Graphical User Interfaces

A fundamental design guideline is "Make simple things easy, and difficult things possible."[1]

The original design goal of the graphical user interface (GUI) library in Java l.o was to allow the programmer to build a GUI that looks good on all platforms. That goal was not achieved. Instead, the Java l.o *Abstract Windowing Toolkit* (AWT) produced a GUI that looked equally mediocre on all systems. In addition, it was restrictive; you could use only four fonts and you couldn't access any of the more sophisticated GUI elements that exist in your operating system. The Java 1.0 AWT programming model was also awkward and non-object-oriented. A student in one of my seminars (who had been at Sun during the creation of Java) explained why: The original AWT had been conceived, designed, and implemented in a month. Certainly a marvel of productivity, and also an object lesson in why design is important.

The situation improved with the Java 1.1 AWT event model, which takes a much clearer, object-oriented approach, along with the addition of JavaBeans, a component programming model that is oriented toward the easy creation of visual programming environments. Java 2 (JDK 1.2) finished the transformation away from the old Java 1.0 AWT by essentially replacing everything with the *Java Foundation Classes* (JFC), the GUI portion of which is called "Swing." These are a rich set of easy-to-use, easy-to-understand JavaBeans that can be dragged and dropped (as well as hand programmed) to create a reasonable GUI. The "revision 3" rule of the software industry (a product isn't good until revision 3) seems to hold true with programming languages as well.

This chapter introduces the modern Java Swing library and makes the reasonable assumption that Swing is Sun's final destination GUI library for Java.[2] If for some reason you need to use the original "old" AWT (because you're supporting old code or you have browser limitations), you can find that introduction in the 1st edition of this book, downloadable at *www.MindView.net*. Note that some AWT components remain in Java, and in some situations you must use them.

Please be aware that this is not a comprehensive glossary of either all the Swing components or all the methods for the described classes. What you see here is intended to be a simple introduction. The Swing library is vast, and the goal of this chapter is only to get you started with the essentials and comfortable with the concepts. If you need to do more than what you see here, then Swing can probably give you what you want if you're willing to do the research.

I assume here that you have downloaded and installed the JDK documentation from *http://java.sun.com* and will browse the **javax.swing** classes in that documentation to see the full details and methods of the Swing library. You can also search the Web, but the best place to start is Sun's own Swing Tutorial at *http://java.sun.com/docs/books/tutorial/uiswing*.

---

[1] A variation on this is called "the principle of least astonishment," which essentially says, "Don't surprise the user."

[2] Note that IBM created a new open-source GUI library for their Eclipse editor *(www.Eclipse.org)*, which you may want to consider as an alternative to Swing. This will be introduced later in the chapter.

There are numerous (rather thick) books dedicated solely to Swing, and you'll want to go to those if you need more depth, or if you want to modify the default Swing behavior.

As you learn about Swing, you'll discover:

1. Swing is a much improved programming model compared to many other languages and development environments (not to suggest that it's perfect, but a step forward on the path). JavaBeans (introduced toward the end of this chapter) is the framework for that library.

2. "GUI builders" (visual programming environments) are a *de rigueur* aspect of a complete Java development environment. JavaBeans and Swing allow the GUI builder to write code for you as you place components onto forms using graphical tools. This rapidly speeds development during GUI building, and also allows for greater experimentation and thus the ability to try out more designs and presumably come up with better ones.

3. Because Swing is reasonably straightforward, even if you do use a GUI builder rather than coding by hand, the resulting code should still be comprehensible. This solves a big problem with GUI builders from the past, which could easily generate unreadable code.

Swing contains all the components that you expect to see in a modern UI: everything from buttons that contain pictures to trees and tables. It's a big library, but it's designed to have appropriate complexity for the task at hand; if something is simple, you don't have to write much code, but as you try to do more complex things, your code becomes proportionally more complex.

Much of what you'll like about Swing might be called "orthogonality of use." That is, once you pick up the general ideas about the library, you can usually apply them everywhere. Primarily because of the standard naming conventions, while I was writing these examples I could usually guess successfully at the method names. This is certainly a hallmark of good library design. In addition, you can generally plug components into other components and things will work correctly.

Keyboard navigation is automatic; you can run a Swing application without using the mouse, and this doesn't require any extra programming. Scrolling support is effortless; you simply wrap your component in a **JScrollPane** as you add it to your form. Features such as tool tips typically require a single line of code to use.

For portability, Swing is written entirely in Java.

Swing also supports a rather radical feature called "pluggable look and feel," which means that the appearance of the UI can be dynamically changed to suit the expectations of users working under different platforms and operating systems. It's even possible (albeit difficult) to invent your own look and feel. You can find some of these on the Web.[3]

Despite all of its positive aspects, Swing is not for everyone nor has it solved all the user interface problems that its designers intended. At the end of the chapter, we'll look at two alternative solutions to Swing: the IBM-sponsored SWT, developed for the Eclipse editor but freely available as an open-source, standalone GUI library, and Macromedia's Flex tool for developing Flash client-side front ends for Web applications.

[3] My favorite example of this is Ken Arnold's "Napkin" look and feel, which makes the windows look like they were scribbled on a napkin. See *http://napkinlaf.sourceforge.net*.

Applets

When Java first appeared, much of the brouhaha around the language came from the *applet*, a program that can be delivered across the Internet to run (inside a so-called *sandbox*, for security) in a Web browser. People foresaw the Java applet as the next stage in the evolution of the Internet, and many of the original books on Java assumed that the reason you were interested in the language was that you wanted to write applets.

For various reasons, this revolution never happened. A large part of the problem was that most machines don't include the necessary Java software to run applets, and downloading and installing a 10 MB package in order to run something you've casually encountered on the Web is not something most users are willing to do. Many users are even frightened by the idea. Java applets as a client-side application delivery system never achieved critical mass, and although you will still occasionally see an applet, they have generally been relegated to the backwaters of computing.

This doesn't mean that applets are not an interesting and valuable technology. If you are in a situation where you can ensure that users have a JRE installed (such as inside a corporate environment), then applets (or JNLP/Java Web Start, described later in this chapter) might be the perfect way to distribute client programs and automatically update everyone's machine without the usual cost and effort of distributing and installing new software.

You'll find an introduction to the technology of applets in the online supplements to this book at *www.MindView.net*.

Swing basics

Most Swing applications will be built inside a basic **JFrame**, which creates the window in whatever operating system you're using. The title of the window can be set using the **JFrame** constructor, like this:

```
//: gui/HelloSwing.java
import javax.swing.*;

public class HelloSwing {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Hello Swing");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 100);
    frame.setVisible(true);
  }
} ///:~
```

setDefaultCloseOperation( ) tells the **JFrame** what to do when the user executes a shutdown maneuver. The **EXIT_ON_CLOSE** constant tells it to exit the program. Without this call, the default behavior is to do nothing, so the application wouldn't close.

setSize( ) sets the size of the window in pixels.

Notice the last line:

```
frame.setVisible(true);
```

Without this, you won't see anything on the screen.

We can make things a little more interesting by adding a **JLabel** to the **JFrame:**

```
//: gui/HelloLabel.java
import javax.swing.*;
import java.util.concurrent.*;

public class HelloLabel {
  public static void main(String[] args) throws Exception {
    JFrame frame = new JFrame("Hello Swing");
    JLabel label = new JLabel("A Label");
    frame.add(label);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 100);
    frame.setVisible(true);
    TimeUnit.SECONDS.sleep(1);
    label.setText("Hey! This is Different!");
  }
} ///:~
```

After one second, the text of the **JLabel** changes. While this is entertaining and safe for such a trivial program, it's really not a good idea for the **main( )** thread to write directly to the GUI components. Swing has its own thread dedicated to receiving UI events and updating the screen. If you start manipulating the screen with other threads, you can have the collisions and deadlock described in the *Concurrency* chapter.

Instead, other threads—like **main( )**, here—should submit tasks to be executed by the Swing *event dispatch thread.*[4] You do this by handing a task to **SwingUtilities.invokeLater( )**, which puts it on the *event queue* to be (eventually) executed by the event dispatch thread. If we do this with the previous example, it looks like this:

```
//: gui/SubmitLabelManipulationTask.java
import javax.swing.*;
import java.util.concurrent.*;

public class SubmitLabelManipulationTask {
  public static void main(String[] args) throws Exception {
    JFrame frame = new JFrame("Hello Swing");
    final JLabel label = new JLabel("A Label");
    frame.add(label);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 100);
    frame.setVisible(true);
    TimeUnit.SECONDS.sleep(1);
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        label.setText("Hey! This is Different!");
      }
    });
  }
} ///:~
```

Now you are no longer manipulating the **JLabel** directly. Instead, you submit a **Runnable**, and the event dispatch thread will do the actual manipulation, when it gets to that task in the event queue. And when it's executing this **Runnable**, it's not doing anything else, so there won't be any collisions—*if* all the code in your program follows this approach of submitting manipulations through **SwingUtilities.invokeLater( )**. This includes starting the program itself—**main( )** should not call the Swing methods as it does in the above program, but

[4] Technically, the event dispatch thread comes from the AWT library.

instead should submit a task to the event queue.[5] So the properly written program will look something like this:

```
//: gui/SubmitSwingProgram.java
import javax.swing.*;
import java.util.concurrent.*;

public class SubmitSwingProgram extends JFrame {
  JLabel label;
  public SubmitSwingProgram() {
    super("Hello Swing");
    label = new JLabel("A Label");
    add(label);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(300, 100);
    setVisible(true);
  }
  static SubmitSwingProgram ssp;
  public static void main(String[] args) throws Exception {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() { ssp = new SubmitSwingProgram(); }
    });
    TimeUnit.SECONDS.sleep(1);
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        ssp.label.setText("Hey! This is Different!");
      }
    });
  }
} ///:~
```

Notice that the call to **sleep( )** is *not* inside the constructor. If you put it there, the original **JLabel** text never appears, for one thing, because the constructor doesn't complete until after the **sleep( )** finishes and the new label is inserted. But if **sleep( )** is inside the constructor, or inside any UI operation, it means that you're halting the event dispatch thread during the **sleep( )**, which is generally a bad idea.

Exercise 1: (1) Modify **HelloSwing.java** to prove to yourself that the application will not close without the call to **setDefaultCloseOperation( )**.

Exercise 2: (2) Modify **HelloLabel.java** to show that label addition is dynamic, by adding a random number of labels.

A display framework

We can combine the ideas above and reduce redundant code by creating a display framework for use in the Swing examples in the rest of this chapter:

```
//: net/mindview/util/SwingConsole.java
// Tool for running Swing demos from the
// console, both applets and JFrames.
package net.mindview.util;
import javax.swing.*;

public class SwingConsole {
  public static void
```

[5] This practice was added in Java SE5, so you will see lots of older programs that don't do it. That doesn't mean the authors were ignorant. The suggested practices seem to be constantly evolving.

```
  run(final JFrame f, final int width, final int height) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        f.setTitle(f.getClass().getSimpleName());
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(width, height);
        f.setVisible(true);
      }
    });
  }
} ///:~
```

This is a tool you may want to use yourself, so it's placed in the library **net.mindview.util**. To use it, your application must be in a **JFrame** (which all the examples in this book are). The **static run( )** method sets the title of the window to the simple class name of the **JFrame**.

Exercise 3: (3) Modify **SubmitSwingProgram.java** so that it uses **SwingConsole**.

Making a button

Making a button is quite simple: You just call the **JButton** constructor with the label you want on the button. You'll see later that you can do fancier things, like putting graphic images on buttons.

Usually, you'll want to create a field for the button inside your class so that you can refer to it later.

The **JButton** is a component—its own little window—that will automatically get repainted as part of an update. This means that you don't explicitly paint a button or any other kind of control; you simply place them on the form and let them automatically take care of painting themselves. You'll usually place a button on a form inside the constructor:

```
//: gui/Button1.java
// Putting buttons on a Swing application.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Button1 extends JFrame {
  private JButton
    b1 = new JButton("Button 1"),
    b2 = new JButton("Button 2");
  public Button1() {
    setLayout(new FlowLayout());
    add(b1);
    add(b2);
  }
  public static void main(String[] args) {
    run(new Button1(), 200, 100);
  }
} ///:~
```

Something new has been added here: Before any elements are placed on the **JFrame**, it is given a "layout manager," of type **FlowLayout**. The layout manager is the way that the pane implicitly decides where to place controls on a form. The normal behavior of a **JFrame** is to use the **BorderLayout**, but that won't work here because (as you will learn later in this chapter) it defaults to covering each control entirely with every new one that is added.

However, **FlowLayout** causes the controls to flow evenly onto the form, left to right and top to bottom.

Exercise 4: (1) Verify that without the **setLayout( )** call in **Buttoni.java**, only one button will appear in the resulting program.

Capturing an event

If you compile and run the preceding program, nothing happens when you press the buttons. This is where you must step in and write some code to determine what will happen. The basis of event-driven programming, which comprises a lot of what a GUI is about, is connecting events to the code that responds to those events.

The way this is accomplished in Swing is by cleanly separating the interface (the graphical components) from the implementation (the code that you want to run when an event happens to a component). Each Swing component can report all the events that might happen to it, and it can report each kind of event individually. So if you're not interested in, for example, whether the mouse is being moved over your button, you don't register your interest in that event. It's a very straightforward and elegant way to handle event-driven programming, and once you understand the basic concepts, you can easily use Swing components that you haven't seen before—in fact, this model extends to anything that can be classified as a JavaBean (discussed later in the chapter).

At first, we will just focus on the main event of interest for the components being used. In the case of a **JButton**, this "event of interest" is that the button is pressed. To register your interest in a button press, you call the **JButton**'s **addActionListener( )** method. This method expects an argument that is an object that implements the **ActionListener** interface. That interface contains a single method called **actionPerformed( )**. So to attach code to a **JButton**, implement the **ActionListener** interface in a class, and register an object of that class with the **JButton** via **addActionListener( )**. The **actionPerformed( )** method will then be called when the button is pressed (this is normally referred to as a *callback)*.

But what should the result of pressing that button be? We'd like to see something change on the screen, so a new Swing component will be introduced: the **JTextField**. This is a place where text can be typed by the end user or, in this case, inserted by the program. Although there are a number of ways to create a **JTextField**, the simplest is just to tell the constructor how wide you want that field to be. Once the **JTextField** is placed on the form, you can modify its contents by using the **setText( )** method (there are many other methods in **JTextField**, but you must look these up in the JDK documentation from *http://java.sun.com)*. Here is what it looks like:

```
//: gui/Button2.java
// Responding to button presses.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Button2 extends JFrame {
  private JButton
    b1 = new JButton("Button 1"),
    b2 = new JButton("Button 2");
  private JTextField txt = new JTextField(10);
  class ButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String name = ((JButton)e.getSource()).getText();
      txt.setText(name);
```

```
    }
  }
  private ButtonListener bl = new ButtonListener();
  public Button2() {
    b1.addActionListener(bl);
    b2.addActionListener(bl);
    setLayout(new FlowLayout());
    add(b1);
    add(b2);
    add(txt);
  }
  public static void main(String[] args) {
    run(new Button2(), 200, 150);
  }
} ///:~
```

Creating a **JTextField** and placing it on the canvas takes the same steps as for **JButtons** or for any Swing component. The difference in the preceding program is in the creation of the aforementioned **ActionListener** class **ButtonListener**. The argument to **actionPerformed( )** is of type **ActionEvent**, which contains all the information about the event and where it came from. In this case, I wanted to describe the button that was pressed; **getSource( )** produces the object where the event originated, and I assumed (using a cast) that the object is a **JButton**. **getText( )** returns the text that's on the button, and this is placed in the **JTextField** to prove that the code was actually called when the button was pressed.

In the constructor, **addActionListener( )** is used to register the **ButtonListener** object with both the buttons.

It is often more convenient to code the **ActionListener** as an anonymous inner class, especially since you tend to use only a single instance of each listener class. **Button2.java** can be modified to use an anonymous inner class as follows:

```
//: gui/Button2b.java
// Using anonymous inner classes.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Button2b extends JFrame {
  private JButton
    b1 = new JButton("Button 1"),
    b2 = new JButton("Button 2");
  private JTextField txt = new JTextField(10);
  private ActionListener bl = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      String name = ((JButton)e.getSource()).getText();
      txt.setText(name);
    }
  };
  public Button2b() {
    b1.addActionListener(bl);
    b2.addActionListener(bl);
    setLayout(new FlowLayout());
    add(b1);
    add(b2);
    add(txt);
  }
  public static void main(String[] args) {
    run(new Button2b(), 200, 150);
```

```
  }
} ///:~
```

The approach of using an anonymous inner class will be preferred (when possible) for the examples in this book.

Exercise 5: (4) Create an application using the **SwingConsole** class. Include one text field and three buttons. When you press each button, make different text appear in the text field.

Text areas

A **JTextArea** is like a **JTextField** except that it can have multiple lines and has more functionality. A particularly useful method is **append( )**; with this you can easily pour output into the **JTextArea**. Because you can scroll backwards, this is an improvement over command-line programs that print to standard output. As an example, the following program fills a **JTextArea** with the output from the **Countries** generator in the *Containers in Depth* chapter:

```
//: gui/TextArea.java
// Using the JTextArea control.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

public class TextArea extends JFrame {
  private JButton
    b = new JButton("Add Data"),
    c = new JButton("Clear Data");
  private JTextArea t = new JTextArea(20, 40);
  private Map<String,String> m =
    new HashMap<String,String>();
  public TextArea() {
    // Use up all the data:
    m.putAll(Countries.capitals());
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        for(Map.Entry me : m.entrySet())
          t.append(me.getKey() + ": "+ me.getValue()+"\n");
      }
    });
    c.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        t.setText("");
      }
    });
    setLayout(new FlowLayout());
    add(new JScrollPane(t));
    add(b);
    add(c);
  }
  public static void main(String[] args) {
    run(new TextArea(), 475, 425);
  }
} ///:~
```

In the constructor, the **Map** is filled with all the countries and their capitals. Note that for both buttons, the **ActionListener** is created and added without defining an intermediate variable, since you never need to refer to that listener again during the program. The "Add Data" button formats and appends all the data, and the "Clear Data" button uses **setText( )** to remove all the text from the **JTextArea**.

As the **JTextArea** is added to the **JFrame**, it is wrapped in a **JScrollPane** to control scrolling when too much text is placed on the screen. That's all you must do in order to produce full scrolling capabilities. Having tried to figure out how to do the equivalent in some other GUI programming environments, I am very impressed with the simplicity and good design of components like **JScrollPane**.

Exercise 6: (7) Turn **strings/TestRegularExpression.java** into an interactive Swing program that allows you to put an input string in one **JTextArea** and a regular expression in a **JTextField**. The results should be displayed in a second **JTextArea**.

Exercise 7: (5) Create an application using **SwingConsole**, and add all the Swing components that have an **addActionListener( )** method. (Look these up in the JDK documentation from *http://java.sun.com*. Hint: Search for **addActionListener( )** using the index.) Capture their events and display an appropriate message for each inside a text field.

Exercise 8: (6) Almost every Swing component is derived from **Component**, which has a **setCursor( )** method. Look this up in the JDK documentation. Create an application and change the cursor to one of the stock cursors in the **Cursor** class.

Controlling layout

The way that you place components on a form in Java is probably different from any other GUI system you've used. First, it's all code; there are no "resources" that control placement of components. Second, the way components are placed on a form is controlled not by absolute positioning but by a "layout manager" that decides how the components lie based on the order that you **add( )** them. The size, shape, and placement of components will be remarkably different from one layout manager to another. In addition, the layout managers adapt to the dimensions of your applet or application window, so if the window dimension is changed, the size, shape, and placement of the components can change in response.

JApplet, **JFrame**, **JWindow**, **JDialog**, **JPanel**, etc., can all contain and display **Component**s. **In Container**, there's a method called **setLayout( )** that allows you to choose a different layout manager. In this section we'll explore the various layout managers by placing buttons in them (since that's the simplest thing to do). These examples won't capture the button events because they are only intended to show how the buttons are laid out.

BorderLayout

Unless you tell it otherwise, a **JFrame** will use **BorderLayout** as its default layout scheme. Without any other instruction, this takes whatever you **add( )** to it and places it in the center, stretching the object all the way out to the edges.

BorderLayout has the concept of four border regions and a center area. When you add something to a panel that's using a **BorderLayout**, you can use the overloaded **add( )** method that takes a constant value as its first argument. This value can be any of the following:

BorderLayout.NORTH	Top
BorderLayout.SOUTH	Bottom
BorderLayout.EAST	Right
BorderLayout.WEST	Left
BorderLayout.CENTER	Fill the middle, up to the other components or to the edges

If you don't specify an area to place the object, it defaults to **CENTER**.

In this example, the default layout is used, since **JFrame** defaults to **BorderLayout**:

```
//: gui/BorderLayout1.java
// Demonstrates BorderLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class BorderLayout1 extends JFrame {
  public BorderLayout1() {
    add(BorderLayout.NORTH, new JButton("North"));
    add(BorderLayout.SOUTH, new JButton("South"));
    add(BorderLayout.EAST, new JButton("East"));
    add(BorderLayout.WEST, new JButton("West"));
    add(BorderLayout.CENTER, new JButton("Center"));
  }
  public static void main(String[] args) {
    run(new BorderLayout1(), 300, 250);
  }
} ///:~
```

For every placement but **CENTER**, the element that you add is compressed to fit in the smallest amount of space along one dimension while it is stretched to the maximum along the other dimension. **CENTER**, however, spreads out in both dimensions to occupy the middle.

FlowLayout

This simply "flows" the components onto the form, from left to right until the top space is full, then moves down a row and continues flowing.

Here's an example that sets the layout manager to **FlowLayout** and then places buttons on the form. You'll notice that with **FlowLayout**, the components take on their "natural" size. A **JButton**, for example, will be the size of its string.

```
//: gui/FlowLayout1.java
// Demonstrates FlowLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class FlowLayout1 extends JFrame {
  public FlowLayout1() {
    setLayout(new FlowLayout());
    for(int i = 0; i < 20; i++)
      add(new JButton("Button " + i));
  }
```

```
  public static void main(String[] args) {
    run(new FlowLayout1(), 300, 300);
  }
} ///:~
```

All components will be compacted to their smallest size in a **FlowLayout**, so you might get a little bit of surprising behavior. For example, because a **JLabel** will be the size of its string, attempting to right-justify its text yields an unchanged display when using **FlowLayout**.

Notice that if you resize the window, the layout manager will reflow the components accordingly.

GridLayout

A **GridLayout** allows you to build a table of components, and as you add them, they are placed left to right and top to bottom in the grid. In the constructor, you specify the number of rows and columns that you need, and these are laid out in equal proportions.

```
//: gui/GridLayout1.java
// Demonstrates GridLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class GridLayout1 extends JFrame {
  public GridLayout1() {
    setLayout(new GridLayout(7,3));
    for(int i = 0; i < 20; i++)
      add(new JButton("Button " + i));
  }
  public static void main(String[] args) {
    run(new GridLayout1(), 300, 300);
  }
} ///:~
```

In this case there are 21 slots but only 20 buttons. The last slot is left empty because no "balancing" goes on with a **GridLayout**.

GridBagLayout

The **GridBagLayout** provides you with tremendous control in deciding exactly how the regions of your window will lay themselves out and reformat themselves when the window is resized. However, it's also the most complicated layout manager, and is quite difficult to understand. It is intended primarily for automatic code generation by a GUI builder (GUI builders might use **GridBagLayout** instead of absolute placement). If your design is so complicated that you feel you need to use **GridBagLayout**, then you should be using a GUI builder tool to generate that design. If you feel you must know the intricate details, I'll refer you to one of the dedicated Swing books as a starting point.

As an alternative, you may want to consider **TableLayout**, which is *not* part of the Swing library but which can be downloaded from *http://java.sun.com*. This component is layered on top of **GridBagLayout** and hides most of its complexity, so it can greatly simplify this approach.

Absolute positioning

It is also possible to set the absolute position of the graphical components:

1. Set a **null** layout manager for your **Container: setLayout(null)**.

2. Call **setBounds( )** or **reshape( )** (depending on the language version) for each component, passing a bounding rectangle in pixel coordinates. You can do this in the constructor or in **paint( )**, depending on what you want to achieve.

Some GUI builders use this approach extensively, but this is usually not the best way to generate code.

BoxLayout

Because people had so much trouble understanding and working with **GridBagLayout**, Swing also includes **BoxLayout**, which gives you many of the benefits of **GridBagLayout** without the complexity. You can often use it when you need to do hand-coded layouts (again, if your design becomes too complex, use a GUI builder that generates layouts for you). **BoxLayout** allows you to control the placement of components either vertically or horizontally, and to control the space between the components using something called "struts and glue." You can find some basic examples of **BoxLayout** in the online supplements for this book at *www.MindView.net*.

The best approach?

Swing is powerful; it can get a lot done with a few lines of code. The examples shown in this book are quite simple, and for learning purposes it makes sense to write them by hand. You can actually accomplish quite a bit by combining simple layouts. At some point, however, it stops making sense to hand-code GUI forms; it becomes too complicated and is not a good use of your programming time. The Java and Swing designers oriented the language and libraries to support GUI-building tools, which have been created for the express purpose of making your programming experience easier. As long as you understand what's going on with layouts and how to deal with events (described next), it's not particularly important that you actually know the details of how to lay out components by hand; let the appropriate tool do that for you (Java is, after all, designed to increase programmer productivity).

