of columns. This is why the layout is called a *grid bag*. It's a collection of small grids joined together.

The location and size of each component in a grid bag are determined by a set of constraints linked to it. The constraints are contained in an object of type **GridBagConstraints**. Constraints include the height and width of a cell, and the placement of a component, its alignment, and its anchor point within the cell.

The general procedure for using a grid bag is to first create a new **GridBagLayout** object and to make it the current layout manager. Then, set the constraints that apply to each component that will be added to the grid bag. Finally, add the components to the layout manager. Although **GridBagLayout** is a bit more complicated than the other layout managers, it is still quite easy to use once you understand how it works.

GridBagLayout defines only one constructor, which is shown here:

```
GridBagLayout( )
```

GridBagLayout defines several methods, of which many are protected and not for general use. There is one method, however, that you must use: **setConstraints( )**. It is shown here:

```
void setConstraints(Component comp, GridBagConstraints cons)
```

Here, *comp* is the component for which the constraints specified by *cons* apply. This method sets the constraints that apply to each component in the grid bag.

The key to successfully using **GridBagLayout** is the proper setting of the constraints, which are stored in a **GridBagConstraints** object. **GridBagConstraints** defines several fields that you can set to govern the size, placement, and spacing of a component. These are shown in Table 26-1. Several are described in greater detail in the following discussion.

Field	Purpose
int anchor	Specifies the location of a component within a cell. The default is **GridBagConstraints.CENTER**.
int fill	Specifies how a component is resized if the component is smaller than its cell. Valid values are **GridBagConstraints.NONE** (the default), **GridBagConstraints.HORIZONTAL**, **GridBagConstraints.VERTICAL**, **GridBagConstraints.BOTH**.
int gridheight	Specifies the height of component in terms of cells. The default is 1.
int gridwidth	Specifies the width of component in terms of cells. The default is 1.
int gridx	Specifies the X coordinate of the cell to which the component will be added. The default value is **GridBagConstraints.RELATIVE**.
int gridy	Specifies the Y coordinate of the cell to which the component will be added. The default value is **GridBagConstraints.RELATIVE**.
Insets insets	Specifies the insets. Default insets are all zero.
int ipadx	Specifies extra horizontal space that surrounds a component within a cell. The default is 0.
int ipady	Specifies extra vertical space that surrounds a component within a cell. The default is 0.

Table 26-1 Constraint Fields Defined by **GridBagConstraints**

Field	Purpose
double weightx	Specifies a weight value that determines the horizontal spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window.
double weighty	Specifies a weight value that determines the vertical spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window.

Table 26-1 Constraint Fields Defined by **GridBagConstraints** *(continued)*

GridBagConstraints also defines several static fields that contain standard constraint values, such as **GridBagConstraints.CENTER** and **GridBagConstraints.VERTICAL**.

When a component is smaller than its cell, you can use the **anchor** field to specify where within the cell the component's top-left corner will be located. There are three types of values that you can give to **anchor**. The first are absolute:

GridBagConstraints.CENTER	GridBagConstraints.SOUTH
GridBagConstraints.EAST	GridBagConstraints.SOUTHEAST
GridBagConstraints.NORTH	GridBagConstraints.SOUTHWEST
GridBagConstraints.NORTHEAST	GridBagConstraints.WEST
GridBagConstraints.NORTHWEST	

As their names imply, these values cause the component to be placed at the specific locations.

The second type of values that can be given to **anchor** is relative, which means the values are relative to the container's orientation, which might differ for non-Western languages. The relative values are shown here:

GridBagConstraints.FIRST_LINE_END	GridBagConstraints.LINE_END
GridBagConstraints.FIRST_LINE_START	GridBagConstraints.LINE_START
GridBagConstraints.LAST_LINE_END	GridBagConstraints.PAGE_END
GridBagConstraints.LAST_LINE_START	GridBagConstraints.PAGE_START

Their names describe the placement.

The third type of values that can be given to **anchor** allows you to position components relative to the baseline of the row. These values are shown here:

GridBagConstraints.BASELINE	GridBagConstraints.BASELINE_LEADING
GridBagConstraints.BASELINE_TRAILING	GridBagConstraints.ABOVE_BASELINE
GridBagConstraints.ABOVE_BASELINE_LEADING	GridBagConstraints.ABOVE_BASELINE_ TRAILING
GridBagConstraints.BELOW_BASELINE	GridBagConstraints.BELOW_BASELINE_ LEADING
GridBagConstraints. BELOW_BASELINE_TRAILING	

The horizontal position can be either centered, against the leading edge (LEADING), or against the trailing edge (TRAILING).

