.18. Here is the `EmployeeLists` class, including a standard `main` method to see the layout in action:

```
import java.awt.Dimension;
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.Container;
import javax.swing.*;

public class EmployeeLists extends JFrame
{  protected JButton hire = new JButton("Hire");
   protected JButton fire = new JButton("Fire");
   protected JButton restore = new JButton("Restore");
   protected JTextField name = new JTextField();
   protected DefaultListModel peopleData = new DefaultListModel();
```

```
    protected DefaultListModel toFireData = new DefaultListModel();
    protected JList peopleList = new JList(peopleData);
    protected JList toFireList = new JList(toFireData);
    public EmployeeLists()
    {  super("JList Test");
       JPanel buttons = new JPanel(new FlowLayout());
       buttons.add(hire); buttons.add(fire); buttons.add(restore);
       JPanel input = new JPanel(new BorderLayout());
       input.add("Center", new JPanelBox(name , "Employee Name"));
       input.add("East", new JPanelBox(buttons, "Action"));
       JScrollPane peoplePane = new JScrollPane(peopleList);
       JScrollPane toFirePane = new JScrollPane(toFireList);
       peoplePane.setPreferredSize(new Dimension(200,200));
       toFirePane.setPreferredSize(new Dimension(200,200));
       Container content = getContentPane();
       content.setLayout(new BorderLayout());
       content.add("North", input);
       content.add("East", new JPanelBox(toFirePane, "Probation List"));
       content.add("West", new JPanelBox(peoplePane, "Employees List"));
       peopleList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
       toFireList.setSelectionMode(
                       ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
       validate(); pack(); setVisible(true);
    }
    public static void main(String args[])
    {  EmployeeLists jlt = new EmployeeLists(); }
}
```

Note that we allow single selections only in the employee list, but multiple selections (the default) in the probation list. We have also marked all fields as protected rather than private so that the event handling class we are going to define next can easily access them. The class should compile as is, providing the following look:

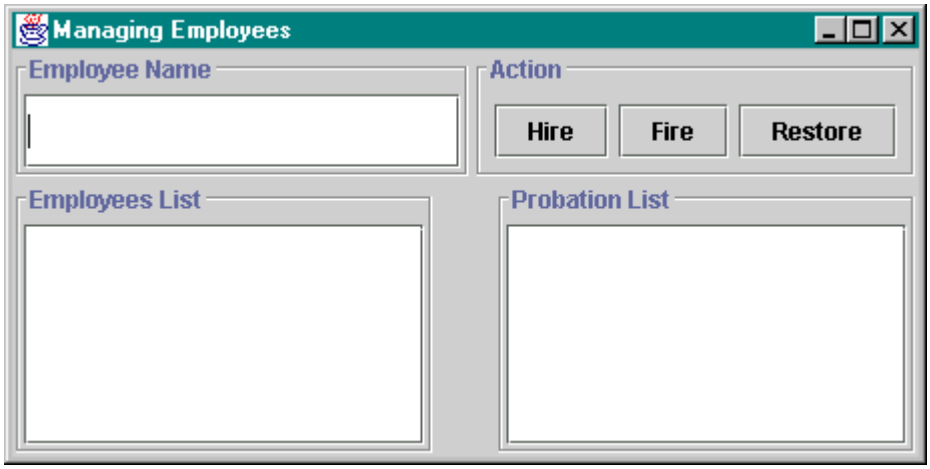


Figure 6.3.6: Hiring and Firing: handling multiple selections in a list

Of course the class will not actually do anything because no events are generated by any elements and no listeners are installed. Instead of finishing our class by adding the necessary code to that class, we establish a second class `EmployeeEventHandler` that will react to the user's request. In order for this event handler to have access to the protected fields of the `main` class, we use a reference field `master` that points to the `EmployeeLists` instance and is initialized through the constructor.

The `EmployeeEventHandler` class extends `WindowAdapter` so that it can properly close the program when the user clicks on the close box. It will implement `ActionListener` as well as `ListSelectionListener` to react to action events and list selection events. Therefore, we need to define as a minimum the methods `actionPerformed` and `valueChanged`.

The `actionPerformed` method delegates its task to other methods:

- `public void hire()`: checks if a name has been entered into the text field, adds it to the data for the employee list, removes the text from the input field and makes sure the input field receives the focus
- `public void fire()`: checks if any names have been selected in the probation list, then removes those names from the corresponding data model
- `public void restore()`: checks if any names have been selected in the probation list, then moves those names from the probation data model to the employee data model, restoring them as regular employees.

These methods are not hard, but we should explain how the `fire` method works since it deals with multiple selected values:

- the method is called if the user whishes to remove names from the probation list
- it defines a reference to the array of selected indices of the probation list
- if that array has anything in it, we start a loop to remove the corresponding entries from the data model

For example, if the user wants to remove entries 0, 3, 4, and 5 from the probation list, our event handler needs to remove the 0th, 3rd, 4th, and 5th entry in that corresponding data model. However, if we remove the 0th entry *first*, all entries above will move down by one, and what used to be the 3rd entry is now the 2nd. Now we have a problem trying to remove the original 3rd entry ! The solution is to use a reverse loop that removes first the 5th entry, then the 4th, and so on:

```
private void fire()
{  int indices[] = master.toFireList.getSelectedIndices();
   if (indices.length > 0)
      for (int i = indices.length-1; i >= 0; i--)
         master.toFireData.remove(indices[i]);
}
```

The `valueChanged` method moves a name from the employee list to the probation list by removing and adding it to the corresponding data models. It also ensures that the name just moved to the probation list is visible and selected. Here is the complete class:

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

public class EmployeeEventHandler extends WindowAdapter
       implements ActionListener, ListSelectionListener
{  private EmployeeLists master;
   public EmployeeEventHandler(EmployeeLists _master)
   {  master = _master; }
   private void hire()
   {  if (!master.name.getText().equals(""))
      {  master.peopleData.addElement(master.name.getText());
         master.name.setText("");
         master.name.requestFocus();
      }
   }
```

```
    private void fire()
    {  /* as defined above */ }
    private void restore()
    {   int indices[] = master.toFireList.getSelectedIndices();
        if (indices.length > 0)
        {  for (int i = indices.length-1; i >= 0; i--)
           {   master.peopleData.addElement(
                           master.toFireData.get(indices[i]));
               master.toFireData.remove(indices[i]);
           }
       }
    }
    public void actionPerformed(ActionEvent ae)
    {  if ((ae.getSource() == master.hire) ||
           (ae.getSource() == master.name))
          hire();
       else if (ae.getSource() == master.fire)
          fire();
       else if (ae.getSource() == master.restore)
          restore();
    }
    public void valueChanged(ListSelectionEvent lse)
    {  if (!lse.getValueIsAdjusting())
       {  master.toFireData.addElement(
                                 master.peopleList.getSelectedValue());
          master.peopleData.remove(master.peopleList.getSelectedIndex());
          master.toFireList.ensureIndexIsVisible(
                                       master.toFireData.getSize()-1);
          master.toFireList.setSelectedIndex(
                                       master.toFireData.getSize()-1);
       }
    }
    public void windowClosing(WindowEvent we)
    {  System.exit(0); }
}
```

The final step is to attach this event handler to the `EmployeeLists` class. To do that, we need to activate the buttons and the text field, add a list selection listener to the employee list, and add a window listener to the frame. We therefore add the following lines to the end of the constructor of `EmployeeLists`:

```
public EmployeeLists()
{  // all code as before, plus:
   EmployeeEventHandler handler = new EmployeeEventHandler(this);
   hire.addActionListener(handler);
   fire.addActionListener(handler);
   restore.addActionListener(handler);
   name.addActionListener(handler);
   peopleList.addListSelectionListener(handler);
   addWindowListener(handler);
}
```

■

The next text component is `JTextArea`, which allows for multi-line text. It is similar to its AWT counterpart, but does not support scrolling on its own. There are also some other differences so that this class is not code-compatible to `java.awt.TextField`.

Definition 6.3.16: The `JTextArea` Class

This class represents a multi-line area that displays editable or non-editable plain text. It is different from `java.awt.TextField` *in several respects. Most importantly:*

- *it does not provide automatic scrolling capabilities but can be embedded in a* `JScrollPane`
- *it does not add a* `TextListener` *to monitor changes to the text but uses a* `DocumentListener` *instead, which can be added to the* `Document` *returned by the inherited* `getDocument` *method*
- *it supports different document classes as data models (which we will not discuss), using the* `PlainText` *document model by default*
- *it allows automatic line-wrapping, controlled by* `setLineWrap` *and* `setWrapStyleWord` *(to wrap lines at word boundaries)*

The Java API defines this class as follows:

```
public class JTextArea extends JTextComponent
{  // selected constructors
   public JTextArea()
   public JTextArea(String text)
   public JTextArea(int rows, int columns)
   public JTextArea(String text, int rows, int columns)
   public JTextArea(Document doc)
   // selected methods
   public void insert(String str, int pos)
   public void append(String str)
   public void replaceRange(String str, int start, int end)
   public void setLineWrap(boolean wrap)
   public void setWrapStyleWord(boolean word)
}
```

Example 6.3.17:

One of the (few) deficiencies of the `JLabel` class is that it supports only one line of text (aside from icons). Use a `JTextArea` with scrollbars to define a multi-line, non-editable text field with automatic word-wrapping at word boundaries.

We will add a text area with 3 rows and 10 columns to a `JFrame`, set it to automatically wrap lines at word boundaries, and add some long text to it to see how it works. To support scrolling, we embed the field in a `JScrollPane`. Here is the code:

```
import javax.swing.*;

public class MultiLineLabelTest extends JFrame
{  private JTextArea label = new JTextArea(3, 10);
   public MultiLineLabelTest()
   {  super("MultiLineLabel Test");
      label.setLineWrap(true);
      label.setWrapStyleWord(true);
      label.setEditable(false);
      String s = " This text should be displayed using ";
      s += "multiple scrolling lines"
      label.setText(s);
      getContentPane().add("Center", new JScrollPane(label));
      validate(); pack(); setVisible(true);
   }
   public static void main(String args[])
   {  MultiLineLabelTest mllt = new MultiLineLabelTest(); }
```

```
}
```

When the program executes, you will see the scrollable text area:

Figure 6.3.7: `JTextArea` *with automatic scrolling enabled*

■

This example, while simple, should be sufficient to convert most `TextArea` fields to `JTextArea` ones. An added level of complexity is the use of a `DocumentListener` instead of a `TextListener`. As the definition mentioned, if you need to monitor changes that the user makes to the text represented by a `JTextArea`, you must attach a `DocumentListener` to the data model connected to the text area.

Definition 6.3.18: The `DocumentListener` and `DocumentEvent` Classes

The `DocumentListener` *is can be used for document change notifications and replaces* `java.awt.TextListener`. *It uses a* `DocumentEvent` *to represent changes to the text. The Java API defines these classes as follows:*

```
public abstract interface DocumentListener extends EventListener
{  // methods to be implemented
   public void insertUpdate(DocumentEvent e)
   public void removeUpdate(DocumentEvent e)
   public void changedUpdate(DocumentEvent e)
}
public abstract interface DocumentEvent
{  // selected methods
   public int getOffset()
   public int getLength()
}
```

Example 6.3.19:

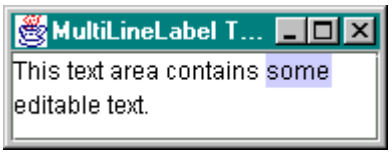
Modify example 6.3.17 to allow editing in the `JTextArea`. Add a `DocumentListener` to the text area and display all changes to the text to standard output. Add and remove some text and explain the output.

The code is similar to above, except that our class should now implement `DocumentListener`. Then we can attach `this` to the text field as document listener, provided that we implement the necessary methods. We add a private method to display the properties of the `DocumentEvent` to standard output:

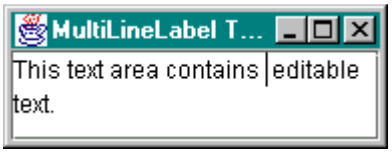
```
import javax.swing.*;
import javax.swing.event.*;

public class JTextAreaTest extends JFrame implements DocumentListener
{  private JTextArea label = new JTextArea(3, 10);
   public JTextAreaTest()
   {  super("MultiLineLabel Test");
      label.setLineWrap(true);
      label.setWrapStyleWord(true);
```

```
            label.setText("This text area contains some editable text.");
            getContentPane().add("Center", new JScrollPane(label));
            label.getDocument().addDocumentListener(this);
            validate(); pack(); setVisible(true);
        }
        public void changedUpdate(DocumentEvent e)
        {  System.out.println("Change Event: \n" + eventInfo(e)); }
        public void insertUpdate(DocumentEvent e)
        {  System.out.println("Insert Event: \n" + eventInfo(e)); }
        public void removeUpdate(DocumentEvent e)
        {  System.out.println("Remove Event: \n" + eventInfo(e)); }
        private String eventInfo(DocumentEvent e)
        {  return "Length: " +e.getLength()+ ", Offset: " +e.getOffset(); }
        public static void main(String args[])
        {  JTextAreaTest tat = new JTextAreaTest(); }
    }
```

When we run the program, we will mark the work `some`, remove it by hitting delete, and add the word `an` in its place. The following document events will be generated:

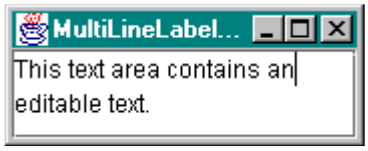


Marking text does not generate any document event.



Cutting text generates a `REMOVE` document event where `Length` is the number of characters removed and `Offset` is where the change started:

```
Remove Event: Length: 4, Offset: 24
```



Inserting the letters 'a' and 'n' generates two `INSERT` events of length 1, with offset indicating where the character is inserted:

```
Insert Event: Length: 1, Offset: 24
Insert Event: Length: 1, Offset: 25
```

Figure 6.3.8: Illustrating `DocumentEvent`

■

Our final text component class is somewhat similar to a `JTextArea` but with the extra ability to display formatted text, images, and tables.

Definition 6.3.20: The `JEditorPane` Class

This class represents a text component that can handle the following default types of text:

- `text/plain`: *plain text such as the one used for a* `JTextArea`
- `text/html`: *full support for the HTML 3.2 formatting language*
- `text/rtf`: *limited support for Rich Text Format*

The class provides methods to read text directly from a file or URL via the `setPage` *methods or constructors. It does not support automatic scrolling, but can be embedded in a* `JScrollPane`.

Perhaps the most useful – and surprisingly easy way – of using this class is to display non-editable HTML text. The Java API defines this class as follows:

```
public class JEditorPane extends JTextComponent
{  // selected constructors
   public JEditorPane()
   public JEditorPane(String pathToFile) throws IOException
   public JEditorPane(URL urlToFile) throws IOException
   public JEditorPane(String type, String text)
   // selected methods
   public void addHyperlinkListener(HyperlinkListener listener)
   public void setPage(String pathToFile) throws IOException
   public void setPage(URL urlToFile) throws IOException
   public final void setContentType(String type)
}
```

where `pathToFile` *is either a full* `URL` *(web address) of a document located on the web or, if the document is located in the same directory as the source code, the* `String`[26]*:*

```
"file:" + System.getProperty("user.dir")
        + System.getProperty("file.separator")
        + "actualFileName.html";
```

We can not explore the full capabilities of this class, but it is very convenient to show formatted, non-editable text inside a Java program. Probably the most useful content type is HTML, so we will only use this content type. Of course this means that you will need some familiarity with HTML formatting tags. Again we will not be able to explore all of the capabilities that HTML offers but restrict ourselves to a very small subset of HTML formatting commands:

Definition 6.3.21: The HTML Formatting Language

The HyperText Markup Language provides a flexible, platform-independent, and non-proprietary way to represent formatted documents including images, tables, and hyperlinks. A document formatted according to the HTML standard consists of plain text and HTML formatting tags. All formatting tags are enclosed in angular brackets and turn some formatting feature on or off. Tags that turn formatting off are prefaced by a slash. All HTML documents must start with the <HTML> tag and end with the </HTML> tag. Examples of other HTML formatting tags are[27]*:*

- `<B> … </B>`: *text inside the tags is bold*
- `<I> … </I>`: *text inside the tags is italics*
- `<H#>…</H#>`: *various levels of headlines, where # is a number from 1 to 6*
- `<UL> … </UL>`: *enclosed text represents an unsorted list*
- `<LI>… </LI>`: *a bulleted item, usually inside an unsorted list*
- `<IMG SRC='image.gif'>`: *display the specified image at this position, where image must be of GIF or JPEG format*
- `<A HREF='url'>…</A>`: *creates a hyperlink (anchored hyper-reference) to 'url'*
- `<P>`: *A paragraph break*

[26] The `pathToFile` must be a fully qualified `URL` (Uniform Resource Locator). Just using the file name will not work. We will later discuss how to construct URLs in detail.

[27] We will return to the HTML formatting language in chapter 10.2. and explain in some more detail what actually happens when trying to locate and load a web page formatted in HTML.

Example 6.3.22:

Create a plain text file using the above formatting tags. Save the file as `sample.html` and load it into your web browser.

We can create HTML documents using the same editor we use to create Java source code. Here is a sample file, as it is displayed in the editor:

```
<HTML>
<H1>A Sample Document</H1>
Using <I>HTML</I> formatting tags, text can be displayed in <B>bold</B> or
<I>italics</I> or even in <B><I>bold and italics</I></B> easily. In addition
there can be simple lists:
<UL>
<LI><B>Item 1</B> in our list</LI>
<UL>
<LI><B>Item 1.2</B> in a sublist within the first one</LI>
<LI><B>Item 1.2</B> in the same sublist</LI>
</UL>
<LI><B>Item 2</B> in the original list</LI>
</UL>
And finally, here is a small <IMG SRC="new.gif"> image inside our document.
</HTML>
```

When we save this document as `sample.html` and load it into a web browser such as Netscape (using the `File | Open Page` menu options), it will look similar to the following:

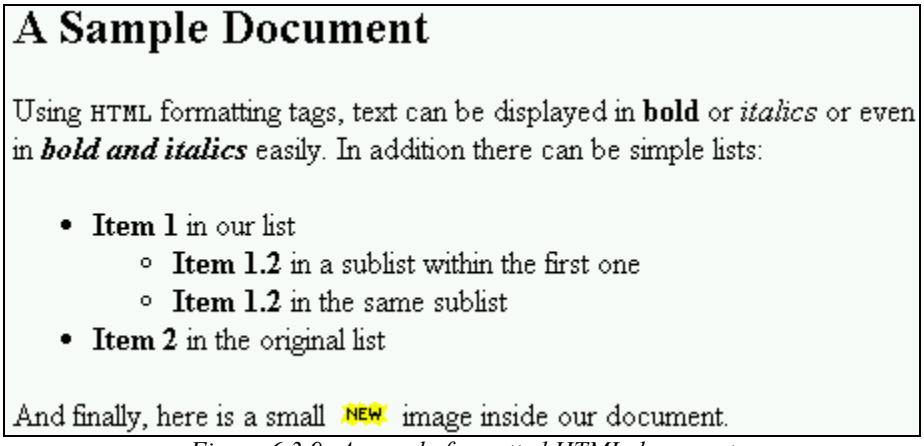


Figure 6.3.9: A sample formatted HTML document

Of course this will only display correctly if the image file `new.gif` is located in the same directory as the HTML source code.

■

Example 6.3.23:

Create a Java program that displays the above file `sample.html` in a frame.

The `JEditorPane` directly supports reading HTML documents, so all we have to do is ensure that the text content is of type `text/html`, make the component non-editable, read the file as defined above, and add it to a frame via a scroll pane. Here is the code:

```
import javax.swing.*;
```

```
import java.io.*;

public class HTMLReader extends JFrame
{  private JEditorPane editor = new JEditorPane();
   public HTMLReader()
   {  super("HTML Reader");
      editor.setContentType("text/html");
      editor.setEditable(false);
      String name = "file:" + System.getProperty("user.dir")
                            + System.getProperty("file.separator")
                            + "sample.html";
      try
      {  editor.setPage(name); }
      catch (IOException ioe)
      {  System.err.println("IO Exception: " + ioe); }
      getContentPane().add("Center", new JScrollPane(editor));
      validate(); pack(); setVisible(true);
   }
   public static void main(String args[])
   {  HTMLReader htmlreader = new HTMLReader(); }
}
```

The look of the resulting program is almost the same as in the previous figure so we do not need to repeat the screen shot.

■

Here is a more complete program, illustrating that a `JEditorPane` can be used to create HTML code on the fly, as well as react to clicks on hyperlinks embedded in an HTML document and read documents straight from the web. Before we can do that, we need to define the `HyperlinkListener` and associated `HyperlinkEvent` classes:

Definition 6.3.24: The `HyperlinkListener` and `HyperlinkEvent` Classes

A `HyperLinkListener` *is notified if the user clicks on or moves into or out of a hyperlink. A* `HyperlinkEvent` *represents information of what happened to a hyperlink. The Java API defines these classes as follows:*

```
public abstract interface HyperlinkListener extends EventListener
{  // methods
   public void hyperlinkUpdate(HyperlinkEvent e)
}
public class HyperlinkEvent extends EventObject
{  // selected methods
   public HyperlinkEvent.EventType getEventType()
   public URL getURL()
}
```

where the valid constants in `HyperLinkEvent.EventType` *are* `ACTIVATED`, `ENTERED`, *or* `EXITED`.

Example 6.3.25:

Create a fully functioning program that displays bookmarks of web addresses. Each bookmarked address should contain a title, a URL, a category, and a description. Your program should display all bookmarks by title, show the full description when the user

selects a particular title, and display the corresponding web page when the user clicks on the hyperlinked URL in the description.[28]

That seems to be a tall order. We basically need to create a simplified (much simplified …) version of a web browser. Fortunately, the `JEditorPane` is going to do all of the hard work so our program is simpler than it sounds. First we need to represent web bookmarks, including their title, URL, category, and description. We create a straightforward `Bookmark` class for that, including a few sample bookmarks to get us started.