The Swing event model

In the Swing event model, a component can initiate ("fire") an event. Each type of event is represented by a distinct class. When an event is fired, it is received by one or more "listeners," which act on that event. Thus, the source of an event and the place where the event is handled can be separate. Since you typically use Swing components as they are, but need to write custom code that is called when the components receive an event, this is an excellent example of the separation of interface from implementation.

Each event listener is an object of a class that implements a particular type of listener interface. So as a programmer, all you do is create a listener object and register it with the component that's firing the event. This registration is performed by calling an **addXXXListener( )** method in the event-firing component, in which "**XXX**" represents the type of event listened for. You can easily know what types of events can be handled by noticing the names of the "addListener" methods, and if you try to listen for the wrong events, you'll discover your mistake at compile time. You'll see later in the chapter that

JavaBeans also use the names of the "addListener" methods to determine what events a Bean can handle.

All of your event logic, then, will go inside a listener class. When you create a listener class, the sole restriction is that it must implement the appropriate interface. You can create a global listener class, but this is a situation in which inner classes tend to be quite useful, not only because they provide a logical grouping of your listener classes inside the UI or business logic classes they are serving, but also because an inner-class object keeps a reference to its parent object, which provides a nice way to call across class and subsystem boundaries.

All the examples so far in this chapter have been using the Swing event model, but the remainder of this section will fill out the details of that model.

Event and listener types

All Swing components include **addXXXListener( )** and **removeXXXListener( )** methods so that the appropriate types of listeners can be added and removed from each component. You'll notice that the "**XXX**" in each case also represents the argument for the method, for example, **addMyListener(MyListener m)**. The following table includes the basic associated events, listeners, and methods, along with the basic components that support those particular events by providing the **addXXXListener( )** and **removeXXXListener( )** methods. You should keep in mind that the event model is designed to be extensible, so you may encounter other events and listener types that are not covered in this table.

Event, listener interface, and add- and remove-methods	**Components supporting this event**
ActionEvent **ActionListener** **addActionListener( )** **removeActionListener( )**	**JButton, JList, JTextField, JMenuItem** and its derivatives including **JCheckBoxMenuItem, JMenu,** and **JRadioButtonMenuItem**
AdjustmentEvent **AdjustmentListener** **addAdjustmentListener( )** **removeAdjustmentListener( )**	**JScrollbar** and anything you create that implements the **Adjustable** interface
ComponentEvent **ComponentListener** **addComponentListener( )** **removeComponentListener( )**	***Component** and its derivatives, including **JButton, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollbar, JTextArea,** and **JTextField**
ContainerEvent **addContainerListener( )** **removeContainerListener( )**	**Container** and its derivatives, **JScrollPane**, **Window**, **JDialog**, **JFileDialog**, and **JFrame**
FocusEvent **FocusListener** **addFocusListener( )** **removeFocusListener( )**	**Component** and **derivatives***
KeyEvent **KeyListener** **addKeyListener( )** **removeKeyListener( )**	**Component** and **derivatives***

Event, listener interface, and add- and remove-methods	Components supporting this event
MouseEvent (for both clicks and motion) **MouseListener** **addMouseListener( )** **removeMouseListener( )**	**Component** and **derivatives***
MouseEvent[6] (for both clicks and motion) **MouseMotionListener** **addMouseMotionListener( )** **removeMouseMotionListener( )**	**Component** and **derivatives***
WindowEvent **WindowListener** **addWindowListener( )** **removeWindowListener( )**	**Window** and its derivatives, including **JDialog**, **JFileDialog**, and **JFrame**
ItemEvent **ItemListener** **addItemListener( )** **removeItemListener( )**	**JCheckBox**, **JCheckBoxMenuItem**, **JComboBox**, **JList**, and anything that implements the **ItemSelectable** interface
TextEvent **TextListener** **addTextListener( )** **removeTextListener( )**	Anything derived from **JTextComponent**, including **JTextArea** and **JTextField**

You can see that each type of component supports only certain types of events. It turns out to be rather tedious to look up all the events supported by each component. A simpler approach is to modify the **ShowMethods.java** program from the *Type Information* chapter so that it displays all the event listeners supported by any Swing component that you enter.

The *Type Information* chapter introduced *reflection* and used that feature to look up methods for a particular class—either the entire list of methods or a subset of those whose names match a keyword that you provide. The magic of reflection is that it can automatically show you *all* the methods for a class without forcing you to walk up the inheritance hierarchy, examining the base classes at each level. Thus, it provides a valuable timesaving tool for programming; because the names of most Java methods are made nicely verbose and descriptive, you can search for the method names that contain a particular word of interest. When you find what you think you're looking for, check the JDK documentation.

Here is the more useful GUI version of **ShowMethods.java**, specialized to look for the "addListener" methods in Swing components:

```
//: gui/ShowAddListeners.java
// Display the "addXXXListener" methods of any Swing class.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.regex.*;
import static net.mindview.util.SwingConsole.*;

public class ShowAddListeners extends JFrame {
```

[6] There is no **MouseMotionEvent** even though it seems like there ought to be. Clicking and motion is combined into **MouseEvent**, so this second appearance of **MouseEvent** in the table is not an error.

```
  private JTextField name = new JTextField(25);
  private JTextArea results = new JTextArea(40, 65);
  private static Pattern addListener =
    Pattern.compile("(add\\w+?Listener\\(.*?\\))");
  private static Pattern qualifier =
    Pattern.compile("\\w+\\.");
  class NameL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String nm = name.getText().trim();
      if(nm.length() == 0) {
        results.setText("No match");
        return;
      }
      Class<?> kind;
      try {
        kind = Class.forName("javax.swing." + nm);
      } catch(ClassNotFoundException ex) {
        results.setText("No match");
        return;
      }
      Method[] methods = kind.getMethods();
      results.setText("");
      for(Method m : methods) {
        Matcher matcher =
          addListener.matcher(m.toString());
        if(matcher.find())
          results.append(qualifier.matcher(
            matcher.group(1)).replaceAll("") + "\n");
      }
    }
  }
  public ShowAddListeners() {
    NameL nameListener = new NameL();
    name.addActionListener(nameListener);
    JPanel top = new JPanel();
    top.add(new JLabel("Swing class name (press Enter):"));
    top.add(name);
    add(BorderLayout.NORTH, top);
    add(new JScrollPane(results));
    // Initial data and test:
    name.setText("JTextArea");
    nameListener.actionPerformed(
      new ActionEvent("", 0 ,""));
  }
  public static void main(String[] args) {
    run(new ShowAddListeners(), 500, 400);
  }
} ///:~
```

You enter the Swing class name that you want to look up in the **name JTextField**. The results are extracted using regular expressions, and displayed in a **JTextArea**.

You'll notice that there are no buttons or other components to indicate that you want the search to begin. That's because the **JTextField** is monitored by an **ActionListener**. Whenever you make a change and press Enter, the list is immediately updated. If the text field isn't empty, it is used inside **Class.forName( )** to try to look up the class. If the name is incorrect, **Class.forName( )** will fail, which means that it throws an exception. This is trapped, and the **JTextArea** is set to "No match." But if you type in a correct name (capitalization counts), **Class.forName( )** is successful, and **getMethods( )** will return an array of **Method** objects.

Two regular expressions are used here. The first, **addListener**, looks for "add" followed by any word characters, followed by "Listener" and the argument list in parentheses. Notice that this whole regular expression is surrounded by non-escaped parentheses, which means it will be accessible as a regular expression "group" when it matches. Inside **NameL.ActionPerformed( )**, a **Matcher** is created by passing each **Method** object to the **Pattern.matcher( )** method. When **find( )** is called for this **Matcher** object, it returns **true** only if a match occurs, and in that case you can select the first matching parenthesized group by calling **group(1)**. This string still contains qualifiers, so to strip them off, the **qualifier Pattern** object is used just as it was in **ShowMethods.java**.

At the end of the constructor, an initial value is placed in **name** and the action event is run to provide a test with initial data.

This program is a convenient way to investigate the capabilities of a Swing component. Once you know which events a particular component supports, you don't need to look anything up to react to that event. You simply:

1. Take the name of the event class and remove the word "**Event**." Add the word "**Listener**" to what remains. This is the listener interface you must implement in your inner class.

2. Implement the interface above and write out the methods for the events you want to capture. For example, you might be looking for mouse movements, so you write code for the **mouseMoved( )** method of the **MouseMotionListener** interface. (You must implement the other methods, of course, but there's often a shortcut for this, which you'll see soon.)

3. Create an object of the listener class in Step 2. Register it with your component with the method produced by prefixing "**add**" to your listener name. For example, **addMouseMotionListener( )**.

Here are some of the listener interfaces:

Listener interface w/ adapter	**Methods in interface**
ActionListener	**actionPerformed(ActionEvent)**
AdjustmentListener	**adjustmentValueChanged(AdjustmentEvent)**
ComponentListener ComponentAdapter	**componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent)**
ContainerListener ContainerAdapter	**componentAdded(ContainerEvent) componentRemoved(ContainerEvent)**
FocusListener FocusAdapter	**focusGained(FocusEvent) focusLost(FocusEvent)**
KeyListener KeyAdapter	**keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)**
MouseListener MouseAdapter	**mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)**

Listener interface w/ adapter	Methods in interface
MouseMotionListener **MouseMotionAdapter**	**mouseDragged(MouseEvent)** **mouseMoved(MouseEvent)**
WindowListener **WindowAdapter**	**windowOpened(WindowEvent)** **windowClosing(WindowEvent)** **windowClosed(WindowEvent)** **windowActivated(WindowEvent)** **windowDeactivated(WindowEvent)** **windowIconified(WindowEvent)** **windowDeiconified(WindowEvent)**
ItemListener	**itemStateChanged(ItemEvent)**

This is not an exhaustive listing, partly because the event model allows you to create your own event types and associated listeners. Thus, you'll regularly come across libraries that have invented their own events, and the knowledge gained in this chapter will allow you to figure out how to use these events.

Using listener adapters for simplicity

In the table above, you can see that some listener interfaces have only one method. These are trivial to implement. However, the listener interfaces that have multiple methods can be less pleasant to use. For example, if you want to capture a mouse click (that isn't already captured for you, for example, by a button), then you need to write a method for **mouseClicked( )**. But since **MouseListener** is an interface, you must implement all of the other methods even if they don't do anything. This can be annoying.

To solve the problem, some (but not all) of the listener interfaces that have more than one method are provided with *adapters*, the names of which you can see in the table above. Each adapter provides default empty methods for each of the interface methods. When you inherit from the adapter, you override only the methods you need to change. For example, the typical **MouseListener** you'll use looks like this:

```
class MyMouseListener extends MouseAdapter {
  public void mouseClicked(MouseEvent e) {
    // Respond to mouse click...
  }
}
```

The whole point of the adapters is to make the creation of listener classes easy.

There is a downside to adapters, however, in the form of a pitfall. Suppose you write a **MouseAdapter** like the previous one:

```
class MyMouseListener extends MouseAdapter {
  public void MouseClicked(MouseEvent e) {
    // Respond to mouse click...
  }
}
```

This doesn't work, but it will drive you crazy trying to figure out why, since everything will compile and run fine—except that your method won't be called for a mouse click. Can you see the problem? It's in the name of the method: **MouseClicked( )** instead of **mouseClicked( )**. A simple slip in capitalization results in the addition of a completely new method. However, this is not the method that's called when the mouse is clicked, so you don't get the

desired results. Despite the inconvenience, an interface will guarantee that the methods are properly implemented.

An improved alternative way to guarantee that you are in fact overriding a method is to use the built-in **@Override** annotation in the code above.

Exercise 9: (5) Starting with **ShowAddListeners.java**, create a program with the full functionality of **typeinfo.ShowMethods.java**.

Tracking multiple events

To prove to yourself that these events are in fact being fired, it's worth creating a program that tracks behavior in a **JButton** beyond whether it has been pressed. This example also shows you how to inherit your own button object from **JButton.**[7]

In the code below, the **MyButton** class is an inner class of **TrackEvent**, so **MyButton** can reach into the parent window and manipulate its text fields, which is necessary in order to write the status information into the fields of the parent. Of course, this is a limited solution, since **MyButton** can be used only in conjunction with **TrackEvent**. This kind of code is sometimes called "highly coupled":

```
//: gui/TrackEvent.java
// Show events as they happen.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

public class TrackEvent extends JFrame {
  private HashMap<String,JTextField> h =
    new HashMap<String,JTextField>();
  private String[] event = {
    "focusGained", "focusLost", "keyPressed",
    "keyReleased", "keyTyped", "mouseClicked",
    "mouseEntered", "mouseExited", "mousePressed",
    "mouseReleased", "mouseDragged", "mouseMoved"
  };
  private MyButton
    b1 = new MyButton(Color.BLUE, "test1"),
    b2 = new MyButton(Color.RED, "test2");
  class MyButton extends JButton {
    void report(String field, String msg) {
      h.get(field).setText(msg);
    }
    FocusListener fl = new FocusListener() {
      public void focusGained(FocusEvent e) {
        report("focusGained", e.paramString());
      }
      public void focusLost(FocusEvent e) {
        report("focusLost", e.paramString());
      }
    };
    KeyListener kl = new KeyListener() {
      public void keyPressed(KeyEvent e) {
        report("keyPressed", e.paramString());
```

[7] In Java 1.0/1.1 you could *not* usefully inherit from the button object. This was only one of numerous fundamental design flaws.

```
      }
      public void keyReleased(KeyEvent e) {
        report("keyReleased", e.paramString());
      }
      public void keyTyped(KeyEvent e) {
        report("keyTyped", e.paramString());
      }
    };
    MouseListener ml = new MouseListener() {
      public void mouseClicked(MouseEvent e) {
        report("mouseClicked", e.paramString());
      }
      public void mouseEntered(MouseEvent e) {
        report("mouseEntered", e.paramString());
      }
      public void mouseExited(MouseEvent e) {
        report("mouseExited", e.paramString());
      }
      public void mousePressed(MouseEvent e) {
        report("mousePressed", e.paramString());
      }
      public void mouseReleased(MouseEvent e) {
        report("mouseReleased", e.paramString());
      }
    };
    MouseMotionListener mml = new MouseMotionListener() {
      public void mouseDragged(MouseEvent e) {
        report("mouseDragged", e.paramString());
      }
      public void mouseMoved(MouseEvent e) {
        report("mouseMoved", e.paramString());
      }
    };
    public MyButton(Color color, String label) {
      super(label);
      setBackground(color);
      addFocusListener(fl);
      addKeyListener(kl);
      addMouseListener(ml);
      addMouseMotionListener(mml);
    }
  }
  public TrackEvent() {
    setLayout(new GridLayout(event.length + 1, 2));
    for(String evt : event) {
      JTextField t = new JTextField();
      t.setEditable(false);
      add(new JLabel(evt, JLabel.RIGHT));
      add(t);
      h.put(evt, t);
    }
    add(b1);
    add(b2);
  }
  public static void main(String[] args) {
    run(new TrackEvent(), 700, 500);
  }
} ///:~
```

In the **MyButton** constructor, the button's color is set with a call to **SetBackground( )**. The listeners are all installed with simple method calls.

The **TrackEvent** class contains a **HashMap** to hold the strings representing the type of event and **JTextField**s where information about that event is held. Of course, these could have been created statically rather than putting them in a **HashMap**, but I think you'll agree that it's a lot easier to use and change. In particular, if you need to add or remove a new type of event in **TrackEvent**, you simply add or remove a string in the **event** array— everything else happens automatically.

When **report( )** is called, it is given the name of the event and the parameter string from the event. It uses the **HashMap h** in the outer class to look up the actual **JTextField** associated with that event name and then places the parameter string into that field.

This example is fun to play with because you can really see what's going on with the events in your program.

Exercise 10: (6) Create an application using **SwingConsole**, with a **JButton** and a **JTextField**. Write and attach the appropriate listener so that if the button has the focus, characters typed into it will appear in the **JTextField**.

Exercise 11: (4) Inherit a new type of button from **JButton**. Each time you press this button, it should change its color to a randomly selected value. See **ColorBoxes.java** (later in this chapter) for an example of how to generate a random color value.

Exercise 12: (4) Monitor a new type of event in **TrackEvent.java** by adding the new event-handling code. You'll need to discover on your own the type of event that you want to monitor.

A selection of Swing components

Now that you understand layout managers and the event model, you're ready to see how Swing components can be used. This section is a non-exhaustive tour of the Swing components and features that you'll probably use most of the time. Each example is intended to be reasonably small so that you can easily lift the code and use it in your own programs.

Keep in mind:

1. You can easily see what each of these examples looks like during execution by compiling and running the downloadable source code for this chapter *(www.MindView.net)*.

2. The JDK documentation from *http://java.sun.com* contains all of the Swing classes and methods (only a few are shown here).

3. Because of the naming convention used for Swing events, it's fairly easy to guess how to write and install a handler for a particular type of event. Use the lookup program **ShowAddListeners.java** from earlier in this chapter to aid in your investigation of a particular component.

4. When things start to get complicated you should graduate to a GUI builder.

Buttons

Swing includes a number of different types of buttons. All buttons, check boxes, radio buttons, and even menu items are inherited from **AbstractButton** (which, since menu items are included, would probably have been better named "AbstractSelector" or something

equally general). You'll see the use of menu items shortly, but the following example shows the various types of buttons available:

```
//: gui/Buttons.java
// Various Swing buttons.
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.plaf.basic.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Buttons extends JFrame {
  private JButton jb = new JButton("JButton");
  private BasicArrowButton
    up = new BasicArrowButton(BasicArrowButton.NORTH),
    down = new BasicArrowButton(BasicArrowButton.SOUTH),
    right = new BasicArrowButton(BasicArrowButton.EAST),
    left = new BasicArrowButton(BasicArrowButton.WEST);
  public Buttons() {
    setLayout(new FlowLayout());
    add(jb);
    add(new JToggleButton("JToggleButton"));
    add(new JCheckBox("JCheckBox"));
    add(new JRadioButton("JRadioButton"));
    JPanel jp = new JPanel();
    jp.setBorder(new TitledBorder("Directions"));
    jp.add(up);
    jp.add(down);
    jp.add(left);
    jp.add(right);
    add(jp);
  }
  public static void main(String[] args) {
    run(new Buttons(), 350, 200);
  }
} ///:~
```

This begins with the **BasicArrowButton** from **javax.swing.plaf.basic**, then continues with the various specific types of buttons. When you run the example, you'll see that the toggle button holds its last position, in or out. But the check boxes and radio buttons behave identically to each other, just clicking on or off (they are inherited from **JToggleButton**).

Button groups

If you want radio buttons to behave in an "exclusive or" fashion, you must add them to a "button group." But, as the following example demonstrates, any **AbstractButton** can be added to a **ButtonGroup**.

To avoid repeating a lot of code, this example uses reflection to generate the groups of different types of buttons. This is seen in **makeBPanel( )**, which creates a button group in a **JPanel**. The second argument to **makeBPanel( )** is an array of **String**. For each **String**, a button of the class represented by the first argument is added to the **JPanel:**

```
//: gui/ButtonGroups.java
// Uses reflection to create groups
// of different types of AbstractButton.
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.lang.reflect.*;
```

```
import static net.mindview.util.SwingConsole.*;

public class ButtonGroups extends JFrame {
  private static String[] ids = {
    "June", "Ward", "Beaver", "Wally", "Eddie", "Lumpy"
  };
  static JPanel makeBPanel(
    Class<? extends AbstractButton> kind, String[] ids) {
    ButtonGroup bg = new ButtonGroup();
    JPanel jp = new JPanel();
    String title = kind.getName();
    title = title.substring(title.lastIndexOf('.') + 1);
    jp.setBorder(new TitledBorder(title));
    for(String id : ids) {
      AbstractButton ab = new JButton("failed");
      try {
        // Get the dynamic constructor method
        // that takes a String argument:
        Constructor ctor =
          kind.getConstructor(String.class);
        // Create a new object:
        ab = (AbstractButton)ctor.newInstance(id);
      } catch(Exception ex) {
        System.err.println("can't create " + kind);
      }
      bg.add(ab);
      jp.add(ab);
    }
    return jp;
  }
  public ButtonGroups() {
    setLayout(new FlowLayout());
    add(makeBPanel(JButton.class, ids));
    add(makeBPanel(JToggleButton.class, ids));
    add(makeBPanel(JCheckBox.class, ids));
    add(makeBPanel(JRadioButton.class, ids));
  }
  public static void main(String[] args) {
    run(new ButtonGroups(), 500, 350);
  }
} ///:~
```

The title for the border is taken from the name of the class, stripping off all the path information. The **AbstractButton** is initialized to a **JButton** that has the label "failed," so if you ignore the exception message, you'll still see the problem on the screen. The **getConstructor( )** method produces a **Constructor** object that takes the array of arguments of the types in the list of Classes passed to **getConstructor( )**. Then all you do is call **newInstance( )**, passing it a list of arguments—in this case, just the **String** from the **ids** array.

To get "exclusive or" behavior with buttons, you create a button group and add each button for which you want that behavior to the group. When you run the program, you'll see that all the buttons except **JButton** exhibit this "exclusive or" behavior.

Icons

You can use an **Icon** inside a **JLabel** or anything that inherits from **AbstractButton** (including **JButton**, **JCheckBox**, **JRadioButton**, and the different kinds of **JMenuItem**). Using **Icon**s with **JLabel**s is quite straightforward (you'll see an example later). The following example explores all the additional ways you can use **Icons** with buttons and their descendants.

You can use any **GIF** files you want, but the ones used in this example are part of this book's code distribution, available at *www.MindView.net*. To open a file and bring in the image, simply create an **ImageIcon** and hand it the file name. From then on, you can use the resulting **Icon** in your program.

```
//: gui/Faces.java
// Icon behavior in JButtons.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Faces extends JFrame {
  private static Icon[] faces;
  private JButton jb, jb2 = new JButton("Disable");
  private boolean mad = false;
  public Faces() {
    faces = new Icon[]{
      new ImageIcon(getClass().getResource("Face0.gif")),
      new ImageIcon(getClass().getResource("Face1.gif")),
      new ImageIcon(getClass().getResource("Face2.gif")),
      new ImageIcon(getClass().getResource("Face3.gif")),
      new ImageIcon(getClass().getResource("Face4.gif")),
    };
    jb = new JButton("JButton", faces[3]);
    setLayout(new FlowLayout());
    jb.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if(mad) {
          jb.setIcon(faces[3]);
          mad = false;
        } else {
          jb.setIcon(faces[0]);
          mad = true;
        }
        jb.setVerticalAlignment(JButton.TOP);
        jb.setHorizontalAlignment(JButton.LEFT);
      }
    });
    jb.setRolloverEnabled(true);
    jb.setRolloverIcon(faces[1]);
    jb.setPressedIcon(faces[2]);
    jb.setDisabledIcon(faces[4]);
    jb.setToolTipText("Yow!");
    add(jb);
    jb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if(jb.isEnabled()) {
          jb.setEnabled(false);
          jb2.setText("Enable");
        } else {
          jb.setEnabled(true);
          jb2.setText("Disable");
        }
      }
    });
    add(jb2);
  }
  public static void main(String[] args) {
    run(new Faces(), 250, 125);
  }
} ///:~
```

An **Icon** can be used as an argument for many different Swing component constructors, but you can also use **setIcon( )** to add or change an **Icon**. This example also shows how a **JButton** (or any **AbstractButton**) can set the various different sorts of icons that appear when things happen to that button: when it's pressed, disabled, or "rolled over" (the mouse moves over it without clicking). You'll see that this gives the button a nice animated feel.

Tool tips

The previous example added a "tool tip" to the button. Almost all of the classes that you'll be using to create your user interfaces are derived from **JComponent**, which contains a method called **setToolTipText(String)**. So, for virtually anything you place on your form, all you need to do is say (for an object j c of any **JComponent**-derived class):

```
jc.setToolTipText("My tip");
```

When the mouse stays over that **JComponent** for a predetermined period of time, a tiny box containing your text will pop up next to the mouse.

Text fields

This example shows what **JTextField**s can do:

```
//: gui/TextFields.java
// Text fields and Java events.
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class TextFields extends JFrame {
  private JButton
    b1 = new JButton("Get Text"),
    b2 = new JButton("Set Text");
  private JTextField
    t1 = new JTextField(30),
    t2 = new JTextField(30),
    t3 = new JTextField(30);
  private String s = "";
  private UpperCaseDocument ucd = new UpperCaseDocument();
  public TextFields() {
    t1.setDocument(ucd);
    ucd.addDocumentListener(new T1());
    b1.addActionListener(new B1());
    b2.addActionListener(new B2());
    t1.addActionListener(new T1A());
    setLayout(new FlowLayout());
    add(b1);
    add(b2);
    add(t1);
    add(t2);
    add(t3);
  }
  class T1 implements DocumentListener {
    public void changedUpdate(DocumentEvent e) {}
    public void insertUpdate(DocumentEvent e) {
      t2.setText(t1.getText());
      t3.setText("Text: "+ t1.getText());
```

```
    }
    public void removeUpdate(DocumentEvent e) {
      t2.setText(t1.getText());
    }
  }
  class T1A implements ActionListener {
    private int count = 0;
    public void actionPerformed(ActionEvent e) {
      t3.setText("t1 Action Event " + count++);
    }
  }
  class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(t1.getSelectedText() == null)
        s = t1.getText();
      else
        s = t1.getSelectedText();
      t1.setEditable(true);
    }
  }
  class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      ucd.setUpperCase(false);
      t1.setText("Inserted by Button 2: " + s);
      ucd.setUpperCase(true);
      t1.setEditable(false);
    }
  }
  public static void main(String[] args) {
    run(new TextFields(), 375, 200);
  }
}

class UpperCaseDocument extends PlainDocument {
  private boolean upperCase = true;
  public void setUpperCase(boolean flag) {
    upperCase = flag;
  }
  public void
  insertString(int offset, String str, AttributeSet attSet)
  throws BadLocationException {
    if(upperCase) str = str.toUpperCase();
    super.insertString(offset, str, attSet);
  }
} ///:~
```

The **JTextField t3** is included as a place to report when the action listener for the **JTextField t1** is fired. You'll see that the action listener for a **JTextField** is fired only when you press the Enter key.