The **weightx** and **weighty** fields are both quite important and quite confusing at first glance. In general, their values determine how much of the extra space within a container is allocated to each row and column. By default, both these values are zero. When all values within a row or a column are zero, extra space is distributed evenly between the edges of the window. By increasing the weight, you increase that row or column's allocation of space proportional to the other rows or columns. The best way to understand how these values work is to experiment with them a bit.

The **gridwidth** variable lets you specify the width of a cell in terms of cell units. The default is 1. To specify that a component use the remaining space in a row, use **GridBagConstraints.REMAINDER**. To specify that a component use the next-to-last cell in a row, use **GridBagConstraints.RELATIVE**. The **gridheight** constraint works the same way, but in the vertical direction.

You can specify a padding value that will be used to increase the minimum size of a cell. To pad horizontally, assign a value to **ipadx**. To pad vertically, assign a value to **ipady**.

Here is an example that uses **GridBagLayout** to demonstrate several of the points just discussed:

```
// Use GridBagLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="GridBagDemo" width=250 height=200>
  </applet>
*/

public class GridBagDemo extends Applet
  implements ItemListener {

  String msg = "";
  Checkbox windows, android, solaris, mac;

  public void init() {
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);

    // Define check boxes.
    windows = new Checkbox("Windows ", null, true);
    android = new Checkbox("Android");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("Mac OS");

    // Define the grid bag.

    // Use default row weight of 0 for first row.
    gbc.weightx = 1.0; // use a column weight of 1
```

```
    gbc.ipadx = 200; // pad by 200 units
    gbc.insets = new Insets(4, 4, 0, 0); // inset slightly from top left

    gbc.anchor = GridBagConstraints.NORTHEAST;

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(windows, gbc);

    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(android, gbc);

    // Give second row a weight of 1.
    gbc.weighty = 1.0;

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(solaris, gbc);

    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(mac, gbc);

    // Add the components.
    add(windows);
    add(android);
    add(solaris);
    add(mac);

    // Register to receive item events.
    windows.addItemListener(this);
    android.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
  }

  // Repaint when status of a check box changes.
  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Display current state of the check boxes.
  public void paint(Graphics g) {
    msg = "Current state: ";
    g.drawString(msg, 6, 80);
    msg = "  Windows: " + windows.getState();
    g.drawString(msg, 6, 100);
    msg = "  Android: " + android.getState();
    g.drawString(msg, 6, 120);
    msg = "  Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = "  Mac: " + mac.getState();
    g.drawString(msg, 6, 160);
  }
}
```

Sample output produced by the program is shown here.

In this layout, the operating system check boxes are positioned in a 2×2 grid. Each cell has a horizontal padding of 200. Each component is inset slightly (by 4 units) from the top left. The column weight is set to 1, which causes any extra horizontal space to be distributed evenly between the columns. The first row uses a default weight of 0; the second has a weight of 1. This means that any extra vertical space is added to the second row.

GridBagLayout is a powerful layout manager. It is worth taking some time to experiment with and explore. Once you understand what the various settings do, you can use **GridBagLayout** to position components with a high degree of precision.

Menu Bars and Menus

A top-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu. This concept is implemented in the AWT by the following classes: **MenuBar**, **Menu**, and **MenuItem**. In general, a menu bar contains one or more **Menu** objects. Each **Menu** object contains a list of **MenuItem** objects. Each **MenuItem** object represents something that can be selected by the user. Since **Menu** is a subclass of **MenuItem**, a hierarchy of nested submenus can be created. It is also possible to include checkable menu items. These are menu options of type **CheckboxMenuItem** and will have a check mark next to them when they are selected.

To create a menu bar, first create an instance of **MenuBar**. This class defines only the default constructor. Next, create instances of **Menu** that will define the selections displayed on the bar. Following are the constructors for **Menu**:

Menu( ) throws HeadlessException
Menu(String *optionName*) throws HeadlessException
Menu(String *optionName*, boolean *removable*) throws HeadlessException

Here, *optionName* specifies the name of the menu selection. If *removable* is **true**, the menu can be removed and allowed to float free. Otherwise, it will remain attached to the menu bar. (Removable menus are implementation-dependent.) The first form creates an empty menu.