```
public class Bookmark
{  protected String title, address, description, category;
   public Bookmark()
   {  title = address = description = category = ""; }
   public Bookmark(String _title, String _address)
   {  title = _title;
      address = _address;
      description = category = "";
   }
   public String toString()
   {  return title;  }
   public static Bookmark[] getSampleBookmarks()
   {  Bookmark url[] = new Bookmark[4];
      url[0] = new Bookmark("Seton Hall University",
                            "http://www.shu.edu/");
      url[0].description = "The Catholic University of New Jersey";
      url[0].category = "Universities";
      url[1] = new Bookmark("Dartmouth College",
                            "http://www.dartmouth.edu/");
      url[1].description = "An Ivy League College in New Hampshire";
      url[1].category = "Universities";
      url[2] = new Bookmark("Yahoo", "http://www.yahoo.com/");
      url[2].description = "The most popular search engine on the web";
      url[2].category = "Search Engines";
      url[3] = new Bookmark("Excite", "http://www.excite.edu/");
      url[3].description = "A pretty good alternative to Yahoo.";
      url[3].category = "Search Engines";
      return url;
   }
}
```

Next, we need to worry about the program we are supposed to create. It will extend `JFrame`, as usual and implement both `ItemChangeListener` and `HyperlinkListener`. That way it can react to changes in the list of bookmarks as well as to clicks on hyperlinks. Clearly our program needs a `JList` and an associated data model. It also needs a `JEditorPane` to display information formatted in HTML. And actually, that's already it. Here is the layout of our class with the implementation of some of methods following below:

```
public class BookmarkManager extends JFrame
             implements ListSelectionListener, HyperlinkListener
{  private DefaultListModel data = new DefaultListModel();
   private JList list = new JList(data);
   private JEditorPane info = new JEditorPane();
   private class WindowCloser extends WindowAdapter
   {  public void windowClosing(WindowEvent we)
```

[28] We could expand on this example by allowing for bookmark editing and displaying bookmarks in a tree-like structure using the category information for grouping. After finishing chapter 9, you could further expand the example to retrieve bookmarks from a file. After finishing chapter 10, you could again expand the example to allow others to save and retrieve bookmarks via client/server programs.

```
      {  System.exit(0); }
   }
   public BookmarkManager()
   {  super("Bookmark Manager");
      Bookmark samples[] = Bookmark.getSampleBookmarks();
      for (int i = 0; i < samples.length; i++)
         data.addElement(samples[i]);
      info.setEditable(false);
      JScrollPane scrollList = new JScrollPane(list);
      JScrollPane scrollInfo = new JScrollPane(info);
      scrollList.setPreferredSize(new Dimension(150, 200));
      scrollInfo.setPreferredSize(new Dimension(300, 200));
      JSplitPane split = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                              new JPanelBox(scrollList,"Bookmarks"),
                              new JPanelBox(scrollInfo, "Description"));
      getContentPane().add("Center", split);
      list.addListSelectionListener(this);
      info.addHyperlinkListener(this);
      addWindowListener(new WindowCloser());
      validate(); pack(); setVisible(true);
   }
   public void valueChanged(ListSelectionEvent lse)
   {  /* displays formatted information about selected bookmark */ }
   public void hyperlinkUpdate(HyperlinkEvent hle)
   {  /* loads selected URL into editor */ }
   private void showBookmark(int i)
   {  /* formats the information in a bookmark in HTML */ }
   public static void main(String args[])
   {  BookmarkManager bm = new BookmarkManager(); }
}
```

Note that we added a `ListSelectionListener` to the list and a `HyperlinkListener` to the editor pane. The `showBookmark` method will retrieve and typecast a `Bookmark` from the data model and put appropriate HTML formatting tags around it. Then it sets the editor pane to that formatted text. Depending on your formatting preferences, it might look similar to this:

```
private void showBookmark(int i)
{  Bookmark mark = (Bookmark)data.get(i);
   String s = "<HTML><H1>" + mark.title + "</H1><UL>";
   s += "<LI><B>Address: </B><A HREF='" + mark.address +"'>";
   s += mark.address + "</A></LI>";
   s += "<LI><B>Category: </B><I>" + mark.category + "</I></LI>";
   s += "<LI><B>Description: </B>"+mark.description+"</LI></UL></HTML>";
   info.setContentType("text/html");
   info.setText(s);
}
```

In particular, the last two lines set the content type of the editor pane to HTML and put the formatted string in the editor. The `valueChanged` method waits until the user has made a selection, retrieves the index of the selected list item, and uses the `showBookmark` method to display the formatted bookmark:

```
public void valueChanged(ListSelectionEvent lse)
{  if (!lse.getValueIsAdjusting())
      showBookmark(list.getSelectedIndex());
}
```

Finally, `hyperLinkUpdate` is called when a user clicks on a hyperlink in the editor. There is at least one hyperlink available, coming from the formatted version of our bookmark. When a user clicks that link, the appropriate web page should be loaded into the editor. While that is in principle tricky to

accomplish (a connection to a web server must be made, an HTML page must be loaded and formated, etc – see chapter 10 for details), the `JEditorPane` method `showPage` handles all details. Since that method throws an `IOException`, we must embed it in a standard `try-catch` block as follows:

```
public void hyperlinkUpdate(HyperlinkEvent hle)
{  if (hle.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
   {  try
      {  info.setPage(hle.getURL()); }
      catch(IOException ioe)
      {  info.setText("<B>" +hle.getURL() +"unloadable</B><P>" + ioe); }
   }
}
```

Note that we display the error message, if any, right in our editor. All that remains to do is put the pieces together, import the right classes – including the `java.io.IOException` class – and test the program. Before testing starts, though, make sure your computer is connected to the Internet. You should first dialup an Internet Service Provider, then start this class to see its full functionality[29] (and limitations). The program should initially display bookmarks as follows:

Figure 6.3.10: `BookmarkManager` *displaying a particular bookmark*

If you click on the hyperlink http://www.yahoo.com, and you are properly connected to the Internet, the program will locate Yahoo's web page right into the editor:

XYZZY

Figure 6.3.11: `BookmarkManager` *after clicking on the link for Yahoo's web page*

As a matter of fact, all Yahoo's hyperlinks also work, so you can click on any link and even perform searches right from our little Java program. Of course our program is not a complete web browser. For one thing, the standard "back" button is missing and pages comprised of "frames" may not work correctly. Also, many pages on the web will not display properly due to formatting limitations of the `JEditorPane` and incorrect HTML code in web pages. Still, for the effort invested this is a pretty neat little program.

If you clicked on the Yahoo link while not properly connected to the Internet, our program will display the following error message (it may take a few seconds):

[29] In chapter 10.1 we explain in detail what happens when your computer dials into the Internet via an Internet Service Provider.

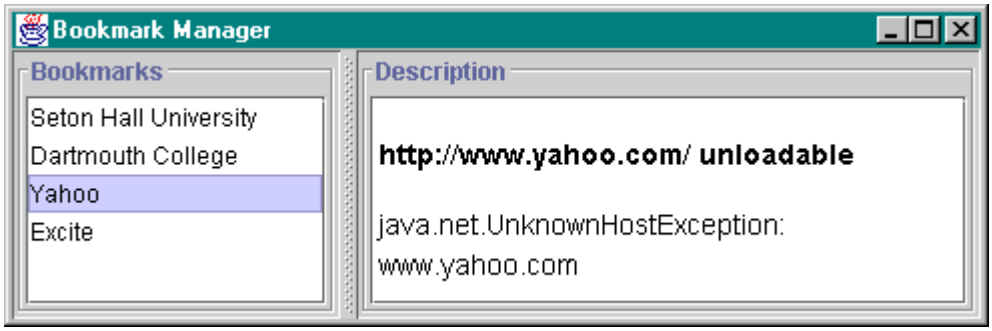


Figure 6.3.12: `BookmarkManager` *tried to connect unsuccessfully to Yahoo's web site*

■

Holding Swinging Dialogs

Dialogs are another area where Swing excels over the AWT. It provides a `JDialog` class that is, for the most part, code-compatible to `java.awt.Dialog`, but it also provides an easy-to-use `JOptionPane` class that allows you to quickly generate several standard types of dialogs, complete with standard icons and buttons. Therefore, we will cover this class first before mentioning the more traditional `JDialog` class.

Definition 6.3.26: The `JOptionPane` Class

This class provides several convenient static methods to pop up a "message", "input", "confirmation", or "options" dialog box, including appropriate icons and buttons. All dialog boxes are modal, i.e. they block input to their parent class. If the parent component is `null`*, the dialog appears in the middle of the screen, otherwise it appears centered inside its parent. The methods are (for the constants used, see table 6.3.13):*

```
public class JOptionPane extends JComponent implements Accessible
{  // selected methods - ALL METHODS ARE PUBLIC
   static void showMessageDialog(Component parent, Object message)
   static void showMessageDialog(Component parent, Object message,
                                 String title, int messageType)
   static String showInputDialog(Component parent, Object message)
   static String showInputDialog(Component parent, Object message,
                                 String title, int messageType)
   static int showConfirmDialog(Component parent, Object message)
   static int showConfirmDialog(Component parent, Object message,
                               String title, int optionType,
                               int messageType)
   static int showOptionDialog(Component parent, Object message,
                              String title, int optionType,
                              int messageType, Icon icon,
                              Object[] options, Object initialValue)
}
```

Constant	Name
messageType	`ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, PLAIN_MESSAGE`
optionType	`DEFAULT_OPTION, YES_NO_OPTION,`

	YES_NO_CANCEL_OPTION, OK_CANCEL_OPTION
return values for "Confirm" and "Option" dialogs	YES_OPTION, NO_OPTION, CANCEL_OPTION, OK_OPTION, CLOSED_OPTION

Table 6.3.13: Constants used in `JOptionPane`

Example 6.3.27:

Create a program with three buttons, one for each of the dialog types except `"Option"` dialog. Use drop-down combo boxes to let the user select the `messageType` and `optionType` and display a message corresponding to the button pressed and the option(s) selected. Test each of the dialog boxes.

We use three buttons, as requested, and two combo boxes. To initialize the combo boxes, we define arrays of strings corresponding to the text of the various options. We also define additional arrays of integers corresponding to the actual options. In the `actionPerformed` method, we check which string is currently selected in the combo boxes, then pick that entry from the integer array for that option and construct the appropriate dialog box. Here is the code, using once again the `JPanelBox` class from example 6.2.18:

```
import java.awt.FlowLayout;
import javax.swing.*;
import java.awt.event.*;

public class JOptionPaneTest extends JFrame implements ActionListener
{  private int MESSAGE_TYPE[] =
      {JOptionPane.ERROR_MESSAGE, JOptionPane.INFORMATION_MESSAGE,
       JOptionPane.WARNING_MESSAGE, JOptionPane.QUESTION_MESSAGE,
       JOptionPane.PLAIN_MESSAGE};
   private int OPTION_TYPE[] =
      {JOptionPane.DEFAULT_OPTION, JOptionPane.YES_NO_OPTION,
       JOptionPane.YES_NO_CANCEL_OPTION, JOptionPane.OK_CANCEL_OPTION };
   private String MESSAGE_STRING[] =
      {"ERROR_MESSAGE", "INFORMATION_MESSAGE", "WARNING_MESSAGE",
       "QUESTION_MESSAGE", "PLAIN_MESSAGE"};
   private String OPTION_STRING[] =
      {"DEFAULT_OPTION", "YES_NO_OPTION",
       "YES_NO_CANCEL_OPTION", "OK_CANCEL_OPTION"};
   private JButton message = new JButton("Message");
   private JButton input = new JButton("Simple Input");
   private JButton confirm = new JButton("Confirm");
   private JComboBox messageType = new JComboBox(MESSAGE_STRING);
   private JComboBox optionType  = new JComboBox(OPTION_STRING);
   public JOptionPaneTest()
   {  super("Standard Dialog Test");
      JPanelBox buttons = new JPanelBox(new FlowLayout(), "Types");
      buttons.add(message);  message.addActionListener(this);
      buttons.add(input);    input.addActionListener(this);
      buttons.add(confirm);  confirm.addActionListener(this);
      JPanelBox options = new JPanelBox(new FlowLayout(), "Options");
      options.add(new JLabel("Message Types:"));
      options.add(messageType);
      options.add(new JLabel("Option Types:"));
      options.add(optionType);
      getContentPane().add("North", buttons);
      getContentPane().add("South", options);
      validate(); pack(); setVisible(true);
   }
   public void actionPerformed(ActionEvent ae)
   { if (ae.getSource() == message)
```

```
            JOptionPane.showMessageDialog(this, "Test Message", "Message",
               MESSAGE_TYPE[messageType.getSelectedIndex()]);
         else if (ae.getSource() == input)
            JOptionPane.showInputDialog(this, "Test Input", "Input",
               MESSAGE_TYPE[messageType.getSelectedIndex()]);
         else if (ae.getSource() == confirm)
            JOptionPane.showConfirmDialog(this, "Test Confirm", "Confirm",
               OPTION_TYPE[optionType.getSelectedIndex()],
               MESSAGE_TYPE[messageType.getSelectedIndex()]);
      }
      public static void main(String args[])
      {  JOptionPaneTest jopt = new JOptionPaneTest(); }
   }
```

The program should execute fine and present the user with several buttons and combo box combinations of constants:

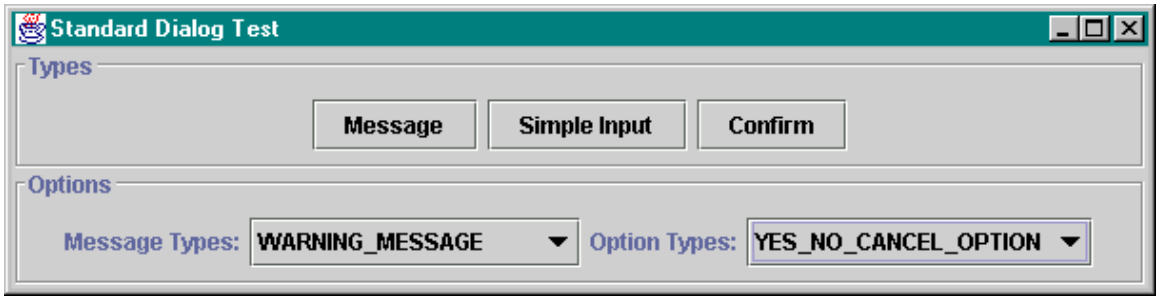


Figure 6.3.14: The `JOptionPaneTest` *program in action*

Here are some of the dialog boxes from the above program with various types and options chosen, in the Java look as well as in the Windows look:

Figure 6.3.15: Java Look of various types of message boxes

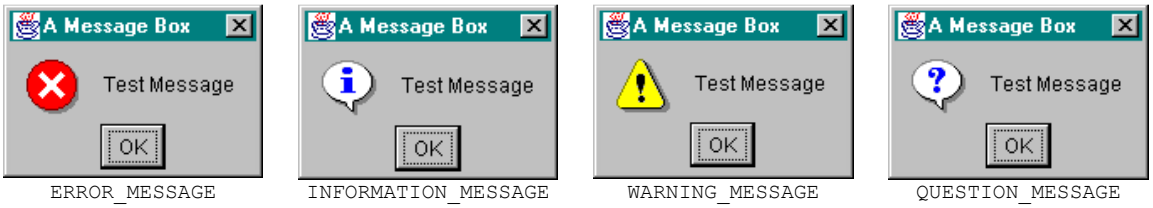


Figure 6.3.16: Windows Look of various types of message boxes

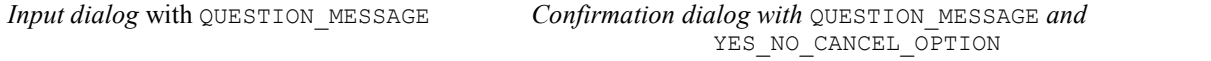


Figure 6.3.17: Java Look of input and confirmation dialogs with various options

■

The general options dialog box is somewhat more complicated to use, but also more flexible. It can probably replace many modal instances of using `JDialog` by using an appropriate instance of a `JPanel` as `message`. You do get ease of use as well as automatic icons and positioning so there are many reasons to use `JOptionPane` over `JDialog`. Here is a simple example using the general options dialog.

Example 6.3.28:

Create a program that brings up a general dialog box containing two text fields for user input. If the user clicks `OK` and has entered text in both fields, show the input in a label. If the user clicks `OK` but one or both text fields contain no text, display a suitable error message. If the user closes the dialog any other way, do nothing.

We will extend `JFrame` and add an active button and a status label. We also define a `JPanel` containing the text fields for our options dialog, and use that panel as input to the dialog. When the user clicks `OK` we can query that panel for the values the user entered and act accordingly. For convenience, we define that panel as a separate class:

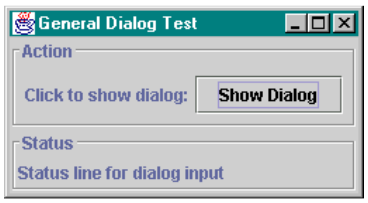
```
import java.awt.GridLayout;
import javax.swing.*;

public class OptionsPanel extends JPanel
{  protected static String BUTTONS[] = {"Sure, why not",
                                        "I don't think so"};
   private JTextField name  = new JTextField();
   private JTextField email = new JTextField();
   public OptionsPanel()
   {  setLayout(new GridLayout(2,2));
      add(new JLabel("Name:"));  add(name);
      add(new JLabel("Email:")); add(email);
   }
   public String getName()
   {  return name.getText(); }
   public String getEmail()
   {  return email.getText(); }
}
```

In addition to the text fields, we added some labels to the layout. More importantly, we defined a protected static array of strings representing the text we want to see on the buttons of the dialog when it appears. With this class defined we can create our main program. Its main action is in the `actionPerformed` method. If the `show` button is clicked, we first create an instance of an `OptionsPanel` as defined above. Then we use that panel as well as the static array of strings `BUTTONS` as input to the `showOptionDialog` method. When that modal dialog is dismissed, we check which button was pressed and act accordingly. Here is the code:

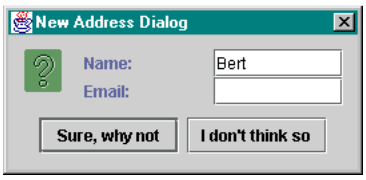
```
import java.awt.FlowLayout;
import java.awt.event.*;
import javax.swing.*;

public class GeneralDialogTest extends JFrame implements ActionListener
{  private JButton show = new JButton("Show Dialog");
   private JLabel  status = new JLabel("Status line for dialog input");
```