The **JTextField t1** has several listeners attached to it. The **T1** listener is a **DocumentListener** that responds to any change in the "document" (the contents of the **JTextField**, in this case). It automatically copies all text from **t1** into **t2**. In addition, **t1**'s document is set to a derived class of **PlainDocument**, called **UpperCaseDocument**, which forces all characters to uppercase. It automatically detects backspaces and performs the deletion, adjusting the caret and handling everything as you expect.

Exercise 13: (3) Modify **TextFields.java** so that the characters in t2 retain the original case that they were typed in, instead of automatically being forced to uppercase.

Borders

JComponent contains a method called **setBorder( )**, which allows you to place various interesting borders on any visible component. The following example demonstrates a number of the different borders that are available, using a method called **showBorder( )** that creates a **JPanel** and puts on the border in each case. Also, it uses RTTI to find the name of the border that you're using (stripping off all the path information), then puts that name in a **JLabel** in the middle of the panel:

```
//: gui/Borders.java
// Different Swing borders.
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Borders extends JFrame {
  static JPanel showBorder(Border b) {
    JPanel jp = new JPanel();
    jp.setLayout(new BorderLayout());
    String nm = b.getClass().toString();
    nm = nm.substring(nm.lastIndexOf('.') + 1);
    jp.add(new JLabel(nm, JLabel.CENTER),
      BorderLayout.CENTER);
    jp.setBorder(b);
    return jp;
  }
  public Borders() {
    setLayout(new GridLayout(2,4));
    add(showBorder(new TitledBorder("Title")));
    add(showBorder(new EtchedBorder()));
    add(showBorder(new LineBorder(Color.BLUE)));
    add(showBorder(
      new MatteBorder(5,5,30,30,Color.GREEN)));
    add(showBorder(
      new BevelBorder(BevelBorder.RAISED)));
    add(showBorder(
      new SoftBevelBorder(BevelBorder.LOWERED)));
    add(showBorder(new CompoundBorder(
      new EtchedBorder(),
      new LineBorder(Color.RED))));
  }
  public static void main(String[] args) {
    run(new Borders(), 500, 300);
  }
} ///:~
```

You can also create your own borders and put them inside buttons, labels, etc.—anything derived from **JComponent**.

A mini-editor

The **JTextPane** control provides a great deal of support for editing, without much effort. The following example makes very simple use of this component, ignoring the bulk of its functionality:

```
//: gui/TextPane.java
// The JTextPane control is a little editor.
import javax.swing.*;
```

```
import java.awt.*;
import java.awt.event.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

public class TextPane extends JFrame {
  private JButton b = new JButton("Add Text");
  private JTextPane tp = new JTextPane();
  private static Generator sg =
    new RandomGenerator.String(7);
  public TextPane() {
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        for(int i = 1; i < 10; i++)
          tp.setText(tp.getText() + sg.next() + "\n");
      }
    });
    add(new JScrollPane(tp));
    add(BorderLayout.SOUTH, b);
  }
  public static void main(String[] args) {
    run(new TextPane(), 475, 425);
  }
} ///:~
```

The button adds randomly generated text. The intent of the **JTextPane** is to allow text to be edited in place, so you will see that there is no **append( )** method. In this case (admittedly, a poor use of the capabilities of **JTextPane**), the text must be captured, modified, and placed back into the pane using **setText( )**.

Elements are added to the **JFrame** using its default **BorderLayout**. The **JTextPane** is added (inside a **JScrollPane**) without specifying a region, so it just fills the center of the pane out to the edges. The **JButton** is added to the **SOUTH**, so the component will fit itself into that region; in this case, the button will nest down at the bottom of the screen.

Notice the built-in features of **JTextPane**, such as automatic line wrapping. There are numerous other features that you can look up using the JDK documentation.

Exercise 14: (2) Modify **TextPane.java** to use a **JTextArea** instead of a **JTextPane**.

Check boxes

A check box provides a way to make a single on/off choice. It consists of a tiny box and a label. The box typically holds a little "x" (or some other indication that it is set) or is empty, depending on whether that item was selected.

You'll normally create a **JCheckBox** using a constructor that takes the label as an argument. You can get and set the state, and also get and set the label if you want to read or change it after the **JCheckBox** has been created.

Whenever a **JCheckBox** is set or cleared, an event occurs, which you can capture the same way you do a button: by using an **ActionListener**. The following example uses a **JTextArea** to enumerate all the check boxes that have been checked:

```
//: gui/CheckBoxes.java
// Using JCheckBoxes.
import javax.swing.*;
import java.awt.*;
```

```
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class CheckBoxes extends JFrame {
  private JTextArea t = new JTextArea(6, 15);
  private JCheckBox
    cb1 = new JCheckBox("Check Box 1"),
    cb2 = new JCheckBox("Check Box 2"),
    cb3 = new JCheckBox("Check Box 3");
  public CheckBoxes() {
    cb1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        trace("1", cb1);
      }
    });
    cb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        trace("2", cb2);
      }
    });
    cb3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        trace("3", cb3);
      }
    });
    setLayout(new FlowLayout());
    add(new JScrollPane(t));
    add(cb1);
    add(cb2);
    add(cb3);
  }
  private void trace(String b, JCheckBox cb) {
    if(cb.isSelected())
      t.append("Box " + b + " Set\n");
    else
      t.append("Box " + b + " Cleared\n");
  }
  public static void main(String[] args) {
    run(new CheckBoxes(), 200, 300);
  }
} ///:~
```

The **trace( )** method sends the name of the selected **JCheckBox** and its current state to the **JTextArea** using **append( )**, so you'll see a cumulative list of the check boxes that were selected, along with their state.

Exercise 15: (5) Add a check box to the application created in Exercise 5, capture the event, and insert different text into the text field.

Radio buttons

The concept of radio buttons in GUI programming comes from pre-electronic car radios with mechanical buttons: When you push one in, any other buttons pop out. Thus, it allows you to force a single choice among many.

To set up an associated group of **JRadioButton**s, you add them to a **ButtonGroup** (you can have any number of **ButtonGroup**s on a form). One of the buttons can be optionally set to **true** (using the second argument in the constructor). If you try to set more than one radio button to **true**, then only the last one set will be **true**.

Here's a simple example of the use of radio buttons, showing event capture using an **ActionListener:**

```
//: gui/RadioButtons.java
// Using JRadioButtons.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class RadioButtons extends JFrame {
  private JTextField t = new JTextField(15);
  private ButtonGroup g = new ButtonGroup();
  private JRadioButton
    rb1 = new JRadioButton("one", false),
    rb2 = new JRadioButton("two", false),
    rb3 = new JRadioButton("three", false);
  private ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      t.setText("Radio button " +
        ((JRadioButton)e.getSource()).getText());
    }
  };
  public RadioButtons() {
    rb1.addActionListener(al);
    rb2.addActionListener(al);
    rb3.addActionListener(al);
    g.add(rb1); g.add(rb2); g.add(rb3);
    t.setEditable(false);
    setLayout(new FlowLayout());
    add(t);
    add(rb1);
    add(rb2);
    add(rb3);
  }
  public static void main(String[] args) {
    run(new RadioButtons(), 200, 125);
  }
} ///:~
```

To display the state, a text field is used. This field is set to non-editable because it's used only to display data, not to collect it. Thus it is an alternative to using a **JLabel**.

Combo boxes (drop-down lists)

Like a group of radio buttons, a drop-down list is a way to force the user to select only one element from a group of possibilities. However, it's a more compact way to accomplish this, and it's easier to change the elements of the list without surprising the user. (You can change radio buttons dynamically, but that tends to be visibly jarring.)

By default, **JComboBox** box is not like the combo box in Windows, which lets you select from a list *or* type in your own selection. To produce this behavior you must call **setEditable( )**. With a **JComboBox** box, you choose one and only one element from the list. In the following example, the **JComboBox** box starts with a certain number of entries, and then new entries are added to the box when a button is pressed.

```
//: gui/ComboBoxes.java
// Using drop-down lists.
import javax.swing.*;
import java.awt.*;
```

```
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class ComboBoxes extends JFrame {
  private String[] description = {
    "Ebullient", "Obtuse", "Recalcitrant", "Brilliant",
    "Somnescent", "Timorous", "Florid", "Putrescent"
  };
  private JTextField t = new JTextField(15);
  private JComboBox c = new JComboBox();
  private JButton b = new JButton("Add items");
  private int count = 0;
  public ComboBoxes() {
    for(int i = 0; i < 4; i++)
      c.addItem(description[count++]);
    t.setEditable(false);
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if(count < description.length)
          c.addItem(description[count++]);
      }
    });
    c.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        t.setText("index: "+ c.getSelectedIndex() + "   " +
         ((JComboBox)e.getSource()).getSelectedItem());
      }
    });
    setLayout(new FlowLayout());
    add(t);
    add(c);
    add(b);
  }
  public static void main(String[] args) {
    run(new ComboBoxes(), 200, 175);
  }
} ///:~
```

The **JTextField** displays the "selected index," which is the sequence number of the currently selected element, as well as the text of the selected item in the combo box.

List boxes

List boxes are significantly different from **JComboBox** boxes, and not just in appearance. While a **JComboBox** box drops down when you activate it, a **JList** occupies some fixed number of lines on a screen all the time and doesn't change. If you want to see the items in a list, you simply call **getSelectedValues( )**, which produces an array of **String** of the items that have been selected.

A **JList** allows multiple selection; if you control-click on more than one item (holding down the Control key while performing additional mouse clicks), the original item stays highlighted and you can select as many as you want. If you select an item, then shift-click on another item, all the items in the span between the two are selected. To remove an item from a group, you can control-click it.

```
//: gui/List.java
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import java.awt.*;
```

```
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class List extends JFrame {
  private String[] flavors = {
    "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
    "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie"
  };
  private DefaultListModel lItems = new DefaultListModel();
  private JList lst = new JList(lItems);
  private JTextArea t =
    new JTextArea(flavors.length, 20);
  private JButton b = new JButton("Add Item");
  private ActionListener bl = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      if(count < flavors.length) {
        lItems.add(0, flavors[count++]);
      } else {
        // Disable, since there are no more
        // flavors left to be added to the List
        b.setEnabled(false);
      }
    }
  };
  private ListSelectionListener ll =
    new ListSelectionListener() {
      public void valueChanged(ListSelectionEvent e) {
        if(e.getValueIsAdjusting()) return;
        t.setText("");
        for(Object item : lst.getSelectedValues())
          t.append(item + "\n");
      }
    };
  private int count = 0;
  public List() {
    t.setEditable(false);
    setLayout(new FlowLayout());
    // Create Borders for components:
    Border brd = BorderFactory.createMatteBorder(
      1, 1, 2, 2, Color.BLACK);
    lst.setBorder(brd);
    t.setBorder(brd);
    // Add the first four items to the List
    for(int i = 0; i < 4; i++)
      lItems.addElement(flavors[count++]);
    add(t);
    add(lst);
    add(b);
    // Register event listeners
    lst.addListSelectionListener(ll);
    b.addActionListener(bl);
  }
  public static void main(String[] args) {
    run(new List(), 250, 375);
  }
} ///:~
```

You can see that borders have also been added to the lists.

If you just want to put an array of **String**s into a **JList**, there's a much simpler solution; you pass the array to the **JList** constructor, and it builds the list automatically. The only reason

for using the "list model" in the preceding example is so that the list can be manipulated during the execution of the program.

JLists do not automatically provide direct support for scrolling. Of course, all you need to do is wrap the **JList** in a **JScrollPane**, and the details are automatically managed for you.

Exercise 16: (5) Simplify **List.java** by passing the array to the constructor and eliminating the dynamic addition of elements to the list.

Tabbed panes

The **JTabbedPane** allows you to create a "tabbed dialog," which has filefolder tabs running across one edge. When you press a tab, it brings forward a different dialog.

```
//: gui/TabbedPane1.java
// Demonstrates the Tabbed Pane.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class TabbedPane1 extends JFrame {
  private String[] flavors = {
    "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
    "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie"
  };
  private JTabbedPane tabs = new JTabbedPane();
  private JTextField txt = new JTextField(20);
  public TabbedPane1() {
    int i = 0;
    for(String flavor : flavors)
      tabs.addTab(flavors[i],
        new JButton("Tabbed pane " + i++));
    tabs.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        txt.setText("Tab selected: " +
          tabs.getSelectedIndex());
      }
    });
    add(BorderLayout.SOUTH, txt);
    add(tabs);
  }
  public static void main(String[] args) {
    run(new TabbedPane1(), 400, 250);
  }
} ///:~
```

When you run the program, you'll see that the **JTabbedPane** automatically stacks the tabs if there are too many of them to fit on one row. You can see this by resizing the window when you run the program from the console command line.

Message boxes

Windowing environments commonly contain a standard set of message boxes that allow you to quickly post information to the user or to capture information from the user. In Swing, these message boxes are contained in **JOptionPane**. You have many different possibilities (some quite sophisticated), but the ones you'll most commonly use are probably the message

dialog and confirmation dialog, invoked using the **static JOptionPane.showMessageDialog( )** and **JOptionPane.showConfirmDialog( )**. The following example shows a subset of the message boxes available with **JOptionPane:**

```
//: gui/MessageBoxes.java
// Demonstrates JOptionPane.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class MessageBoxes extends JFrame {
  private JButton[] b = {
    new JButton("Alert"), new JButton("Yes/No"),
    new JButton("Color"), new JButton("Input"),
    new JButton("3 Vals")
  };
  private JTextField txt = new JTextField(15);
  private ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      String id = ((JButton)e.getSource()).getText();
      if(id.equals("Alert"))
        JOptionPane.showMessageDialog(null,
          "There's a bug on you!", "Hey!",
          JOptionPane.ERROR_MESSAGE);
      else if(id.equals("Yes/No"))
        JOptionPane.showConfirmDialog(null,
          "or no", "choose yes",
          JOptionPane.YES_NO_OPTION);
      else if(id.equals("Color")) {
        Object[] options = { "Red", "Green" };
        int sel = JOptionPane.showOptionDialog(
          null, "Choose a Color!", "Warning",
          JOptionPane.DEFAULT_OPTION,
          JOptionPane.WARNING_MESSAGE, null,
          options, options[0]);
        if(sel != JOptionPane.CLOSED_OPTION)
          txt.setText("Color Selected: " + options[sel]);
      } else if(id.equals("Input")) {
        String val = JOptionPane.showInputDialog(
            "How many fingers do you see?");
        txt.setText(val);
      } else if(id.equals("3 Vals")) {
        Object[] selections = {"First", "Second", "Third"};
        Object val = JOptionPane.showInputDialog(
          null, "Choose one", "Input",
          JOptionPane.INFORMATION_MESSAGE,
          null, selections, selections[0]);
        if(val != null)
          txt.setText(val.toString());
      }
    }
  };
  public MessageBoxes() {
    setLayout(new FlowLayout());
    for(int i = 0; i < b.length; i++) {
      b[i].addActionListener(al);
      add(b[i]);
    }
    add(txt);
  }
  public static void main(String[] args) {
    run(new MessageBoxes(), 200, 200);
```

```
  }
} ///:~
```

To write a single **ActionListener**, I've used the somewhat risky approach of checking the **String** labels on the buttons. The problem with this is that it's easy to get the label a little bit wrong, typically in capitalization, and this bug can be hard to spot.

Note that **showOptionDialog( )** and **showInputDialog( )** provide return objects that contain the value entered by the user.

Exercise 17: (5) Create an application using **SwingConsole**. In the JDK documentation from *http://java.sun.com*, find the **JPasswordField** and add this to the program. If the user types in the correct password, use **JOptionPane** to provide a success message to the user.

Exercise 18: (4) Modify **MessageBoxes.java** so that it has an individual **ActionListener** for each button (instead of matching the button text).

Menus

Each component capable of holding a menu, including **JApplet**, **JFrame**, **JDialog**, and their descendants, has a **setJMenuBar( )** method that accepts a **JMenuBar** (you can have only one **JMenuBar** on a particular component). You add **JMenus** to the **JMenuBar**, and **JMenuItem**s to the **JMenu**s. Each **JMenuItem** can have an **ActionListener** attached to it, to be fired when that menu item is selected.

With Java and Swing you must hand assemble all the menus in source code. Here is a very simple menu example:

```
//: gui/SimpleMenus.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class SimpleMenus extends JFrame {
  private JTextField t = new JTextField(15);
  private ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      t.setText(((JMenuItem)e.getSource()).getText());
    }
  };
  private JMenu[] menus = {
    new JMenu("Winken"), new JMenu("Blinken"),
    new JMenu("Nod")
  };
  private JMenuItem[] items = {
    new JMenuItem("Fee"), new JMenuItem("Fi"),
    new JMenuItem("Fo"),  new JMenuItem("Zip"),
    new JMenuItem("Zap"), new JMenuItem("Zot"),
    new JMenuItem("Olly"), new JMenuItem("Oxen"),
    new JMenuItem("Free")
  };
  public SimpleMenus() {
    for(int i = 0; i < items.length; i++) {
      items[i].addActionListener(al);
      menus[i % 3].add(items[i]);
    }
    JMenuBar mb = new JMenuBar();
```

```
    for(JMenu jm : menus)
      mb.add(jm);
    setJMenuBar(mb);
    setLayout(new FlowLayout());
    add(t);
  }
  public static void main(String[] args) {
    run(new SimpleMenus(), 200, 150);
  }
} ///:~
```

The use of the modulus operator in "**i%3**" distributes the menu items among the three **JMenu**s. Each **JMenuItem** must have an **ActionListener** attached to it; here, the same **ActionListener** is used everywhere, but you'll usually need an individual one for each **JMenuItem**.

JMenuItem inherits **AbstractButton**, so it has some button-like behaviors. By itself, it provides an item that can be placed on a drop-down menu. There are also three types inherited from **JMenuItem: JMenu**, to hold other JMenuItems (so you can have cascading menus); **JCheckBoxMenuItem**, which produces a check mark to indicate whether that menu item is selected; and **JRadioButtonMenuItem**, which contains a radio button.

As a more sophisticated example, here are the ice cream flavors again, used to create menus. This example also shows cascading menus, keyboard mnemonics, **JCheckBoxMenuItem**s, and the way that you can dynamically change menus:

```
//: gui/Menus.java
// Submenus, check box menu items, swapping menus,
// mnemonics (shortcuts) and action commands.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Menus extends JFrame {
  private String[] flavors = {
    "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
    "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie"
  };
  private JTextField t = new JTextField("No flavor", 30);
  private JMenuBar mb1 = new JMenuBar();
  private JMenu
    f = new JMenu("File"),
    m = new JMenu("Flavors"),
    s = new JMenu("Safety");
  // Alternative approach:
  private JCheckBoxMenuItem[] safety = {
    new JCheckBoxMenuItem("Guard"),
    new JCheckBoxMenuItem("Hide")
  };
  private JMenuItem[] file = { new JMenuItem("Open") };
  // A second menu bar to swap to:
  private JMenuBar mb2 = new JMenuBar();
  private JMenu fooBar = new JMenu("fooBar");
  private JMenuItem[] other = {
    // Adding a menu shortcut (mnemonic) is very
    // simple, but only JMenuItems can have them
    // in their constructors:
    new JMenuItem("Foo", KeyEvent.VK_F),
    new JMenuItem("Bar", KeyEvent.VK_A),
```

```
    // No shortcut:
    new JMenuItem("Baz"),
  };
  private JButton b = new JButton("Swap Menus");
  class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JMenuBar m = getJMenuBar();
      setJMenuBar(m == mb1 ? mb2 : mb1);
      validate(); // Refresh the frame
    }
  }
  class ML implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JMenuItem target = (JMenuItem)e.getSource();
      String actionCommand = target.getActionCommand();
      if(actionCommand.equals("Open")) {
        String s = t.getText();
        boolean chosen = false;
        for(String flavor : flavors)
          if(s.equals(flavor))
            chosen = true;
        if(!chosen)
          t.setText("Choose a flavor first!");
        else
          t.setText("Opening " + s + ". Mmm, mm!");
      }
    }
  }
  class FL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JMenuItem target = (JMenuItem)e.getSource();
      t.setText(target.getText());
    }
  }
  // Alternatively, you can create a different
  // class for each different MenuItem. Then you
  // don't have to figure out which one it is:
  class FooL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Foo selected");
    }
  }
  class BarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Bar selected");
    }
  }
  class BazL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Baz selected");
    }
  }
  class CMIL implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
      JCheckBoxMenuItem target =
        (JCheckBoxMenuItem)e.getSource();
      String actionCommand = target.getActionCommand();
      if(actionCommand.equals("Guard"))
        t.setText("Guard the Ice Cream! " +
          "Guarding is " + target.getState());
      else if(actionCommand.equals("Hide"))
        t.setText("Hide the Ice Cream! " +
          "Is it hidden? " + target.getState());
```

```
    }
  }
  public Menus() {
    ML ml = new ML();
    CMIL cmil = new CMIL();
    safety[0].setActionCommand("Guard");
    safety[0].setMnemonic(KeyEvent.VK_G);
    safety[0].addItemListener(cmil);
    safety[1].setActionCommand("Hide");
    safety[1].setMnemonic(KeyEvent.VK_H);
    safety[1].addItemListener(cmil);
    other[0].addActionListener(new FooL());
    other[1].addActionListener(new BarL());
    other[2].addActionListener(new BazL());
    FL fl = new FL();
    int n = 0;
    for(String flavor : flavors) {
      JMenuItem mi = new JMenuItem(flavor);
      mi.addActionListener(fl);
      m.add(mi);
      // Add separators at intervals:
      if((n++ + 1) % 3 == 0)
        m.addSeparator();
    }
    for(JCheckBoxMenuItem sfty : safety)
      s.add(sfty);
    s.setMnemonic(KeyEvent.VK_A);
    f.add(s);
    f.setMnemonic(KeyEvent.VK_F);
    for(int i = 0; i < file.length; i++) {
      file[i].addActionListener(ml);
      f.add(file[i]);
    }
    mb1.add(f);
    mb1.add(m);
    setJMenuBar(mb1);
    t.setEditable(false);
    add(t, BorderLayout.CENTER);
    // Set up the system for swapping menus:
    b.addActionListener(new BL());
    b.setMnemonic(KeyEvent.VK_S);
    add(b, BorderLayout.NORTH);
    for(JMenuItem oth : other)
      fooBar.add(oth);
    fooBar.setMnemonic(KeyEvent.VK_B);
    mb2.add(fooBar);
  }
  public static void main(String[] args) {
    run(new Menus(), 300, 200);
  }
} ///:~
```

In this program I placed the menu items into arrays and then stepped through each array, calling **add( )** for each **JMenuItem**. This makes adding or subtracting a menu item somewhat less tedious.

This program creates two **JMenuBar**s to demonstrate that menu bars can be actively swapped while the program is running. You can see how a **JMenuBar** is made up of **JMenus**, and each **JMenu** is made up of **JMenuItems**, **JCheckBoxMenuItem**s, or even other **JMenu**s (which produce submenus). When a **JMenuBar** is assembled, it can be installed into the current program with the **setJMenuBar( )** method. Note that when the

button is pressed, it checks to see which menu is currently installed by calling **getJMenuBar( )**, then it puts the other menu bar in its place.

When testing for "Open," notice that spelling and capitalization are critical, but Java signals no error if there is no match with "Open." This kind of string comparison is a source of programming errors.

The checking and unchecking of the menu items is taken care of automatically. The code handling the **JCheckBoxMenuItem**s shows two different ways to determine what was checked: string matching (the less-safe approach, although you'll see it used) and matching on the event target object. As shown, the **getState( )** method can be used to reveal the state. You can also change the state of a **JCheckBoxMenuItem** with **setState( )**.

The events for menus are a bit inconsistent and can lead to confusion: **JMenuItem**s use **ActionListener**s, but **JCheckBoxMenuItem**s use **ItemListener**s. The **JMenu** objects can also support **ActionListener**s, but that's not usually helpful. In general, you'll attach listeners to each **JMenuItem**, **JCheckBoxMenuItem**, or **JRadioButtonMenuItem**, but the example shows **ItemListener**s and **ActionListener**s attached to the various menu components.

Swing supports mnemonics, or "keyboard shortcuts," so you can select anything derived from **AbstractButton** (button, menu item, etc.) by using the keyboard instead of the mouse. These are quite simple; for **JMenuItem**, you can use the overloaded constructor that takes, as a second argument, the identifier for the key. However, most **AbstractButton**s do not have constructors like this, so the more general way to solve the problem is to use the **setMnemonic( )** method. The preceding example adds mnemonics to the button and some of the menu items; shortcut indicators automatically appear on the components.

You can also see the use of **setActionCommand( )**. This seems a bit strange because in each case, the "action command" is exactly the same as the label on the menu component. Why not just use the label instead of this alternative string? The problem is internationalization. If you retarget this program to another language, you want to change only the label in the menu, and not change the code (which would no doubt introduce new errors). By using **setActionCommand( )**, the "action command" can be immutable, but the menu label can change. All the code works with the "action command," so it's unaffected by changes to the menu labels. Note that in this program, not all the menu components are examined for their action commands, so those that aren't do not have their action command set.