Individual menu items are of type **MenuItem**. It defines these constructors:

MenuItem( ) throws HeadlessException
MenuItem(String *itemName*) throws HeadlessException
MenuItem(String *itemName*, MenuShortcut *keyAccel*) throws HeadlessException

Here, *itemName* is the name shown in the menu, and *keyAccel* is the menu shortcut for this item.

You can disable or enable a menu item by using the **setEnabled( )** method. Its form is shown here:

```
void setEnabled(boolean enabledFlag)
```

If the argument *enabledFlag* is **true**, the menu item is enabled. If **false**, the menu item is disabled.

You can determine an item's status by calling **isEnabled( )**. This method is shown here:

```
boolean isEnabled( )
```

isEnabled( ) returns **true** if the menu item on which it is called is enabled. Otherwise, it returns **false**.

You can change the name of a menu item by calling **setLabel( )**. You can retrieve the current name by using **getLabel( )**. These methods are as follows:

```
void setLabel(String newName)
String getLabel( )
```

Here, *newName* becomes the new name of the invoking menu item. **getLabel( )** returns the current name.

You can create a checkable menu item by using a subclass of **MenuItem** called **CheckboxMenuItem**. It has these constructors:

```
CheckboxMenuItem( ) throws HeadlessException
CheckboxMenuItem(String itemName) throws HeadlessException
CheckboxMenuItem(String itemName, boolean on) throws HeadlessException
```

Here, *itemName* is the name shown in the menu. Checkable items operate as toggles. Each time one is selected, its state changes. In the first two forms, the checkable entry is unchecked. In the third form, if *on* is **true**, the checkable entry is initially checked. Otherwise, it is cleared.

You can obtain the status of a checkable item by calling **getState( )**. You can set it to a known state by using **setState( )**. These methods are shown here:

```
boolean getState( )
void setState(boolean checked)
```

If the item is checked, **getState( )** returns **true**. Otherwise, it returns **false**. To check an item, pass **true** to **setState( )**. To clear an item, pass **false**.

Once you have created a menu item, you must add the item to a **Menu** object by using **add( )**, which has the following general form:

```
MenuItem add(MenuItem item)
```

Here, *item* is the item being added. Items are added to a menu in the order in which the calls to **add( )** take place. The *item* is returned.

Once you have added all items to a **Menu** object, you can add that object to the menu bar by using this version of **add( )** defined by **MenuBar**:

```
Menu add(Menu menu)
```

Here, *menu* is the menu being added. The *menu* is returned.

Menus generate events only when an item of type **MenuItem** or **CheckboxMenuItem** is selected. They do not generate events when a menu bar is accessed to display a drop-down menu, for example. Each time a menu item is selected, an **ActionEvent** object is generated. By default, the action command string is the name of the menu item. However, you can specify a different action command string by calling **setActionCommand( )** on the menu item. Each time a check box menu item is checked or unchecked, an **ItemEvent** object is generated. Thus, you must implement the **ActionListener** and/or **ItemListener** interfaces in order to handle these menu events.

The **getItem( )** method of **ItemEvent** returns a reference to the item that generated this event. The general form of this method is shown here:

Object getItem( )