```
    public GeneralDialogTest()
    {  super("General Dialog Test");
       JPanel buttons = new JPanel(new FlowLayout());
       buttons.add(new JLabel("Click to show dialog:"));
       buttons.add(show); show.addActionListener(this);
       getContentPane().add("North", new JPanelBox(buttons, "Action"));
       getContentPane().add("South", new JPanelBox(status, "Status"));
       validate(); pack(); setVisible(true);
    }
    public void actionPerformed(ActionEvent ae)
    {  if (ae.getSource() == show)
       {  OptionsPanel options = new OptionsPanel();
          int resp = JOptionPane.showOptionDialog(this, options,
                          "New Address Dialog", JOptionPane.YES_NO_OPTION,
                          JOptionPane.QUESTION_MESSAGE, null,
                          OptionsPanel.BUTTONS, OptionsPanel.BUTTONS[0]);
          if (resp == JOptionPane.YES_OPTION)
          {  if (!(options.getName().equals("")) &&
                 !(options.getEmail().equals("")))
             status.setText("Name: " + options.getName() +
                            ", Email: " + options.getEmail());
             else
                JOptionPane.showMessageDialog(this,
                          "You must enter a value in both text fields",
                          "Input Error", JOptionPane.ERROR_MESSAGE);
          }
       }
    }
    public static void main(String args[])
    {  GeneralDialogTest gdt = new GeneralDialogTest(); }
}
```

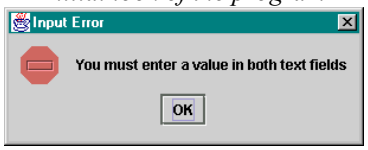
This class will compile and execute fine, resulting in the following behavior.



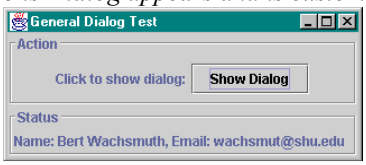
Initial look of the program



Options Dialog appears and is customized



Error message if one text field is empty



Status changed if name and email fields were entered

Figure 6.3.18: Various stages of the `GeneralDialogTest` *program*

■

Finally there is the `JDialog` class, a general-purpose dialog class similar to `java.awt.Dialog`. In fact, since `JDialog` is so similar to its AWT counterpart, we do not need to spend much time on this class.

Definition 6.3.29: The `JDialog` Class

This class allows modal (blocking) or non-modal (not blocking) dialog windows that are usually attached to other frames or dialogs and whose behavior is coupled to that of its parent. The class is code-compatible to `java.awt.Dialog`*, which it extends. The only difference is that components and layout managers must be added to the content pane obtained via* `getContentPane` *instead of directly to the dialog. Dialogs are non-modal unless specified otherwise. The Java API defines this class as follows:*

```
public class JDialog extends Dialog
       implements WindowConstants, Accessible, RootPaneContainer
{  // constructors
   public JDialog()
   public JDialog(Frame parent)
   public JDialog(Frame parent, boolean modal)
   public JDialog(Frame parent, String title)
   public JDialog(Frame parent, String title, boolean modal)
   public JDialog(Dialog parent)
   public JDialog(Dialog parent, boolean modal)
   public JDialog(Dialog parent, String title)
   public JDialog(Dialog parent, String title, boolean modal)
   // selected methods
   public void setJMenuBar(JMenuBar menu)
   public Container getContentPane()
}
```

Since the `JDialog` *class has a constructor requiring no input, it can be instantiated from an applet, different from* `java.awt.Dialog`.

Since `JDialog` extends `Dialog` its behavior is so similar to its AWT counterpart that we can refer to the *Dialog* class, section 4.6, for examples. There are two more dialog classes in Swing that we will not explain in details:

- The `JFileChooser` class replaces `java.awt.FileDialog` and allows selection of files for reading and writing. Since we will not discuss file handling until chapter 9, we will not discuss this class here.
- The `JColorChooser` class allows convenient selection of colors. We have no need for this class in any of the examples in this text, so we will not discuss it here.

Both classes are pretty straightforward and if you need of these classes for a particular program you should have little trouble consulting the Java API for details and using them accordingly.

6.4. Trees, Tables, and Graphics

In the previous section we have seen the model/view approach applied to lists and text components. While certainly different from the AWT, these elements do have an equivalent, albeit less functional, AWT counterpart. Next we will introduce components that do not have any equivalent AWT structure, the `JTree` and `JTable` components. We will also discuss custom graphics in detail, so that we lean how to replace the `java.awt.Canvas` class.

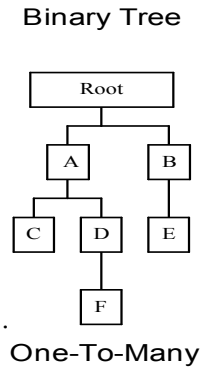
At this point we have introduced enough Swing components so that you should be able to convert most AWT-based programs to Swing unless they use custom graphics. Therefore, the remaining sections are purely optional. However, to *effectively* use Swing, we do recommend that you read on, especially since some of the examples in this and the next section will include more substantial programs.

The `JTree` and `JTable` classes are quite complex and we will only discuss them briefly. We will leave out the part that allows direct editing of these components within their graphical representation and use them only as display and selection elements for simplicity. For additional details, you can always refer to the Java API.

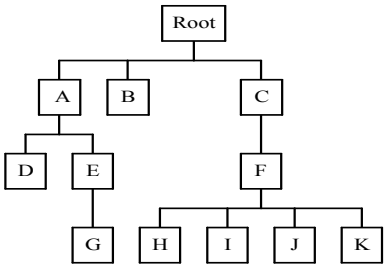
Trees

We will discuss the `JTree` class first. It is a class that can represent data that is hierarchically organized, different from lists, which represent sequentially organized data. Hierarchical data, or trees, come in many different flavors. Generally, all trees consist of nodes, and one node is singled out as the root node. Each node can have children, and in general one child could have several parents. The root node is characterized as the node without a parent. Hierarchical structures come in different flavors:

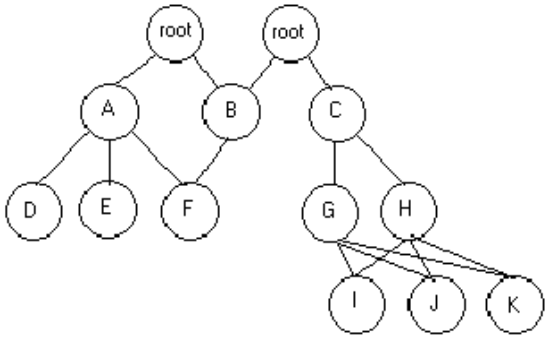
One-to-Two relationship: Each node in a tree can have at most two children and all children have exactly one parent. Such a tree is also called binary tree



One-to-Many relationship: Each node in a tree can have none to several children and all children have exactly one parent.



Many-to-Many relationship: Each node can have multiple children and children can have multiple parents. This structure could also have more than one root node.



The `JTree` class represents one-to-many trees and allows the user to select nodes as well as expand and collapse branches of the tree.

Definition 6.4.1: The `JTree` Class

This class can display data that is hierarchically organized in a tree-like structure. A `JTree` *has one root node and each node can have any number of children, but each child (except the root) has exactly one parent. Nodes without children are called leafs, the only node without parents is the root. All nodes to follow starting from the root to reach a particular node in the tree are called the "path to that node". The class works in conjunction with several other classes, in particular* `TreePath` *and* `TreeNode`*. Selections of nodes in a tree are detected by a* `TreeSelectionListener`*. The class does not support scrolling but can be embedded in a* `JScrollPane`*. The Java API defines* `JTree` *as follows:*

```
public class JTree extends JComponent
       implements Scrollable, Accessible
{  // selected constructor
   public JTree(TreeNode root)
   // selected methods
   public void addTreeSelectionListener(TreeSelectionListener tsl)
   public void expandPath(TreePath path)
   public void collapsePath(TreePath path)
   public boolean isExpanded(TreePath path)
   public boolean isCollapsed(TreePath path)
   public void scrollPathToVisible(TreePath path)
   public void setSelectionPath(TreePath path)
   public TreePath getSelectionPath()
   public boolean isPathSelected(TreePath path)
   public void clearSelection()
}
```

Before we can use `JTree` we need to know how to create a tree node. Java provides a `TreeNode` interface as well as a `DefaultMutableTreeNode` class that is ready-to-use and sufficient for most applications.

Definition 6.4.2: The `DefaultMutableTreeNode` Class

Swing provides the `DefaultMutableTreeNode` *class to represent a changeable node in a tree that could be a root node, a parent node, or a leaf node. The class is part of the* `javax.swing.tree` *package, and the Java API defines it as follows:*

```
public class DefaultMutableTreeNode extends Object
       implements Cloneable, MutableTreeNode, Serializable
{  // selected constructor
   public DefaultMutableTreeNode(Object userObject)
   // selected method
   public void add(MutableTreeNode newChild)
   public void remove(int childIndex)
   public TreeNode getParent()
   public TreeNode getChildAt(int index)
   public int getChildCount()
   public TreeNode[] getPath()
   public TreeNode getRoot()
   public boolean isRoot()
   public boolean isLeaf()
   public Enumeration breadthFirstEnumeration()
   public Enumeration depthFirstEnumeration()
}
```

Finally, we need to know something about how the various nodes of a tree are connected. Java again has a ready-made class for that called `TreePath`.

Definition 6.4.3: The `TreePath` Class

The `TreePath` *class represents a path to a node. The class is part of the* `javax.swing.tree` *package, and the Java API defines it as follows:*

```
public class TreePath extends Object implements Serializable
{  // selected constructor
   public TreePath(Object[] path)
   // selected methods
   public Object[] getPath()
   public int getPathCount()
   public Object getPathComponent(int element)
   public TreePath getParentPath()
}
```

Definition 6.4.2 mentions an `Enumeration`, which we will discuss in detail in chapter 8. For now, the next example will illustrate how to use an enumeration to traverse all nodes in a tree without providing an exact definition of `Enumeration` at this time. The `Enumeration` class is part of the `java.util` package and must be imported if used.

Example 6.4.4:

Create a tree representing a hierarchical structure of strings representing web sites. The web sites should be organized by category, and all sites within one category should be leafs of the tree with parent being their category. Specifically, use two categories "Universities" and "Search Engines", and the web addresses "www.shu.edu", "www.dartmouth.edu", "www.yahoo.com", and "www.excite.com". The tree root should be a node labeled "Web Sites". Provide buttons to expand and collapse the entire tree.

First, let's review the structure we want to create. There is one parent node, two categories that are children of that parent, and each category in turn has two children that are leafs. A rough representation of this structure would look as follows:

```
Web Sites
   Universities
      www.shu.edu
      www.dartmouth.edu
   Search Engines
      www.yahoo.com
      www.excite.com
```

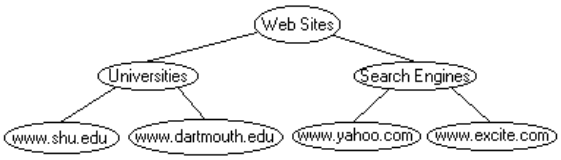


Figure 6.4.1: Tree structure of web sites

Therefore, we need to create a `JTree` with seven nodes and make sure they represent this particular structure. We also put a scroll pane around the tree to enable scrolling and add two buttons to expand and collapse the entire tree. To represent our tree, we define its root node as a field to have a convenient starting point to reach all nodes. The children are added within a method `createTreeStructure`, as follows:

```
import java.util.Enumeration;[30]
import java.awt.FlowLayout;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;

public class SimpleTree extends JFrame implements ActionListener
{  private DefaultMutableTreeNode root =
                        new DefaultMutableTreeNode("Web Sites");
   private JTree tree = new JTree(root);
   private JButton expand   = new JButton("Expand");
   private JButton collapse = new JButton("Collapse");
   private JButton quit     = new JButton("Quit");
   private SimpleTree()
   {  super("A Simple Tree");
      createTreeStructure();
      JPanel buttons = new JPanel(new FlowLayout());
      buttons.add(collapse);  collapse.addActionListener(this);
      buttons.add(expand);    expand.addActionListener(this);
      buttons.add(quit);      quit.addActionListener(this);
      getContentPane().add("Center", new JScrollPane(tree));
      getContentPane().add("South", buttons);
      validate(); pack(); setVisible(true);
   }
   private void createTreeStructure()
   {  DefaultMutableTreeNode edu =
                     new DefaultMutableTreeNode("Universities");
      DefaultMutableTreeNode com =
                     new DefaultMutableTreeNode("Search Engines");
      root.add(edu);
      root.add(com);
      edu.add(new DefaultMutableTreeNode("www.shu.edu"));
      edu.add(new DefaultMutableTreeNode("www.dartmouth.edu"));
      com.add(new DefaultMutableTreeNode("www.yahoo.com"));
      com.add(new DefaultMutableTreeNode("www.excite.com"));
   }
   private void collapseTree()
   {  /* collapses all branches of the tree */   }
   private void expandTree()
   {  /* expands all branches of the tree */ }
   public void actionPerformed(ActionEvent ae)
   {  if (ae.getSource() == expand)
         expandTree();
      else if (ae.getSource() == collapse)
```

[30] The `Enumeration` class needs to be imported from the `java.util` package for the `collapseTree` and `expandTree` methods.

```
            collapseTree();
         else if (ae.getSource() == quit)
            System.exit(0);
      }
      public static void main(String args[])
      {  SimpleTree st = new SimpleTree(); }
   }
```

This class will create a representation of our web site structure as shown in figure 6.4.2. Initially, only the root node will be visible but you can double-click on it to expand it, and again double-click on its children to expand those until you reach the leaf nodes. You can also double-click on an expanded node with children to collapse it. All details are handled entirely by the `JTree` class and the various nodes it contains.

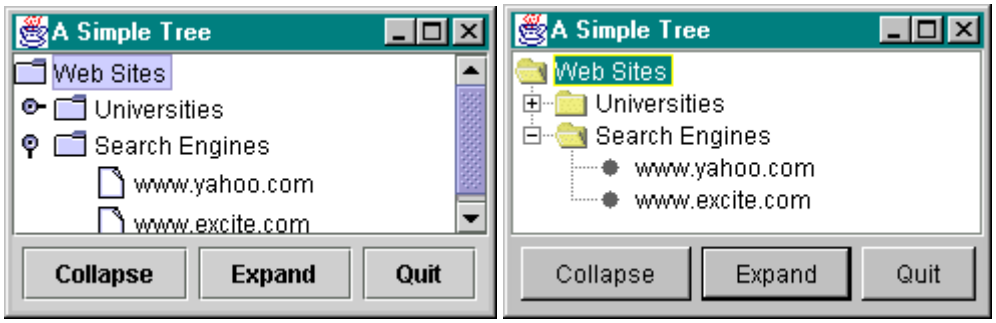


Figure 6.4.2: The Tree representation of `SimpleTree` *in Java (left) and Window (right) look and feel*

Of course we still have to provide the code to expand and collapse the entire tree by clicking the appropriate buttons. We will use the `breadthFirstEnumeration` and `depthFirstEnumeration` methods for that. Both methods return an `Enumeration`, which returns a sequential structure with methods `hasMoreElements` and `nextElement` that can be used in a `for` loop. The details of an `Enumeration` will be explained in chapter 8. For now, here is the code that will expand all tree nodes:

```
public void expandTree()
{  for(Enumeration nodes = root.breadthFirstEnumeration();
       nodes.hasMoreElements(); )
   {  DefaultMutableTreeNode node =
          (DefaultMutableTreeNode)nodes.nextElement();
      tree.expandPath(new TreePath(node.getPath()));
   }
}
```

It works as follows:

- the initializer in the `for` loop defines a sequential structure `nodes` of type `Enumeration` of all nodes starting from the `root`, in a breadth-first order. That means that first all children of the root are listed, then all children of those children, and so on.
- the loop uses the sequential structure `nodes` and its method `hasMoreElements` to define the condition that terminates the loop
- the modification part of the for loop is left empty (!) and is instead provided inside the loop by the method `nextElement`
- the `nextElement` method returns the next element of the sequential structure, typecasts it into an appropriate tree `node`, and then expands the path leading to that `node`
- to find the `TreePath` leading to a particular node, we ask the `node` to return its path, then use the array of objects returned to construct a new `TreePath` as input to the `expandPath` method

The `collapseTree` method can be created accordingly, but it will use a depth-first enumeration to collapse the inner-most branches of the tree first:

```
public void collapseTree()
{  for(Enumeration nodes = root.depthFirstEnumeration();
       nodes.hasMoreElements(); )
   {  DefaultMutableTreeNode node =
             (DefaultMutableTreeNode)nodes.nextElement();
      tree.collapsePath(new TreePath(node.getPath()));
   }
}
```

When these methods are added to our class, the example is complete. Now selective branches can be expanded and collapsed by double clicking on the corresponding nodes in the tree and the entire tree can be completely collapsed or expanded using the appropriate buttons.

■

If we want to monitor which nodes in a tree are currently selected, we can add a `TreeSelectionListener` to the tree.

Definition 6.4.5: The `TreeSelectionListener` and `TreeSelectionEvent` Classes

The `TreeSelectionListener` *is notified when a selection in a* `JTree` *changes. The* `TreeSelectionEvent` *represents the current selection in a tree.*

```
public abstract interface TreeSelectionListener extends EventListener
{  // method
   public void valueChanged(TreeSelectionEvent e)
}
public class TreeSelectionEvent extends EventObject
{  // selected method
   public TreePath getPath()
}
```

Example 6.4.6:

Add a status line to the above `SimpleTree` example that will contain the full path of the node that is currently selected by the user.

Now we need to track the selections made by the user so we attach a `TreeSelectionListener` to our tree, which works similar to adding a `ListSelectionListener` to a list:

- we make sure our class implements `TreeSelectionListener`
- we use `this` as input to the `addTreeSelectionListener` method of the tree
- we implement the required method `valueChanged`
- we use the `getPath` method of the `TreeSelectionEvent` to inquire which element is currently selected
- `getPath` returns a `TreePath`, and a further `getPath` method invoked on that tree path returns an array of objects leading to our node, so we display the values of this array in a `for` loop to get the full path to the selected node

Thus, we need to change the `SimpleExample` class as follows, with the changes displayed in bold and italics.