The bulk of the work happens in the listeners. **BL** performs the **JMenuBar** swapping. In **ML**, the "figure out who rang" approach is taken by getting the source of the **ActionEvent** and casting it to a **JMenuItem**, then getting the action command string to pass it through a cascaded if statement.

The **FL** listener is simple even though it's handling all the different flavors in the flavor menu. This approach is useful if you have enough simplicity in your logic, but in general, you'll want to take the approach used with **FooL**, **BarL**, and **BazL**, in which each is attached to only a single menu component, so no extra detection logic is necessary, and you know exactly who called the listener. Even with the profusion of classes generated this way, the code inside tends to be smaller, and the process is more foolproof.

You can see that menu code quickly gets long-winded and messy. This is another case where the use of a GUI builder is the appropriate solution. A good tool will also handle the maintenance of the menus.

Exercise 19: (3) Modify **Menus.java** to use radio buttons instead of check boxes on the menus.

Exercise 20: (6) Create a program that breaks a text file into words. Distribute those words as labels on menus and submenus.

Pop-up menus

The most straightforward way to implement a **JPopupMenu** is to create an inner class that extends **MouseAdapter**, then add an object of that inner class to each component that you want to produce pop-up behavior:

```
//: gui/Popup.java
// Creating popup menus with Swing.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Popup extends JFrame {
  private JPopupMenu popup = new JPopupMenu();
  private JTextField t = new JTextField(10);
  public Popup() {
    setLayout(new FlowLayout());
    add(t);
    ActionListener al = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        t.setText(((JMenuItem)e.getSource()).getText());
      }
    };
    JMenuItem m = new JMenuItem("Hither");
    m.addActionListener(al);
    popup.add(m);
    m = new JMenuItem("Yon");
    m.addActionListener(al);
    popup.add(m);
    m = new JMenuItem("Afar");
    m.addActionListener(al);
    popup.add(m);
    popup.addSeparator();
    m = new JMenuItem("Stay Here");
    m.addActionListener(al);
    popup.add(m);
    PopupListener pl = new PopupListener();
    addMouseListener(pl);
    t.addMouseListener(pl);
  }
  class PopupListener extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      maybeShowPopup(e);
    }
    public void mouseReleased(MouseEvent e) {
      maybeShowPopup(e);
    }
    private void maybeShowPopup(MouseEvent e) {
      if(e.isPopupTrigger())
        popup.show(e.getComponent(), e.getX(), e.getY());
    }
  }
  public static void main(String[] args) {
    run(new Popup(), 300, 200);
  }
} ///:~
```

The same **ActionListener** is added to each **JMenuItem**. It fetches the text from the menu label and inserts it into the **JTextField**.

Drawing

In a good GUI framework, drawing should be reasonably easy—and it is, in the Swing library. The problem with any drawing example is that the calculations that determine where things go are typically a lot more complicated than the calls to the drawing routines, and these calculations are often mixed together with the drawing calls, so it can seem that the interface is more complicated than it actually is.

For simplicity, consider the problem of representing data on the screenhere, the data will be provided by the built-in **Math.sin( )** method, which produces a mathematical sine function. To make things a little more interesting, and to further demonstrate how easy it is to use Swing components, a slider will be placed at the bottom of the form to dynamically control the number of sine wave cycles that are displayed. In addition, if you resize the window, you'll see that the sine wave refits itself to the new window size.

Although any **JComponent** may be painted and thus used as a canvas, if you just want a straightforward drawing surface, you will typically inherit from a **JPanel**. The only method you need to override is **paintComponent( )**, which is called whenever that component must be repainted (you normally don't need to worry about this, because the decision is managed by Swing). When it is called, Swing passes a **Graphics** object to the method, and you can then use this object to draw or paint on the surface.

In the following example, all the intelligence concerning painting is in the **SineDraw** class; the **SineWave** class simply configures the program and the slider control. Inside **SineDraw**, the **setCycles( )** method provides a hook to allow another object—the slider control, in this case—to control the number of cycles.

```
//: gui/SineWave.java
// Drawing with Swing, using a JSlider.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

class SineDraw extends JPanel {
  private static final int SCALEFACTOR = 200;
  private int cycles;
  private int points;
  private double[] sines;
  private int[] pts;
  public SineDraw() { setCycles(5); }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    int maxWidth = getWidth();
    double hstep = (double)maxWidth / (double)points;
    int maxHeight = getHeight();
    pts = new int[points];
    for(int i = 0; i < points; i++)
      pts[i] =
        (int)(sines[i] * maxHeight/2 * .95 + maxHeight/2);
    g.setColor(Color.RED);
    for(int i = 1; i < points; i++) {
      int x1 = (int)((i - 1) * hstep);
      int x2 = (int)(i * hstep);
      int y1 = pts[i-1];
      int y2 = pts[i];
```

```
      g.drawLine(x1, y1, x2, y2);
    }
  }
  public void setCycles(int newCycles) {
    cycles = newCycles;
    points = SCALEFACTOR * cycles * 2;
    sines = new double[points];
    for(int i = 0; i < points; i++) {
      double radians = (Math.PI / SCALEFACTOR) * i;
      sines[i] = Math.sin(radians);
    }
    repaint();
  }
}

public class SineWave extends JFrame {
  private SineDraw sines = new SineDraw();
  private JSlider adjustCycles = new JSlider(1, 30, 5);
  public SineWave() {
    add(sines);
    adjustCycles.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        sines.setCycles(
          ((JSlider)e.getSource()).getValue());
      }
    });
    add(BorderLayout.SOUTH, adjustCycles);
  }
  public static void main(String[] args) {
    run(new SineWave(), 700, 400);
  }
} ///:~
```

All of the fields and arrays are used in the calculation of the sine wave points; **cycles** indicates the number of complete sine waves desired, **points** contains the total number of points that will be graphed, **sines** contains the sine function values, and **pts** contains the y-coordinates of the points that will be drawn on the **JPanel**. The **setCycles( )** method creates the arrays according to the number of points needed and fills the **sines** array with numbers. By calling **repaint( )**, **setCycles( )** forces **paintComponent( )** to be called so the rest of the calculation and redraw will take place.

The first thing you must do when you override **paintComponent( )** is to call the base-class version of the method. Then you are free to do whatever you like; normally, this means using the **Graphics** methods that you can find in the documentation for **java.awt.Graphics** (in the JDK documentation from *http://java.sun.com)* to draw and paint pixels onto the **JPanel**. Here, you can see that almost all the code is involved in performing the calculations; the only two method calls that actually manipulate the screen are **setColor( )** and **drawLine( )**. You will probably have a similar experience when creating your own program that displays graphical data; you'll spend most of your time figuring out what it is you want to draw, but the actual drawing process will be quite simple.

When I created this program, the bulk of my time was spent in getting the sine wave to display. Once I did that, I thought it would be nice to dynamically change the number of cycles. My programming experiences when trying to do such things in other languages made me a bit reluctant to try this, but it turned out to be the easiest part of the project. I created a **JSlider** (the arguments are the leftmost value of the **JSlider**, the rightmost value, and the starting value, respectively, but there are other constructors as well) and dropped it into the **JFrame**. Then I looked at the JDK documentation and noticed that the only listener was the **addChangeListener**, which was triggered whenever the slider was changed enough for it to produce a different value. The only method for this was the obviously named **stateChanged( )**, which provided a **ChangeEvent** object so that I could look backward to

the source of the change and find the new value. Calling the **sines** object's **setCycles( )** enabled the new value to be incorporated and the **JPanel** to be redrawn.

In general, you will find that most of your Swing problems can be solved by following a similar process, and you'll find that it's generally quite simple, even if you haven't used a particular component before.

If your problem is more complex, there are other, more sophisticated alternatives for drawing, including third-party JavaBeans components and the Java 2D API. These solutions are beyond the scope of this book, but you should look them up if your drawing code becomes too onerous.

Exercise 21: (5) Modify **SineWave.java** to turn **SineDraw** into a JavaBean by adding "getter" and "setter" methods.

Exercise 22: (7) Create an application using **SwingConsole**. This should have three sliders, one each for the red, green, and blue values in **java.awt.Color**. The rest of the form should be a **JPanel** that displays the color determined by the three sliders. Also include non-editable text fields that show the current RGB values.

Exercise 23: (8) Using **SineWave.java** as a starting point, create a program that displays a rotating square on the screen. One slider should control the speed of rotation, and a second slider should control the size of the box.

Exercise 24: (7) Remember the "sketching box" toy with two knobs, one that controls the vertical movement of the drawing point, and one that controls the horizontal movement? Create a variation of this toy, using **SineWave.java** to get you started. Instead of knobs, use sliders. Add a button that will erase the entire sketch.

Exercise 25: (8) Starting with **SineWave.java**, create a program (an application using the **SwingConsole** class) that draws an animated sine wave that appears to scroll past the viewing window like an oscilloscope, driving the animation with a **java.util.Timer**. The speed of the animation should be controlled with a **javax.swing.JSlider** control.

Exercise 26: (5) Modify the previous exercise so that multiple sine wave panels are created within the application. The number of sine wave panels should be controlled by command-line parameters.

Exercise 27: (5) Modify Exercise 25 so that the **javax.swing.Timer** class is used to drive the animation. Note the difference between this and **java.util.Timer**.

Exercise 28: (7) Create a dice class (just a class, without a GUI). Create five dice and throw them repeatedly. Draw the curve showing the sum of the dots from each throw, and show the curve evolving dynamically as you throw more and more times.

Dialog boxes

A dialog box is a window that pops up out of another window. Its purpose is to deal with some specific issue without cluttering the original window with those details. Dialog boxes are commonly used in windowed programming environments.

To create a dialog box, you inherit from **JDialog**, which is just another kind of **Window**, like a **JFrame**. A **JDialog** has a layout manager (which defaults to **BorderLayout)**, and you add event listeners to deal with events. Here's a very simple example:

```
//: gui/Dialogs.java
// Creating and using Dialog Boxes.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

class MyDialog extends JDialog {
  public MyDialog(JFrame parent) {
    super(parent, "My dialog", true);
    setLayout(new FlowLayout());
    add(new JLabel("Here is my dialog"));
    JButton ok = new JButton("OK");
    ok.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        dispose(); // Closes the dialog
      }
    });
    add(ok);
    setSize(150,125);
  }
}

public class Dialogs extends JFrame {
  private JButton b1 = new JButton("Dialog Box");
  private MyDialog dlg = new MyDialog(null);
  public Dialogs() {
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        dlg.setVisible(true);
      }
    });
    add(b1);
  }
  public static void main(String[] args) {
    run(new Dialogs(), 125, 75);
  }
} ///:~
```

Once the **JDialog** is created, **setVisible(true)** must be called to display and activate it. When the dialog window is closed, you must release the resources used by the dialog's window by calling **dispose( )**.

The following example is more complex; the dialog box is made up of a grid (using **GridLayout**) of a special kind of button that is defined here as class **ToeButton**. This button draws a frame around itself and, depending on its state, a blank, an "x," or an "o" in the middle. It starts out blank, and then depending on whose turn it is, changes to an "x" or an "o." However, it will also flip back and forth between "x" and "o" when you click on the button, to provide an interesting variation on the tic-tac-toe concept. In addition, the dialog box can be set up for any number of rows and columns by changing numbers in the main application window.

```
//: gui/TicTacToe.java
// Dialog boxes and creating your own components.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class TicTacToe extends JFrame {
  private JTextField
    rows = new JTextField("3"),
```

```
    cols = new JTextField("3");
  private enum State { BLANK, XX, OO }
  static class ToeDialog extends JDialog {
    private State turn = State.XX; // Start with x's turn
    ToeDialog(int cellsWide, int cellsHigh) {
      setTitle("The game itself");
      setLayout(new GridLayout(cellsWide, cellsHigh));
      for(int i = 0; i < cellsWide * cellsHigh; i++)
        add(new ToeButton());
      setSize(cellsWide * 50, cellsHigh * 50);
      setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    }
    class ToeButton extends JPanel {
      private State state = State.BLANK;
      public ToeButton() { addMouseListener(new ML()); }
      public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int
          x1 = 0, y1 = 0,
          x2 = getSize().width - 1,
          y2 = getSize().height - 1;
        g.drawRect(x1, y1, x2, y2);
        x1 = x2/4;
        y1 = y2/4;
        int wide = x2/2, high = y2/2;
        if(state == State.XX) {
          g.drawLine(x1, y1, x1 + wide, y1 + high);
          g.drawLine(x1, y1 + high, x1 + wide, y1);
        }
        if(state == State.OO)
          g.drawOval(x1, y1, x1 + wide/2, y1 + high/2);
      }
      class ML extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
          if(state == State.BLANK) {
            state = turn;
            turn =
              (turn == State.XX ? State.OO : State.XX);
          }
          else
            state =
              (state == State.XX ? State.OO : State.XX);
          repaint();
        }
      }
    }
  }
  class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JDialog d = new ToeDialog(
        new Integer(rows.getText()),
        new Integer(cols.getText()));
      d.setVisible(true);
    }
  }
  public TicTacToe() {
    JPanel p = new JPanel();
    p.setLayout(new GridLayout(2,2));
    p.add(new JLabel("Rows", JLabel.CENTER));
    p.add(rows);
    p.add(new JLabel("Columns", JLabel.CENTER));
    p.add(cols);
    add(p, BorderLayout.NORTH);
```

```
    JButton b = new JButton("go");
    b.addActionListener(new BL());
    add(b, BorderLayout.SOUTH);
  }
  public static void main(String[] args) {
    run(new TicTacToe(), 200, 200);
  }
} ///:~
```

Because **statics** can only be at the outer level of the class, inner classes cannot have **static** data or nested classes.

The **paintComponent( )** method draws the square around the panel and the "x" or the "o." This is full of tedious calculations, but it's straightforward.

A mouse click is captured by the **MouseListener**, which first checks to see if the panel has anything written on it. If not, the parent window is queried to find out whose turn it is, which establishes the state of the **ToeButton**. Via the inner-class mechanism, the **ToeButton** then reaches back into the parent and changes the turn. If the button is already displaying an "x" or an "o," then that is flopped. You can see in these calculations the convenient use of the ternary **if-else** described in the *Operators* chapter. After a state change, the **ToeButton** is repainted.

The constructor for **ToeDialog** is quite simple: It adds into a **GridLayout** as many buttons as you request, then resizes it for 50 pixels on a side for each button.

TicTacToe sets up the whole application by creating the **JTextField**s (for inputting the rows and columns of the button grid) and the "go" button with its **ActionListener**. When the button is pressed, the data in the **JTextField**s must be fetched, and, since they are in **String** form, turned into **int**s using the **Integer** constructor that takes a **String** argument.

File dialogs

Some operating systems have a number of special built-in dialog boxes to handle the selection of things such as fonts, colors, printers, and the like. Virtually all graphical operating systems support the opening and saving of files, so Java's **JFileChooser** encapsulates these for easy use.

The following application exercises two forms of **JFileChooser** dialogs, one for opening and one for saving. Most of the code should by now be familiar, and all the interesting activities happen in the action listeners for the two different button clicks:

```
//: gui/FileChooserTest.java
// Demonstration of File dialog boxes.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class FileChooserTest extends JFrame {
  private JTextField
    fileName = new JTextField(),
    dir = new JTextField();
  private JButton
    open = new JButton("Open"),
    save = new JButton("Save");
  public FileChooserTest() {
    JPanel p = new JPanel();
```

```
    open.addActionListener(new OpenL());
    p.add(open);
    save.addActionListener(new SaveL());
    p.add(save);
    add(p, BorderLayout.SOUTH);
    dir.setEditable(false);
    fileName.setEditable(false);
    p = new JPanel();
    p.setLayout(new GridLayout(2,1));
    p.add(fileName);
    p.add(dir);
    add(p, BorderLayout.NORTH);
  }
  class OpenL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JFileChooser c = new JFileChooser();
      // Demonstrate "Open" dialog:
      int rVal = c.showOpenDialog(FileChooserTest.this);
      if(rVal == JFileChooser.APPROVE_OPTION) {
        fileName.setText(c.getSelectedFile().getName());
        dir.setText(c.getCurrentDirectory().toString());
      }
      if(rVal == JFileChooser.CANCEL_OPTION) {
        fileName.setText("You pressed cancel");
        dir.setText("");
      }
    }
  }
  class SaveL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JFileChooser c = new JFileChooser();
      // Demonstrate "Save" dialog:
      int rVal = c.showSaveDialog(FileChooserTest.this);
      if(rVal == JFileChooser.APPROVE_OPTION) {
        fileName.setText(c.getSelectedFile().getName());
        dir.setText(c.getCurrentDirectory().toString());
      }
      if(rVal == JFileChooser.CANCEL_OPTION) {
        fileName.setText("You pressed cancel");
        dir.setText("");
      }
    }
  }
  public static void main(String[] args) {
    run(new FileChooserTest(), 250, 150);
  }
} ///:~
```

Note that there are many variations you can apply to **JFileChooser**, including filters to narrow the file names that you will allow.

For an "open file" dialog, you call **showOpenDialog( )**, and for a "save file" dialog, you call **showSaveDialog( )**. These commands don't return until the dialog is closed. The **JFileChooser** object still exists, so you can read data from it. The methods **getSelectedFile( )** and **getCurrentDirectory( )** are two ways you can interrogate the results of the operation. If these return **null**, it means the user canceled out of the dialog.

Exercise 29: (3) In the JDK documentation for **javax.swing**, look up the **JColorChooser**. Write a program with a button that brings up the color chooser as a dialog.

HTML on Swing components

Any component that can take text can also take HTML text, which it will reformat according to HTML rules. This means you can very easily add fancy text to a Swing component. For example:

```
//: gui/HTMLButton.java
// Putting HTML text on Swing components.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class HTMLButton extends JFrame {
  private JButton b = new JButton(
    "<html><b><font size=+2>" +
    "<center>Hello!<br><i>Press me now!");
  public HTMLButton() {
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        add(new JLabel("<html>" +
          "<i><font size=+4>Kapow!"));
        // Force a re-layout to include the new label:
        validate();
      }
    });
    setLayout(new FlowLayout());
    add(b);
  }
  public static void main(String[] args) {
    run(new HTMLButton(), 200, 500);
  }
} ///:~
```

You must start the text with "<html>," and then you can use normal HTML tags. Note that you are not forced to include the normal closing tags.

The **ActionListener** adds a new **JLabel** to the form, which also contains HTML text. However, this label is not added during construction, so you must call the container's **validate( )** method in order to force a re-layout of the components (and thus the display of the new label).

You can also use HTML text for **JTabbedPane**, **JMenuItem**, **JToolTip**, **JRadioButton**, and **JCheckBox**.

Exercise 30: (3) Write a program that shows the use of HTML text on all the items from the previous paragraph.

Sliders and progress bars

A slider (which has already been used in **SineWave.java)** allows the user to input data by moving a point back and forth, which is intuitive in some situations (volume controls, for example). A progress bar displays data in a relative fashion from "full" to "empty" so the user gets a perspective. My favorite example for these is to simply hook the slider to the progress bar so when you move the slider, the progress bar changes accordingly. The following example also demonstrates the **ProgressMonitor**, a more fullfeatured pop-up dialog:

```
//: gui/Progress.java
```

```
// Using sliders, progress bars and progress monitors.
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Progress extends JFrame {
  private JProgressBar pb = new JProgressBar();
  private ProgressMonitor pm = new ProgressMonitor(
    this, "Monitoring Progress", "Test", 0, 100);
  private JSlider sb =
    new JSlider(JSlider.HORIZONTAL, 0, 100, 60);
  public Progress() {
    setLayout(new GridLayout(2,1));
    add(pb);
    pm.setProgress(0);
    pm.setMillisToPopup(1000);
    sb.setValue(0);
    sb.setPaintTicks(true);
    sb.setMajorTickSpacing(20);
    sb.setMinorTickSpacing(5);
    sb.setBorder(new TitledBorder("Slide Me"));
    pb.setModel(sb.getModel()); // Share model
    add(sb);
    sb.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        pm.setProgress(sb.getValue());
      }
    });
  }
  public static void main(String[] args) {
    run(new Progress(), 300, 200);
  }
} ///:~
```

The key to hooking the slider and progress bar components together is in sharing their model, in the line:

```
pb.setModel(sb.getModel());
```

Of course, you could also control the two using a listener, but using the model is more straightforward for simple situations. The **ProgressMonitor** does not have a model and so the listener approach is required. Note that the **ProgressMonitor** only moves forward, and once it reaches the end it closes. The **JProgressBar** is fairly straightforward, but the **JSlider** has a lot of options, such as the orientation and major and minor tick marks. Notice how straightforward it is to add a titled border.

Exercise 31: (8) Create an "asymptotic progress indicator" that gets slower and slower as it approaches the finish point. Add random erratic behavior so it will periodically look like it's starting to speed up.

Exercise 32: (6) Modify **Progress.java** so that it does not share models, but instead uses a listener to connect the slider and progress bar.

Selecting look & feel

"Pluggable look & feel" allows your program to emulate the look and feel of various operating environments. You can even dynamically change the look and feel while the program is

executing. However, you generally just want to do one of two things: either select the "cross-platform" look and feel (which is Swing's "metal"), or select the look and feel for the system you are currently on so your Java program looks like it was created specifically for that system (this is almost certainly the best choice in most cases, to avoid confounding the user). The code to select either of these behaviors is quite simple, but you must execute it *before* you create any visual components, because the components will be made based on the current look and feel, and will not be changed just because you happen to change the look and feel midway during the program (that process is more complicated and uncommon, and is relegated to Swing-specific books).

Actually, if you want to use the cross-platform ("metal") look and feel that is characteristic of Swing programs, you don't have to do anything—it's the default. But if you want instead to use the current operating environment's look and feel,[8] you just insert the following code, typically at the beginning of your **main( )**, but at least before any components are added:

```
try {
UIManager.setLookAndFeet(
  UIManager.getSystemLookAndFeelClassName());
} catch(Exception e) {
  throw new RuntimeException(e);
}
```

You don't actually need anything in the **catch** clause because the **UIManager** will default to the cross-platform look and feel if your attempts to set up any of the alternatives fail. However, during debugging, the exception can be quite useful, so you may at least want to see some results via the **catch** clause.

Here is a program that takes a command-line argument to select a look and feel, and shows how several different components look under the chosen look and feel:

```
//: gui/LookAndFeel.java
// Selecting different looks & feels.
// {Args: motif}
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class LookAndFeel extends JFrame {
  private String[] choices =
    "Eeny Meeny Minnie Mickey Moe Larry Curly".split(" ");
  private Component[] samples = {
    new JButton("JButton"),
    new JTextField("JTextField"),
    new JLabel("JLabel"),
    new JCheckBox("JCheckBox"),
    new JRadioButton("Radio"),
    new JComboBox(choices),
    new JList(choices),
  };
  public LookAndFeel() {
    super("Look And Feel");
    setLayout(new FlowLayout());
    for(Component component : samples)
      add(component);
  }
  private static void usageError() {
    System.out.println(
      "Usage:LookAndFeel [cross|system|motif]");
```

[8] You may argue about whether the Swing rendering does justice to your operating environment.

```
    System.exit(1);
  }
  public static void main(String[] args) {
    if(args.length == 0) usageError();
    if(args[0].equals("cross")) {
      try {
        UIManager.setLookAndFeel(UIManager.
          getCrossPlatformLookAndFeelClassName());
      } catch(Exception e) {
        e.printStackTrace();
      }
    } else if(args[0].equals("system")) {
      try {
        UIManager.setLookAndFeel(UIManager.
          getSystemLookAndFeelClassName());
      } catch(Exception e) {
        e.printStackTrace();
      }
    } else if(args[0].equals("motif")) {
      try {
        UIManager.setLookAndFeel("com.sun.java."+
          "swing.plaf.motif.MotifLookAndFeel");
      } catch(Exception e) {
        e.printStackTrace();
      }
    } else usageError();
    // Note the look & feel must be set before
    // any components are created.
    run(new LookAndFeel(), 300, 300);
  }
} ///:~
```

You can see that one option is to explicitly specify a string for a look and feel, as seen with **MotifLookAndFeel**. However, that one and the default "metal" look and feel are the only ones that can legally be used on any platform; even though there are look-and-feel strings for Windows and Macintosh, those can only be used on their respective platforms (these are produced when you call **getSystemLookAndFeelClassName( )** and you're on that particular platform).

It is also possible to create a custom look and feel package, for example, if you are building a framework for a company that wants a distinctive appearance. This is a big job and is far beyond the scope of this book (in fact, you'll discover it is beyond the scope of many dedicated Swing books!).

Trees, tables & clipboard

You can find a brief introduction and examples for these topics in the online supplements for this chapter at *www.MindView.net*.

JNLP and Java Web Start

It's possible to *sign* an applet for security purposes. This is shown in the online supplement for this chapter at *www.MindView.net*. Signed applets are powerful and can effectively take the place of an application, but they must run inside a Web browser. This requires the extra overhead of the browser running on the client machine, and also means that the user

interface of the applet is limited and often visually confusing. The Web browser has its own set of menus and toolbars, which will appear above the applet.[9]

The *Java Network Launch Protocol* (JNLP) solves the problem without sacrificing the advantages of applets. With a JNLP application, you can download and install a standalone Java application onto the client's machine. This can be run from the command prompt, a desktop icon, or the application manager that is installed with your JNLP implementation. The application can even be run from the Web site from which it was originally downloaded.