Following is an example that adds a series of nested menus to a pop-up window. The item selected is displayed in the window. The state of the two check box menu items is also displayed.

```
// Illustrate menus.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="MenuDemo" width=250 height=250>
  </applet>
*/

// Create a subclass of Frame.
class MenuFrame extends Frame {
  String msg = "";
  CheckboxMenuItem debug, test;

  MenuFrame(String title) {
    super(title);

    // create menu bar and add it to frame
    MenuBar mbar = new MenuBar();
    setMenuBar(mbar);

    // create the menu items
    Menu file = new Menu("File");
    MenuItem item1, item2, item3, item4, item5;
    file.add(item1 = new MenuItem("New..."));
    file.add(item2 = new MenuItem("Open..."));
    file.add(item3 = new MenuItem("Close"));
    file.add(item4 = new MenuItem("-"));
    file.add(item5 = new MenuItem("Quit..."));
    mbar.add(file);

    Menu edit = new Menu("Edit");
    MenuItem item6, item7, item8, item9;
    edit.add(item6 = new MenuItem("Cut"));
    edit.add(item7 = new MenuItem("Copy"));
```

```
    edit.add(item8 = new MenuItem("Paste"));
    edit.add(item9 = new MenuItem("-"));

    Menu sub = new Menu("Special");
    MenuItem item10, item11, item12;
    sub.add(item10 = new MenuItem("First"));
    sub.add(item11 = new MenuItem("Second"));
    sub.add(item12 = new MenuItem("Third"));
    edit.add(sub);

    // these are checkable menu items
    debug = new CheckboxMenuItem("Debug");
    edit.add(debug);
    test = new CheckboxMenuItem("Testing");
    edit.add(test);

    mbar.add(edit);

    // create an object to handle action and item events
    MyMenuHandler handler = new MyMenuHandler(this);
    // register it to receive those events
    item1.addActionListener(handler);
    item2.addActionListener(handler);
    item3.addActionListener(handler);
    item4.addActionListener(handler);
    item5.addActionListener(handler);
    item6.addActionListener(handler);
    item7.addActionListener(handler);
    item8.addActionListener(handler);
    item9.addActionListener(handler);
    item10.addActionListener(handler);
    item11.addActionListener(handler);
    item12.addActionListener(handler);
    debug.addItemListener(handler);
    test.addItemListener(handler);

    // create an object to handle window events
    MyWindowAdapter adapter = new MyWindowAdapter(this);
    // register it to receive those events
    addWindowListener(adapter);
  }

  public void paint(Graphics g) {
    g.drawString(msg, 10, 200);

    if(debug.getState())
      g.drawString("Debug is on.", 10, 220);
     else
      g.drawString("Debug is off.", 10, 220);

     if(test.getState())
       g.drawString("Testing is on.", 10, 240);
     else
```

```
        g.drawString("Testing is off.", 10, 240);
    }
}

class MyWindowAdapter extends WindowAdapter {
  MenuFrame menuFrame;

  public MyWindowAdapter(MenuFrame menuFrame) {
    this.menuFrame = menuFrame;
  }

  public void windowClosing(WindowEvent we) {
    menuFrame.setVisible(false);
  }
}

class MyMenuHandler implements ActionListener, ItemListener {
  MenuFrame menuFrame;

  public MyMenuHandler(MenuFrame menuFrame) {
    this.menuFrame = menuFrame;
  }

  // Handle action events.
  public void actionPerformed(ActionEvent ae) {
    String msg = "You selected ";
    String arg = ae.getActionCommand();
    if(arg.equals("New..."))
      msg += "New.";
    else if(arg.equals("Open..."))
      msg += "Open.";
    else if(arg.equals("Close"))
      msg += "Close.";
    else if(arg.equals("Quit..."))
      msg += "Quit.";
    else if(arg.equals("Edit"))
      msg += "Edit.";
    else if(arg.equals("Cut"))
      msg += "Cut.";
    else if(arg.equals("Copy"))
      msg += "Copy.";
    else if(arg.equals("Paste"))
      msg += "Paste.";
    else if(arg.equals("First"))
      msg += "First.";
    else if(arg.equals("Second"))
      msg += "Second.";
    else if(arg.equals("Third"))
      msg += "Third.";
    else if(arg.equals("Debug"))
      msg += "Debug.";
    else if(arg.equals("Testing"))
      msg += "Testing.";
```

```
    menuFrame.msg = msg;
    menuFrame.repaint();
  }

  // Handle item events.
  public void itemStateChanged(ItemEvent ie) {
    menuFrame.repaint();
  }
}

// Create frame window.
public class MenuDemo extends Applet {
  Frame f;

  public void init() {
    f = new MenuFrame("Menu Demo");
    int width = Integer.parseInt(getParameter("width"));
    int height = Integer.parseInt(getParameter("height"));

    setSize(new Dimension(width, height));

    f.setSize(width, height);
    f.setVisible(true);
  }

  public void start() {
    f.setVisible(true);
  }

  public void stop() {
    f.setVisible(false);
  }
}
```

Sample output from the **MenuDemo** applet is shown in Figure 26-8.