```
// import all classes as before and add:
import javax.swing.event.*;

public class SimpleTreeWithListener extends JFrame
       implements ActionListener, TreeSelectionListener
{  // as before, with one added field for the status line
   private JLabel status = new JLabel("Tree selection status");
   private SimpleTreeWithListener()
   {  // as before, with the following lines added
      getContentPane().add("North", status);
      tree.addTreeSelectionListener(this);
      validate(); pack(); setVisible(true);
   }
   private void createTreeStructure()
   {  /* no changes */ }
   private void collapseTree()
   {  /* no changes */   }
   private void expandTree()
   {  /* no changes */   }
   public void actionPerformed(ActionEvent ae)
   {  /* no changes */   }
   public void valueChanged(TreeSelectionEvent tse)
   {  Object path[] = tse.getPath().getPath();
      String s = path[0].toString();
      for (int i = 1; i < path.length; i++)
         s += " - " + path[i].toString();
      status.setText(s);
   }
   public static void main(String args[])
   { SimpleTreeWithListener st = new SimpleTreeWithListener(); }
}
```

Here is the result of this new class, where the full path to the currently selected node (root, parent, or leaf) is displayed in the label at the top.

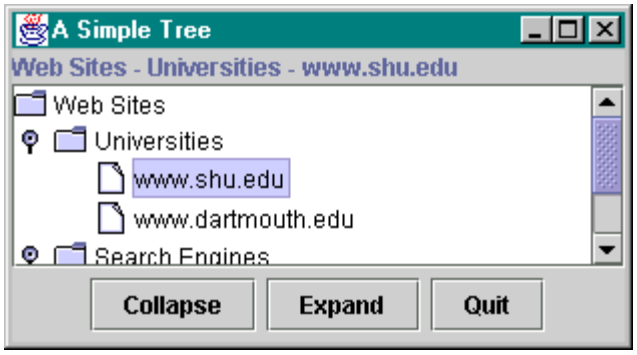


Figure 6.4.3: `TreeWithListener` *class, showing the path to selected node*

■

Tables

Our last Swing class to cover in at least some detail is `JTable`. As the name implies, it provides a table-like representation of data, similar to the look of a spreadsheet program like Microsoft Excel. Just as `JTree`, this class is quite complex so we will only discuss how to use it for relatively simple

examples. `JTable` is another example of a model/view design and we need to define the class as well as the appropriate model before we can do any examples.

Definition 6.4.7: The `JTable` Class

This class represents a two-dimensional view on data in table form. The column width can be determined interactively or through appropriate methods. The `JTable` *class implements a model/view split design and represents the view on an appropriate data model. Individual cells in the table are identified by integers, indicating the row and column of a cell. The class does not support scrolling but can be embedded in a* `JScrollPane`*. The Java API defines this class as follows:*

```
public class JTable extends JComponent
       implements TableModelListener, Scrollable,
                  TableColumnModelListener, ListSelectionListener,
                  CellEditorListener, Accessible
{  // selected constants
   public static int AUTO_RESIZE_OFF, AUTO_RESIZE_ALL_COLUMNS,
                     AUTO_RESIZE_NEXT_COLUMN, AUTO_RESIZE_LAST_COLUMN,
                     AUTO_RESIZE_SUBSEQUENT_COLUMNS;
   // selected constructor
   public JTable(TableModel tm)
   // selected methods
   public int getSelectedRow()
   public int getSelectedColumn()
   public void setAutoResizeMode(int mode)
   public void setRowSelectionAllowed(boolean flag)
   public void setColumnSelectionAllowed(boolean flag)
   public void setCellSelectionEnabled(boolean flag)
   public void selectAll()
   public void clearSelection()
}
```

As usual for model/view splits, we need to define a data model that can be used with this class. Swing provides a `DefaultTableModel` class, but it is recommended to extend that class, or `AbstractTableModel`, instead of using it directly (see definition 6.4.10). For our purposes it is best to extend `DefaultTableModel` to make sure we can control whether cells can be edited or not.

Definition 6.4.8: The `DefaultTableModel` Class

This class can store data suitable for representation in a `JTable`*. It stores data dynamically, using only as much memory as necessary at any given time.* [31] *The Java API defines this class as follows*[32]*:*

```
public class DefaultTableModel extends AbstractTableModel
       implements Serializable
{  // selected constructor
   public DefaultTableModel()
   public void addColumn(Object columnName)
   public void addColumn(Object columnName, Object[] columnData)
```

[31] We will explore dynamic data structures in detail in chapter 8. The `DefaultTableModel` actually uses a `Vector` of `Vectors` to store its data.

[32] There is no method to add a column at a specified index because the `JTable` view allows for interactive rearrangements of columns.

```
   public void addRow(Object[] rowData)
   public void insertRow(int row, Object[] rowData)
   public void removeRow(int row)
   public int getRowCount()
   public int getColumnCount()
   public String getColumnName(int column)
   public boolean isCellEditable(int row, int column)
   public Object getValueAt(int row, int column)
   public void setValueAt(Object aValue, int row, int column)
}
```

The `DefaultTableModel` *allows for data editing by default, which can be disabled by overriding* `isCellEditable` *to return* `false`.

Example 6.4.9:

Use a `JTable` class to create an invoice-like program. Each row on the invoice should consist of an ID (`int`), a Description (`String`), the unit price (`double`), the quantity (`int`), and the total price (`double`). The program should enter all data for now, but there should be a separate field where the program will display the computed total amount. You do not have to worry about formatting the doubles to correctly represent currency values (i.e. two digits after the period, see "*Formatting Decimals*", section 1.5).

We need to create a `JTable`, tie it to a particular data model, and add some sample data in the specified format to the model. Looking through our methods for adding data to a `DefaultTableModel` we see that we can only add arrays of `Object`. Therefore, we need to use wrapper classes to convert `int` and `double` types to their corresponding wrapper objects `Integer` and `Double`. To see how our program works in principle, here is its layout:

```
import java.awt.Dimension;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;

public class Invoicer extends JFrame
{  private DefaultTableModel data = new DefaultTableModel();
   private JTable table = new JTable(data);
   private JLabel price = new JLabel("$0.0", SwingConstants.RIGHT);
   private class WindowCloser extends WindowAdapter
   {  public void windowClosing(WindowEvent we)
      {  System.exit(0); }
   }
   public Invoicer()
   {  super("Simple Table");
      createDataColumns();
      createSampleData();
      JScrollPane scrollTable = new JScrollPane(table);
      price.setText("$" + total());
      scrollTable.setPreferredSize(new Dimension(400,100));
      getContentPane().add("Center", new JPanelBox(scrollTable, "Data"));
      getContentPane().add("South", new JPanelBox(price, "Total Cost"));
      addWindowListener(new WindowCloser());
      validate(); pack(); setVisible(true);
   }
   private void createDataColumns()
   { /* create columns with appropriate headers */ }
   private void createSampleData()
   { /* create some sample data */
   private Object[] dataRow(int id, String desc, double price, int num)
```

```
   {  /* utility method to convert basic types into array of Objects */ }
   private double total()
   { /* computes the total in the last column of the table */ }
   public static void main(String args[])
   { Invoicer st = new Invoicer(); }
}
```

The method `createDataColumns` is the easiest to understand. It simply adds five columns to our data model with appropriate headers:

```
private void createDataColumns()
{  data.addColumn("ID");
   data.addColumn("Description");
   data.addColumn("Unit Price");
   data.addColumn("Quantity");
   data.addColumn("Price");
}
```

The `createSampleData` uses the `addRow` method to add some sample data. Since that method requires an array of objects, it uses in turn the `dataRow` method to create an array of objects from some sample data. Here are both of these methods:

```
private void createSampleData()
{  data.addRow(dataRow(19, "IBM 770 Laptop", 4329.99, 1));
   data.addRow(dataRow(207, "External ZIP drive", 99.95, 1));
   data.addRow(dataRow(1008, "SDRAM Memory Chips, 64MB", 104.99, 4));
   data.addRow(dataRow(44, "IBM 21' Triniton Monitor", 699.95, 1));
   data.addRow(dataRow(105, "External Keyboard, black", 59.99, 1));
   data.addRow(dataRow(207, "External Mouse, black", 29.99, 1));
   data.addRow(dataRow(45, "IBM Port Replicator", 145.99, 1));
}
private Object[] dataRow(int id, String desc, double price, int num)
{  Object row[] = new Object[5];
   row[0] = new Integer(id);
   row[1] = desc;
   row[2] = new Double(price);
   row[3] = new Integer(num);
   row[4] = new Double(price * num);
   return row;
}
```

The last method to implement is the `total` method. It loops through all rows, retrieving the value in the last column (containing the `price`). Since the `getValueAt` method returns an object, we first typecast it into a `Double`, then retrieve its `double` value and add it to a running sum:

```
private double total()
{  double sum = 0.0;
   for (int row = 0; row < data.getRowCount(); row++)
      sum += ((Double)data.getValueAt(row, 4)).doubleValue();
   return sum;
}
```

Now all methods are implemented, and the program should run. It will create an "invoice" similar to the following figure:

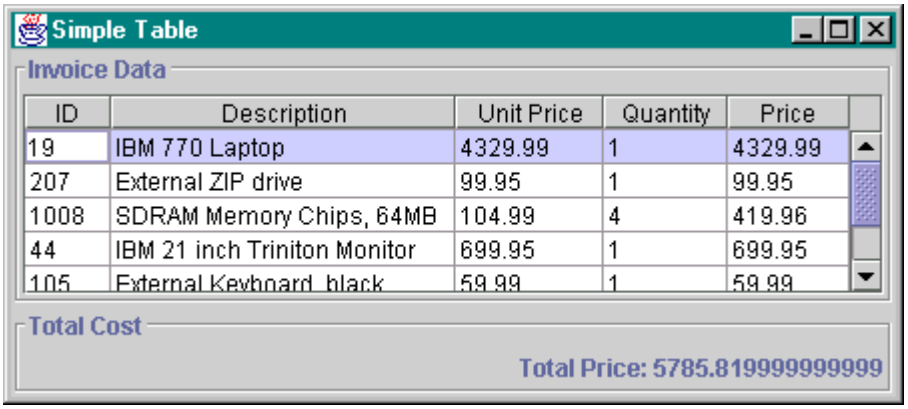


Figure 6.4.4: The `Invoicer` *program in action*

To be sure, when you first start the program all columns will have equal width. The width of a column can be adjusted interactively by dragging the column boundaries. In fact, entire columns can be switched with other columns by dragging them around. Try it yourself.

■

Actually, there is a slight problem with our class: the `DefaultTableModel` lets you edit the cells by double-clicking on them (again, try it) ! At first glance that might sound very convenient and we could try to add some listener to tell us when a cell has been edited so that we can adjust the total if necessary. However, the default editor will convert edited objects into strings, and then our invoice will not work any more because the object containing the price will automatically convert from its original `Double` type to `String`. We will fix that error in example 6.4.11 according to our next rule of thumb.

Definition 6.4.10: Rule of Thumb for Editing Table Cells

A `JTable` *together with a* `DefaultTableModel` *allows editing of all cells by default. Edited cells are converted to* `String`*, so that editing will only work correctly if the original data was already of type* `String`*.*

To disable editing, create a new class that extends `DefaultTableModel` *and overwrite the* `public boolean isCellEditable(int row, int col)` *method to always return* `false`*.*[33]

Example 6.4.11:

In example 6.4.9 we used a `DefaultTableModel`, hence the table cells in that `Invoicer` program were editable. Substitute your own model for the default model so that all cells become non-editable.

First we need to create our own model extending `DefaultTableModel`. The only method we will overwrite is `isCellEditable`, so the new model is easy:

```
import javax.swing.table.*;

public class NoEditTableModel extends DefaultTableModel
{  public boolean isCellEditable(int row, int col)
```

[33] To be safe, you should *always* extend `DefaultTableModel`, disable editing, and use that class as data model for your tables, unless you exclusively deal with String objects.

```
    {  return false; }
}
```

That's the entire class. All we have to do in the `Invoicer` program is replace the line defining the data model. Instead of:

```
private DefaultTableModel data = new DefaultTableModel();
```

we substitute our own model:

```
private NoEditTableModel data = new NoEditTableModel();
```

After recompiling the `Invoicer` class, everything will work as before and all cells are now non-editable (make sure to try it).

■

We will conclude this section with a more elaborate version of the `Invoicer` program:

Example 6.4.12:

Create a second version of our `Invoicer` program that contains options to remove a selected row, to add a row, and to edit a row. All editing should take place in appropriate dialogs. Make sure the total price is recomputed if necessary.

We want to accomplish this modification with as little change to our original class as possible. Therefore, we will try to move most of our new code into other classes. An easy way to achieve that is to attach a popup menu to our table containing the options to add, modify, and delete rows, and to use a separate class to define and react to the popup menu selections. As described in definition 6.2.21, to bring up a popup menu we need to detect a right-click via a mouse listener. Therefore the change to our existing `Invoicer` class will consist of adding a mouse listener implemented in a separate class. In order for the new class to have access to the fields of `Invoicer` we also mark its fields as `protected`. The new class will be defined as:

```
public class InvoicePopup extends MouseAdapter implements ActionListener
{ /* brings up a popup menu and reacts to selections from that menu */ }
```

The change in the `Invoicer` class will be:

- we change the accessibility of the fields `data`, `price`, and `table` from `private` to `protected`
- we change the method `total` from `private` to `protected`
- we install a mouse listener to the table via the line

  ```
  table.addMouseListener(new InvoicePopup(this));
  ```

- we also add a tooltip to the table reminding the user to click the right mouse button to choose an option, via the line

  ```
  table.setToolTipText("Right-Click for menu");
  ```

Both lines of code should be at the end of the `Invoicer` constructor.

Now we need to think about the `InvoicerPopup` class. Since we add that class as a mouse listener to a table, the class must extend `MouseAdapter` (or implement `MouseListener`). It also needs to react to

menu selections so that it must implement `ActionListener`. One of the jobs of this class is to bring up a dialog to enter new data or modify existing invoice items. We will use the `showOptionDialog` method for that and hence we need a panel of input fields that we can pass to that method. Therefore we will first create a class `InvoicePanel` that determines a mask for data entry and modification.

`InvoicePanel` should contain input fields for the items of an invoice, but some of those items are numbers instead of strings. A `JTextField` would not work for those types, so we first create a `JNumberField` class that can return an `int` or `double`, as requested (compare example 5.1.16). To do that we start with a `JTextField` and create methods that convert strings to `int` or `double`, respectively. Since that may or may not work depending on the user input, those methods will throw a potential `NumberFormatException`. Here is that class:

```
import javax.swing.*;

public class JNumberField extends JTextField
{  public int getInteger() throws NumberFormatException
   {  return Integer.valueOf(getText()).intValue();   }
   public double getDouble() throws NumberFormatException
   {  return Double.valueOf(getText()).doubleValue(); }
}
```

With that class in place, we can define our `InvoicePanel` to consist of text and number fields, appropriately arranged, as well as methods to conveniently set and get the values of those fields:

```
import java.awt.BorderLayout;
import java.awt.GridLayout;
import javax.swing.*;

public class InvoicePanel extends JPanel
{  protected JNumberField id    = new JNumberField();
   protected JTextField   desc  = new JTextField();
   protected JNumberField price = new JNumberField();
   protected JNumberField num   = new JNumberField();
   public InvoicePanel()
   {  JPanel labels = new JPanel(new GridLayout(4,1));
      labels.add(new JLabel(" ID "));
      labels.add(new JLabel(" Description "));
      labels.add(new JLabel(" Price "));
      labels.add(new JLabel(" Num Units"));
      JPanel inputs = new JPanel(new GridLayout(4,1));
      inputs.add(id);      inputs.add(desc);
      inputs.add(price);   inputs.add(num);
      setLayout(new BorderLayout());
      add("West", labels); add("Center", inputs);
   }
   public Object[] getData() throws NumberFormatException
   {  Object data[] = new Object[5];
      data[0] = new Integer(id.getInteger());
      data[1] = desc.getText();
      data[2] = new Double(price.getDouble());
      data[3] = new Integer(num.getInteger());
      data[4] = new Double(price.getDouble() * num.getInteger());
      return data;
   }
   public void setData(String[] data)
   {  id.setText(data[0]);
      desc.setText(data[1]);
      price.setText(data[2]);
      num.setText(data[3]);
   }
```

```
}
```

Now we are ready to create our `InvoicePop` method, extending `WindowAdapter` and implementing `ActionListener` as mentioned before. The class will receive as input a reference to `Invoice` so that it can use the non-private fields and methods of that class. It also gets a reference to the table so that the popup menu can be placed correctly over that component. Aside from constructing and showing the menu, the class will react to the menu choices using three methods `add`, `delete`, and `modify`, as follows:

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

public class InvoicePopup extends MouseAdapter implements ActionListener
{  private JPopupMenu menu = new JPopupMenu("Table Options");
   private JMenuItem add = new JMenuItem("Add Item");
   private JMenuItem del = new JMenuItem("Delete Item");
   private JMenuItem mod = new JMenuItem("Modify Item");
   private Invoicer parent = null;
   public InvoicePopup(Invoicer _parent)
   {  parent = _parent;
      menu.add(add);  add.addActionListener(this);
      menu.add(mod);  mod.addActionListener(this);
      menu.add(new JPopupMenu.Separator());
      menu.add(del);  del.addActionListener(this);
   }
   public void mousePressed(MouseEvent me)
   {  if (SwingUtilities.isRightMouseButton(me))
         menu.show(parent.table, me.getX(), me.getY());
   }
   public void actionPerformed(ActionEvent ae)
   {  if (ae.getSource() == add)
         add();
      else if (ae.getSource() == del)
         delete();
      else if (ae.getSource() == mod)
         modify();
   }
   private void delete()
   { /* deletes the currently selected row from the invoice table */ }
   private void add()
   { /* brings up dialog box to add a new Incoice item to the table */ }
   private void modify()
   { /* brings up dialog box with selected Invoice item for editing */ }
}
```

The `delete` method is the easiest of the three. It finds out which, if any, row is currently selected in the table, retrieves the description of that invoice item, and brings up a dialog box to confirm deletion. If the user clicks OK, the selected row is removed from the data model (which in turn will update the table automatically).

```
public void delete()
{  int row = parent.table.getSelectedRow();
   if (row >= 0)
   {  String msg = "Delete '" + parent.data.getValueAt(row, 1) + "'?";
      if (JOptionPane.showConfirmDialog(parent, msg, "Confirm Deletion",
            JOptionPane.OK_CANCEL_OPTION, JOptionPane.WARNING_MESSAGE)
                  == JOptionPane.YES_OPTION)
      {  parent.data.removeRow(row);
         parent.price.setText("$" + parent.total());
      }
```

```
   }
}
```

The `add` method is more complicated in theory: it should bring up a dialog box where the user can enter new data, check the types of the data entered, and insert that data into a new row of the invoice. But fortunately `InvoicePanel` handles all details, so the actual `add` method turns out to be not hard at all.

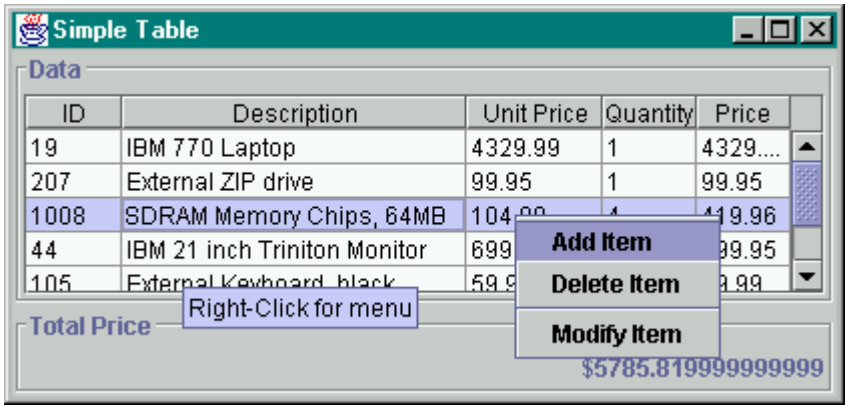
```
public void add()
{  InvoicePanel invoice = new InvoicePanel();
   if (JOptionPane.showOptionDialog(parent, invoice, "New Invoice Item",
             JOptionPane.OK_CANCEL_OPTION, JOptionPane.QUESTION_MESSAGE,
             null, null, null) == JOptionPane.YES_OPTION)
   {  try
      {  parent.data.addRow(invoice.getData());
         parent.price.setText("$" + parent.total());
      }
      catch(NumberFormatException nfe)
      {  JOptionPane.showMessageDialog(parent, "Invalid numeric format",
                "Error Message", JOptionPane.ERROR_MESSAGE);
      }
   }
}
```

The `getData` method from the `InvoicePanel` will throw an exception if the user has entered incorrect numerical values, so we catch that exception and notify the user via a message dialog that data could not be added.