A JNLP application can dynamically download resources from the Internet at run time, and can automatically check the version if the user is connected to the Internet. This means that it has all of the advantages of an applet together with the advantages of standalone applications.

Like applets, JNLP applications need to be treated with some caution by the client's system. Because of this, JNLP applications are subject to the same sandbox security restrictions as applets. Like applets, they can be deployed in signed JAR files, giving the user the option to trust the signer. Unlike applets, if they are deployed in an unsigned JAR file, they can still request access to certain resources of the client's system by means of services in the JNLP API. The user must approve these requests during program execution.

JNLP describes a protocol, not an implementation, so you will need an implementation in order to use it. Java Web Start, or JAWS, is Sun's freely available official reference implementation and is distributed as part of Java SE5- If you are using it for development, you must ensure that the JAR file (**javaws.jar**) is in your classpath; the easiest solution is to add **javaws.jar** to your classpath from its normal Java installation path in **jre/lib**. If you are deploying your JNLP application from a Web server, you must ensure that your server recognizes the MIME type **application/x-java-jnlp-file**. If you are using a recent version of the Tomcat server *(http://jakarta.apache.org/tomcat)* this is pre-configured. Consult the user guide for your particular server.

Creating a JNLP application is not difficult. You create a standard application that is archived in a JAR file, and then you provide a launch file, which is a simple XML file that gives the client system all the information it needs to download and install your application. If you choose not to sign your JAR file, then you must use the services supplied by the JNLP API for each type of resource you want to access on the user's machine.

Here is a variation of **FileChooserTest.java** using the JNLP services to open the file chooser, so that the class can be deployed as a JNLP application in an unsigned JAR file.

```
//: gui/jnlp/JnlpFileChooser.java
// Opening files on a local machine with JNLP.
// {Requires: javax.jnlp.FileOpenService;
// You must have javaws.jar in your classpath}
// To create the jnlpfilechooser.jar file, do this:
// cd ..
// cd ..
// jar cvf gui/jnlp/jnlpfilechooser.jar gui/jnlp/*.class
package gui.jnlp;
import javax.jnlp.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class JnlpFileChooser extends JFrame {
  private JTextField fileName = new JTextField();
```

[9] Jeremy Meyer developed this section.

```
  private JButton
    open = new JButton("Open"),
    save = new JButton("Save");
  private JEditorPane ep = new JEditorPane();
  private JScrollPane jsp = new JScrollPane();
  private FileContents fileContents;
  public JnlpFileChooser() {
    JPanel p = new JPanel();
    open.addActionListener(new OpenL());
    p.add(open);
    save.addActionListener(new SaveL());
    p.add(save);
    jsp.getViewport().add(ep);
    add(jsp, BorderLayout.CENTER);
    add(p, BorderLayout.SOUTH);
    fileName.setEditable(false);
    p = new JPanel();
    p.setLayout(new GridLayout(2,1));
    p.add(fileName);
    add(p, BorderLayout.NORTH);
    ep.setContentType("text");
    save.setEnabled(false);
  }
  class OpenL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      FileOpenService fs = null;
      try {
        fs = (FileOpenService)ServiceManager.lookup(
          "javax.jnlp.FileOpenService");
      } catch(UnavailableServiceException use) {
        throw new RuntimeException(use);
      }
      if(fs != null) {
        try {
          fileContents = fs.openFileDialog(".",
            new String[]{"txt", "*"});
          if(fileContents == null)
            return;
          fileName.setText(fileContents.getName());
          ep.read(fileContents.getInputStream(), null);
        } catch(Exception exc) {
          throw new RuntimeException(exc);
        }
        save.setEnabled(true);
      }
    }
  }
  class SaveL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      FileSaveService fs = null;
      try {
        fs = (FileSaveService)ServiceManager.lookup(
          "javax.jnlp.FileSaveService");
      } catch(UnavailableServiceException use) {
        throw new RuntimeException(use);
      }
      if(fs != null) {
        try {
          fileContents = fs.saveFileDialog(".",
            new String[]{"txt"},
            new ByteArrayInputStream(
              ep.getText().getBytes()),
            fileContents.getName());
```

```
          if(fileContents == null)
            return;
          fileName.setText(fileContents.getName());
        } catch(Exception exc) {
          throw new RuntimeException(exc);
        }
      }
    }
  }
  public static void main(String[] args) {
    JnlpFileChooser fc = new JnlpFileChooser();
    fc.setSize(400, 300);
    fc.setVisible(true);
  }
} ///:~
```

Note that the **FileOpenService** and the **FileSaveService** classes are imported from the **javax.jnlp** package and that nowhere in the code is the **JFileChooser** dialog box referred to directly. The two services used here must be requested using the **ServiceManager.lookup( )** method, and the resources on the client system can only be accessed via the objects returned from this method. In this case, the files on the client's file system are being written to and read from using the **FileContent** interface, provided by the JNLP. Any attempt to access the resources directly by using, say, a **File** or a **FileReader** object would cause a **SecurityException** to be thrown in the same way that it would if you tried to use them from an unsigned applet. If you want to use these classes and not be restricted to the JNLP service interfaces, you must sign the JAR file.

The commented **jar** command in **JnlpFileChooser.java** will produce the necessary JAR file. Here is an appropriate launch file for the preceding example.

```
//:! gui/jnlp/filechooser.jnlp
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec = "1.0+"
  codebase="file:C:/AAA-TIJ4/code/gui/jnlp"
  href="filechooser.jnlp">
  <information>
    <title>FileChooser demo application</title>
    <vendor>Mindview Inc.</vendor>
    <description>
      Jnlp File chooser Application
    </description>
    <description kind="short">
      Demonstrates opening, reading and writing a text file
    </description>
    <icon href="mindview.gif"/>
    <offline-allowed/>
  </information>
  <resources>
    <j2se version="1.3+"
      href="http://java.sun.com/products/autodl/j2se"/>
    <jar href="jnlpfilechooser.jar" download="eager"/>
  </resources>
  <application-desc
    main-class="gui.jnlp.JnlpFileChooser"/>
</jnlp>
///:~
```

You'll find this launch file in the source-code download for this book (from *www.MindView.net)* saved as **filechooser.jnlp** without the first and last lines, in the same directory as the JAR file. As you can see, it is an XML file with one **<jnlp>** tag. This has a few sub-elements, which are mostly selfexplanatory.

The **spec** attribute of the **jnlp** element tells the client system what version of the JNLP the application can be run with. The **codebase** attribute points to the URL where this launch file and the resources can be found. Here, it points to a directory on the local machine, which is a good means of testing the application. *Note that you'll need to change this path so that it indicates the appropriate directory on your machine, in order for the program to load successfully*. The **href** attribute must specify the name of this file.

The **information** tag has various sub-elements that provide information about the application. These are used by the Java Web Start administrative console or equivalent, which installs the JNLP application and allows the user to run it from the command line, make shortcuts, and so on.

The **resources** tag serves a similar purpose as the applet tag in an HTML file. The **J2se** sub-element specifies the J2SE version required to run the application, and the **jar** sub-element specifies the JAR file in which the class is archived. The **jar** element has an attribute **download**, which can have the values "eager" or "lazy" that tell the JNLP implementation whether or not the entire archive needs to be downloaded before the application can be run.

The **application-desc** attribute tells the JNLP implementation which class is the executable class, or entry point, to the JAR file.

Another useful sub-element of the **jnlp** tag is the **security** tag, not shown here. Here's what a **security** tag looks like:

```
<security>
   <all-permissions/>
<security/>
```

You use the **security** tag when your application is deployed in a signed JAR file. It is not needed in the preceding example because the local resources are all accessed via the JNLP services.

There are a few other tags available, the details of which can be found in the specification at *http://java.sun.com/products/javawehstart/downloadspec. html.*

To launch the program, you need a download page containing a hypertext link to the **.jnlp** file. Here's what it looks like (without the first and last lines):

```
//:! gui/jnlp/filechooser.html
<html>
Follow the instructions in JnlpFileChooser.java to
build jnlpfilechooser.jar, then:
<a href="filechooser.jnlp">click here</a>
</html>
///:~
```

Once you have downloaded the application once, you can configure it by using the administrative console. If you are using Java Web Start on Windows, then you will be prompted to make a shortcut to your application the second time you use it. This behavior is configurable.

Only two of the JNLP services are covered here, but there are seven services in the current release. Each is designed for a specific task such as printing, or cutting and pasting to the clipboard. You can find more information at *http://java.sun.com*.

Concurrency & Swing

When you program with Swing you're using threads. You saw this at the beginning of this chapter when you learned that everything should be submitted to the Swing event dispatch thread through **SwingUtilities.invokeLater( )**. However, the fact that you don't have to explicitly create a **Thread** object means that threading issues can catch you by surprise. You must keep in mind that there is a Swing event dispatch thread, which is always there, handling all the Swing events by pulling each one out of the event queue and executing it in turn. By remembering the event dispatch thread you'll help ensure that your application won't suffer from deadlocking or race conditions.

This section addresses threading issues that arise when working with Swing.

Long-running tasks

One of the most fundamental mistakes you can make when programming with a graphical user interface is to accidentally use the event dispatch thread to run a long task. Here's a simple example:

```
//: gui/LongRunningTask.java
// A badly designed program.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import static net.mindview.util.SwingConsole.*;

public class LongRunningTask extends JFrame {
  private JButton
    b1 = new JButton("Start Long Running Task"),
    b2 = new JButton("End Long Running Task");
  public LongRunningTask() {
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          TimeUnit.SECONDS.sleep(3);
        } catch(InterruptedException e) {
          System.out.println("Task interrupted");
          return;
        }
        System.out.println("Task completed");
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        // Interrupt yourself?
        Thread.currentThread().interrupt();
      }
    });
    setLayout(new FlowLayout());
    add(b1);
    add(b2);
  }
  public static void main(String[] args) {
    run(new LongRunningTask(), 200, 150);
  }
} ///:~
```

When you press **b1**, the event dispatch thread is suddenly occupied in performing the long-running task. You'll see that the button doesn't even pop back out, because the event dispatch thread that would normally repaint the screen is busy. And you cannot do anything else, like press **b2**, because the program won't respond until **b1**'s task is complete and the event dispatch thread is once again available. The code in **b2** is a flawed attempt to solve the problem by interrupting the event dispatch thread.

The answer, of course, is to execute long-running processes in separate threads. Here, the single-thread **Executor** is used, which automatically queues pending tasks and executes them one at a time:

```
//: gui/InterruptableLongRunningTask.java
// Long-running tasks in threads.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import static net.mindview.util.SwingConsole.*;

class Task implements Runnable {
  private static int counter = 0;
  private final int id = counter++;
  public void run() {
    System.out.println(this + " started");
    try {
      TimeUnit.SECONDS.sleep(3);
    } catch(InterruptedException e) {
      System.out.println(this + " interrupted");
      return;
    }
    System.out.println(this + " completed");
  }
  public String toString() { return "Task " + id; }
  public long id() { return id; }
};

public class InterruptableLongRunningTask extends JFrame {
  private JButton
    b1 = new JButton("Start Long Running Task"),
    b2 = new JButton("End Long Running Task");
  ExecutorService executor =
    Executors.newSingleThreadExecutor();
  public InterruptableLongRunningTask() {
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        Task task = new Task();
        executor.execute(task);
        System.out.println(task + " added to the queue");
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        executor.shutdownNow(); // Heavy-handed
      }
    });
    setLayout(new FlowLayout());
    add(b1);
    add(b2);
  }
  public static void main(String[] args) {
    run(new InterruptableLongRunningTask(), 200, 150);
  }
```

```
} ///:~
```

This is better, but when you press **b2**, it calls **shutdownNow( )** on the **ExecutorService**, thereby disabling it. If you try to add more tasks, you get an exception. Thus, pressing **b2** makes the program inoperable. What we'd like to do is to shut down the current task (and cancel pending tasks) without stopping everything. The Java SE5 **Callable/Future** mechanism described in the *Concurrency* chapter is just what we need. We'll define a new class called **TaskManager**, which contains *tuples* that hold the **Callable** representing the task and the **Future** that comes back from the **Callable**. The reason the tuple is necessary is because it allows us to keep track of the original task, so that we may get extra information that is not available from the **Future**. Here it is:

```
//: net/mindview/util/TaskItem.java
// A Future and the Callable that produced it.
package net.mindview.util;
import java.util.concurrent.*;

public class TaskItem<R,C extends Callable<R>> {
  public final Future<R> future;
  public final C task;
  public TaskItem(Future<R> future, C task) {
    this.future = future;
    this.task = task;
  }
} ///:~
```

In the **java.util.concurrent** library, the task is not available via the **Future** by default because the task would not necessarily still be around when you get the result from the **Future**. Here, we force the task to stay around by storing it.

TaskManager is placed in **net.mindview.util** so it is available as a general-purpose utility:

```
//: net/mindview/util/TaskManager.java
// Managing and executing a queue of tasks.
package net.mindview.util;
import java.util.concurrent.*;
import java.util.*;

public class TaskManager<R,C extends Callable<R>>
extends ArrayList<TaskItem<R,C>> {
  private ExecutorService exec =
    Executors.newSingleThreadExecutor();
  public void add(C task) {
    add(new TaskItem<R,C>(exec.submit(task),task));
  }
  public List<R> getResults() {
    Iterator<TaskItem<R,C>> items = iterator();
    List<R> results = new ArrayList<R>();
    while(items.hasNext()) {
      TaskItem<R,C> item = items.next();
      if(item.future.isDone()) {
        try {
          results.add(item.future.get());
        } catch(Exception e) {
          throw new RuntimeException(e);
        }
        items.remove();
      }
    }
    return results;
```

```
  }
  public List<String> purge() {
    Iterator<TaskItem<R,C>> items = iterator();
    List<String> results = new ArrayList<String>();
    while(items.hasNext()) {
      TaskItem<R,C> item = items.next();
      // Leave completed tasks for results reporting:
      if(!item.future.isDone()) {
        results.add("Cancelling " + item.task);
        item.future.cancel(true); // May interrupt
        items.remove();
      }
    }
    return results;
  }
} ///:~
```

TaskManager is an **ArrayList** of **Taskltem**. It also contains a singlethread **Executor**, so when you call **add( )** with a **Callable**, it submits the **Callable** and stores the resulting **Future** along with the original task. This way, if you need to do anything with the task, you have a reference to that task. As a simple example, in **purge( )** the task's **toString( )** is used.

This can now be used to manage the long-running tasks in our example:

```
//: gui/InterruptableLongRunningCallable.java
// Using Callables for long-running tasks.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

class CallableTask extends Task
implements Callable<String> {
  public String call() {
    run();
    return "Return value of " + this;
  }
}

public class
InterruptableLongRunningCallable extends JFrame {
  private JButton
    b1 = new JButton("Start Long Running Task"),
    b2 = new JButton("End Long Running Task"),
    b3 = new JButton("Get results");
  private TaskManager<String,CallableTask> manager =
    new TaskManager<String,CallableTask>();
  public InterruptableLongRunningCallable() {
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        CallableTask task = new CallableTask();
        manager.add(task);
        System.out.println(task + " added to the queue");
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        for(String result : manager.purge())
          System.out.println(result);
```

```
      }
    });
    b3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        // Sample call to a Task method:
        for(TaskItem<String,CallableTask> tt :
            manager)
          tt.task.id(); // No cast required
        for(String result : manager.getResults())
          System.out.println(result);
      }
    });
    setLayout(new FlowLayout());
    add(b1);
    add(b2);
    add(b3);
  }
  public static void main(String[] args) {
    run(new InterruptableLongRunningCallable(), 200, 150);
  }
} ///:~
```

As you can see, **CallableTask** does exactly the same thing as **Task** except that it returns a result—in this case a **String** identifying the task.

Non-Swing utilities (not part of the standard Java distribution) called **SwingWorker** (from the Sun Web site) and *Foxtrot* (from *http://foxtrot.sourceforge.net)* were created to solve a similar problem, but at this writing, those utilities had not been modified to take advantage of the Java SE5 **Callable/Future** mechanism.

It's often important to give the end user some kind of visual cue that a task is running, and of its progress. This is normally done through either a **JProgressBar** or a **ProgressMonitor**. This example uses a **ProgressMonitor**:

```
//: gui/MonitoredLongRunningCallable.java
// Displaying task progress with ProgressMonitors.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

class MonitoredCallable implements Callable<String> {
  private static int counter = 0;
  private final int id = counter++;
  private final ProgressMonitor monitor;
  private final static int MAX = 8;
  public MonitoredCallable(ProgressMonitor monitor) {
    this.monitor = monitor;
    monitor.setNote(toString());
    monitor.setMaximum(MAX - 1);
    monitor.setMillisToPopup(500);
  }
  public String call() {
    System.out.println(this + " started");
    try {
      for(int i = 0; i < MAX; i++) {
        TimeUnit.MILLISECONDS.sleep(500);
        if(monitor.isCanceled())
          Thread.currentThread().interrupt();
```

```
        final int progress = i;
        SwingUtilities.invokeLater(
          new Runnable() {
            public void run() {
              monitor.setProgress(progress);
            }
          }
        );
      }
    } catch(InterruptedException e) {
      monitor.close();
      System.out.println(this + " interrupted");
      return "Result: " + this + " interrupted";
    }
    System.out.println(this + " completed");
    return "Result: " + this + " completed";
  }
  public String toString() { return "Task " + id; }
};

public class MonitoredLongRunningCallable extends JFrame {
  private JButton
    b1 = new JButton("Start Long Running Task"),
    b2 = new JButton("End Long Running Task"),
    b3 = new JButton("Get results");
  private TaskManager<String,MonitoredCallable> manager =
    new TaskManager<String,MonitoredCallable>();
  public MonitoredLongRunningCallable() {
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        MonitoredCallable task = new MonitoredCallable(
          new ProgressMonitor(
            MonitoredLongRunningCallable.this,
            "Long-Running Task", "", 0, 0)
        );
        manager.add(task);
        System.out.println(task + " added to the queue");
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        for(String result : manager.purge())
          System.out.println(result);
      }
    });
    b3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        for(String result : manager.getResults())
          System.out.println(result);
      }
    });
    setLayout(new FlowLayout());
    add(b1);
    add(b2);
    add(b3);
  }
  public static void main(String[] args) {
    run(new MonitoredLongRunningCallable(), 200, 500);
  }
} ///:~
```

The **MonitoredCallable** constructor takes a **ProgressMonitor** as an argument, and its **call( )** method updates the **ProgressMonitor** every half second. Notice that a

MonitoredCallable is a separate task and thus should not try to control the UI directly, so **SwingUtilities.invokeLater( )** is used to submit the progress change information to the **monitor**. Sun's Swing Tutorial (on *http://java.sun.com)* shows an alternate approach of using a Swing **Timer**, which checks the status of the task and updates the monitor.

If the "cancel" button is pressed on the monitor, **monitor.isCanceled( )** will return **true**. Here, the task just calls **interrupt**) on its own thread, which will land it in the **catch** clause where the **monitor** is terminated with the **close( )** method.

The rest of the code is effectively the same as before, except for the creation of the **ProgressMonitor** as part of the **MonitoredLongRunningCallable** constructor.

Exercise 33: (6) Modify **InterruptableLongRunningCallable.java** so that it runs all the tasks in parallel rather than sequentially.

Visual threading

The following example makes a **Runnable JPanel** class that paints different colors on itself. This application is set up to take values from the command line to determine how big the grid of colors is and how long to **sleep( )** between color changes. By playing with these values, you may discover some interesting and possibly inexplicable features in the threading implementation on your platform:

```
//: gui/ColorBoxes.java
// A visual demonstration of threading.
import javax.swing.*;
import java.awt.*;
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

class CBox extends JPanel implements Runnable {
  private int pause;
  private static Random rand = new Random();
  private Color color = new Color(0);
  public void paintComponent(Graphics g) {
    g.setColor(color);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
  public CBox(int pause) { this.pause = pause; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        color = new Color(rand.nextInt(0xFFFFFF));
        repaint(); // Asynchronously request a paint()
        TimeUnit.MILLISECONDS.sleep(pause);
      }
    } catch(InterruptedException e) {
      // Acceptable way to exit
    }
  }
}

public class ColorBoxes extends JFrame {
  private int grid = 12;
  private int pause = 50;
  private static ExecutorService exec =
    Executors.newCachedThreadPool();
```

```
  public void setUp() {
    setLayout(new GridLayout(grid, grid));
    for(int i = 0; i < grid * grid; i++) {
      CBox cb = new CBox(pause);
      add(cb);
      exec.execute(cb);
    }
  }
  public static void main(String[] args) {
    ColorBoxes boxes = new ColorBoxes();
    if(args.length > 0)
      boxes.grid = new Integer(args[0]);
    if(args.length > 1)
      boxes.pause = new Integer(args[1]);
    boxes.setUp();
    run(boxes, 500, 400);
  }
} ///:~
```

ColorBoxes configures a **GridLayout** so that it has **grid** cells in each dimension. Then it adds the appropriate number of **CBox** objects to fill the grid, passing the **pause** value to each one. In **main( )** you can see how **pause** and **grid** have default values that can be changed if you pass in command-line arguments.

CBox is where all the work takes place. This is inherited from **JPanel** and it implements the **Runnable** interface so that each **JPanel** can also be an independent task. These tasks are driven by a thread pool **ExecutorService**.

The current cell color is **color**. Colors are created using the **Color** constructor that takes a 24-bit number, which in this case is created randomly.

paintComponent( ) is quite simple; it just sets the color to **color** and fills the entire **JPanel** with that color.

In **run( )**, you see the infinite loop that sets the **color** to a new random color and then calls **repaint( )** to show it. Then the thread goes to **sleep( )** for the amount of time specified on the command line.

The call to **repaint( )** in **run( )** deserves examination. At first glance, it may seem like we're creating a lot of threads, each of which is forcing a paint. It might appear that this is violating the principle that you should only submit tasks to the event queue. However, these threads are not actually modifying the shared resource. When they call **repaint( )**, it doesn't force a paint at that time, but only sets a "dirty flag" indicating that the next time the event dispatch thread is ready to repaint things, this area is a candidate for repainting. Thus the program doesn't cause Swing threading problems.

When the event dispatch thread actually does perform a **paint( )**, it first calls **paintComponent( )**, then **paintBorder( )** and **paintChildren( )**. If you need to override **paint( )** in a derived component, you must remember to call the base-class version of **paint( )** so that the proper actions are still performed.

Precisely because this design is flexible and threading is tied to each **JPanel** element, you can experiment by making as many threads as you want. (In reality, there is a restriction imposed by the number of threads your JVM can comfortably handle.)

This program also makes an interesting benchmark, since it can show dramatic performance and behavioral differences between one JVM threading implementation and another, as well as on different platforms.

Exercise 34: (4) Modify **ColorBoxes.java** so that it begins by sprinkling points ("stars") across the canvas, then randomly changes the colors of those "stars."

Visual programming and JavaBeans

So far in this book you've seen how valuable Java is for creating reusable pieces of code. The "most reusable" unit of code has been the class, since it comprises a cohesive unit of characteristics (fields) and behaviors (methods) that can be reused either directly via composition or through inheritance.

Inheritance and polymorphism are essential parts of object-oriented programming, but in the majority of cases when you're putting together an application, what you really want is *components* that do exactly what you need. You'd like to drop these parts into your design like the chips an electronic engineer puts on a circuit board. It seems that there should be some way to accelerate this "modular assembly" style of programming.

"Visual programming" first became successful—*very* successful—with Microsoft's Visual BASIC (VB), followed by a second-generation design in Borland's Delphi (which was the primary inspiration for the JavaBeans design). With these programming tools the components are represented visually, which makes sense since they usually display some kind of visual component such as a button or a text field. The visual representation, in fact, is often exactly the way the component will look in the running program. So part of the process of visual programming involves dragging a component from a palette and dropping it onto your form. The Application Builder Integrated Development Environment (IDE) writes code as you do this, and that code will cause the component to be created in the running program.

Simply dropping the component onto a form is usually not enough to complete the program. Often, you must change the characteristics of a component, such as its color, the text that's on it, the database it's connected to, etc. Characteristics that can be modified at design time are referred to as *properties*. You can manipulate the properties of your component inside the IDE, and when you create the program, this configuration data is saved so that it can be rejuvenated when the program is started.

By now you're probably used to the idea that an object is more than characteristics; it's also a set of behaviors. At design time, the behaviors of a visual component are partially represented by *events*, meaning "Here's something that can happen to the component." Ordinarily, you decide what you want to happen when an event occurs by tying code to that event.

Here's the critical part: The IDE uses reflection to dynamically interrogate the component and find out which properties and events the component supports. Once it knows what they are, it can display the properties and allow you to change them (saving the state when you build the program), and also display the events. In general, you do something like double-clicking on an event, and the IDE creates a code body and ties it to that particular event. All you must do at that point is write the code that executes when the event occurs.

All this adds up to a lot of work that's done for you by the IDE. As a result, you can focus on what the program looks like and what it is supposed to do, and rely on the IDE to manage the connection details for you. The reason that visual programming tools have been so successful is that they dramatically speed up the process of building an application—certainly the user interface, but often other portions of the application as well.

What is a JavaBean?

After the dust settles, then, a component is really just a block of code, typically embodied in a class. The key issue is the ability for the IDE to discover the properties and events for that component. To create a VB component, the programmer originally had to write a fairly complicated piece of code following certain conventions to expose the properties and events (it got easier as the years passed). Delphi was a second-generation visual programming tool, and the language was actively designed around visual programming, so it was much easier to create a visual component. However, Java has brought the creation of visual components to its most advanced state with JavaBeans, because a Bean is just a class. You don't have to write any extra code or use special language extensions in order to make something a Bean. The only thing you need to do, in fact, is slightly modify the way that you name your methods. It is the method name that tells the IDE whether this is a property, an event, or just an ordinary method.