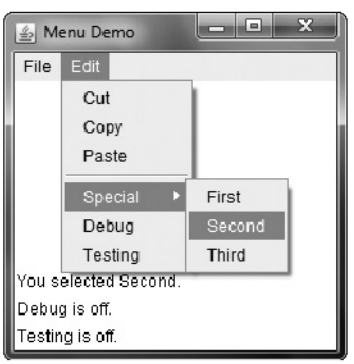


Figure 26-8 Sample output from the **MenuDemo** applet

There is one other menu-related class that you might find interesting: **PopupMenu**. It works just like **Menu**, but produces a menu that can be displayed at a specific location. **PopupMenu** provides a flexible, useful alternative for some types of menuing situations.

Dialog Boxes

Often, you will want to use a *dialog box* to hold a set of related controls. Dialog boxes are primarily used to obtain user input and are often child windows of a top-level window. Dialog boxes don't have menu bars, but in other respects, they function like frame windows. (You can add controls to them, for example, in the same way that you add controls to a frame window.) Dialog boxes may be modal or modeless. When a *modal* dialog box is active, all input is directed to it until it is closed. This means that you cannot access other parts of your program until you have closed the dialog box. When a *modeless* dialog box is active, input focus can be directed to another window in your program. Thus, other parts of your program remain active and accessible. In the AWT, dialog boxes are of type **Dialog**. Two commonly used constructors are shown here:

Dialog(Frame *parentWindow*, boolean *mode*)
Dialog(Frame *parentWindow*, String *title*, boolean *mode*)

Here, *parentWindow* is the owner of the dialog box. If *mode* is **true**, the dialog box is modal. Otherwise, it is modeless. The title of the dialog box can be passed in *title*. Generally, you will subclass **Dialog**, adding the functionality required by your application.

Following is a modified version of the preceding menu program that displays a modeless dialog box when the New option is chosen. Notice that when the dialog box is closed, **dispose( )** is called. This method is defined by **Window**, and it frees all system resources associated with the dialog box window.

```
// Demonstrate Dialog box.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="DialogDemo" width=250 height=250>
  </applet>
*/

// Create a subclass of Dialog.
class SampleDialog extends Dialog implements ActionListener {
  SampleDialog(Frame parent, String title) {
    super(parent, title, false);
    setLayout(new FlowLayout());
    setSize(300, 200);

    add(new Label("Press this button:"));
    Button b;
    add(b = new Button("Cancel"));
    b.addActionListener(this);
  }
```