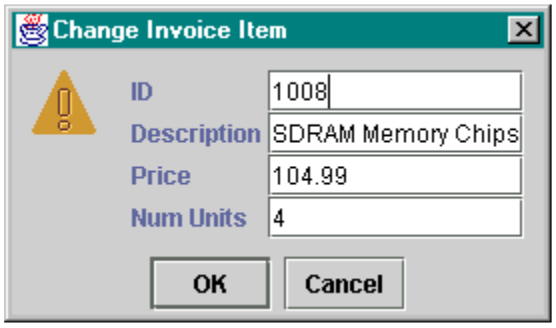
The final method `modify` is the most complicated, but again with the help that `InvoicePanel` provides it will not be so bad. First we check which row, if any, is selected in the table. Then we retrieve the data from that row and use the `setData` method to insert it into an `InvoicePanel`. After that we bring up an options dialog with that panel, which will present the data to the user for editing. When the user clicks OK the modified data will replace the one in the current row if possible. If there is an input error, we again use a message box to inform the user that data could not be modified.

```
public void modify()
{  InvoicePanel invoice = new InvoicePanel();
   int row = parent.table.getSelectedRow();
   if (row >= 0)
   {  String data[] = new String[4];
      for (int i = 0; i < data.length; i++)
         data[i] = parent.data.getValueAt(row, i).toString();
      invoice.setData(data);
      if (JOptionPane.showOptionDialog(parent, invoice, "Change Item",
               JOptionPane.OK_CANCEL_OPTION, JOptionPane.WARNING_MESSAGE,
               null, null, null) == JOptionPane.YES_OPTION)
      {  try
         {  for (int i = 0; i < invoice.getData().length; i++)
               parent.data.setValueAt(invoice.getData()[i], row, i);
            parent.price.setText("$" + parent.total());
         }
         catch(NumberFormatException nfe)
        {  JOptionPane.showMessageDialog(parent, "Invalid number",
                     "Error Message", JOptionPane.ERROR_MESSAGE);
         }
      }
   }
}
```

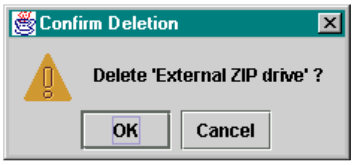
Note that we use three null parameters in `showOptionDialog`. That has the effect of using the default icon specified for a `WARNING_MESSAGE` and the buttons specified by `OK_CANCEL_OPTION`. Here is the improved version of the `Invoicer` in action.



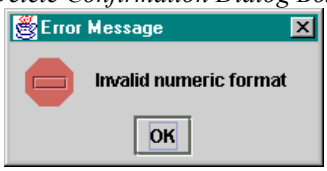
New Invoicer program with ToolTip and Popup Menu activated



Modification Dialog



Delete Confirmation Dialog Box



Error Message Dialog Box

Figure 6.4.5: Various dialogs for the new `Invoicer` program

One minor but annoying problem with our program is that if the user makes an error entering a number, the data entry dialog disappears before the error message appears. That does not give the user a convenient way to simply fix the mistake, all data must be re-entered instead. It is possible to remove the 'auto-closing' feature from the dialog boxes that appear via a `showOptionDialog`, but it involves concepts we have not defined. Please see the Java API for details. Alternatively, we could use an instance of a `JDialog` whose behavior can be controlled completely and easily (but we would not get automatic icons).

■

Graphing and Painting in Swing

In chapter 4 we introduced the `java.awt.Canvas` class whose sole purpose was to contain graphics or images. To use it, we had to extend the class, overwrite the `paint` method, and put all drawing routines in that method. Swing does *not* provide a comparable class, so graphics should be contained in another class. Actually, we have already seen that the `JLabel` class can easily accommodate

images. It can in fact also contain graphics just as easily, but another class is usually better for that purpose: the `JPanel` class.

Definition 6.4.13: Painting in the `JPanel` Class

Since there is no class in Swing comparable to `java.awt.Canvas`*, a class that needs to create custom graphics should extend* `JPanel`*. The extended class should overwrite* `public void paintComponent(Graphics g)` *and place any custom graphics code in that method. The first line in that method should call the* `paintComponent` *of the super class. The method* `paint` *that is also inherited by a* `JPane` *should not be overwritten or changed. Using* `JPanel` *instead of* `Canvas` *provides several immediate benefits:*

- `JPanel` *is a "lightweight" Swing component and can overlap with other components*
- `JPanel` *automatically paints its components using an efficient technique called "double buffering" (compare definition 6.5.12)*
- `JPanel` *supports scrolling by using* `JScrollPanel`
- `JPanel` *can contain standard borders (as usual for Swing components)*

The class also contains a `repaint` *method without arguments that eventually calls* `paintComponent` *with the appropriate* `Graphics` *object as input (compare to section 4.7) to redo all graphics.*

Example 6.4.14:

Create a `JPanel` class that draws some graphics shapes in some colors as well as a `String` in some font. Add a border around the panel, define a tool tip, and attach a popup menu with a single choice that calls `System.exit`. Then add the panel to a frame using `JScrollPane` and test the program.

Of course we need two classes, one extending `JPanel` and one extending `JFrame`. The `JPanel` class will contain all drawing code in its `paintComponent` method, as defined above. The code to define the tooltip and the menu goes in the constructor. The class must implement `ActionListener` to react to the choice of the popup menu and it needs a mouse listener that we implement as an inner class to bring up the popup menu on a right mouse click. The trick to enable proper scrolling is to overwrite the method `getPreferredSize` to specify how large our drawing *wants* to be. Only with that method defined correctly can the `JScrollPanel` defined in the frame know how to handle scrolling. Here is the drawing class (try to guess the image that will be drawn):

```
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Font;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class GraphicsPanel extends JPanel implements ActionListener
{  private JPopupMenu menu = new JPopupMenu("Menu");
   private JMenuItem  exit = new JMenuItem("Exit");
   private class Trigger extends MouseAdapter
   {  public void mousePressed(MouseEvent me)
      {  if (SwingUtilities.isRightMouseButton(me))
            menu.show(GraphicsPanel.this, me.getX(), me.getY());
```

```
      }
   }
   public GraphicsPanel()
   {  menu.add(exit);
      exit.addActionListener(this);
      setBackground(Color.white);
      setToolTipText("Right-click to exit");
      addMouseListener(new Trigger());
   }
   public Dimension getPreferredSize()
   {  return new Dimension(270,270); }
   public void paintComponent(Graphics g)
   {  super.paintComponent(g);
      Font font = new Font("Helvetica", Font.BOLD + Font.ITALIC, 14);
      g.setColor(Color.yellow);   g.fillOval(10, 10, 210, 210);
      g.setColor(Color.black);    g.fillArc(40, 40, 150, 150, 180, 180);
      g.setColor(Color.yellow);   g.fillArc(40, 20, 150, 130, 180, 180);
      g.setColor(Color.black);
      g.fillOval(50, 70, 30, 30); g.fillOval(150, 70, 30, 30);
      g.setFont(font); g.drawString("Welcome to Swing", 60, 250);
   }
   public void actionPerformed(ActionEvent ae)
   {  System.exit(0); }
}
```

The frame class is quite simple, since all the action happens in the drawing class. The only thing that our class really does is to add an instance of our `GraphicsPanel`, endowed with a border via our `JPanelBox` class (see example 6.2.18) as its center component. We will set the size of the panel manually to something slightly smaller than its preferred size to illustrate the scrollbars in action:

```
import javax.swing.*;

public class GraphicsTest extends JFrame
{  public GraphicsTest()
   {  super("Graphics Test");
      GraphicsPanel canvas = new GraphicsPanel();
      JScrollPane scrollCanvas = new JScrollPane(canvas);
      JPanelBox borderedCanvas = new JPanelBox(scrollCanvas, "Graphics");
      getContentPane().add("Center", borderedCanvas);
      setSize(200, 200); validate(); setVisible(true);
   }
   public static void main(String args[])
   {  GraphicsTest gt = new GraphicsTest(); }
}
```

Here is the image resulting from executing this class:

Figure 6.4.6: Image resulting from `GraphicsTest` *class*

■

Since Swing class have more functionality that AWT classes, it will quite often not be necessary to use these techniques. Images, for example, can easily be represented in a `JLabel` and positioned with a layout manager. The `JTable` and `JTree` classes provide capabilities that would have to be simulated using a `Canvas` in the AWT. Graphic buttons are easily created by adding an image to the `JButton` class. However, occasionally graphics *are* necessary, and the simple example above should suffice for most situations. To be sure, we will, in the next chapter, explore some more options about handling graphic efficiently, and in chapter 6.6 we will present the Mandelbrot example, making extensive use of drawing for which Swing does not provide any build-in alternatives. Still, the above example will form the foundation for all these situations.

6.5. Images and Sounds

In this chapter we will explore in some more detail the intricacies of loading images. We will show how to load several images and control the loading process, explain a simple animation program, and introduce Swing's sound handling capabilities. We will also explain how to load images and sounds into applets, for which we need to overcome a security restriction build into Java. This section, as well as section 6.6, is entirely optional.

Loading and Filtering Images

We will start our discussion with managing the loading process of more than one image. Of course we have already seen examples of how images can easily be loaded and displayed in a `JLabel`, but there are some techniques that will be useful when trying to work with multiple images, such as during animation.

As mentioned, Java provides extensive support for images in GIF or JPEG format. The simplest way to display an image is to enter it into a `JLabel`, using the file name of the image as input parameter. There will be several situations, however, where it is useful to load and store an image without necessarily displaying it. We may need it for processing before displaying it, or we may want to collect several images to create an animation, starting the animation process only when all images are loaded. Or, as a simple example, we may want to display the status of loading the image,

particularly when loading it over a network connection. The very first thing we need, therefore, is a class that can store a representation of an image.

Definition 6.5.1: The `Image` Class

The `Image` *class is an abstract class in the* `java.awt` *package that can be used to represent images. Since the class is abstract, it can not be used to instantiate any objects; instead, the image must be obtained using appropriate methods. The* `Image` *class is typically used to either store a representation of a* `GIF` *or* `JPEG` *graphics file or to provide access to an off-screen* `Graphics` *object into which Java can draw using its basic drawing methods. Images can be associated with an* `ImageObserver` *that provides access to the loading status of an image. The Java API defines* `Image` *as follows:*

```
public abstract class Image extends Object
{   // selected Methods
   public abstract int getWidth(ImageObserver observer)
   public abstract int getHeight(ImageObserver observer)
   public abstract ImageProducer getSource()
   public abstract Graphics getGraphics()
}
```

Since this class is an abstract class, we need to explore ways to load an image first before we can provide any examples. The loading of the image is actually system-dependent. However, Java provides a special class that forms a bridge between platform-independent Java classes and methods and their system-dependent implementations.

Definition 6.5.2: The `Toolkit` Class

The `Toolkit` *class is an abstract class that provides system-dependent implementations of the platform-independent classes in the* `java.awt` *package. Most methods in the* `Toolkit` *class should not be called directly but a few methods provide convenient access to system-dependent mechanisms and parameters:*

- `public String[] getFontList()`: *The names of the available fonts*
- `public Image getImage(String filename)`*: Loads image from the specified file*
- `public Image getImage(URL url)`*: Loads image from the specified URL*
- `public Dimension getScreenSize()`*: Gets the size of the screen in pixels*

To access to the `Toolkit` *methods, every* `Component` *provides the method* `public Toolkit getToolkit()`*, which returns a* `Toolkit` *instance.*

The `getImage` *methods return immediately and will start the loading process only when the image is actually rendered.*

In particular, the `Toolkit` class provides access to the overloaded methods `getImage`, which in turn return a reference to an `Image` in a valid format. That image can than be used as input to a `JLabel` for display.

Example 6.5.3:

Create a program that loads the image file `cat.jpg`. After issuing a `Toolkit` call to `getImage`, display the width and height of the image. Again display these parameters after displaying the image in a `JLabel`. Compare the output.

Of course our program will only work if there *is* a file `cat.jpg`. We will assume that this file exists in the same directory as the rest of our class. The simple way to load this image would be to use

```
ImageIcon image = new ImageIcon("cat.jpg");
JLabel displayImage = new JLabel(image);
```

A somewhat more elaborate version of these two lines would first load the image into an `Image` class instead of an `ImageIcon`, then use that as input for a `JLabel` as follows:

```
import java.awt.Image;
import javax.swing.*;
import java.io.*;

public class SimpleImageTest extends JFrame
{  private Image image = null;
   private JLabel display = new JLabel();
   public SimpleImageTest()
   {  super("Image Test");
      image = getToolkit().getImage("cat.jpg");
      System.out.println("Size: " + image.getWidth(this) +
                         " by " + image.getHeight(this));
      display.setIcon(new ImageIcon(image));
      getContentPane().add("Center", display);
      validate(); pack(); setVisible(true);
      System.out.println("Size: " + image.getWidth(this) +
                         " by " + image.getHeight(this));
   }
   public static void main(String args[])
   {  SimpleImageTest sit = new SimpleImageTest(); }
}
```

This program works fine and since we have already seen the image before there is no reason to show another screen shot of this program. What *is* interesting, on the other hand, is the width and height of the image that our program displays. It will produce two lines on standard output similar to the following:

```
Size: -1 by -1
Size: 384 by 269
```

This illustrates that a call to `getImage` does not actually start the loading process, hence information about the dimensions of the image is not available. After the image displays, however, its dimensions are available and correct.[34]

■

At first it may not be clear why we should use the `getImage` method from the `ToolKit` when a `JLabel` can just as easily load and display the image. But having the image available separately is useful if the image needs to be processed before displaying.

[34] Note that we use `this` as input to `getWidth` and `getHeight` of the `image`. That works because `JFrame` extends `Frame` which extends `Window` which extends `Container` which extends `Component` which – finally – implements `ImageObserver`.

We will very briefly touch upon filtering images as an example of image processing. Java does provide many (!) different classes and methods to manipulate images and we can not hope to cover all of them. Therefore, we will provide only a very brief example of image filtering.

Definition 6.5.4: Image Filtering using the `RGBImageFilter` and `GrayFilter` Classes

Images can be filtered to change the way they present their pixels. A simple-to-use image filter is provided by the `java.awt.image.RGBImageFilter` *class. To create a filter, you define a class extending* `RGBImageFilter` *and implement one abstract method:*

```
public int filterRGB(int x, int y, int rgb)
```

You should also set the value of `canFilterIndexColorModel` *to* `true` *if the filter's operation does not depend on the pixel's (x,y) location, or to* `false` *otherwise. In a class extending* `Component` *you can then obtain the new filtered image via:*

```
Image newImg = createImage(new FilteredImageSource(
                           img.getSource(), new MyOwnFilter()));
```

One ready-made image filter is provided by the Swing class `GrayFilter`*, which extends* `RGBImageFilter`*. Its constructor takes a* `boolean` *value as input, indicating whether pixels should be brightened, and an integer between 0 and 100 indicating the percentage of graying to be used.*

Example 6.5.5:

Use the `GrayFilter` class to create a gray version of the `cat.jpg` image. Your program should provide buttons to either show the original color version or the grayed version of the image.

We have already seen how to load an image using the `ToolKit`. This time, we use two `Image` fields, load the original image into the first and store a grayed version of it in the second, using the `GrayFilter` class as described in the definition. We define buttons as usual to display either one of these images.

```
import java.awt.FlowLayout;
import java.awt.Image;
import java.awt.image.*;
import java.awt.event.*;
import javax.swing.*;

public class FilteredImageTest extends JFrame implements ActionListener
{  private ImageIcon colorVersion = null;
   private ImageIcon grayVersion  = null;
   private JButton color = new JButton("Color");
   private JButton gray  = new JButton("Gray");
   private JLabel display = new JLabel();
   public FilteredImageTest()
   {  super("Image Test");
      Image colorImage = getToolkit().getImage("cat.jpg");
      Image grayImage = createImage(new FilteredImageSource(
                colorImage.getSource(), new GrayFilter(true, 50)));
      colorVersion = new ImageIcon(colorImage);
      grayVersion  = new ImageIcon(grayImage);
      JPanel buttons = new JPanel(new FlowLayout());
```

```
            buttons.add(color);  color.addActionListener(this);
            buttons.add(gray);   gray.addActionListener(this);
            display.setIcon(colorVersion);
            getContentPane().add("Center", display);
            getContentPane().add("South",  buttons);
            validate(); pack(); setVisible(true);
         }
         public void actionPerformed(ActionEvent ae)
         {  if (ae.getSource() == color)
               display.setIcon(colorVersion);
            else if (ae.getSource() == gray)
               display.setIcon(grayVersion);
         }
         public static void main(String args[])
         { FilteredImageTest sit = new FilteredImageTest(); }
      }
```

Here are the two versions of the image:

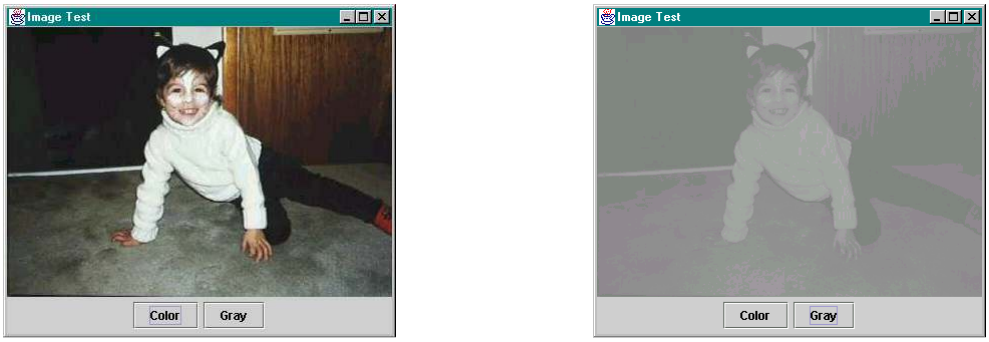


Figure 6.5.1: Original and GrayFiltered `cat.jpg` *image*

■

It is relatively easy to provide your own image filters, but the details will be beyond the scope of this book[35]. Please consult the Java API for details on the `RGBImageFilter` and `FilteredImageSource` classes.

Animation and Enhanced Image Loading

So far we have let the `JLabel` class or `getImage` method determine when exactly the image is loaded and when and how the image is displayed. As it happens, that class will start the loading process, wait until the entire image is downloaded, then display the complete image. The `getImage` method on the other hand returns immediately, not loading the image at all. If we wanted to create an animation using several images, we could load all images into an array via `getImage`, then use that array of images sequentially in a `JLabel` to create an animation. However, the first time that animation is running it would not run effectively, because the `JLabel` class will wait until the entire

[35] A simple inversion filter that creates the photo-negative of a color image would look as follows:

```
public class InverseFilter extends RGBImageFilter
{ public InverseFilter() { canFilterIndexColorModel = true; }
  public int filterRGB(int x, int y, int p)
  { DirectColorModel cm = (DirectColorModel)ColorModel.getRGBdefault();
    int red = 255-cm.getRed(p), green = 255-cm.getGreen(p), blue = 255-cm.getBlue(p);
    return ((cm.getAlpha(p) << 24) | (red << 16) | (green << 8) | blue); }
}
```

image is loaded before displaying it. There would therefore be a delay between images at least as large as it takes to download the next image. To avoid that delay, we want to make sure that all images are loaded completely before beginning the animation process. Java provides the `MediaTracker` class for that purpose.

Definition 6.5.6: The `MediaTracker` Class

The `MediaTracker` *is a utility class that can automatically track the loading progress of one or more images. In its simplest form an instance of a* `MediaTracker` *is created and attached to a* `Component`*, the images are added to it, and then the* `MediaTracker` *is told to either block any further execution until one or more images are completely loaded or to start the loading process for one or more images without necessarily displaying them.*

```
public class MediaTracker extends Object
{ // Constructor
   public MediaTracker(Component comp)
   // selected Methods
   public void addImage(Image image,  int id)
   public boolean checkAll(boolean beginLoading)
   public boolean checkID(int id, boolean beginLoading)
   public void waitForAll() throws InterruptedException
   public void waitForID(int id) throws InterruptedException
   public synchronized boolean isErrorAny()
}
```

The `waitForAll` *and* `waitForID` *methods are blocking methods, i.e. they suspend execution until the image or images are completely loaded. The* `checkAll` *and* `checkID` *methods will start the loading process for the image or images without blocking. Currently, the* `MediaTracker` *can only track image files but in future versions it might also track sound file and other media formats.*

Example 6.5.7:

Create, somehow, several images that form an animation (use stick figures if you must). Then load them using a `MediaTracker`, making sure all images are loaded. Finally, display the animated sequence in an infinite loop. If any errors occur, inform the user via an appropriate message box.