In the JDK documentation, this naming convention is mistakenly termed a "design pattern." This is unfortunate, since design patterns (see *Thinking in Patterns* at *www.MindView.net)* are challenging enough without this sort of confusion. It's not a design pattern, it's just a naming convention, and it's fairly simple:

1. For a property named **xxx**, you typically create two methods: **getXxx( )** and **setXxx( )**. The first letter after "get" or "set" will automatically be lowercased by any tools that look at the methods, in order to produce the property name. The type produced by the "get" method is the same as the type of the argument to the "set" method. The name of the property and the type for the "get" and "set" are not related.

2. For a **boolean** property, you can use the "get" and "set" approach above, but you can also use "is" instead of "get."

3. Ordinary methods of the Bean don't conform to the above naming convention, but they're **public**.

4. For events, you use the Swing "listener" approach. It's exactly the same as you've been seeing: **addBounceListener(BounceListener)** and **removeBounceListener(BounceListener)** to handle a **BounceEvent**. Most of the time, the built-in events and listeners will satisfy your needs, but you can also create your own events and listener interfaces.

We can use these guidelines to create a simple Bean:

```
//: frogbean/Frog.java
// A trivial JavaBean.
package frogbean;
import java.awt.*;
import java.awt.event.*;

class Spots {}

public class Frog {
  private int jumps;
  private Color color;
  private Spots spots;
  private boolean jmpr;
  public int getJumps() { return jumps; }
  public void setJumps(int newJumps) {
    jumps = newJumps;
  }
  public Color getColor() { return color; }
  public void setColor(Color newColor) {
    color = newColor;
  }
  public Spots getSpots() { return spots; }
```

```
  public void setSpots(Spots newSpots) {
    spots = newSpots;
  }
  public boolean isJumper() { return jmpr; }
  public void setJumper(boolean j) { jmpr = j; }
  public void addActionListener(ActionListener l) {
    //...
  }
  public void removeActionListener(ActionListener l) {
    // ...
  }
  public void addKeyListener(KeyListener l) {
    // ...
  }
  public void removeKeyListener(KeyListener l) {
    // ...
  }
  // An "ordinary" public method:
  public void croak() {
    System.out.println("Ribbet!");
  }
} ///:~
```

First, you can see that it's just a class. Usually, all your fields will be **private** and accessible only through methods and properties. Following the naming convention, the properties are **jumps**, **color**, **spots**, and **jumper** (notice the case change of the first letter in the property name). Although the name of the internal identifier is the same as the name of the property in the first three cases, in **jumper** you can see that the property name does not force you to use any particular identifier for internal variables (or, indeed, to even *have* any internal variables for that property).

The events this Bean handles are **ActionEvent** and **KeyEvent**, based on the naming of the "add" and "remove" methods for the associated listener. Finally, you can see that the ordinary method **croak( )** is still part of the Bean simply because it's a **public** method, not because it conforms to any naming scheme.

Extracting **BeanInfo** with the **Introspector**

One of the most critical parts of the JavaBean scheme occurs when you drag a Bean off a palette and drop it onto a form. The IDE must be able to create the Bean (which it can do if there's a default constructor) and then, without access to the Bean's source code, extract all the necessary information to create the property sheet and event handlers.

Part of the solution is already evident from the *Type Information* chapter: Java *reflection* discovers all the methods of an unknown class. This is perfect for solving the JavaBean problem without requiring extra language keywords like those in other visual programming languages. In fact, one of the prime reasons that reflection was added to Java was to support JavaBeans (although reflection also supports object serialization and Remote Method Invocation, and is helpful in ordinary programming). So you might expect that the creator of the IDE would have to reflect each Bean and hunt through its methods to find the properties and events for that Bean.

This is certainly possible, but the Java designers wanted to provide a standard tool, not only to make Beans simpler to use, but also to provide a standard gateway to the creation of more complex Beans. This tool is the **Introspector** class, and the most important method in this class is the **static getBeanInfo( )**. You pass a **Class** reference to this method, and it fully

interrogates that class and returns a **BeanInfo** object which you can dissect to find properties, methods, and events.

Usually, you won't care about any of this; you'll probably get most of your Beans off the shelf, and you won't need to know all the magic that's going on underneath. You'll simply drag Beans onto your form, then configure their properties and write handlers for the events of interest. However, it's an educational exercise to use the **Introspector** to display information about a Bean. Here's a tool that does it:

```
//: gui/BeanDumper.java
// Introspecting a Bean.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.lang.reflect.*;
import static net.mindview.util.SwingConsole.*;

public class BeanDumper extends JFrame {
  private JTextField query = new JTextField(20);
  private JTextArea results = new JTextArea();
  public void print(String s) { results.append(s + "\n"); }
  public void dump(Class<?> bean) {
    results.setText("");
    BeanInfo bi = null;
    try {
      bi = Introspector.getBeanInfo(bean, Object.class);
    } catch(IntrospectionException e) {
      print("Couldn't introspect " +  bean.getName());
      return;
    }
    for(PropertyDescriptor d: bi.getPropertyDescriptors()){
      Class<?> p = d.getPropertyType();
      if(p == null) continue;
      print("Property type:\n  " + p.getName() +
        "Property name:\n  " + d.getName());
      Method readMethod = d.getReadMethod();
      if(readMethod != null)
        print("Read method:\n  " + readMethod);
      Method writeMethod = d.getWriteMethod();
      if(writeMethod != null)
        print("Write method:\n  " + writeMethod);
      print("====================");
    }
    print("Public methods:");
    for(MethodDescriptor m : bi.getMethodDescriptors())
      print(m.getMethod().toString());
    print("======================");
    print("Event support:");
    for(EventSetDescriptor e: bi.getEventSetDescriptors()){
      print("Listener type:\n  " +
        e.getListenerType().getName());
      for(Method lm : e.getListenerMethods())
        print("Listener method:\n  " + lm.getName());
      for(MethodDescriptor lmd :
          e.getListenerMethodDescriptors() )
        print("Method descriptor:\n  " + lmd.getMethod());
      Method addListener= e.getAddListenerMethod();
      print("Add Listener Method:\n  " + addListener);
      Method removeListener = e.getRemoveListenerMethod();
      print("Remove Listener Method:\n  "+ removeListener);
      print("====================");
```

```
    }
  }
  class Dumper implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String name = query.getText();
      Class<?> c = null;
      try {
        c = Class.forName(name);
      } catch(ClassNotFoundException ex) {
        results.setText("Couldn't find " + name);
        return;
      }
      dump(c);
    }
  }
  public BeanDumper() {
    JPanel p = new JPanel();
    p.setLayout(new FlowLayout());
    p.add(new JLabel("Qualified bean name:"));
    p.add(query);
    add(BorderLayout.NORTH, p);
    add(new JScrollPane(results));
    Dumper dmpr = new Dumper();
    query.addActionListener(dmpr);
    query.setText("frogbean.Frog");
    // Force evaluation
    dmpr.actionPerformed(new ActionEvent(dmpr, 0, ""));
  }
  public static void main(String[] args) {
    run(new BeanDumper(), 600, 500);
  }
} ///:~
```

BeanDumper.dump( ) does all the work. First it tries to create a **BeanInfo** object, and if successful, calls the methods of **BeanInfo** that produce information about properties, methods, and events. In **Introspector.getBeanInfo( )**, you'll see there is a second argument that tells the **Introspector** where to stop in the inheritance hierarchy. Here, it stops before it parses all the methods from **Object**, since we're not interested in seeing those.

For properties, **getPropertyDescriptors( )** returns an array of **PropertyDescriptors**. For each **PropertyDescriptor**, you can call **getPropertyType( )** to find the class of object that is passed in and out via the property methods. Then, for each property, you can get its pseudonym (extracted from the method names) with **getName( )**, the method for reading with **getReadMethod( )**, and the method for writing with **getWriteMethod( )**. These last two methods return a **Method** object that can actually be used to invoke the corresponding method on the object (this is part of reflection).

For the **public** methods (including the property methods), **getMethodDescriptors( )** returns an array of **MethodDescriptors**. For each one, you can get the associated **Method** object and print its name.

For the events, **getEventSetDescriptors( )** returns an array of **EventSetDescriptors**. Each of these can be queried to find out the class of the listener, the methods of that listener class, and the add- and removelistener methods. The **BeanDumper** program displays all of this information.

Upon startup, the program forces the evaluation of **frogbean.Frog**. The output, after unnecessary details have been removed, is:

```
Property type:
```

```
  Color
Property name:
  color
Read method:
  public Color getColor()
Write method:
  public void setColor(Color)
====================
Property type:
  boolean
Property name:
  jumper
Read method:
  public boolean isJumper()
Write method:
  public void setJumper(boolean)
====================
Property type:
  int
Property name:
  jumps
Read method:
  public int getJumps()
Write method:
  public void setJumps(int)
====================
Property type:
  frogbean.Spots
Property name:
  spots
Read method:
  public frogbean.Spots getSpots()
Write method:
  public void setSpots(frogbean.Spots)
====================
Public methods:
public void setSpots(frogbean.Spots)
public void setColor(Color)
public void setJumps(int)
public boolean isJumper()
public frogbean.Spots getSpots()
public void croak()
public void addActionListener(ActionListener)
public void addKeyListener(KeyListener)
public Color getColor()
public void setJumper(boolean)
public int getJumps()
public void removeActionListener(ActionListener)
public void removeKeyListener(KeyListener)
====================
Event support:
Listener type:
  KeyListener
Listener method:
  keyPressed
Listener method:
  keyReleased
Listener method:
  keyTyped
Method descriptor:
  public abstract void keyPressed(KeyEvent)
Method descriptor:
  public abstract void keyReleased(KeyEvent)
```

```
Method descriptor:
  public abstract void keyTyped(KeyEvent)
AddListener Method:
  public void addKeyListener(KeyListener)
Remove Listener Method:
  public void removeKeyListener(KeyListener)
====================
Listener type:
  ActionListener
Listener method:
  actionPerformed
Method descriptor:
  public abstract void actionPerformed(ActionEvent)
Add Listener Method:
  public void addActionListener(ActionListener)
Remove Listener Method:
  public void removeActionListener(ActionListener)
====================
```

This reveals most of what the **Introspector** sees as it produces a **BeanInfo** object from your Bean. You can see that the type of the property and its name are independent. Notice the lowercasing of the property name. (The only time this doesn't occur is when the property name begins with more than one capital letter in a row.) And remember that the method names you're seeing here (such as the read and write methods) are actually produced from a **Method** object that can be used to invoke the associated method on the object.

The **public** method list includes the methods that are not associated with a property or an event, such as **croak( )**, as well as those that are. These are all the methods that you can call programmatically for a Bean, and the IDE can choose to list all of these while you're making method calls, to ease your task.

Finally, you can see that the events are fully parsed out into the listener, its methods, and the add- and remove-listener methods. Basically, once you have the **BeanInfo**, you can find out everything of importance for the Bean. You can also call the methods for that Bean, even though you don't have any other information except the object (again, a feature of reflection).

A more sophisticated Bean

This next example is slightly more sophisticated, albeit frivolous. It's a **JPanel** that draws a little circle around the mouse whenever the mouse is moved. When you press the mouse, the word "Bang!" appears in the middle of the screen, and an action listener is fired.

The properties you can change are the size of the circle as well as the color, size, and text of the word that is displayed when you press the mouse. A **BangBean** also has its own **addActionListener( )** and **removeActionListener( )**, so you can attach your own listener that will be fired when the user clicks on the **BangBean**. You should recognize the property and event support:

```
//: bangbean/BangBean.java
// A graphical Bean.
package bangbean;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;

public class
BangBean extends JPanel implements Serializable {
```

```
  private int xm, ym;
  private int cSize = 20; // Circle size
  private String text = "Bang!";
  private int fontSize = 48;
  private Color tColor = Color.RED;
  private ActionListener actionListener;
  public BangBean() {
    addMouseListener(new ML());
    addMouseMotionListener(new MML());
  }
  public int getCircleSize() { return cSize; }
  public void setCircleSize(int newSize) {
    cSize = newSize;
  }
  public String getBangText() { return text; }
  public void setBangText(String newText) {
    text = newText;
  }
  public int getFontSize() { return fontSize; }
  public void setFontSize(int newSize) {
    fontSize = newSize;
  }
  public Color getTextColor() { return tColor; }
  public void setTextColor(Color newColor) {
    tColor = newColor;
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.BLACK);
    g.drawOval(xm - cSize/2, ym - cSize/2, cSize, cSize);
  }
  // This is a unicast listener, which is
  // the simplest form of listener management:
  public void addActionListener(ActionListener l)
  throws TooManyListenersException {
    if(actionListener != null)
      throw new TooManyListenersException();
    actionListener = l;
  }
  public void removeActionListener(ActionListener l) {
    actionListener = null;
  }
  class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      Graphics g = getGraphics();
      g.setColor(tColor);
      g.setFont(
        new Font("TimesRoman", Font.BOLD, fontSize));
      int width = g.getFontMetrics().stringWidth(text);
      g.drawString(text, (getSize().width - width) /2,
        getSize().height/2);
      g.dispose();
      // Call the listener's method:
      if(actionListener != null)
        actionListener.actionPerformed(
          new ActionEvent(BangBean.this,
            ActionEvent.ACTION_PERFORMED, null));
    }
  }
  class MML extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
      xm = e.getX();
      ym = e.getY();
```

```
      repaint();
    }
  }
  public Dimension getPreferredSize() {
    return new Dimension(200, 200);
  }
} ///:~
```

The first thing you'll notice is that **BangBean** implements the **Serializable** interface. This means that the IDE can "pickle" all the information for the **BangBean** by using serialization after the program designer has adjusted the values of the properties. When the Bean is created as part of the running application, these "pickled" properties are restored so that you get exactly what you designed.

When you look at the signature for **addActionListener( )**, you'll see that it can throw a **TooManyListenersException**. This indicates that it is *unicast*, which means it notifies only one listener when the event occurs. Ordinarily, you'll use *multicast* events so that many listeners can be notified of an event. However, that runs into threading issues, so it will be revisited in the next section, "JavaBeans and synchronization." In the meantime, a unicast event sidesteps the problem.

When you click the mouse, the text is put in the middle of the **BangBean**, and if the **actionListener** field is not **null**, its **actionPerformed( )** is called, creating a new **ActionEvent** object in the process. Whenever the mouse is moved, its new coordinates are captured and the canvas is repainted (erasing any text that's on the canvas, as you'll see).

Here is the **BangBeanTest** class to test the Bean:

```
//: bangbean/BangBeanTest.java
// {Timeout: 5} Abort after 5 seconds when testing
package bangbean;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

public class BangBeanTest extends JFrame {
  private JTextField txt = new JTextField(20);
  // During testing, report actions:
  class BBL implements ActionListener {
    private int count = 0;
    public void actionPerformed(ActionEvent e) {
      txt.setText("BangBean action "+ count++);
    }
  }
  public BangBeanTest() {
    BangBean bb = new BangBean();
    try {
      bb.addActionListener(new BBL());
    } catch(TooManyListenersException e) {
      txt.setText("Too many listeners");
    }
    add(bb);
    add(BorderLayout.SOUTH, txt);
  }
  public static void main(String[] args) {
    run(new BangBeanTest(), 400, 500);
  }
} ///:~
```

When a Bean is used in an IDE, this class will not be used, but it's helpful to provide a rapid testing method for each of your Beans. **BangBeanTest** places a **BangBean** within the **JFrame**, attaching a simple **ActionListener** to the **BangBean** to print an event count to the **JTextField** whenever an **ActionEvent** occurs. Usually, of course, the IDE would create most of the code that uses the Bean.

When you run the **BangBean** through **BeanDumper** or put the **BangBean** inside a Bean-enabled development environment, you'll notice that there are many more properties and actions than are evident from the preceding code. That's because **BangBean** is inherited from **JPanel**, and **JPanel** is also a Bean, so you're seeing its properties and events as well.

Exercise 35: (6) Locate and download one or more of the free GUI builder development environments available on the Internet, or use a commercial product if you own one. Discover what is necessary to add **BangBean** to this environment and to use it.

JavaBeans and synchronization

Whenever you create a Bean, you must assume that it will run in a multithreaded environment. This means that:

1. Whenever possible, all the **public** methods of a Bean should be **synchronized**. Of course, this incurs the **synchronized** runtime overhead (which has been significantly reduced in recent versions of the JDK). If that's a problem, methods that will not cause problems in critical sections can be left un**synchronized**, but keep in mind that such methods are not always obvious. Methods that qualify tend to be small (such as **getCircleSize( )** in the following example) and/or "atomic"; that is, the method call executes in such a short amount of code that the object cannot be changed during execution (but review the *Concurrency* chapter— what you may think is atomic might not be). Making such methods un**synchronized** might not have a significant effect on the execution speed of your program. You're better off making all **public** methods of a Bean **synchronized** and removing the **synchronized** keyword on a method only when you know for sure that it makes a difference and that you can safely remove the keyword.

2. When firing a multicast event to a bunch of listeners interested in that event, you must assume that listeners might be added or removed while moving through the list.

The first point is fairly straightforward, but the second point requires a little more thought. **BangBean.java** ducked out of the concurrency question by ignoring the **synchronized** keyword and making the event unicast. Here is a modified version that works in a multithreaded environment and uses multicasting for events:

```
//: gui/BangBean2.java
// You should write your Beans this way so they
// can run in a multithreaded environment.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

public class BangBean2 extends JPanel
implements Serializable {
  private int xm, ym;
  private int cSize = 20; // Circle size
  private String text = "Bang!";
  private int fontSize = 48;
```

```
  private Color tColor = Color.RED;
  private ArrayList<ActionListener> actionListeners =
    new ArrayList<ActionListener>();
  public BangBean2() {
    addMouseListener(new ML());
    addMouseMotionListener(new MM());
  }
  public synchronized int getCircleSize() { return cSize; }
  public synchronized void setCircleSize(int newSize) {
    cSize = newSize;
  }
  public synchronized String getBangText() { return text; }
  public synchronized void setBangText(String newText) {
    text = newText;
  }
  public synchronized int getFontSize(){ return fontSize; }
  public synchronized void setFontSize(int newSize) {
    fontSize = newSize;
  }
  public synchronized Color getTextColor(){ return tColor;}
  public synchronized void setTextColor(Color newColor) {
    tColor = newColor;
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.BLACK);
    g.drawOval(xm - cSize/2, ym - cSize/2, cSize, cSize);
  }
  // This is a multicast listener, which is more typically
  // used than the unicast approach taken in BangBean.java:
  public synchronized void
  addActionListener(ActionListener l) {
    actionListeners.add(l);
  }
  public synchronized void
  removeActionListener(ActionListener l) {
    actionListeners.remove(l);
  }
  // Notice this isn't synchronized:
  public void notifyListeners() {
    ActionEvent a = new ActionEvent(BangBean2.this,
      ActionEvent.ACTION_PERFORMED, null);
    ArrayList<ActionListener> lv = null;
    // Make a shallow copy of the List in case
    // someone adds a listener while we're
    // calling listeners:
    synchronized(this) {
      lv = new ArrayList<ActionListener>(actionListeners);
    }
    // Call all the listener methods:
    for(ActionListener al : lv)
      al.actionPerformed(a);
  }
  class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      Graphics g = getGraphics();
      g.setColor(tColor);
      g.setFont(
        new Font("TimesRoman", Font.BOLD, fontSize));
      int width = g.getFontMetrics().stringWidth(text);
      g.drawString(text, (getSize().width - width) /2,
        getSize().height/2);
      g.dispose();
```

```
      notifyListeners();
    }
  }
  class MM extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
      xm = e.getX();
      ym = e.getY();
      repaint();
    }
  }
  public static void main(String[] args) {
    BangBean2 bb2 = new BangBean2();
    bb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.out.println("ActionEvent" + e);
      }
    });
    bb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.out.println("BangBean2 action");
      }
    });
    bb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.out.println("More action");
      }
    });
    JFrame frame = new JFrame();
    frame.add(bb2);
    run(frame, 300, 300);
  }
} ///:~
```

Adding **synchronized** to the methods is an easy change. However, notice in **addActionListener( )** and **removeActionListener( )** that the **ActionListener**s are now added to and removed from an **ArrayList**, so you can have as many as you want.

You can see that the method **notifyListeners( )** is *not* **synchronized**. It can be called from more than one thread at a time. It's also possible for **addActionListener( )** or **removeActionListener( )** to be called in the middle of a call to **notifyListeners( )**, which is a problem because it traverses the **ArrayList actionListeners**. To alleviate the problem, the **ArrayList** is duplicated inside a **synchronized** clause, using the **ArrayList** constructor which copies the elements of its argument, and the duplicate is traversed. This way, the original **ArrayList** can be manipulated without impact on **notifyListeners( )**.

The **paintComponent( )** method is also not **synchronized**. Deciding whether to synchronize overridden methods is not as clear as when you're just adding your own methods. In this example, it turns out that **paintComponent( )** seems to work OK whether it's **synchronized** or not. But the issues you must consider are:

1. Does the method modify the state of "critical" variables within the object? To discover whether the variables are "critical," you must determine whether they will be read or set by other threads in the program. (In this case, the reading or setting is virtually always accomplished via **synchronized** methods, so you can just examine those.) In the case of **paintComponent( )**, no modification takes place.

2. Does the method depend on the state of these "critical" variables? If a **synchronized** method modifies a variable that your method uses, then you might very well want to make your method **synchronized** as well. Based on this, you might observe that **cSize** is changed by **synchronized** methods, and therefore **paintComponent( )**

should be **synchronized**. Here, however, you can ask, "What's the worst thing that will happen if **cSize** is changed during a **paintComponent( )**?" When you see that it's nothing too bad, and a transient effect at that, you can decide to leave **paintComponent( ) unsynchronized** to prevent the extra overhead from the **synchronized** method call.

3. A third clue is to notice whether the base-class version of **paintComponent( )** is **synchronized**, which it isn't. This isn't an airtight argument, just a clue. In this case, for example, a field that *is* changed via **synchronized** methods (that is, **cSize)** has been mixed into the **paintComponent( )** formula and might have changed the situation. Notice, however, that **synchronized** doesn't inherit; that is, if a method is **synchronized** in the base class, then it *is not* automatically **synchronized** in the derivedclass overridden version.

4. **paint( )** and **paintComponent( )** are methods that must be as fast as possible. Anything that takes processing overhead out of these methods is highly recommended, so if you think you need to synchronize these methods it may be an indicator of bad design.

The test code in **main( )** has been modified from that seen in **BangBeanTest** to demonstrate the multicast ability of **BangBean2** by adding extra listeners.