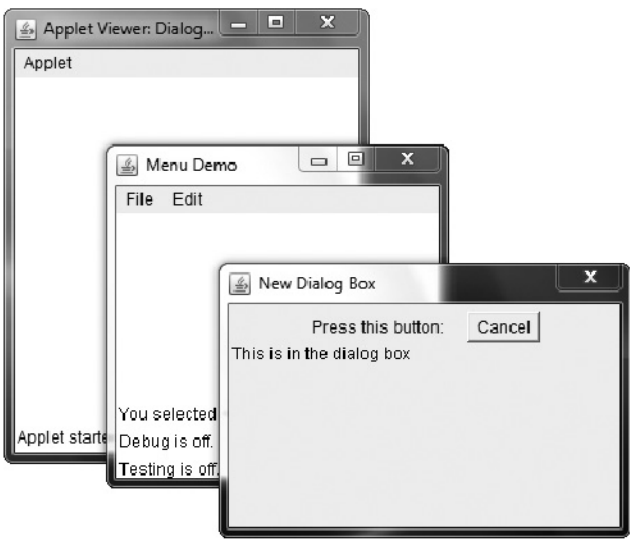
```
  public void actionPerformed(ActionEvent ae) {
    dispose();
  }

  public void paint(Graphics g) {
    g.drawString("This is in the dialog box", 10, 70);
  }
}

// Create a subclass of Frame.
class MenuFrame extends Frame {
  String msg = "";
  CheckboxMenuItem debug, test;

  MenuFrame(String title) {
    super(title);

    // create menu bar and add it to frame
    MenuBar mbar = new MenuBar();
    setMenuBar(mbar);

    // create the menu items
    Menu file = new Menu("File");
    MenuItem item1, item2, item3, item4;
    file.add(item1 = new MenuItem("New..."));
    file.add(item2 = new MenuItem("Open..."));
    file.add(item3 = new MenuItem("Close"));
    file.add(new MenuItem("-"));
    file.add(item4 = new MenuItem("Quit..."));
    mbar.add(file);

    Menu edit = new Menu("Edit");
    MenuItem item5, item6, item7;
    edit.add(item5 = new MenuItem("Cut"));
    edit.add(item6 = new MenuItem("Copy"));
    edit.add(item7 = new MenuItem("Paste"));
    edit.add(new MenuItem("-"));

    Menu sub = new Menu("Special", true);
    MenuItem item8, item9, item10;
    sub.add(item8 = new MenuItem("First"));
    sub.add(item9 = new MenuItem("Second"));
    sub.add(item10 = new MenuItem("Third"));
    edit.add(sub);

    // these are checkable menu items
    debug = new CheckboxMenuItem("Debug");
    edit.add(debug);
    test = new CheckboxMenuItem("Testing");
    edit.add(test);

    mbar.add(edit);
```

```
    // create an object to handle action and item events
    MyMenuHandler handler = new MyMenuHandler(this);
    // register it to receive those events
    item1.addActionListener(handler);
    item2.addActionListener(handler);
    item3.addActionListener(handler);
    item4.addActionListener(handler);
    item5.addActionListener(handler);
    item6.addActionListener(handler);
    item7.addActionListener(handler);
    item8.addActionListener(handler);
    item9.addActionListener(handler);
    item10.addActionListener(handler);
    debug.addItemListener(handler);
    test.addItemListener(handler);

    // create an object to handle window events
    MyWindowAdapter adapter = new MyWindowAdapter(this);

    // register it to receive those events
    addWindowListener(adapter);
  }

  public void paint(Graphics g) {
    g.drawString(msg, 10, 200);

    if(debug.getState())
      g.drawString("Debug is on.", 10, 220);
    else
      g.drawString("Debug is off.", 10, 220);

    if(test.getState())
      g.drawString("Testing is on.", 10, 240);
    else
    g.drawString("Testing is off.", 10, 240);
  }
}

class MyWindowAdapter extends WindowAdapter {
  MenuFrame menuFrame;

  public MyWindowAdapter(MenuFrame menuFrame) {
    this.menuFrame = menuFrame;
  }

  public void windowClosing(WindowEvent we) {
    menuFrame.dispose();
  }
}

class MyMenuHandler implements ActionListener, ItemListener {
  MenuFrame menuFrame;
```

```
  public MyMenuHandler(MenuFrame menuFrame) {
    this.menuFrame = menuFrame;
  }

  // Handle action events.
  public void actionPerformed(ActionEvent ae) {
    String msg = "You selected ";
    String arg = ae.getActionCommand();
    // Activate a dialog box when New is selected.
    if(arg.equals("New...")) {
       msg += "New.";
       SampleDialog d = new
         SampleDialog(menuFrame, "New Dialog Box");
       d.setVisible(true);
    }
    // Try defining other dialog boxes for these options.
    else if(arg.equals("Open..."))
      msg += "Open.";
    else if(arg.equals("Close"))
      msg += "Close.";
    else if(arg.equals("Quit..."))
      msg += "Quit.";
    else if(arg.equals("Edit"))
      msg += "Edit.";
    else if(arg.equals("Cut"))
      msg += "Cut.";
    else if(arg.equals("Copy"))
      msg += "Copy.";
    else if(arg.equals("Paste"))
      msg += "Paste.";
    else if(arg.equals("First"))
      msg += "First.";
    else if(arg.equals("Second"))
      msg += "Second.";
    else if(arg.equals("Third"))
      msg += "Third.";
    else if(arg.equals("Debug"))
      msg += "Debug.";
    else if(arg.equals("Testing"))
      msg += "Testing.";
    menuFrame.msg = msg;
    menuFrame.repaint();
  }

  public void itemStateChanged(ItemEvent ie) {
      menuFrame.repaint();
  }
}

// Create frame window.
public class DialogDemo extends Applet {
  Frame f;
```

```
  public void init() {
    f = new MenuFrame("Menu Demo");
    int width = Integer.parseInt(getParameter("width"));
    int height = Integer.parseInt(getParameter("height"));

    setSize(width, height);

    f.setSize(width, height);
    f.setVisible(true);
  }

  public void start() {
    f.setVisible(true);
  }

  public void stop() {
    f.setVisible(false);
  }
}
```

Here is sample output from the **DialogDemo** applet:



TIP On your own, try defining dialog boxes for the other options presented by the menus.