This example really consists of several steps to take:

- we need to create images suitable for animation
- we need to load those images into a class using a media tracker
- we need to determine a class to extend for our animation
- we need to animate the images

Creating the animation images has nothing to do with Java programming and we will leave it to your creativity. As an example, we'll create a "winking smiley face", using the following images:


image1


image2


image3


image4


image5

Figure 6.5.2: Various images used for animation

For our animation, we arrange the image in the following files, using some duplicates:

`smile0.gif` = *image1*	`smile1.gif` = *image2*	`smile2.gif` = *image3*
`smile3.gif` = *image2*	`smile4.gif` = *image1*	`smile5.gif` = *image4*
`smile6.gif` = *image5*	`smile7.gif` = *image4*	`smile8.gif` = *image1*

That solves our first problem. Next, we will introduce the code snippet to load images controlled by a `MediaTracker`:

```
Image images[] = new Image[9];
MediaTracker tracker = new MediaTracker(this);
for (int i = 0; i < images.length; i++)
{  images[i] = getToolkit().getImage("smile" + i + ".gif");
   tracker.addImage(images[i], i);
}
try
{  tracker.waitForAll(); }
catch(InterruptedException ie)
{  showError("Image loading interrupted: " + ie); }
if (tracker.isErrorAny())
   showError("Error loading one or more images");
```

In other words, we create an array of nine images and an instance of a `MediaTracker` attached to `this` component (which must therefore implement `ImageObserver`). Then we use `getImage` to load the images into the array. This will not actually start the loading process as `getImage` returns immediately. We also add each image to the media tracker. Then we start the loading process for all images and wait until all of them are completely loaded. Since `waitForAll` throws an exception, we embed it in a standard `try-catch` clause, using a method `showError` we still need to implement. Finally, we check on the status of the media tracker to see if there were any errors loading the image. The most likely error would be that a file name is incorrect. If an error occurs, we again call on `showError` to inform the user. That solves our second problem.

As for the third part, we could extend `JPanel` since we are doing some custom drawing or `JLabel` since we are trying to display images. If we did use `JLabel`, though, we would need to wrap our images into `ImageIcon` objects and that would be a waste or memory. So we will use the `JPanel` class and draw the images "by hand" without the help of a `JLabel`.

To finish the class, we need to decide how to do the actual animation. Of course we will use a thread for this, so our class needs to implement the `Runnable` interface. That means we need a field of type `Thread` and a `run` method. We will also use an additional field called `currentImage` of type `int`. The `run` method will, in an infinite loop, advance `currentImage` and call `repaint`, as well as putting our thread to sleep for some time. The `repaint` method will in turn call `paintComponent`, where we draw the current image as specified by `currentImage`. Here is the complete class:

```
import java.awt.MediaTracker;
import java.awt.Dimension;
import java.awt.Image;
import java.awt.BorderLayout;
import java.awt.Graphics;
import javax.swing.*;

public class AnimationPanel extends JPanel implements Runnable
{  private Image images[] = new Image[9];
   private int currentImage = 0;
```

```
   private Thread thread = null;
   public AnimationPanel()
   {  MediaTracker tracker = new MediaTracker(this);
      for (int i = 0; i < images.length; i++)
      {  images[i] = getToolkit().getImage("smile" + i + ".gif");
         tracker.addImage(images[i], i);
      }
      try
      {  tracker.waitForAll(); }
      catch(InterruptedException ie)
      {  showError("Image loading interrupted: " + ie); }
      if (tracker.isErrorAny())
         showError("Error loading one or more images");
      thread = new Thread(this);
      thread.start();
   }
   public void run()
   {  while (true)
      {  try
         {  repaint();
            thread.sleep(100);
            if (currentImage >= images.length-1)
               currentImage = 0;
            else
               currentImage++;
         }
         catch(InterruptedException ie)
         {  showError("Thread interrupted: " + ie); }
      }
   }
   public Dimension getPreferredSize()
   {  return new Dimension(images[0].getWidth(this),
                           images[0].getHeight(this));
   }
   public void paintComponent(Graphics g)
   {  super.paintComponent(g);
      g.drawImage(images[currentImage], 0, 0, this);
   }
   private void showError(String msg)
   {  JOptionPane.showMessageDialog(this, msg, "Error",
                                    JOptionPane.ERROR_MESSAGE);
   }
}
```

Of course this class needs to be embedded in another class before it can run. Here is a sample class that will finish this example:

```
import javax.swing.*;

public class AnimationTest extends JFrame
{  public AnimationTest()
   {  super("Animation Test");
      getContentPane().add("Center", new AnimationPanel());
      validate(); pack(); setVisible(true);
   }
   public static void main(String args[])
   { AnimationTest gt = new AnimationTest(); }
}
```

It is difficult to show an animation on a static page, but given the images shown above it should be easy to imaging the animation showing a winking smiley face.

■

There are many possible improvements for an `AnimationPanel` class. Some possibilities are:

- add a constructor that passes files names into the `AnimatorPanel` class so that those files will be used for animation
- allow for setting of the sleep time and add the possibility of adding a longer pause at the end of the animation
- change the `MediaTracker` so that the first image will be displayed as soon as possible, while loading the remaining images. The actual animation would only start when all images are loaded
- allow for starting, pausing, and restarting the animation at any time

We will not pursue this here, but none of these modifications would be difficult and you are welcome to try them as exercises.

Loading and Displaying Images in Applets

We have so far avoided creating any applets that deal with images, even as icons for buttons. That was no accident, but due to a security restriction on applets: applets are not allowed to read any files from the local disk so we *could* not have loaded an image file from the local disk into an applet. We will discuss applet security in detail in chapters 9 and 10, but we need to make a quick excursion into the land of URL's in order bypass that restriction and support loading images and sounds in applets nonetheless.

Recall that the `getImage` method has an overloaded constructor, requiring either a `String` or a `URL` as input for the image file to load. The same is true for the constructor of an `ImageIcon`. So far we have only used the filename as input string, but now it is time to explore the second version using a `URL`.

Definition 6.5.8: Uniform Resource Locator and the `URL` Class

A Uniform Resource Locator, or URL, is a pointer to a resource on the World Wide Web. In most, but not all, situations, the resource represents a file such as an HTML document an image file, or a sound clip. A URL consists of several parts: the protocol, the machine address, an optional port, the file component, and an optional anchor. If some components of a URL are missing, they can be inherited from the context URL. Java provides a `URL` *class in the* `java.net` *package to represent a Uniform Resource Locator:*

```
public final class URL extends Object
{  // Constructors
   public URL(String address) throws MalformedURLException
   public URL(URL context, String file)
          throws MalformedURLException
   public URL(String protocol, String host, String file)
          throws MalformedURLException
   public URL(String protocol, String host, int port, String file)
          throws MalformedURLException
}
```

In particular, note that we need to catch a `MalformedURLException` if we want to create a new URL. The first and second form of the constructor are perhaps the most useful:

- the first form, requiring only a string as input, is used to enter the entire, complete URL address in much the same way you would enter it in a web browser to visit a particular web page (for example `new URL("http://www.shu.edu/index.html")`).
- the second form, requiring a URL context and a String, can be used when the base URL of a file is matched against the first parameter and the name of the file against the second one (for example `new URL(baseURL, "index.html")`, assuming that `baseURL` is a `URL` such as `URL baseURL = new URL("http://www.shu.edu/")`).

While applets are prevented from reading files from disks, they *are* allowed to read data through a URL connection. So if an applet could construct a URL pointing to a local file, it could use the appropriate `getImage` method or `ImageIcon` constructor to load an image file just fine. In fact, applets have two methods that make it easy to construct such URL's:

Definition 6.5.9: Loading Data into Applets with `getCodeBase` and `getDocumentBase`

Applets are prevented from reading data directly from disk, but they are allowed to read data from a URL. To construct a URL that points to the location of a data file that an applet wants to read, the `JApplet` *class inherits the methods*[36]*:*

- `public URL getDocumentBase()`*: Retrieves the base address of the* `HTML` *document containing the current* `<APPLET>` *tags for this applet in URL form*
- `public URL getCodeBase()`*: Retrieves the base address of the location of the applet class file in URL form.*

The returned base addresses could refer to an address on the Internet or a location on a local disk. The standard protocol using an Internet location is `http`*, while the protocol for local addresses is* `file`*. Applets and the data files they want to read must be located on the same machine.*[37]

Recall that the `Applet` tag for `HTML` documents provides a number of parameters. In particular, the parameter `CODE="ClassFile.class"` specifies which Java class file to load, and the optional `CODEBASE="url"` specifies the base location where `ClassFile.class` is located:

- the `getDocumentBase` method will return the base location of the `HTML` document containing the `<APPLET>` tag or appropriate `<EMBED>` tag
- the `getCodeBase` method will return the base address specified in the `CODEBASE` parameter of the `<APPLET>` or `<EMBED>` tag if the class file and <code>HTML</code> documents are located in different directories.

Example 6.5.10:

Suppose you create a document named `images.html` that is saved on a web server named `www.mathcs.shu.edu` in a folder named `JBD`. That document might contain, aside from any other relevant `HTML` code, the applet tag:

[36] `JApplet` extends `Applet` and `Applet` contains `getDocumentBase` and `getCodeBase`. Compare definition 4.15.
[37] We will discuss applet security restrictions in detail in chapter 9.

```
<APPLET CODEBASE="http://www.shu.edu/~wachsmut/Java"
        CODE="Images.class"
        WIDTH=300 HEIGHT=300>
</APPLET>
```
[38]

Which base addresses will be returned by the `getCodeBase` and `getDocumentBase` methods?

The location of the `HTML` file `images.html` that you would enter in a web browser to access the file would be:

```
http://www.mathcs.shu.edu/JBD/images.html
```

and the location of the Java class file in this example, in its full URL form, is

```
http://www.shu.edu/~wachsmut/Java/Images.class
```

The method `getDocumentBase` will then return `http://www.mathcs.shu.edu/JBD` while the method `getCodeBase` returns `http://www.shu.edu/~wachsmut/Java`.[39]

■

Now it should be clear how we could redo the `JAppletWithButtons` example so that the applet contains the images that were part of the `JFrameWithButtons` program in example 6.2.10.

Example 6.5.11:

Redo example 6.2.12 to create an applet containing three buttons with images. It does not actually matter what happens when you click on the buttons as long as the images appear properly. Assume that the image files are located in a directory `Icons` in the same directory as the class file. Would this applet work when started from a local `appletviewer` program as well as from a web browser off the Internet?

As in example 6.2.10, we assume that we have the following icons saved in a directory `Icons`:

 `new.gif`  `windows.gif`  `xwin.gif`  `java.gif`

For a frame we would define the buttons as fields and initialize them via:

```
JButton windLook = new JButton("Windows",
                               new ImageIcon("Icons/windows.gif"));
```

That would not work for applets, so we need to use the constructor of an `ImageIcon` that uses a `URL` to point to the named image file. But to construct a `URL` we need to catch a `MalFormedURL` exception, which we can not do while initializing a field. Therefore, we first define a button without an icon, then add the appropriate icons in the `init` method. Since the example states that the icons are located in an `Icons` directory off the applet class file, we will use the `getCodeBase` method to create the appropriate `URL`, using the second version of the `URL` constructor. Here is the code:

```
import java.net.*;
```

[38] You could also use the EMBED or OBJECT tags as discussed in Definition 6.2.13, with similar results.
[39] In this case the applet could not use `getDocumentBase` to load images, because the HTML document and the class file are located on different machines. The applet could only load images from `www.shu.edu`, i.e. it must use `getCodeBase` as the base address for loading images.

```
import java.awt.event.*;
import java.awt.FlowLayout;
import javax.swing.*;

public class JAppletWithButtons extends JApplet implements ActionListener
{  JButton windLook = new JButton("Windows");
   JButton unixLook = new JButton("Unix");
   JButton javaLook = new JButton("Java");
   JLabel label = new JLabel("Welcome to Swing", SwingConstants.CENTER);
   public void init()
   {  try
      {  URL windURL = new URL(getCodeBase(), "Icons/windows.gif");
         URL unixURL = new URL(getCodeBase(), "Icons/xwin.gif");
         URL javaURL = new URL(getCodeBase(), "Icons/java.gif");
         URL newURL  = new URL(getCodeBase(), "Icons/new.gif");
         windLook.setIcon(new ImageIcon(windURL));
         unixLook.setIcon(new ImageIcon(unixURL));
         javaLook.setIcon(new ImageIcon(javaURL));
         label.setIcon(new ImageIcon(newURL));
      }
      catch(MalformedURLException murle)
      {  System.err.println("Error loading button images: " + murle); }
      getContentPane().setLayout(new FlowLayout());
      getContentPane().add(label);
      getContentPane().add(windLook);
      getContentPane().add(unixLook);
      getContentPane().add(javaLook);
   }
   public void actionPerformed(ActionEvent ae)
   {  /* whatever should happen if the buttons are pressed */ }
}
```

With the appropriate HTML document in place, the `appletviewer` will show the same image as for the `JFrameWithButtons` program:

Figure 6.5.3: `JAppletWithButtons`*, using URL's to add icons to buttons and label*

Note that `getCodeBase` will take care of all protocol details so the code will work whether the applet is started from a local disk (`file:` protocol) or loaded from a web site (`http:` protocol), as long as the images and the class files are located on the same machine and the images are in a subdirectory called `Icons` off the class file directory.

■

From now on we can interchange standalone programs and applets, as long as the methods that load resources such as images and sound clips support the `URL` input parameter.

Off-Screen Drawing and Double-Buffering

So far all of our custom drawing took place in the `paintComponent` method. To be more precise we should say that all drawings took place within the `Graphics` object provided (automatically) by the `paintComponent` method. That is frequently not sufficient:

- If the drawing takes a lot of computing time, a program might want to prepare the graphics while showing the user some intermediate information, then swapping the entire image into place at once.
- The `paintComponent` method is called every time the class needs to refresh its display. If the code in that method takes a lot of time to compute it will be called all the time, causing unnecessary delays.

The solution to these and other related problems is to draw to an off-screen graphics object instead of the `Graphics` object provided by `paintComponent`. Then, when the time comes, we can replace the `paintComponents's Graphics` object (which is visible) with the off-screen graphics (which used to be invisible) in one quick call. In other words, we need to:

- acquire an off-screen graphics object of the right dimensions
- ensure that all drawing is taking place in the off-screen graphics object
- implement the `paintComponent` method so that it swaps the off-screen graphics into the visible area

To some extend, Swing components support this idea by default using a technique called double-buffering.

Definition 6.5.12: Double-Buffering

> *All Swing components that extend* `JComponent` *use a technique called double-buffering for painting. That means that instead of drawing directly to the visible area, an off-screen graphics object is created and all drawing is completed in the off-screen area. When painting is finished, the off-screen area is swapped into the visible area automatically. Double-buffering helps create graphics that appears smoothly and without flickering but requires additional memory to hold the off-screen drawing area. The* `JComponent` *class offers the method* `public void setDoubleBuffered(boolean flag)` *to turn this feature off if necessary.*

Double-buffering usually works well and you do not have to worry about it, but occasionally you need more control and you need to customize the drawing behavior to substitute your own off-screen drawing technique.

Example 6.5.13:

Create a simple program using Swing components that draws 10,000 filled circles in random colors, with random centers and radius 10 pixels. Start the program, move another window on top, then move the program window back to the front. Redo the same example, using only AWT components. Explain any differences you observe.

The code for the Swing classes is quite simple: the graphics class extends `JPanel`, as usual, and put all the drawing-related code into the `paintComponent` method (the code simply allocates a random color and random coordinates for the center, then draws the corresponding circle using the `fillOval` method). We need a small second class to create a working program, so here are both classes:

```
import java.awt.Graphics;
import java.awt.Color;
```

```
import javax.swing.*;

public class LotsOfCircles extends JPanel
{  public void paintComponent(Graphics g)
   {  super.paintComponent(g);
      for (int i = 0; i < 10000; i++)
      {  Color color = new Color((int)(255*Math.random()),
                                 (int)(255*Math.random()),
                                 (int)(255*Math.random()));
         int x = (int)(getSize().width * Math.random());
         int y = (int)(getSize().height * Math.random());
         g.setColor(color);
         g.fillOval(x-10, y-10, 20, 20);
      }
   }
}

import javax.swing.*;

public class LotsOfCirclesTest extends JFrame
{  public LotsOfCirclesTest()
   { super("Lots of Circles");
     getContentPane().add("Center", new LotsOfCircles());
     setSize(300, 300); validate(); setVisible(true);
   }
   public static void main(String args[])
   {  LotsOfCirclesTest loct = new LotsOfCirclesTest(); }
}
```

Figure 6.5.4: `LotsOfCirclesTest` *program*

The program works fine, but the graphics take a long time to complete. In fact, you will not see anything for quite some time, so be patient when running this program. Since Swing components use, by default, double-buffering, the circles are drawn in an off-screen area first and swapped into the visible area when the `paintComponent` method is finished. That leaves the user looking at an empty window for a while, which is not desirable.

Let's see what the equivalent AWT program would do: we redo our classes, making the following modifications:

- remove all occurrences to Swing in the `import` statements
- change `JPanel` to `Canvas` in `LotsOfCircles`
- change `paintComponent` to `paint` in `LotsOfCircles` and remove the call to `super.paintComponent`
- change `JFrame` to `Frame` in `LotsOfCirclesTest`
- change `getContentPane().add(…)` to `add(…)` in `LotsOfCirclesTest`

Here are the new classes, resulting in the same program using AWT components only:

```
import java.awt.*

public class LotsOfCircles extends Canvas
{  public void paint(Graphics g)
```

```
   { /* as before but without super.paintComponent(g); */ }
}

import java.awt.*;

public class LotsOfCirclesTest extends Frame
{  public LotsOfCirclesTest()
   { super("Lots of Circles");
     add("Center", new LotsOfCircles());
     setSize(300, 300); validate(); setVisible(true);
   }
   public static void main(String args[])
   {  LotsOfCirclesTest loct = new LotsOfCirclesTest(); }
}
```

The program produces an image similar to the one before, but now you can see the circles as they are being drawn. That is somewhat better than before, even though the time it takes to complete the final graphics is just about the same. But in the AWT-based program the user can see something happening, which is better than looking at an empty screen for a while.

However, both programs suffer from one important problem: every time the window needs to be regenerated the entire computation starts again. If you move, for example, another window on top of your program, then move your program back to the foreground, it regenerates the entire picture. Since that takes a fairly long time, the user will get annoyed with that program quickly. Moreover, since random circles are used, every time the picture regenerates it looks different.

In some sense, there will be little we can do: the computation takes time and the code can not be optimized. However, unless the size of the window changes there is really no need to regenerate the computation if our program moves to the background, then to the foreground. All we would need to do is to be able to store the finished drawing somehow without recomputing it every time.

■

Now that we have seen a potential problem with generating complex graphics we need to explore how to do our own off-screen drawing so that we can substitute our own optimized code for the general-purpose double-buffering scheme used by Swing.

Definition 6.5.14: Off-Screen Drawing Area

> *An off-screen drawing area is an* `Image` *object that is not visible. To create an off-screen* `Image` *object and an attached off-screen* `Graphics` *object, the* `Component` *method*
>
> ```
> public Image createImage(int width, int height)
> ```
>
> *can be used, together with the* `Image` *method*
>
> ```
> public Graphics getGraphics()
> ```
>
> *The* `createImage` *method returns a reference to an* `Image` *object with the specified* `width` *and* `height`*, while* `getGraphics` *returns a reference to a* `Graphics` *object attached to an image. Note*

that `createImage` *will return* `null` *if either* `width` *or* `height` *is zero, or if used in a constructor before the underlying object is laid out completely*[40].

In other words, to create an off-screen graphics object, you could use code similar to the following:

```
// define fields
Image offImage = null;
Graphics offGraphics = null;
...
// initialize the fields
offImage = getImage(width, height);
offGraphics = offImage.getGraphics();
```

To help utilizing off-screen graphics object, the following guidelines might be useful:

Definition 6.5.15: Rule of Thumb for using Off-Screen Drawing Areas

To use an off-screen drawing area you may want to follow these suggestions:

- *Define integer fields* `height` *and* `width` *to store the current size of the component*
- *Define fields* `Image offImage` *and* `Graphics offGraphics` *and initialize them whenever the actual dimensions of the component are different from* `height` *and* `width`*, using a custom method* `initOffGraphics`
- *Move all drawing code from* `paintComponent` *to, say,* `public void offPaint(Graphics g)`
- *Call* `offPaint(offGraphics)` *at the appropriate time explicitly*
- *Implement the* `paintComponent` *method similar to the following code:*

```
public void paintComponent(Graphics g)
{  super.paintComponent(g);
   if ((getSize().width != width) || (getSize().height != height))
      initOffGraphics();
   if (offImage != null)
      g.drawImage(offImage, 0, 0, this);
}
```

For long computations consider using a thread to call `offPaint`.