Packaging a Bean

Before you can bring a JavaBean into a Bean-enabled IDE, it must be put into a Bean container, which is a JAR file that includes all the Bean classes as well as a "manifest" file that says, "This is a Bean." A manifest file is simply a text file that follows a particular form. For the **BangBean**, the manifest file looks like this:

```
Manifest-Version: 1.0

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

The first line indicates the version of the manifest scheme, which until further notice from Sun is 1.0. The second line (empty lines are ignored) names the **BangBean.class** file, and the third says, "It's a Bean." Without the third line, the program builder tool will not recognize the class as a Bean.

The only tricky part is that you must make sure that you get the proper path in the "Name:" field. If you look back at **BangBean.java**, you'll see it's in **package bangbean** (and thus in a subdirectory called **bangbean** that's off of the classpath), and the name in the manifest file must include this package information. In addition, you must place the manifest file in the directory *above* the root of your package path, which in this case means placing the file in the directory above the "bangbean" subdirectory. Then you must invoke **jar** from the same directory as the manifest file, as follows:

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

This assumes that you want the resulting JAR file to be named **BangBean.jar**, and that you've put the manifest in a file called **BangBean.mf**.

You might wonder, "What about all the other classes that were generated when I compiled **BangBean.java**?" Well, they all ended up inside the **bangbean** subdirectory, and you'll see that the last argument for the above **jar** command line is the **bangbean** subdirectory. When you give **jar** the name of a subdirectory, it packages that entire subdirectory into the JAR file (including, in this case, the original **BangBean.java** source-code file—you might not choose

to include the source with your own Beans). In addition, if you turn around and unpack the JAR file you've just created, you'll discover that your manifest file isn't inside, but that **jar** has created its own manifest file (based partly on yours) called **MANIFEST.MF** and placed it inside the subdirectory **META-INF** (for "meta-information"). If you open this manifest file, you'll also notice that digital signature information has been added by **jar** for each file, of the form:

```
Digest-Algorithms: SHA MD5
SHA-Digest: pDpEAG9NaeCx8aFtqPI4udSX/O0=
MD5-Digest: 04NcSlhE3Smnzlp2hj6qeg==
```

In general, you don't need to worry about any of this, and if you make changes, you can just modify your original manifest file and reinvoke **jar** to create a new JAR file for your Bean. You can also add other Beans to the JAR file simply by adding their information to your manifest.

One thing to notice is that you'll probably want to put each Bean in its own subdirectory, since when you create a JAR file you hand the **jar** utility the name of a subdirectory, and it puts everything in that subdirectory into the JAR file. You can see that both **Frog** and **BangBean** are in their own subdirectories.

Once you have your Bean properly inside a JAR file, you can bring it into a Beans-enabled IDE. The way you do this varies from one tool to the next, but Sun provides a freely available test bed for JavaBeans in its "Bean Builder." (Download from *http://java.sun.com/beans.)* You place a Bean into the Bean Builder by simply copying the JAR file into the correct subdirectory.

Exercise 36: (4) Add **Frog.class** to the manifest file in this section and run **jar** to create a JAR file containing both **Frog** and **BangBean**. Now either download and install the Bean Builder from Sun, or use your own Beans-enabled program builder tool and add the JAR file to your environment so you can test the two Beans.

Exercise 37: (5) Create your own JavaBean called **Valve** that contains two properties: a **boolean** called "on" and an **int** called "level." Create a manifest file, use **jar** to package your Bean, then load it into the Bean Builder or into a Beans-enabled program builder tool so that you can test it.

More complex Bean support

You can see how remarkably simple it is to make a Bean, but you aren't limited to what you've seen here. The JavaBeans architecture provides a simple point of entry but can also scale to more complex situations. These situations are beyond the scope of this book, but they will be briefly introduced here. You can find more details at *http://java.sun.com/beans*.

One place where you can add sophistication is with properties. The examples you've seen here have shown only single properties, but it's also possible to represent multiple properties in an array. This is called an *indexed property*. You simply provide the appropriate methods (again following a naming convention for the method names), and the **Introspector** recognizes an indexed property so that your IDE can respond appropriately.

Properties can be *bound*, which means that they will notify other objects via a **PropertyChangeEvent**. The other objects can then choose to change themselves based on the change to the Bean.

Properties can be *constrained*, which means that other objects can veto a change to that property if it is unacceptable. The other objects are notified by using a

PropertyChangeEvent, and they can throw a **PropertyVetoException** to prevent the change from happening and to restore the old values.

You can also change the way your Bean is represented at design time:

1. You can provide a custom property sheet for your particular Bean. The ordinary property sheet will be used for all other Beans, but yours is automatically invoked when your Bean is selected.

2. You can create a custom editor for a particular property, so the ordinary property sheet is used, but when your special property is being edited, your editor will automatically be invoked.

3. You can provide a custom **BeanInfo** class for your Bean that produces information different from the default created by the **Introspector**.

4. It's also possible to turn "expert" mode on and off in all **FeatureDescriptor**s to distinguish between basic features and more complicated ones.

More to Beans

There are a number of books about JavaBeans; for example, *JavaBeans* by Elliotte Rusty Harold (IDG, 1998).

Alternatives to Swing

Although the Swing library is the GUI sanctioned by Sun, it is by no means the only way to create graphical user interfaces. Two important alternatives are *Macromedia Flash*, using Macromedia's *Flex* programming system, for client-side GUIs over the Web, and the open-source Eclipse *Standard Widget Toolkit* (SWT) library for desktop applications.

Why would you consider alternatives? For Web clients, you can make a fairly strong argument that applets have failed. Considering how long they've been around (since the beginning) and the initial hype and promise around applets, coming across a Web application that uses applets is still a surprise. Even Sun doesn't use applets everywhere. Here's an example:

http://java.sun.com/developer/onlineTraining/new2java/javamap/intro.html

An interactive map of Java features on the Sun site seems a very likely candidate for a Java applet, and yet they did it in Flash. This appears to be a tacit acknowledgement that applets have not been a success. More importantly, the Flash Player is installed on upwards of 98 percent of computing platforms, so it can be considered an accepted standard. As you'll see, the Flex system provides a very powerful client-side programming environment, certainly more powerful than JavaScript and with a look and feel that is often preferable to an applet. If you want to use applets, you must still convince the client to download the JRE, whereas the Flash Player is small and fast to download by comparison.

For desktop applications, one problem with Swing is that users *notice* that they are using a different kind of application, because the look and feel of Swing applications is different from the normal desktop. Users are not generally interested in new looks and feels in an application; they are trying to get work done and prefer that an application look and feel like all their other applications. SWT creates applications that look like native applications, and because the library uses native components as much as possible, the applications tend to run faster than equivalent Swing applications.

Building Flash Web clients with Flex

Because the lightweight Macromedia Flash virtual machine is so ubiquitous, most people will be able to use a Flash-based interface without installing anything, and it will look and behave the same way across all systems and platforms.[10]

With *Macromedia Flex*, you can develop Flash user interfaces for Java applications. Flex consists of an XML- and script-based programming model, similar to programming models such as HTML and JavaScript, along with a robust library of components. You use the MXML syntax to declare layout management and widget controls, and you use dynamic scripting to add event-handling and service invocation code which links the user interface to Java classes, data models, Web services, etc. The Flex compiler takes your MXML and script files and compiles them into bytecode. The Flash virtual machine on the client operates like the Java Virtual Machine in that it interprets compiled bytecode. The Flash bytecode format is known as SWF, and SWF files are produced by the Flex compiler.

Note that there's an open-source alternative to Flex at *http://openlaszlo.org;* this has a structure that's similar to Flex but may be a preferable alternative for some. Other tools also exist to create Flash applications in different ways.

Hello, Flex

Consider this MXML code, which defines a user interface (note that the first and last lines will not appear in the code that you download as part of this book's source-code package):

```
//:! gui/flex/helloflex1.mxml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.macromedia.com/2003/mxml"
  backgroundColor="#ffffff">
  <mx:Label id="output" text="Hello, Flex!" />
</mx:Application>
///:~
```

MXML files are XML documents, so they begin with an XML version/encoding directive. The outermost MXML element is the **Application** element, which is the topmost visual and logical container for a Flex user interface. You can declare tags representing visual controls, such as the **Label** element above, inside the **Application** element. Controls are always placed within a container, and containers encapsulate layout managers, among other mechanisms, so they manage the layout of the controls within them. In the simplest case, as in the above example, the **Application** acts as the container. The **Application's** default layout manager merely places controls vertically down the interface in the order in which they are declared.

ActionScript is a version of ECMAScript, or JavaScript, which looks quite similar to Java and supports classes and strong typing in addition to dynamic scripting. By adding a script to the example, we can introduce behavior. Here, the MXML **Script** control is used to place ActionScript directly into the MXML file:

```
//:! gui/flex/helloflex2.mxml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
```

[10] Sean Neville created the core of the material in this section.

```
  xmlns:mx="http://www.macromedia.com/2003/mxml"
  backgroundColor="#ffffff">
  <mx:Script>
    <![CDATA[
    function updateOutput() {
      output.text = "Hello! " + input.text;
    }
    ]]>
  </mx:Script>
  <mx:TextInput id="input" width="200"
    change="updateOutput()" />
  <mx:Label id="output" text="Hello!" />
</mx:Application>
///:~
```

A **TextInput** control accepts user input, and a **Label** displays the data as it is being typed. Note that the **id** attribute of each control becomes accessible in the script as a variable name, so the script can reference instances of the MXML tags. In the **TextInput** field, you can see that the **change** attribute is connected to the **updateOutput( )** function so that the function is called whenever any kind of change occurs.

Compiling MXML

The easiest way to get started using Flex is with the free trial, which you can download at *www.macromedia.com/software/flex/trial*.[11] The product is packaged in a number of editions, from free trials to enterprise server versions, and Macromedia offers additional tools for developing Flex applications. Exact packaging is subject to change, so check the Macromedia site for specifics. Also note that you may need to modify the **jvm.config** file in the Flex installation **bin** directory.

To compile the MXML code into Flash bytecode, you have two options:

1. You can place the MXML file in a Java Web application, alongside JSP and HTML pages in a WAR file, and have requests for the **.mxml** file compiled at run time whenever a browser requests the MXML document's URL.

2. You can compile the MXML file using the Flex command-line compiler, **mxmlc**.

The first option, Web-based runtime compilation, requires a servlet container (such as Apache Tomcat) in addition to Flex. The servlet container's WAR file(s) must be updated with Flex configuration information, such as servlet mappings which are added to the **web.xml** descriptor, and it must include the Flex JAR files—these steps are handled automatically when you install Flex. After the WAR file is configured, you can place the MXML files in the Web application and request the document's URL through any browser. Flex will compile the application upon the first request, similar to the JSP model, and will thereafter deliver the compiled and cached SWF within an HTML shell.

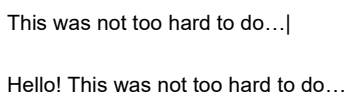
The second option does not require a server. When you invoke the Flex **mxmlc** compiler on the command line, you produce SWF files. You can deploy these as you desire. The **mxmlc** executable is located in the **bin** directory of a Flex installation, and invoking it with no arguments will provide a list of valid command-line options. Typically, you'll specify the location of the Flex client component library as the value of the **-flexlib** command-line option, but in very simple examples like the two that we've seen so far, the Flex compiler will assume the location of the component library. So you can compile the first two examples like this:

[11] Note that you must download Flex, and not FlexBuilder. The latter is an IDE design tool.

```
mxmlc.exe helloflex1.mxml
mxmlc.exe helloflex2.mxml
```

This produces a **helloflex2.swf** file which can be run in Flash, or placed alongside HTML on any HTTP server (once Flash has been loaded into your Web browser, you can often just double-click on the SWF file to start it up in the browser).

For **helloflex2.swf**, you'll see the following user interface running in the Flash Player:



In more complex applications, you can separate MXML and ActionScript by referencing functions in external ActionScript files. From MXML, you use the following syntax for the **Script** control:

```
 <mx:Script source="MyExternalScript.as" />
```

This code allows the MXML controls to reference functions located in a file named **MyExternalScript.as** as if they were located within the MXML file.

MXML and ActionScript

MXML is declarative shorthand for ActionScript classes. Whenever you see an MXML tag, there exists an ActionScript class of the same name. When the Flex compiler parses MXML, it first transforms the XML into ActionScript and loads the referenced ActionScript classes, and then compiles and links the ActionScript into an SWF.

You can write an entire Flex application in ActionScript alone, without using any MXML. Thus, MXML is a convenience. User interface components such as containers and controls are typically declared using MXML, while logic such as event handling and other client logic is handled through ActionScript and Java.

You can create your own MXML controls and reference them using MXML by writing ActionScript classes. You may also combine existing MXML containers and controls in a new MXML document that can then be referenced as a tag in another MXML document. The Macromedia Web site contains more information about how to do this.

Containers and controls

The visual core of the Flex component library is a set of containers which manage layout, and an array of controls which go inside those containers. Containers include panels, vertical and horizontal boxes, tiles, accordions, divided boxes, grids, and more. Controls are user interface widgets such as buttons, text areas, sliders, calendars, data grids, and so forth.

The remainder of this section will show a Flex application that displays and sorts a list of audio files. This application demonstrates containers, controls, and how to connect to Java from Flash.

We start the MXML file by placing a **DataGrid** control (one of the more sophisticated Flex controls) within a **Panel** container:

```
//:! gui/flex/songs.mxml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
```

```
  xmlns:mx="http://www.macromedia.com/2003/mxml"
  backgroundColor="#B9CAD2" pageTitle="Flex Song Manager"
  initialize="getSongs()">
  <mx:Script source="songScript.as" />
  <mx:Style source="songStyles.css"/>
  <mx:Panel id="songListPanel"
    titleStyleDeclaration="headerText"
    title="Flex MP3 Library">
    <mx:HBox verticalAlign="bottom">
      <mx:DataGrid id="songGrid"
        cellPress="selectSong(event)" rowCount="8">
        <mx:columns>
          <mx:Array>
            <mx:DataGridColumn columnName="name"
              headerText="Song Name" width="120" />
            <mx:DataGridColumn columnName="artist"
              headerText="Artist" width="180" />
            <mx:DataGridColumn columnName="album"
              headerText="Album" width="160" />
          </mx:Array>
        </mx:columns>
      </mx:DataGrid>
      <mx:VBox>
        <mx:HBox height="100" >
          <mx:Image id="albumImage" source=""
            height="80" width="100"
            mouseOverEffect="resizeBig"
            mouseOutEffect="resizeSmall" />
          <mx:TextArea id="songInfo"
            styleName="boldText" height="100%" width="120"
            vScrollPolicy="off" borderStyle="none" />
        </mx:HBox>
        <mx:MediaPlayback id="songPlayer"
          contentPath=""
          mediaType="MP3"
          height="70"
          width="230"
          controllerPolicy="on"
          autoPlay="false"
          visible="false" />
      </mx:VBox>
    </mx:HBox>
    <mx:ControlBar horizontalAlign="right">
      <mx:Button id="refreshSongsButton"
        label="Refresh Songs" width="100"
        toolTip="Refresh Song List"
        click="songService.getSongs()" />
    </mx:ControlBar>
  </mx:Panel>
  <mx:Effect>
    <mx:Resize name="resizeBig" heightTo="100"
      duration="500"/>
    <mx:Resize name="resizeSmall" heightTo="80"
      duration="500"/>
  </mx:Effect>
  <mx:RemoteObject id="songService"
    source="gui.flex.SongService"
    result="onSongs(event.result)"
    fault="alert(event.fault.faultstring, 'Error')">
    <mx:method name="getSongs"/>
  </mx:RemoteObject>
</mx:Application>
///:~
```

The **DataGrid** contains nested tags for its array of columns. When you see an attribute or a nested element on a control, you know that it corresponds to some property, event, or encapsulated object in the underlying ActionScript class. The **DataGrid** has an **id** attribute with the value **songGrid**, so ActionScript and MXML tags can reference the grid programmatically by using **songGrid** as a variable name. The **DataGrid** exposes many more properties than those shown here; the complete API for MXML controls and containers can be found online at *http ://livedocs. macromedia. com/flex/is/asdocs_ en/index.html.*

The **DataGrid** is followed by a **VBox** containing an **Image** to show the front of the album along with song information, and a **MediaPlayback** control that will play MP3 files. This example streams the content in order to reduce the size of the compiled SWF. When you embed images, audio, and video files into a Flex application instead of streaming them, the files become part of the compiled SWF and are delivered along with your user interface assets instead of streamed on demand at run time.

The Flash Player contains embedded codecs for playing and streaming audio and video in a variety of formats. Flash and Flex support the use of the Web's most common image formats, and Flex also has the ability to translate *scalable vector graphics* (SVG) files into SWF resources that can be embedded in Flex clients.

Effects and styles

The Flash Player renders graphics using vectors, so it can perform highly expressive transformations at run time. Flex *effects* provide a small taste of these sorts of animations. Effects are transformations that you can apply to controls and containers using MXML syntax.

The **Effect** tag shown in the MXML produces two results: The first nested tag dynamically grows an image when the mouse hovers over it, and the second dynamically shrinks that image when the mouse moves away. These effects are applied to the mouse events available on the **Image** control for **albumlmage**.

Flex also provides effects for common animations like transitions, wipes, and modulating alpha channels. In addition to the built-in effects, Flex supports the Flash drawing API for truly innovative animations. Deeper exploration of this topic involves graphic design and animation, and is beyond the scope of this section.

Standard styling is available through Flex's support for Cascading Style Sheets (CSS). If you attach a CSS file to an MXML file, the Flex controls will follow those styles. For this example, **songStyles.css** contains the following CSS declaration:

```
//:! gui/flex/songStyles.css
.headerText {
  font-family: Arial, "_sans";
  font-size: 16;
  font-weight: bold;
}

.boldText {
  font-family: Arial, "_sans";
  font-size: 11;
  font-weight: bold;
}
///:~
```

This file is imported and used in the song library application via the **Style** tag in the MXML file. After the style sheet is imported, its declarations can be applied to Flex controls in the

MXML file. As an example, the style sheet's **boldText** declaration is used by the **TextArea** control with the **songInfo id**.

Events

A user interface is a state machine; it performs actions as state changes occur. In Flex, these changes are managed through events. The Flex class library contains a wide variety of controls with extensive events covering all aspects of mouse movement and keyboard usage.

The **click** attribute of a **Button**, for example, represents one of the events available on that control. The value assigned to **click** can be a function or an inline bit of script. In the MXML file, for example, the **ControlBar** holds the **refreshSongsButton** to refresh the list of songs. You can see from the tag that when the **click** event occurs, **songService.getSongs( )** is called. In this example, the **click** event of the **Button** refers to the **RemoteObject** which corresponds to the Java method.

Connecting to Java

The **RemoteObject** tag at the end of the MXML file sets up the connection to the external Java class, **gui.flex.SongService**. The Flex client will use the **getSongs( )** method in the Java class to retrieve the data for the **DataGrid**. To do so, it must appear as a *service*—an endpoint with which the client can exchange messages. The service defined in the **RemoteObject** tag has a **source** attribute which denotes the Java class of the **RemoteObject**, and it specifies an ActionScript callback function, **onSongs( )**, to be invoked when the Java method returns. The nested **method** tag declares the method **getSongs( )**, which makes that Java method accessible to the rest of the Flex application.

All invocations of services in Flex return asynchronously, through events fired to these callback functions. The **RemoteObject** also raises an alert dialog control in the event of an error.

The **getSongs( )** method may now be invoked from Flash using ActionScript:

```
songService.getSongs();
```

Because of the MXML configuration, this will call **getSongs( )** in the **SongService** class:

```
//: gui/flex/SongService.java
package gui.flex;
import java.util.*;

public class SongService {
  private List<Song> songs = new ArrayList<Song>();
  public SongService() { fillTestData(); }
  public List<Song> getSongs() { return songs; }
  public void addSong(Song song) { songs.add(song); }
  public void removeSong(Song song) { songs.remove(song); }
  private void fillTestData() {
    addSong(new Song("Chocolate", "Snow Patrol",
      "Final Straw", "sp-final-straw.jpg",
      "chocolate.mp3"));
    addSong(new Song("Concerto No. 2 in E", "Hilary Hahn",
      "Bach: Violin Concertos", "hahn.jpg",
      "bachviolin2.mp3"));
    addSong(new Song("'Round Midnight", "Wes Montgomery",
      "The Artistry of Wes Montgomery",
      "wesmontgomery.jpg", "roundmidnight.mp3"));
```

```
  }
} ///:~
```

Each **Song** object is just a data container:

```
//: gui/flex/Song.java
package gui.flex;

public class Song implements java.io.Serializable {
  private String name;
  private String artist;
  private String album;
  private String albumImageUrl;
  private String songMediaUrl;
  public Song() {}
  public Song(String name, String artist, String album,
  String albumImageUrl, String songMediaUrl) {
    this.name = name;
    this.artist = artist;
    this.album = album;
    this.albumImageUrl = albumImageUrl;
    this.songMediaUrl = songMediaUrl;
  }
  public void setAlbum(String album) { this.album = album;}
  public String getAlbum() { return album; }
  public void setAlbumImageUrl(String albumImageUrl) {
    this.albumImageUrl = albumImageUrl;
  }
  public String getAlbumImageUrl() { return albumImageUrl;}
  public void setArtist(String artist) {
    this.artist = artist;
  }
  public String getArtist() { return artist; }
  public void setName(String name) { this.name = name; }
  public String getName() { return name; }
  public void setSongMediaUrl(String songMediaUrl) {
    this.songMediaUrl = songMediaUrl;
  }
  public String getSongMediaUrl() { return songMediaUrl; }
} ///:~
```

When the application is initialized or you press the **refreshSongsButton**, **getSongs( )** is called, and upon returning, the ActionScript **onSongs(event.result)** is called to populate the **songGrid**.

Here is the ActionScript listing, which is included with the MXML file's **Script** control:

```
//: gui/flex/songScript.as
function getSongs() {
  songService.getSongs();
}

function selectSong(event) {
  var song = songGrid.getItemAt(event.itemIndex);
  showSongInfo(song);
}

function showSongInfo(song) {
  songInfo.text = song.name + newline;
  songInfo.text += song.artist + newline;
  songInfo.text += song.album + newline;
```

```
  albumImage.source = song.albumImageUrl;
  songPlayer.contentPath = song.songMediaUrl;
  songPlayer.visible = true;
}

function onSongs(songs) {
  songGrid.dataProvider = songs;
} ///:~
```

To handle **DataGrid** cell selections, we add the **cellPress** event attribute to the **DataGrid** declaration in the MXML file:

```
cellPress="selectSong(event)"
```

When the user clicks on a song in the **DataGrid**, this will call **selectSong( )** in the ActionScript above.

Data models and data binding

Controls can directly invoke services, and ActionScript event callbacks give you a chance to programmatically update the visual controls when services return data. While the script which updates the controls is straightforward, it can get verbose and cumbersome, and its functionality is so common that Flex handles the behavior automatically, with data binding.

In its simplest form, data binding allows controls to reference data directly instead of requiring glue code to copy data into a control. When the data is updated, the control which references it is also automatically updated without any need for programmer intervention. The Flex infrastructure correctly responds to the data change events, and updates all controls which are bound to the data.

Here is a simple example of data binding syntax:

```
<mx:Slider id="mySlider"/>
<mx:Text text="{mySlider.value}" />
```

To perform data binding, you place references within curly braces: {}. Everything within those curly braces is deemed an expression for Flex to evaluate.

The value of the first control, a **Slider** widget, is displayed by the second control, a **Text** field. As the **Slider** changes, the **Text** field's **text** property is automatically updated. This way, the developer does not need to handle the **Slider**'s change events in order to update the **Text** field.

Some controls, such as the **Tree** control and the **DataGrid** in the song library application, are more sophisticated. These controls have a **dataprovider** property to facilitate binding to collections of data. The ActionScript **onSongs( )** function shows how the **SongService.getSongs( )** method is bound to the **dataprovider** of the Flex **DataGrid**. As declared in the **RemoteObject** tag in the MXML file, this function is the callback that ActionScript invokes whenever the Java method returns.

A more sophisticated application with more complex data modeling, such as an enterprise application making use of Data Transfer Objects or a messaging application with data conforming to complex schemas, may encourage further decoupling of the source of data from the controls. In Flex development, we perform this decoupling by declaring a "Model" object, which is a generic MXML container for data. The model contains no logic. It mirrors the Data Transfer Object found in enterprise development, and the structures of other programming languages. By using the model, we can databind our controls to the model, and

at the same time have the model databind its properties to service inputs and outputs. This decouples the sources of data, the services, from the visual consumers of the data, facilitating use of the *Model- View-Controller* (MVC) pattern. In larger, more sophisticated applications, the initial complexity caused by inserting a model is often only a small tax compared to the value of a cleanly decoupled MVC application.

In addition to Java objects, Flex can also access SOAP-based Web services and RESTful HTTP services using the **WebService** and **HttpService** controls, respectively. Access to all services is subject to security authorization constraints.

Building and deploying

With the earlier examples, you could get away without a **-flexlib** flag on the command line, but to compile this program, you must specify the location of the **flex-config.xml** file using the **-flexlib** flag. For my installation, the following command works, but you'll have to modify it for your own configuration (the command is a single line, which has been wrapped):

```
//:! gui/flex/buiId-command.txt
mxmlc -flexlib C:/"Program
Files"/Macromedia/Flex/jrun4/servers/default/flex/WEB-INF/flex
songs.mxml
///:~
```

This command will build the application into an SWF file which you can view in your browser, but the book's code distribution file contains no MP3 files or JPG files, so you won't see anything but the framework when you run the application.

In addition, you must configure a server in order to successfully talk to the Java files from the Flex application. The Flex trial package comes with the JRun server, and you can start this through your computer's menus once Flex is installed, or via the command line:

```
jrun -start default
```

You can verify that the server has been successfully started by opening *http://localhost:8700/samples* in a Web browser and viewing the various samples (this is also a good way to get more familiar with the abilities of Flex).

Instead of compiling the application on the command line, you can compile it via the server. To do this, drop the song source files, CSS style sheet, etc., into the **jrun4/servers/default/flex** directory and access them in a browser by opening *http://localhost:8700/flex/songs.mxml.*

To successfully run the app, you must configure both the Java side and the Flex side.

Java: The compiled **Song.java** and **SongService.java** files must be placed in your **WEB-INF/classes** directory. This is where you drop WAR classes according to the J2EE specification. Alternatively, you can JAR the files and drop the result in **WEB-INF/lib**. It must be in a directory that matches its Java package structure. If you're using JRun, these would be placed in **jrun4/servers/defaulIt/flex/WEB-INF/classes/gui/flex/Song.cIass** and **jrun4/servers/default/flex/WEBINF/classes/gui/flex/SongService.class**. You also need the image and MP3 support files available in the Web app (for JRun, **jrun4/servers/default/flex** is the Web app root).

Flex: For security reasons, Flex cannot access Java objects unless you give permission by modifying your **flex-config.xml** file. For JRun, this is located at **jrun4/servers/default/flex/WEB-INF/flex/flex-config.xml**. **Go** to the **<remote-**

objects> entry in that file, look at the **<whitelist>** section within, and see the following note:

```
<!--
For security, the whitelist is locked down by default. Uncomment the source element
below to enable access to all classes during development.