Example 6.5.16:

Modify the "10,000 circles" program from example 6.5.13 so that it utilizes a custom-made off-screen drawing area (but no thread) instead of drawing directly to the visible area. Run the program, cover it with another window, then bring it back to the foreground. Also resize the window to make sure that it works correctly. Describe what happens and what the differences are to the previous implementation in example 6.5.13.

To understand a common error when using the `createImage` method, we will *not* follow the above suggestion but try to make do with somewhat easier code, not saving the current `width` and `height` and instead creating the offscreen image right in the constructor.

[40] This is a common source of error in using an off-screen area: the method `getGraphics` is used in a constructor, but because it is called *before* the component is laid out, it returns `null`. The common solution is to (a) always check if the off-screen image is null before using it and (b) initialize it in a method different from the constructor.

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Image;
import javax.swing.*;

public class LotsOfCircles extends JPanel // this class will not work
{  private Image offImage = null;
   private Graphics offGraphics = null;
   public LotsOfCircles()
   {  super();
      offImage = createImage(getSize().width, getSize().height);
      offGraphics = offImage.getGraphics();
      offPaint(offGraphics);
   }
   public void offPaint(Graphics g)
   {  for (int i = 0; i < 10000; i++)
      {  Color color = new Color((int)(255*Math.random()),
                                 (int)(255*Math.random()),
                                 (int)(255*Math.random()));
         int x = (int)(getSize().width * Math.random());
         int y = (int)(getSize().height * Math.random());
         int r = 10;
         g.setColor(color);
         g.fillOval(x-r, y-r, 2*r, 2*r);
      }
   }
   public void paintComponent(Graphics g)
   {  super.paintComponent(g);
      if (offImage != null)
         g.drawImage(offImage,0,0,null);
   }
}
```

In the constructor we initialize the off-screen image and attached graphics object. Then we call `offPaint` with `offGraphics` as input to generate the image. Finally, the `paintComponent` method is simplified to swap the off-screen graphics into the visible graphics context.

The class `LotsOfCirclesTest` remains unchanged from example 6.5.13 but to our dismay we get the following runtime error when running the program:

```
C:\jbd\06\>java LotsOfCirclesTest
Exception in thread "main" java.lang.NullPointerException
        at LotsOfCircles.<init>(LotsOfCircles.java:11)
        at LotsOfCirclesTest.<init>(LotsOfCirclesTest.java:6)
        at LotsOfCirclesTest.main(LotsOfCirclesTest.java:10)
```

The code leading to the null pointer exception is the line

```
offGraphics = offImage.getGraphics();
```

Just before that line we create `offImage` with input parameters `getSize().width` and `getSize().height` to make sure the offscreen image has the same size as the image in the `JPanel`. However, the `JPanel`, at that time, has not been laid out yet so its width and height are –1. Hence, `offImage` is `null` and therefore we can not call its `getGraphics` method.

To fix that error, we must make sure to generate the offscreen image only when the panel has a well-define width and height. Moreover, we should do it in such a way that the offscreen image and associated graphics is regenerated whenever the size of the panel changes.

To accomplish that, we follow the above rule of thumb guidelines and keep track of the current dimensions of the panel. In the `paintComponent` method we check if the dimensions have changed. If so, `paintComponent` will regenerate the offscreen image and associated drawing. Since `paintComponent` is called every time the panel is resized or redrawn, this will ensure that the offscreen image will be in total sync with the size of the panel at all times. Here is the code that works:

```
public class LotsOfCircles extends JPanel  // this class will work fine
{  private Image offImage = null;
   private Graphics offGraphics = null;
   private int width = 0, height = 0;
   public LotsOfCircles()
   {  super();
      setDoubleBuffered(false);
   }
   public void offPaint(Graphics g)
   {  for (int i = 0; i < 10000; i++)
      {  Color color = new Color((int)(255*Math.random()),
                                 (int)(255*Math.random()),
                                 (int)(255*Math.random()));
         int x = (int)(getSize().width * Math.random());
         int y = (int)(getSize().height * Math.random());
         int r = 10;
         g.setColor(color);
         g.fillOval(x-r, y-r, 2*r, 2*r);
      }
   }
   public void initOffGraphics()
   {  width = getSize().width;
      height = getSize().height;
      offImage = createImage(width, height);
      offGraphics = offImage.getGraphics();
      offPaint(offGraphics);
   }
   public void paintComponent(Graphics g)
   {  super.paintComponent(g);
      if ((width != getSize().width) || (height != getSize().height))
         initOffGraphics();
      if (offImage != null)
         g.drawImage(offImage,0,0,this);
   }
}
```

In other words, if `paintComponent` detects that the actual size of the component is different from the one recorded in `width` and `height`, the offscreen image is regenerated and its associated graphics is recomputed. If the dimensions are unchanged, `paintComponent` simply swaps the offscreen image into the visible area. If `offImage` is `null` because the process is just starting, the `paintComponent` method does nothing. Note that we turned off default buffering by calling `setDoubleBuffered(false)` in the constructor.

This time the program runs perfectly. There are no runtime exceptions and the image is generated correctly (but there is still a brief period where the user looks at an empty screen). If the program is brought to the background, then again to the foreground, the image is not regenerated but the saved offscreen image is swapped into the foreground. That happens without noticeable delay, so there is no unnecessary computation. Therefore, this technique works better than the default double buffering scheme in this case. If, on the other hand the window is resized, the `paintComponent` detects the new size and regenerates the image appropriately.

■

The last thing for us to worry about is that there is a short period where the user looks at an empty program. Worse than that, for that time the program is actually quite busy and will not react to any user directives such as a click on the standard close box. To improve that behavior, we need to employ a thread to handle the generation of the graphics asynchronously.

Example 6.5.17:

Redo example 6.5.16, this time using a thread to generate the offscreen image.

The idea is simple: instead of directly calling `offPaint`, we start a thread that in turn calls `offPaint` in its `run` method. Since the thread runs concurrently to our program, the image is generated asynchronously. To improve what the user sees we also call repaint every 100th time in the `run` method to make that part of the picture visible that has been computed so far. Finally, if the offscreen image needs to be regenerated, we restart the thread so that our program can regenerate a new picture immediately after the user resizes the window. The new code is in bold and italics.

```
public class LotsOfCircles extends JPanel implements Runnable
{  private Image offImage = null;
   private Graphics offGraphics = null;
   private Thread thread = null;
   private int counter = 0;
   private int width = 0, height = 0;
   public LotsOfCircles()
   {  super();
      setDoubleBuffered(false);
   }
   public void run()
   {  Thread currentThread = Thread.currentThread();
      while ((thread == currentThread) && (counter < 10000))
      {  offPaint(offGraphics);
         counter++;
         if ((counter % 100) == 0)
            repaint();
         thread.yield();
      }
   }
   public void offPaint(Graphics g)
   {  Color color = new Color((int)(255*Math.random()),
                              (int)(255*Math.random()),
                              (int)(255*Math.random()));
      int x = (int)(getSize().width * Math.random());
      int y = (int)(getSize().height * Math.random());
      int r = 10;
      g.setColor(color);
      g.fillOval(x-r, y-r, 2*r, 2*r);
   }
   public void initOffImage()
   {  thread = null;
      width = getSize().width;
      height = getSize().height;
      offImage = createImage(width, height);
      offGraphics = offImage.getGraphics();
      counter = 0;
      thread = new Thread(this);
      thread.start();
   }
   public void paintComponent(Graphics g)
   {  if ((width != getSize().width) || (height != getSize().height))
         initOffImage();
```

```
            if (offImage != null)
               g.drawImage(offImage,0,0,null);
         }
      }
```

It is important to remove the fixed `for` loop from the `offPaint` method in example 6.5.16. If that loop would still be there, a thread drawing the current image would continue even if the image was resized. A second thread would start *in addition*, performing the new computations. Repeated resizing of the frame would start extra threads, eating into system resources unnecessarily. Since our code instead implements part of the recommendation in definition 5.2.11, we can be sure that the old thread stops as soon as a new thread is instantiated. As soon as `thread` is different from `currentThread` in the `run` method, the "old" `run` method will stop and the "new" one will start for the new `thread`. Note that we are also cooperating nicely with other system resources by calling the `yield` method inside the `run` method.

Now we can have our cake and eat it, too. First, our program can continue to react to user input because the computation takes place in a separate thread. And – which is important – we call the `repaint` method every 100th time within `offPaint` to make that part of the picture visible that has been computed so far. When we run this applet, the screen seems to change in several discrete steps, but once the image is computed, it regenerates (almost) immediately, unless the window is resized. If the window is resized, a new image is computed automatically without having to wait for the old one to finish.

■

To be sure, this version does take *longer* to compute the final picture: threads have a computational overhead that can not be neglected, and we call repaint at every 100th step. However, to a user this image would *appear* to be generated faster, which is really what we want. Also, our applet can now contain other GUI elements that the user could use concurrently while the graphics are computed even if the computation is not yet finished.

Working with Sounds

Now that we have a reasonable understand of how to work with graphics we want to add sound to our programs. Audio files, like image files, are stored in a variety of formats. The most common ones are WAV files, primarily on Windows machines, AU files on Unix systems, and AIFF or SND files on Macintosh computers. Just as for images, there are a variety of utility programs available to convert audio files from one format to another, and there are differences in quality associated with different formats.

Definition 6.5.18: Loading Audio Files

Java supports audio files in the following formats: `AIFF`, `AU`, `WAV`, `MIDI type 0` *and* `1`, *and* `RMF`[41]. *Audio files can be loaded into an* `AudioClip` *object via the* `static` *method:*

```
public static final AudioClip newAudioClip(URL url)
```

[41] Versions of Java before 1.2 only support AU audio files, and there is no static `newAudioClip` method available, making it difficult to support sound in non-applet programs.

That method is part of the `java.applet.Applet` *class and can be used by applets and non-applets. Java supports 8 and 16 bit audio data in mono and stereo with sample rates from 8kHz to 48kHz in linear and u-law encoding. Rendering quality defaults to 16-bit stereo data at 22kHz. If this quality is not supported it automatically switches to 8-bit or mono output.*

Applets can construct a URL using `getCodeBase` *or* `getDocumentBase` *as described in definition 6.5.9. Standalone programs can reference a local audio file via the string*

```
String fileURL = "file:" + System.getProperty("user.dir")
                         + System.getProperty("file.separator")
                         + "actualFileName.ext";
```

Loading and playing sounds is actually easier, in some sense, then loading and displaying images because methods related to sounds are much more limited than those related to images. On the other hand, while sounds are easy to handle, Java offers fewer choices when dealing with sounds. There are, for example, a variety of mechanisms to control the loading process of images (most easily via the `MediaTracker`), but there is currently no such support for loading sounds.[42] Since the `newAudioClip` method returns an `AudioClip`, we need to define that class before continuing.

Definition 6.5.19: The `AudioClip` Interface

The `AudioClip` *interface is a simple abstraction for playing a sound clip.* `AudioClip` *items can be played separately or together, and each* `AudioClip` *plays independently of others. There are only three methods defined and implemented:*

`play`	*To play an entire audio clip once. Will restart the sound if it is already playing.*
`loop`	*To play an audio clip in an infinite loop. Will restart the sound if it is already playing.*
`stop`	*To stop an audio clip that is currently playing or playing in a loop. Has no effect if sound not playing.*

Java provides no convenient methods to control how much of an audio clip has already been played, how long an `AudioClip` *is, or any other helpful operation. The* `AudioClip` *interface is part of the* `java.applet` *package.*

Example 6.5.20:

Assuming that you have two audio files named `applause.wav` and `music.wav`[43], create a program that contains five buttons, two to play and stop the first audio clip and three to play, stop, and loop the other. Then experiment whether both clips can play simultaneously. The audio files should be located in a directory `Audio` that is an immediate subdirectory to the class directory.

Since the audio clip methods are so simple, the entire program is completely routine. The only noteworthy code is the construction of the URL pointing to the audio files located in the `Audio` directory. We must construct a valid URL, using various system properties as defined above.

[42] Future versions of Java will have enhanced sound support but for now we will have to be content with Java's current abilities.

[43] In a standard Windows installation you can find `"The Microsoft Sound.wav"` in the `\Windows\Media` folder and `"Applause.wav"` can be found in `\Windows\Media\Office97` (if MS Office 97 is installed. Both files can be copied to the current directory and renamed appropriately as sample sound files to use.

```
import java.net.*;
import java.applet.*;
import java.awt.event.*;
import java.awt.Container;
import java.awt.FlowLayout;
import javax.swing.*;

public class SoundTest extends JFrame implements ActionListener
{  private JButton playMusic = new JButton("Play Music");
   private JButton loopMusic = new JButton("Loop Music");
   private JButton stopMusic = new JButton("Stop Music");
   private JButton playSound = new JButton("Play Sound");
   private JButton stopSound = new JButton("Stop Sound");
   private AudioClip music = null, sound = null;
   public SoundTest()
   {  super("Sound Test");
      try
      {  String separator = System.getProperty("file.separator");
         String preface = "file:" + System.getProperty("user.dir")
                                  + separator + "Audio" + separator;
         music = Applet.newAudioClip(new URL(preface + "music.wav"));
         sound = Applet.newAudioClip(new URL(preface + "applause.wav"));
      }
      catch(MalformedURLException murle)
      {  System.err.println("Error loading files: " + murle); }
      Container content = getContentPane();
      content.setLayout(new FlowLayout());
      content.add(playMusic);  playMusic.addActionListener(this);
      content.add(loopMusic);  loopMusic.addActionListener(this);
      content.add(stopMusic);  stopMusic.addActionListener(this);
      content.add(playSound);  playSound.addActionListener(this);
      content.add(stopSound);  stopSound.addActionListener(this);
      validate(); pack(); setVisible(true);
   }
   public void actionPerformed(ActionEvent ae)
   {  if (ae.getSource() == playMusic)
         music.play();
      else if (ae.getSource() == loopMusic)
         music.loop();
      else if (ae.getSource() == stopMusic)
         music.stop();
      else if (ae.getSource() == playSound)
         sound.play();
      else if (ae.getSource() == stopSound)
         sound.stop();
   }
   public static void main(String args[])
   {  SoundTest st = new SoundTest(); }
}
```

Obviously we can not show what happens when the buttons are clicked. But when you experiment with this program you will notice that both sounds can indeed play simultaneously, producing a composite sound. Unless that is what you want you must make sure that all sounds that are currently playing are turned off (using `stop`) before a new sound starts playing.

■

You can find a complete program integrating threads, animation, and sound in example 6.6.1, the "Shark Attack" game.

Case Study: The SharkAttack Game

At this point we have explored many of the features of Swing and we have seen how to work with images, animation, and sound. We are now in a position to create a more extensive Swing-based program, combining Swing, sound effects, and animation to create a reasonably looking action game. You probably won't be able to sell it, but it's fun to play.

This example is optional. It will introduce few new features but it will illustrate how Swing components and OOP programming techniques work together to create a program that is more complex than our standard examples.

Example 6.6.1:

Create an Applet that shows the fin of a shark swimming around in the "water". While the shark is swimming, some soundtrack should be playing. The user can cause the shark to move up, down, right, left, and let it dive and rise. There should also be some animals that the shark can prey on but it can only catch its prey when the shark is fully submerged. Another sound should play whenever a prey is caught. There should be a count of animals left for the shark to eat, and a clock that shows the time in seconds since the game started.[44]

Before we can begin thinking about any code we need some graphics and sound resources. Let's assume we have two images, `river.jpg` and `swan.gif` which depict a piece of a river and a small, white swan, respectively. They might look as follows:

Figure 6.6.1: The images `river.jpg` *and* `swan.gif`

Note that the image of a swan is in `GIF` format where the background color has been marked as transparent (refer to a graphics manipulation package such as Microsoft Photo Editor for details on how to do this).

Also, we will need some sound files in a supported format. Again, let's assume that we have the files `jaws.au`, which plays some background sound clip suitable for looping and `splash.au` which will be played whenever a swan is captured by our shark.

Finally, we need several images that we can use as icons for control buttons. Let's say we have created the images:

 = `down.gif`  = `up.gif`  = `left.gif`  = `right.gif`

 = `dive.gif`  = `rise.gif`  = `stop.gif`  = `start.gif`

[44] This game is based on an idea by Michael Bosco, a former student who developed a simple but effective version of this game for a "Java and Networking" course.

To create our program we need to decide on the classes we need and how they will interact with each other. We decide to layout our program so that the image of the river will occupy the majority of the screen, with the counter and timer at the bottom of the window. We will keep the controls for the shark in a separate frame that will pop up when the user clicks on the image. It should be fairly obvious that we need classes for the Shark, a Swan (with several instances), the Counter, the Timer, the Controls, as well as an applet to pull everything together. Therefore, we will use six classes plus two inner classes, as well as our familiar `JPanelBox`, as follows:

- **`SharkAttack`**: the main class containing the images, sounds, and counter; moves images around via thread; uses inner class **`Arena`** extending `JPanel` to handle the drawing and **`Clicker`** extending `MouseAdapter` to bring up the control panel if a user wishes.
- **`Shark`**: a representation of a shark by drawing its fin; has methods to move and draw the shark
- **`SharkPrey`**: a representation of a prey animal; has methods to move and draw the image
- **`SharkControls`**: contains controls for moving shark; uses shark methods to guide it through the water
- **`SwanCounter`:** Contains images for living swans or "x" for caught ones, as well as a stop watch with methods to reset counter and the timer
- **`StopWatch`:** Simple class using a thread to count every second

Specifically, the classes will be defined with the following fields and methods:

`SharkAttack extends JApplet implements Runnable`

- **Fields**: `Shark shark, SharkPrey prey[], SharkControls guide, SharkCounter count, StopWatch watch, Image river, Image aSwan, AudioClip music, AudioClip splash, Thread thread, Arena arena`
- **Inner Classes**: `Arena extends JPanel, Clicker extends MouseAdapter`
- **Methods**: `init` *(loads images, sounds, attaches* `SharkControls` *and counter, defines tooltip, adds mouse listener and arena via inner classes),* `start` *(starts thread, resets shark, counter, prey, and timer, loops background music),* `stop` *(stops thread, music, and timer),* `run` *(moves shark and prey animals, checks which animals are caught and updates counter if necessary)*

`Shark extends Object SharkAttack applet`

- **Fields:** `Polygon fin, int deltaX, SharkAttack applet`
- **Methods:** `constructor` *(initializes* `applet` *for call-backs to get width),* `reset` *(moves shark to start position),* `moveUp`, `moveDown`, `turnLeft`, `turnRight`, `dive`, `rise`, `move` *(moves the* `fin` *horizontally, turning it around at the borders),* `getTip` *(returns the current position of the tip of the* `fin` *if fully submerged or (-1, -1) otherwise),* `paintComponent` *(draws the image of a shark fin in current configuration)*

`SharkPrey extends Object`

- **Fields:** `Image animal, Rectangle bounds, int deltaX, int amp, SharkAttack applet`
- **Methods:** `constructor` *(initializes* `applet` *and* `animal`*, picks random numbers for* `deltaX`*,* `amp` *(amplitude), defines* `bounds`*),* `move` *(moves the image in* `sine` *wave, turning it around at the borders),* `isEatenBy` *(returns* `true` *if shark's tip is inside* `bounds`*,* `paintComponent` *(draws the image at its current location)*

`SharkControls extends JPanel implements ActionListener`

- **Fields:** `String icons[], String toolTips[], SharkAttack applet, Shark shark, JButtons[]`

- **Methods:** `constructor` *(initializes* `applet` *and* `shark`*, defines layout and icon buttons),* `actionPerformed`, `showHelp`

`SwanCounter extends JPanel`

- **Fields:** `JLabel swans[], int numSwans, ImageIcon aSwan`
- **Methods:** `constructor` *(initializes* `numSwans` *and* `aSwan`*, defines array of* `JLabel` *containing* `aSwan` *icons),* `fillSlots` *(fills first* `numSwans swans[]` *with* `aSwan` *image, remaining* `swans[]` *with "X"),* `remove` *(decreases* `numSwans`*),* `reset`, `getCount`

`StopWatch extends JLabel implements Runnable`

- **Fields**: `Thread thread, int count`
- **Methods:** `constructor` *(defines default text),* `start`, `stop`, `run` *(updates label text with count ever 1000 milliseconds)*

Once we have determined the class types, methods, and fields, we of course need to implement the classes. The first one we will create is `SharkPrey` to move a swan around the screen. It needs to provide `move` to move the animal image along a predetermined path and `paintComponent` to draw it at its current position. Also, we need to implement `isEatenBy` to determine if this animal has been successfully attacked by the shark. The image that will move around is passed into the class via the class constructor.

To determine when the shark has caught one of our animals, we use the field `bounds` to specify a `Rectangle` that always surrounds the image at its current location. The path that our prey animal will follow is a simple `sine` curve with random amplitude, starting at a random location in the lower part of the image.

When the prey hits the left or right side of the applet it is supposed to turn around. Therefore, the class needs to know the dimensions of the applet so that we will pass a reference to it into this class. That means the class will not compile until you define `SharkAttack` (which in turn will not compile until all other classes are implemented):

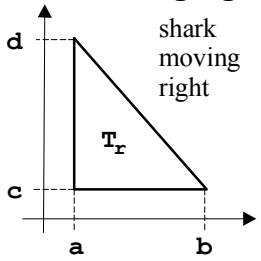
```
import java.awt.*;

public class SharkPrey
{  private SharkAttack   applet   = null;
   private Image         animal   = null;
   private Rectangle     bounds   = new Rectangle();
   private int           deltaX, amp;
   public SharkPrey(Image _animal, SharkAttack _applet)
   {  animal        = _animal;
      applet        = _applet;
      deltaX        = (int)(2*Math.random()) + 1;
      amp           = (int)(55*Math.random()) + 10;
      bounds.x      = (int)(applet.getSize().width*Math.random());
      bounds.y      = 120 + (int)(amp*Math.sin( bounds.x /20.0));
      bounds.height = animal.getHeight(null);
      bounds.width  = animal.getWidth(null);
   }
   public void move()
   {  if ((bounds.x > applet.getSize().width) || (bounds.x < 0))
         deltaX *= (-1);
      bounds.x += deltaX;
      bounds.y = 120 + (int)(amp*Math.sin(bounds.x /20.0));
   }
   public boolean isEatenBy(Shark shark)
   {  return bounds.contains(shark.getTip()); }
   public void paintComponent(Graphics g)
```

```
        {  g.drawImage(animal, bounds.x, bounds.y, null); }
    }
```

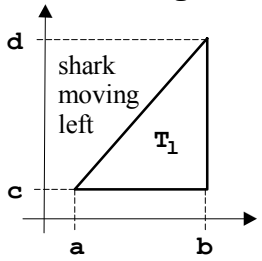
Note that this class does not actually move or draw the image. Instead it provides the *possibility* of moving (and drawing) the animal correctly. Later, the thread in our applet will be responsible for moving the shark and all its prey around the screen.

Next, let's focus on the class to move the shark. This class is more complicated because it not only provides a method to move the shark from right to left, but also methods to switch directions, move the shark up or down, let the shark dive and rise, and determine the position of the top of the `fin`. Perhaps the most complicated affair is how the shark should look like. We will draw the shark's `fin` as a right triangle with the right angle either on the left or right side depending on the direction of the shark. A simple sketch will help:

Shark fin moving right:



Shark fin moving left:



Shark fin coordinates :

$$T_r : \begin{bmatrix} x_0 = a \\ y_0 = c \end{bmatrix}, \begin{bmatrix} x_1 = a \\ y_1 = d \end{bmatrix}, \begin{bmatrix} x_2 = b \\ y_2 = c \end{bmatrix}$$

$$T_l : \begin{bmatrix} x_0 = a \\ y_0 = c \end{bmatrix}, \begin{bmatrix} x_1 = b \\ y_1 = d \end{bmatrix}, \begin{bmatrix} x_2 = b \\ y_2 = c \end{bmatrix}$$

Changing directions:

$T_r \rightarrow T_l$: set x_1 to x_2

$T_l \rightarrow T_r$: set x_1 to x_0

Diving:

T_r : decrease x_2 and y_1

T_l : decrease x_0 and y_1

Rising:

T_r : increase x_2 and y_1

T_l : increase x_0 and y_1

Figure 6.6.2: Schematics of `Shark` *for moving right, left, and diving, rising*

To represent this `fin` we will use a `Polygon`, a useful class from the AWT to represent irregularly shaped closed objects. [45]

Definition 6.6.2: Polygon

The `Polygon` *class from the AWT represents a closed, two-dimensional region bounded by an arbitrary number of line segments. Internally, a polygon is a list of* `(x, y)` *points, where each pair defines a vertex of the polygon. The first and last pairs of* `(x, y)` *points are joined by a line segment that closes the polygon. Polygons can be drawn by using the* `drawPolygon` *or* `fillPolygon` *method from the* `Graphics` *class.*

```
public class Polygon extends Object
{  // public fields
   public int npoints
   public int xpoints[], ypoints[]
   // Constructor
   public Polygon()
   public Polygon(int xpoints[], int ypoints[], int npoints)
   // Methods
   public void translate(int deltaX, int deltaY)
```

[45] The `Graphics` class also provides a method `drawPolyline` to draw a sequence of connected lines that will not necessarily form a closed figure. The input to that method consists of arrays of `x` and `y` coordinates, not of a `Polygon`.

```
      public void addPoint(int x, int y)
      public Rectangle getBounds()
      public boolean contains(Point p)
      public boolean contains(int x, int y)
   }
```

There are a few computations involved to correctly manipulate the shape of `fin` in order to represent "diving" and "rising" but when you look at the above schematics the computations should be clear:

```
import java.awt.*;

public class Shark
{  private Polygon fin = new Polygon(new int[]{ 0, 0,30},
                                    new int[]{60,30,60}, 3);
   private SharkAttack applet    = null;
   private int         deltaX    = 4;
   public Shark(SharkAttack _applet)
   {  applet  = _applet; }
   public void moveUp()
   {  fin.translate(0,-6); }
   public void moveDown()
   {  fin.translate(0,6); }
   public void turnLeft()
   {  fin.xpoints[1] = fin.xpoints[2];
      deltaX = -Math.abs(deltaX);
   }
   public void turnRight()
   {  fin.xpoints[1] = fin.xpoints[0];
      deltaX = Math.abs(deltaX);
   }
   public void dive()
   {  if ((fin.ypoints[2] - fin.ypoints[1]) > 0)
      {  if (deltaX > 0)
            fin.xpoints[2] -= 5;
         else
            fin.xpoints[0] += 5;
         fin.ypoints[1] += 5;
      }
   }
   public void rise()
   {  if ((fin.ypoints[2] - fin.ypoints[1]) < 40)
      {  if (deltaX > 0)
            fin.xpoints[2] += 5;
         else
            fin.xpoints[0] -= 5;
         fin.ypoints[1] -= 5;
      }
   }
   public void move()
   {  if (fin.xpoints[0] > applet.getSize().width)
         turnLeft();
      if (fin.xpoints[0] < 0)
         turnRight();
      fin.translate(deltaX,0);
   }
   public Point getTip()
   {  if ((fin.ypoints[2] - fin.ypoints[1]) < 4)
         return new Point(fin.xpoints[1], fin.ypoints[1]);
      else
         return new Point(-1, -1);
   }
   public void paintComponent(Graphics g)
```

```
      {  g.setColor(Color.lightGray);
         g.fillPolygon(fin);
      }
   }
```

These two classes will do most of the actual work for us and since they are fairly smart the main job of the applet class we will design next is to load the images and sounds and to properly move the shark and the prey animals around. We will use a `MediaTracker` to completely load images, and also add an instance of a `SharkCounter` to the layout of the applet. We also need a `SharkControls` class to react to user input, which will appear when the user clicks anywhere on the image. Most importantly, we use an inner class extending `JPanel` that will handle the actual drawing via its `paintComponent` method. It will draw the river image, the shark, and in a loop all prey animals that are not `null`.

```
import javax.swing.*;
import java.awt.event.*;
import java.applet.AudioClip;
import java.awt.MediaTracker;
import java.awt.Image;
import java.awt.Graphics;

public class SharkAttack extends JApplet implements Runnable
{  private Shark         shark   = null;
   private SharkPrey     prey[]  = null;
   private SharkControls guide = null;
   private SwanCounter   count   = null;
   private class Arena extends JPanel
   {  public void paintComponent(Graphics g)
      {  super.paintComponent(g);
         g.drawImage(river,0,0,
                     getSize().width,getSize().height,this);
         shark.paintComponent(g);
         for (int i = 0; i < prey.length; i++)
            if (prey[i] != null)
               prey[i].paintComponent(g);
      }
   }
   private class Clicker extends MouseAdapter
   {  public void mouseClicked(MouseEvent me)
      {  guide.setVisible(true);
         guide.toFront();
      }
   }
   private Image         river   = null, aSwan   = null;
   private AudioClip     music   = null, splash  = null;
   private Thread        thread  = null;
   private Arena         arena   = new Arena();
   public void init()
   {  river  = getImage(getDocumentBase(),    "Images/river.jpg");
      aSwan  = getImage(getDocumentBase(),    "Images/swan.gif");
      music  = getAudioClip(getDocumentBase(),"Audio/jaws.au");
      splash = getAudioClip(getDocumentBase(),"Audio/splash.au");
      MediaTracker tracker = new MediaTracker(this);
      tracker.addImage(river,0); tracker.addImage(aSwan,1);
      try
      {  tracker.waitForAll(); }
      catch(InterruptedException ie)
      {  JOptionPane.showMessageDialog(this, "Error loading images"); }
      shark = new Shark(this);
      prey  = new SharkPrey[10];
      count = new SwanCounter(new ImageIcon(aSwan), prey.length);
```

```
      getContentPane().add("Center",
                           new JPanelBox(arena, "Shark Attack"));
      getContentPane().add("South", count);
      guide = new SharkControls(this, shark);
      arena.setToolTipText("Click to show controls");
      arena.addMouseListener(new Clicker());
   }
   public void start()
   {  thread = new Thread(this);
      shark.reset();  count.reset();
      thread.start(); music.loop();
      for (int i = 0; i < prey.length; i++)
         prey[i] = new SharkPrey(aSwan, this);
   }
   public void stop()
   {  music.stop(); count.stop(); thread = null; }
   public void run()
   {  Thread currentThread = Thread.currentThread();
      while (thread == currentThread)
      {  shark.move();
         for (int i = 0; i < prey.length; i++)
            if (prey[i] != null)
               prey[i].move();
         try
         {  thread.sleep(200); }
         catch(InterruptedException ie)
         {  System.err.println("Error: " + ie); }
         for (int i = 0; i < prey.length; i++)
         {  if ((prey[i] != null) && (prey[i].isEatenBy(shark)))
            {  splash.play();
               prey[i] = null;
               count.remove();
               if (count.getCount() == 0)
                  stop();
            }
         }
         arena.repaint();
      }
   }
}
```

Recall that the `start` and `stop` methods of a `JApplet` have special meaning. `Start` is called after the applet is instantiated and every time the user revisits the page. `Stop` is called when the user leaves the page. Hence, every time the page containing the applet is visited, the `start` method is called to reset the counter and the shark, start the thread to move the animals around, start the background music, and initialize the prey animals.

The last major class, `SharkControls`, is straightforward: it extends `JFrame` so that it can "float" on top of the applet. It contains movement buttons to control the shark by calling the appropriate methods of the shark and the applet. Since we will use nine buttons, we declare them as an array so that we can initialize them in a loop. We also add appropriate tooltips to the buttons to give the user a clue as to their functionality:

```
import java.net.*;
import java.awt.event.*;
import java.awt.GridLayout;
import javax.swing.*;

public class SharkControls extends JFrame implements ActionListener
{  private String icons[] = {"pause.gif", "up.gif",   "rise.gif",
                             "left.gif",  "swan.gif", "right.gif",
```

```
                              "start.gif", "down.gif", "dive.gif" };
   private String tips[] = {"Stop game", "Shark up",   "Shark rises",
                            "Shark left", "Help",      "Shark right",
                            "Start game", "Shark down", "Shark dives"};
   private SharkAttack applet = null;
   private Shark       shark  = null;
   private JButton     buttons[] = new JButton[9];
   public SharkControls(SharkAttack _applet, Shark _shark)
   {  super("Shark Controls");
      applet = _applet;
      shark = _shark;
      JPanel buttonPanel = new JPanel(new GridLayout(3, 3));
      try
      {  for (int i = 0; i < buttons.length; i++)
         {  URL url = new URL(applet.getCodeBase(), "Images/"+icons[i]);
            buttons[i] = new JButton(new ImageIcon(url));
            buttons[i].addActionListener(this);
            buttons[i].setToolTipText(tips[i]);
            buttonPanel.add(buttons[i]);
         }
      }
      catch(MalformedURLException murle)
      {  System.err.println("Error loading icons: " + murle); }
      getContentPane().add("Center",
                           new JPanelBox(buttonPanel, "Shark Controls"));
      setResizable(false);
      validate(); pack();
   }
   public void actionPerformed(ActionEvent ae)
   {  if (ae.getSource() == buttons[0])
         applet.stop();
      else if (ae.getSource() == buttons[1])
         shark.moveUp();
      else if (ae.getSource() == buttons[2])
         shark.rise();
      else if (ae.getSource() == buttons[3])
         shark.turnLeft();
      else if (ae.getSource() == buttons[4])
         showHelp();
      else if (ae.getSource() == buttons[5])
         shark.turnRight();
      else if (ae.getSource() == buttons[6])
         applet.start();
      else if (ae.getSource() == buttons[7])
         shark.moveDown();
      else if (ae.getSource() == buttons[8])
         shark.dive();
   }
   private void showHelp()
   {  JOptionPane.showMessageDialog(this,
             "Shark Attack\n(c) 1999\nBert Wachsmuth",
             "Shark Attack", JOptionPane.INFORMATION_MESSAGE);
   }
}
```

The remaining utility classes `SwanCounter` and `StopWatch` are simple. `SwanCounter` extends `JPanel` and keeps track of how many swans are still alive by using 10 `JLabels` arranged in a flow layout. Its main public method is `remove` which is called upon by the `run` method of the applet if the shark successfully attacks a swan, but the method doing the actual work is `fillSlots`. That method sets the icon or text of the ten labels depending on the number swans still active.

```
import java.awt.FlowLayout;
```

```
import javax.swing.*;

public class SwanCounter extends JPanel
{  private JLabel     swans[] = null;
   private int        numSwans = 0;
   private ImageIcon  aSwan = null;
   private StopWatch  watch = new StopWatch();
   public SwanCounter(ImageIcon _aSwan, int maxSwans)
   {  numSwans = maxSwans;
      aSwan = _aSwan;
      swans = new JLabel[maxSwans];
      setLayout(new FlowLayout(FlowLayout.LEFT));
      add(watch);
      for (int i = 0; i < swans.length; i++)
      {  swans[i] = new JLabel(aSwan); add(swans[i]); }
   }
   private void fillSlots()
   {  for (int i = 0; i < numSwans; i++)
      {  swans[i].setIcon(aSwan); swans[i].setText(""); }
      for (int i = numSwans; i < swans.length; i++)
      {  swans[i].setIcon(null); swans[i].setText(" X "); }
   }
   public void remove()
   {  numSwans--; fillSlots(); }
   public void reset()
   {  numSwans = swans.length;  fillSlots(); watch.start(); }
   public void stop()
   {  watch.stop(); }
   public int getCount()
   {  return numSwans; }
}
```

`StopWatch` is a simple implementation of a counting thread, using a `JLabel` to display the current value of the counter, incrementing that counter every second.

```
import javax.swing.*;

public class StopWatch extends JLabel implements Runnable
{  private Thread thread = null;
   private int count = 0;
   public StopWatch()
   {  super("Time expired:   0 seconds"); }
   public void start()
   {  count = 0;
      thread = new Thread(this);
      thread.start();
   }
   public void stop()
   {  thread = null; }
   public void run()
   {  Thread currentThread = Thread.currentThread();
      while (thread == currentThread)
      {  setText("Expired: " + count + " seconds");
         count++;
         try
         {  thread.sleep(1000); }
         catch(InterruptedException ie)
         {  System.err.println("Error: " + ie); }
      }
   }
}
```

After all this work its time to enjoy our game – which is difficult to do on paper, so you need to use your imagination when looking at the still images in figure 6.6.3.

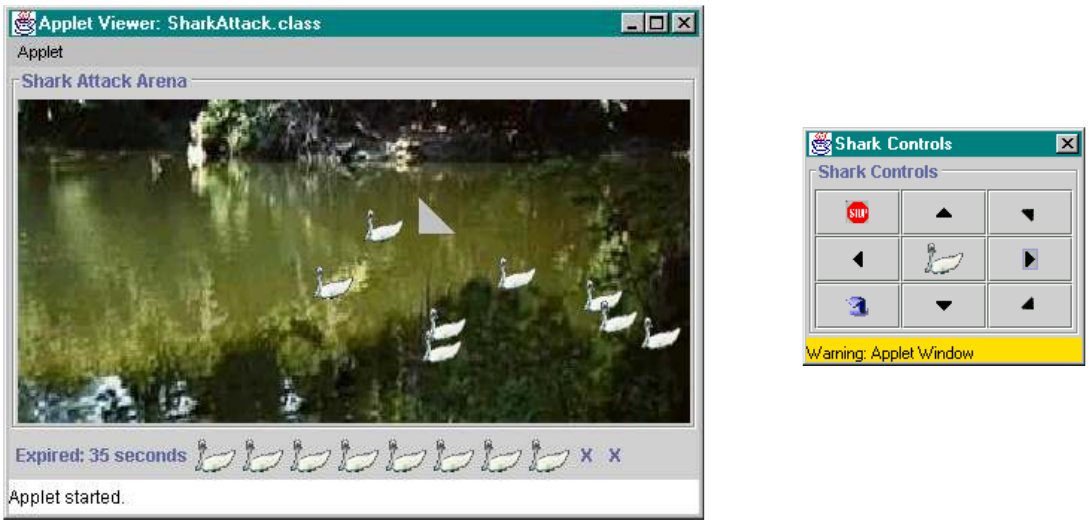


Figure 6.6.3: `SharkAttack` *game screen shot with two swans "eaten"*

When the applet actually executes, the shark continuously moves horizontally. The buttons can redirect the shark, move it up or down, or make it dive or rise. The object of the game is to pass under the prey animals while fully submerged. Every time that happens a "splash" will sound and the corresponding swan will disappear. The swans, of course, also move around in a wave-like fashion (because of the `sine` function in their `move` method), making it somewhat difficult for the shark to catch them. At any time the `restart` button can be used to reposition all swans at random locations to start over. When all swans are gone the counter will stop, showing the time it took to finish the game.

■

There are many additional topics that could have been covered in this chapter. In particular, we did not discuss the Graphics2D package, drag-and-drop support, accessibility features, undo support, image manipulation, and more. However, a detailed discussion of the advanced capabilities that Swing offers could easily fill an entire book so we will have to be content with this discussion.

Therefore, this concludes the discussion of Swing and multimedia and also ends our introduction of the basic capabilities that Java offers to create object-oriented, windows-based, GUI-driven programs. The remaining chapters of this text will not focus on how a program looks, but rather on some advanced programming techniques. Therefore, we will not use Swing in subsequent chapters but stick with the simpler AWT. However, you should feel free to convert every example that uses AWT GUI components into equivalent or better Swing-based programs. We have certainly covered all the tools necessary for that.

Chapter Summary