We strongly recommend not allowing access to all source files in production, since this
exposes Java and Flex system classes. <source>*</source>
-->
```

Uncomment that **<source>** entry to allow access, so that it reads **<source>*</source>**. The meaning of this and other entries is described in the Flex configuration docs.

Exercise 38: (3) Build the "simple example of data binding syntax" shown above.

Exercise 39: (4) The code download for this book does not include the MP3S or JPGs shown in **SongService.java**. Find some MP3S and JPGs, modify **SongService.java** to include their file names, download the Flex trial and build the application.

Creating SWT applications

As previously noted, Swing took the approach of building all the UI components pixel-by-pixel, in order to provide every component desired whether the underlying OS had those components or not. SWT takes the middle ground by using native components if the OS provides them, and synthesizing components if it doesn't. The result is an application that feels to the user like a native application, and often has noticeably faster performance than the equivalent Swing program. In addition, SWT tends to be a less complex programming model than Swing, which can be desirable in a large portion of applications.[12]

Because SWT uses the native OS to do as much of its work as possible, it can automatically take advantage of OS features that may not be available to Swing—for example, Windows has "subpixel rendering" that makes fonts on LCD screens clearer.

It's even possible to create applets using SWT.

This section is not meant to be a comprehensive introduction to SWT; it's just enough to give you a flavor of it, and to see how SWT contrasts with Swing. You'll discover that there are lots of SWT widgets and that they are all reasonably straightforward to use. You can explore the details in the full documentation and many examples that can be found at *www.eclipse.org*. There are also a number of books on programming with SWT, and more on the way.

Installing SWT

SWT applications require downloading and installing the SWT library from the Eclipse project. Go to *www.eclipse.org/downloads/* and choose a mirror. Follow the links to the current Eclipse build and locate a compressed file with a name that begins with "swt" and includes the name of your platform (for example, "win32"). Inside this file you'll find **swt.jar**. The easiest way to install the **swt.jar** file is to put it into your **jre/lib/ext** directory (that way you don't have to make any modifications to your classpath). When you decompress the SWT library, you may find additional files that you need to install in appropriate places for your platform. For example, the Win32 distribution comes with DLL files that need to be placed somewhere in your **java.library.path** (this is usually the same

[12] Chris Grindstaff was very helpful in translating SWT examples and providing SWT information.

as your PATH environment variable, but you can run **object/ShowProperties.java** to discover the actual value of **java.library.path**). Once you've done this, you should be able to transparently compile and execute an SWT application as if it were any other Java program.

The documentation for SWT is in a separate download.

An alternative approach is just to install the Eclipse editor, which includes both SWT and the SWT documentation that you can view through the Eclipse help system.

Hello, SWT

Let's start with the simplest possible "hello world"-style application:

```
//: swt/HelloSWT.java
// {Requires: org.eclipse.swt.widgets.Display; You must
// install the SWT library from http://www.eclipse.org }
import org.eclipse.swt.widgets.*;

public class HelloSWT {
  public static void main(String [] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Hi there, SWT!"); // Title bar
    shell.open();
    while(!shell.isDisposed())
      if(!display.readAndDispatch())
        display.sleep();
    display.dispose();
  }
} ///:~
```

If you download the source code from this book, you'll discover that the "Requires" comment directive ends up in the Ant **build.xml** as a prerequisite for building the **swt** subdirectory; all the files that import **org.eclipse.swt** require that you install the SWT library from *www.eclipse.org*.

The **Display** manages the connection between SWT and the underlying operating system—it is part of a *Bridge* between the operating system and SWT. The **Shell** is the top-level main window, within which all the other components are built. When you call **setText( )**, the argument becomes the label on the title bar of the window.

To display the window and thus the application, you must call **open( )** on the **Shell**.

Whereas Swing hides the event-handling loop from you, SWT forces you to write it explicitly. At the top of the loop, you check to see whether the shell has been disposed—note that this gives you the option of inserting code to perform cleanup activities. But this means that the **main( )** thread *is* the user interface thread. In Swing, a second event-dispatching thread is created behind the scenes, but in SWT your **main( )** thread is what handles the UI. Since by default there's only one thread and not two, this makes it somewhat less likely that you'll clobber the UI with threads.

Notice that you don't have to worry about submitting tasks to the user interface thread like you do in Swing. SWT not only takes care of this for you, it throws an exception if you try to manipulate a widget with the wrong thread. However, if you need to spawn other threads to perform long-running operations, you still need to submit changes in the same way that you do with Swing. For this, SWT provides three methods which can be called on the **Display** object: **asyncExec(Runnable)**, **syncExec(Runnable)** and **timerExec(int**, **Runnable)**.

The activity of your **main( )** thread at this point is to call **readAndDispatch( )** on the **Display** object (this means that there can only be one **Display** object per application). The **readAndDispatch( )** method returns **true** if there are more events in the event queue, waiting to be processed. In that case, you want to call it again, immediately. However, if nothing is pending, you call the **Display** object's **sleep( )** to wait for a short time before checking the event queue again.

Once the program is complete, you must explicitly **dispose( )** of your **Display** object. SWT often requires you to explicitly dispose of resources, because these are usually resources from the underlying operating system, which may otherwise become exhausted.

To prove that the **Shell** is the main window, here's a program that makes a number of **Shell** objects:

```
//: swt/ShellsAreMainWindows.java
import org.eclipse.swt.widgets.*;

public class ShellsAreMainWindows {
  static Shell[] shells = new Shell[10];
  public static void main(String [] args) {
    Display display = new Display();
    for(int i = 0; i < shells.length; i++) {
      shells[i] = new Shell(display);
      shells[i].setText("Shell #" + i);
      shells[i].open();
    }
    while(!shellsDisposed())
      if(!display.readAndDispatch())
        display.sleep();
    display.dispose();
  }
  static boolean shellsDisposed() {
    for(int i = 0; i < shells.length; i++)
      if(shells[i].isDisposed())
        return true;
    return false;
  }
} ///:~
```

When you run it, you'll get ten main windows. The way the program is written, if you close any one of the windows, it will close all of them.

SWT also uses layout managers—different ones than Swing, but the same idea. Here's a slightly more complex example that takes the text from **System.getProperties( )** and adds it to the shell:

```
//: swt/DisplayProperties.java
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import java.io.*;

public class DisplayProperties {
  public static void main(String [] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Display Properties");
    shell.setLayout(new FillLayout());
    Text text = new Text(shell, SWT.WRAP | SWT.V_SCROLL);
    StringWriter props = new StringWriter();
    System.getProperties().list(new PrintWriter(props));
```

```
    text.setText(props.toString());
    shell.open();
    while(!shell.isDisposed())
      if(!display.readAndDispatch())
        display.sleep();
    display.dispose();
  }
} ///:~
```

In SWT, all widgets must have a parent object of the general type **Composite**, and you must provide this parent as the first argument in the widget constructor. You see this in the **Text** constructor, where **shell** is the first argument. Virtually all constructors also take a flag argument that allows you to provide any number of style directives, depending on what that particular widget accepts. Multiple style directives are bitwise-ORed together as seen in this example.

When setting up the **Text( )** object, I added style flags so that it wraps the text, and automatically adds a vertical scroll bar if it needs to. You'll discover that SWT is very constructor-based; there are many attributes of a widget that are difficult or impossible to change except via the constructor. Always check a widget constructor's documentation for the accepted flags. Note that some constructors require a flag argument even when they have no "accepted" flags listed in the documentation. This allows future expansion without modifying the interface.

Eliminating redundant code

Before going on, notice that there are certain things you do for every SWT application, just like there were duplicate actions for Swing programs. For SWT, you always create a **Display**, make a **Shell** from the **Display**, create a **readAndDispatch( )** loop, etc. Of course, in some special cases, you may not do this, but it's common enough that it's worth trying to eliminate the duplicate code as we did with **net.mindview.util.SwingConsole**.

We'll need to force each application to conform to an interface:

```
//: swt/util/SWTApplication.java
package swt.util;
import org.eclipse.swt.widgets.*;

public interface SWTApplication {
  void createContents(Composite parent);
} ///:~
```

The application is handed a **Composite** object (**Shell** is a subclass) and must use this to create all of its contents inside **createContents( )**. **SWTConsole.run( )** calls **createContents( )** at the appropriate point, sets the size of the shell according to what the user passes to **run( )**, opens the shell and then runs the event loop, and finally disposes of the shell at program exit:

```
//: swt/util/SWTConsole.java
package swt.util;
import org.eclipse.swt.widgets.*;

public class SWTConsole {
  public static void
  run(SWTApplication swtApp, int width, int height) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText(swtApp.getClass().getSimpleName());
    swtApp.createContents(shell);
```

```
    shell.setSize(width, height);
    shell.open();
    while(!shell.isDisposed()) {
      if(!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }
} ///:~
```

This also sets the title bar to the name of the **SWTApplication** class, and sets the **width** and **height** of the **Shell**.

We can create a variation of **DisplayProperties.Java** that displays the machine environment, using **SWTConsole**:

```
//: swt/DisplayEnvironment.java
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import java.util.*;

public class DisplayEnvironment implements SWTApplication {
  public void createContents(Composite parent) {
    parent.setLayout(new FillLayout());
    Text text = new Text(parent, SWT.WRAP | SWT.V_SCROLL);
    for(Map.Entry entry: System.getenv().entrySet()) {
      text.append(entry.getKey() + ": " +
        entry.getValue() + "\n");
    }
  }
  public static void main(String [] args) {
    SWTConsole.run(new DisplayEnvironment(), 800, 600);
  }
} ///:~
```

SWTConsole allows us to focus on the interesting aspects of an application rather than the repetitive code.

Exercise 40: (4) Modify **DisplayProperties.java** so that it uses **SWTConsole**.

Exercise 41: (4) Modify **Display Environment.java** so that it does nor use **SWTConsole**.

Menus

To demonstrate basic menus, this example reads its own source code and breaks it into words, then populates the menus with these words:

```
//: swt/Menus.java
// Fun with menus.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import java.util.*;
import net.mindview.util.*;

public class Menus implements SWTApplication {
```

```
  private static Shell shell;
  public void createContents(Composite parent) {
    shell = parent.getShell();
    Menu bar = new Menu(shell, SWT.BAR);
    shell.setMenuBar(bar);
    Set<String> words = new TreeSet<String>(
      new TextFile("Menus.java", "\\W+"));
    Iterator<String> it = words.iterator();
    while(it.next().matches("[0-9]+"))
      ; // Move past the numbers.
    MenuItem[] mItem = new MenuItem[7];
    for(int i = 0; i < mItem.length; i++) {
      mItem[i] = new MenuItem(bar, SWT.CASCADE);
      mItem[i].setText(it.next());
      Menu submenu = new Menu(shell, SWT.DROP_DOWN);
      mItem[i].setMenu(submenu);
    }
    int i = 0;
    while(it.hasNext()) {
      addItem(bar, it, mItem[i]);
      i = (i + 1) % mItem.length;
    }
  }
  static Listener listener = new Listener() {
    public void handleEvent(Event e) {
      System.out.println(e.toString());
    }
  };
  void
  addItem(Menu bar, Iterator<String> it, MenuItem mItem) {
    MenuItem item = new MenuItem(mItem.getMenu(),SWT.PUSH);
    item.addListener(SWT.Selection, listener);
    item.setText(it.next());
  }
  public static void main(String[] args) {
    SWTConsole.run(new Menus(), 600, 200);
  }
} ///:~
```

A **Menu** must be placed on a **Shell**, and **Composite** allows you to fetch its shell with **getShell( )**. **TextFile** is from **net.mindview.util** and has been described earlier in the book; here a **TreeSet** is filled with words so they will appear in sorted order. The initial elements are numbers, which are discarded. Using the stream of words, the top-level menus on the menu bar are named, then the submenus are created and filled with words until there are no more words.

In response to selecting one of the menu items, the **Listener** simply prints the event so you can see what kind of information it contains. When you run the program, you'll see that part of the information includes the label on the menu, so you can base the menu response on that—or you can provide a different listener for each menu (which is the safer approach, for internationalization).

Tabbed panes, buttons, and events

SWT has a rich set of controls, which they call *widgets*. Look at the documentation for **org.eclipse.swt.widgets** to see the basic ones, and **org.eclipse.swt.custom** to see fancier ones.

To demonstrate a few of the basic widgets, this example places a number of sub-examples inside tabbed panes. You'll also see how to create **Composite**s (roughly the same as Swing **JPanel**s) in order to put items within items.

```
//: swt/TabbedPane.java
// Placing SWT components in tabbed panes.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.browser.*;

public class TabbedPane implements SWTApplication {
  private static TabFolder folder;
  private static Shell shell;
  public void createContents(Composite parent) {
    shell = parent.getShell();
    parent.setLayout(new FillLayout());
    folder = new TabFolder(shell, SWT.BORDER);
    labelTab();
    directoryDialogTab();
    buttonTab();
    sliderTab();
    scribbleTab();
    browserTab();
  }
  public static void labelTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("A Label"); // Text on the tab
    tab.setToolTipText("A simple label");
    Label label = new Label(folder, SWT.CENTER);
    label.setText("Label text");
    tab.setControl(label);
  }
  public static void directoryDialogTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("Directory Dialog");
    tab.setToolTipText("Select a directory");
    final Button b = new Button(folder, SWT.PUSH);
    b.setText("Select a Directory");
    b.addListener(SWT.MouseDown, new Listener() {
        public void handleEvent(Event e) {
          DirectoryDialog dd = new DirectoryDialog(shell);
          String path = dd.open();
          if(path != null)
            b.setText(path);
        }
      });
    tab.setControl(b);
  }
  public static void buttonTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("Buttons");
    tab.setToolTipText("Different kinds of Buttons");
    Composite composite = new Composite(folder, SWT.NONE);
    composite.setLayout(new GridLayout(4, true));
    for(int dir : new int[]{
        SWT.UP, SWT.RIGHT, SWT.LEFT, SWT.DOWN
      }) {
      Button b = new Button(composite, SWT.ARROW | dir);
      b.addListener(SWT.MouseDown, listener);
```

```
    }
    newButton(composite, SWT.CHECK, "Check button");
    newButton(composite, SWT.PUSH, "Push button");
    newButton(composite, SWT.RADIO, "Radio button");
    newButton(composite, SWT.TOGGLE, "Toggle button");
    newButton(composite, SWT.FLAT, "Flat button");
    tab.setControl(composite);
  }
  private static Listener listener = new Listener() {
      public void handleEvent(Event e) {
        MessageBox m = new MessageBox(shell, SWT.OK);
        m.setMessage(e.toString());
        m.open();
      }
    };
  private static void newButton(Composite composite,
    int type, String label) {
    Button b = new Button(composite, type);
    b.setText(label);
    b.addListener(SWT.MouseDown, listener);
  }
  public static void sliderTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("Sliders and Progress bars");
    tab.setToolTipText("Tied Slider to ProgressBar");
    Composite composite = new Composite(folder, SWT.NONE);
    composite.setLayout(new GridLayout(2, true));
    final Slider slider =
      new Slider(composite, SWT.HORIZONTAL);
    final ProgressBar progress =
      new ProgressBar(composite, SWT.HORIZONTAL);
    slider.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent event) {
          progress.setSelection(slider.getSelection());
        }
      });
    tab.setControl(composite);
  }
  public static void scribbleTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("Scribble");
    tab.setToolTipText("Simple graphics: drawing");
    final Canvas canvas = new Canvas(folder, SWT.NONE);
    ScribbleMouseListener sml= new ScribbleMouseListener();
    canvas.addMouseListener(sml);
    canvas.addMouseMoveListener(sml);
    tab.setControl(canvas);
  }
  private static class ScribbleMouseListener
    extends MouseAdapter implements MouseMoveListener {
    private Point p = new Point(0, 0);
    public void mouseMove(MouseEvent e) {
      if((e.stateMask & SWT.BUTTON1) == 0)
        return;
      GC gc = new GC((Canvas)e.widget);
      gc.drawLine(p.x, p.y, e.x, e.y);
      gc.dispose();
      updatePoint(e);
    }
    public void mouseDown(MouseEvent e) { updatePoint(e); }
    private void updatePoint(MouseEvent e) {
      p.x = e.x;
      p.y = e.y;
```

```
    }
  }
  public static void browserTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("A Browser");
    tab.setToolTipText("A Web browser");
    Browser browser = null;
    try {
      browser = new Browser(folder, SWT.NONE);
    } catch(SWTError e) {
      Label label = new Label(folder, SWT.BORDER);
      label.setText("Could not initialize browser");
      tab.setControl(label);
    }
    if(browser != null) {
      browser.setUrl("http://www.mindview.net");
      tab.setControl(browser);
    }
  }
  public static void main(String[] args) {
    SWTConsole.run(new TabbedPane(), 800, 600);
  }
} ///:~
```

Here, **createContents( )** sets the layout and then calls the methods that each create a different tab. The text on each tab is set with **setText( )** (you can also create buttons and graphics on a tab), and each one also sets its tool tip text. At the end of each method, you'll see a call to **setControl( )**, which places the control that the method created into the dialog space of that particular tab.

labelTab( ) demonstrates a simple text label. **directoryDialogTab( )** holds a button which opens a standard **DirectoryDialog** object so the user can select a directory. The result is set as the button's text.

buttonTab( ) shows the different basic buttons. **sliderTab( )** repeats the Swing example from earlier in the chapter of tying a slider to a progress bar.

scribbleTab( ) is a fun example of graphics. A drawing program is produced from only a few lines of code.

Finally, **browserTab( )** shows the power of the SWT **Browser** component—a full-featured Web browser in a single component.

Graphics

Here's the Swing **SineWave.java** program translated to SWT:

```
//: swt/SineWave.java
// SWT translation of Swing SineWave.java.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;

class SineDraw extends Canvas {
  private static final int SCALEFACTOR = 200;
  private int cycles;
  private int points;
```

```
  private double[] sines;
  private int[] pts;
  public SineDraw(Composite parent, int style) {
    super(parent, style);
    addPaintListener(new PaintListener() {
      public void paintControl(PaintEvent e) {
        int maxWidth = getSize().x;
        double hstep = (double)maxWidth / (double)points;
        int maxHeight = getSize().y;
        pts = new int[points];
        for(int i = 0; i < points; i++)
          pts[i] = (int)((sines[i] * maxHeight / 2 * .95)
            + (maxHeight / 2));
        e.gc.setForeground(
          e.display.getSystemColor(SWT.COLOR_RED));
        for(int i = 1; i < points; i++) {
          int x1 = (int)((i - 1) * hstep);
          int x2 = (int)(i * hstep);
          int y1 = pts[i - 1];
          int y2 = pts[i];
          e.gc.drawLine(x1, y1, x2, y2);
        }
      }
    });
    setCycles(5);
  }
  public void setCycles(int newCycles) {
    cycles = newCycles;
    points = SCALEFACTOR * cycles * 2;
    sines = new double[points];
    for(int i = 0; i < points; i++) {
      double radians = (Math.PI / SCALEFACTOR) * i;
      sines[i] = Math.sin(radians);
    }
    redraw();
  }
}

public class SineWave implements SWTApplication {
  private SineDraw sines;
  private Slider slider;
  public void createContents(Composite parent) {
    parent.setLayout(new GridLayout(1, true));
    sines = new SineDraw(parent, SWT.NONE);
    sines.setLayoutData(
      new GridData(SWT.FILL, SWT.FILL, true, true));
    sines.setFocus();
    slider = new Slider(parent, SWT.HORIZONTAL);
    slider.setValues(5, 1, 30, 1, 1, 1);
    slider.setLayoutData(
      new GridData(SWT.FILL, SWT.DEFAULT, true, false));
    slider.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent event) {
        sines.setCycles(slider.getSelection());
      }
    });
  }
  public static void main(String[] args) {
    SWTConsole.run(new SineWave(), 700, 400);
  }
} ///:~
```

Instead of **JPanel**, the basic drawing surface in SWT is **Canvas**.

If you compare this version of the program with the Swing version, you'll see that **SineDraw** is virtually identical. In SWT, you get the graphics context gc from the event object that's handed to the **PaintListener**, and in Swing the **Graphics** object is handed directly to the **paintComponent( )** method. But the activities performed with the graphics object are the same, and **setCycles( )** is identical.

createContents( ) requires a bit more code than the Swing version, to lay things out and set up the slider and its listener, but again, the basic activities are roughly the same.

Concurrency in SWT

Although AWT/Swing is single-threaded, it's easily possible to violate that single-threadedness in a way that produces a non-deterministic program. Basically, you don't want to have multiple threads writing to the display because they will write over each other in surprising ways.

SWT doesn't allow this—it throws an exception if you try to write to the display using more than one thread. This will prevent a novice programmer from accidentally making this mistake and introducing hard-to-find bugs into a program.

Here is the translation of the Swing **ColorBoxes.java** program in SWT:

```
//: swt/ColorBoxes.java
// SWT translation of Swing ColorBoxes.java.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

class CBox extends Canvas implements Runnable {
  class CBoxPaintListener implements PaintListener {
    public void paintControl(PaintEvent e) {
      Color color = new Color(e.display, cColor);
      e.gc.setBackground(color);
      Point size = getSize();
      e.gc.fillRectangle(0, 0, size.x, size.y);
      color.dispose();
    }
  }
  private static Random rand = new Random();
  private static RGB newColor() {
    return new RGB(rand.nextInt(255),
      rand.nextInt(255), rand.nextInt(255));
  }
  private int pause;
  private RGB cColor = newColor();
  public CBox(Composite parent, int pause) {
    super(parent, SWT.NONE);
    this.pause = pause;
    addPaintListener(new CBoxPaintListener());
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        cColor = newColor();
```

```
        getDisplay().asyncExec(new Runnable() {
          public void run() {
            try { redraw(); } catch(SWTException e) {}
            // SWTException is OK when the parent
            // is terminated from under us.
          }
        });
        TimeUnit.MILLISECONDS.sleep(pause);
      }
    } catch(InterruptedException e) {
      // Acceptable way to exit
    } catch(SWTException e) {
      // Acceptable way to exit: our parent
      // was terminated from under us.
    }
  }
}

public class ColorBoxes implements SWTApplication {
  private int grid = 12;
  private int pause = 50;
  public void createContents(Composite parent) {
    GridLayout gridLayout = new GridLayout(grid, true);
    gridLayout.horizontalSpacing = 0;
    gridLayout.verticalSpacing = 0;
    parent.setLayout(gridLayout);
    ExecutorService exec = new DaemonThreadPoolExecutor();
    for(int i = 0; i < (grid * grid); i++) {
      final CBox cb = new CBox(parent, pause);
      cb.setLayoutData(new GridData(GridData.FILL_BOTH));
      exec.execute(cb);
    }
  }
  public static void main(String[] args) {
    ColorBoxes boxes = new ColorBoxes();
    if(args.length > 0)
      boxes.grid = new Integer(args[0]);
    if(args.length > 1)
      boxes.pause = new Integer(args[1]);
    SWTConsole.run(boxes, 500, 400);
  }
} ///:~
```

As in the previous example, painting is controlled by creating a **PaintListener** with a **paintControl( )** method that is called when the SWT thread is ready to paint your component. The **PaintListener** is registered in the **CBox** constructor.

What's notably different in this version of **CBox** is the **run( )** method, which cannot just call **redraw( )** directly but must submit the **redraw( )** to the **asyncExec( )** method on the **Display** object, which is roughly the same as **SwingUtilities.invokeLater( )**. If you replace this with a direct call to **redraw( )**, you'll see that the program just stops.

When running the program, you will see little visual artifacts—horizontal lines occasionally running through a box. This is because SWT is *not* doublebuffered by default, while Swing is. Try running the Swing version side by side with the SWT version and you'll see it more clearly. You can write code to double-buffer SWT; you'll find examples on the *www.eclipse.org* Web site.

Exercise 42: (4) Modify **swt/ColorBoxes.java** so that it begins by sprinkling points ("stars") across the canvas, then randomly changes the colors of those "stars."

SWT vs. Swing?

It's hard to get a complete picture from such a short introduction, but you should at least start to see that SWT, in many situations, can be a more straightforward way to write code than Swing. However, GUI programming in SWT can still be complex, so your motivation for using SWT should probably be, first, to give the user a more transparent experience when using your application (because the application looks/feels like the other applications on that platform), and second, if the responsiveness provided by SWT is important. Otherwise, Swing may be an appropriate choice.

Exercise 43: (6) Choose any one of the Swing examples that wasn't translated in this section, and translate it to SWT. (Note: This makes a good homework exercise for a class, since the solutions are *not* in the solution guide.)

Summary

The Java GUI libraries have seen some dramatic changes during the lifetime of the language. The Java 1.0 AWT was roundly criticized as being a poor design, and while it allowed you to create portable programs, the resulting GUI was "equally mediocre on all platforms." It was also limiting, awkward, and unpleasant to use compared with the native application development tools available for various platforms.

When Java 1.1 introduced the new event model and JavaBeans, the stage was set—now it was possible to create GUI components that could easily be dragged and dropped inside a visual IDE. In addition, the design of the event model and JavaBeans clearly shows strong consideration for ease of programming and maintainable code (something that was not evident in the 1.0 AWT). But it wasn't until the JFC/Swing classes appeared that the transition was complete. With the Swing components, cross-platform GUI programming can be a civilized experience.

IDEs are where the real revolution lies. If you want a commercial IDE for a proprietary language to get better, you must cross your fingers and hope that the vendor will give you what you want. But Java is an open environment, so not only does it allow for competing IDEs, it encourages them. And for these tools to be taken seriously, they must support JavaBeans. This means a leveled playing field; if a better IDE comes along, you're not tied to the one you've been using. You can pick up and move to the new one and increase your productivity. This kind of competitive environment for GUI IDEs has not been seen before, and the resulting marketplace can generate very positive results for programmer productivity.

This chapter was only meant to give you an introduction to the power of GUI programming and to get you started so that you can see how relatively simple it is to feel your way through the libraries. What you've seen so far will probably suffice for a good portion of your UI design needs. However, there's a lot more to Swing, SWT and Flash/Flex; these are meant to be fully powered UI design toolkits. There's probably a way to accomplish just about everything you can imagine.

Resources

Ben Galbraith's online presentations at *www.galbraiths.org/presentations* provide some nice coverage of both Swing and SWT.

Solutions to selected exercises can be found in the electronic document *The Thinking in Java Annotated Solution Guide*, available for sale from *www.MindView.net*.