FileDialog

Java provides a built-in dialog box that lets the user specify a file. To create a file dialog box, instantiate an object of type **FileDialog**. This causes a file dialog box to be displayed.

Usually, this is the standard file dialog box provided by the operating system. Here are three **FileDialog** constructors:

FileDialog(Frame *parent*)
FileDialog(Frame *parent*, String *boxName*)
FileDialog(Frame *parent*, String *boxName*, int *how*)

Here, *parent* is the owner of the dialog box. The *boxName* parameter specifies the name displayed in the box's title bar. If *boxName* is omitted, the title of the dialog box is empty. If *how* is **FileDialog.LOAD**, then the box is selecting a file for reading. If *how* is **FileDialog.SAVE**, the box is selecting a file for writing. If *how* is omitted, the box is selecting a file for reading.

FileDialog provides methods that allow you to determine the name of the file and its path as selected by the user. Here are two examples:

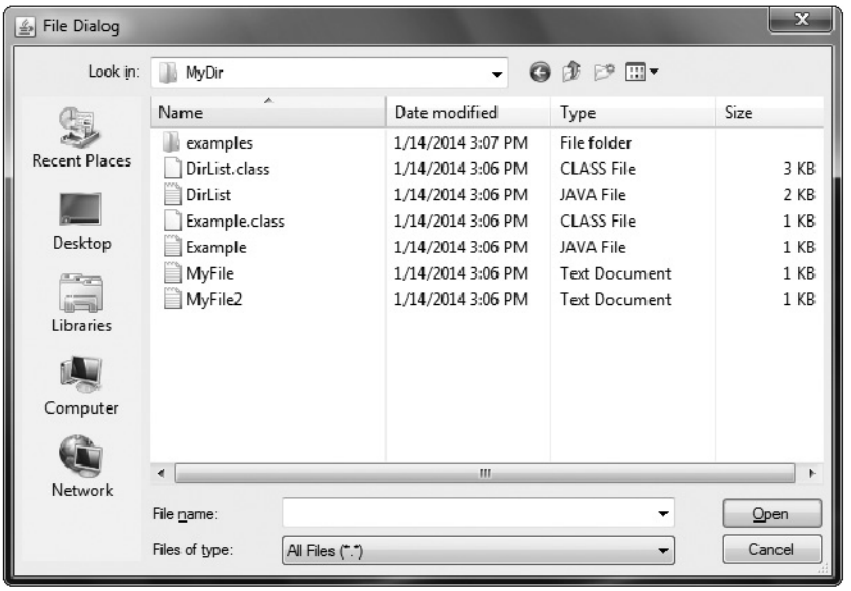
String getDirectory( )
String getFile( )

These methods return the directory and the filename, respectively.

The following program activates the standard file dialog box:

```
/* Demonstrate File Dialog box.

   This is an application, not an applet.
*/
import java.awt.*;
import java.awt.event.*;

//  Create a subclass of Frame.
class SampleFrame extends Frame {
  SampleFrame(String title) {
    super(title);

    // remove the window when closed
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
}

// Demonstrate FileDialog.
class FileDialogDemo {
  public static void main(String args[]) {
    // create a frame that owns the dialog
    Frame f = new SampleFrame("File Dialog Demo");
    f.setVisible(true);
    f.setSize(100, 100);

    FileDialog fd = new FileDialog(f, "File Dialog");

    fd.setVisible(true);
  }
}
```

The output generated by this program is shown here. (The precise configuration of the dialog box may vary.)



One last point: Beginning with JDK 7, you can use **FileDialog** to select a list of files. This functionality is supported by the **setMultipleMode( )**, **isMultipleMode( )**, and **getFiles( )** methods.

A Word About Overriding paint()

Before concluding our examination of AWT controls, a short word about overriding **paint( )** is in order. Although not relevant to the simple AWT examples shown in this book, when overriding **paint( )**, there are times when it is necessary to call the superclass implementation of **paint( )**. Therefore, for some programs, you will need to use this **paint( )** skeleton:

```
public void paint(Graphics g) {

  // code to repaint this window

  // Call superclass paint()
  super.paint(g);
}
```

In Java, there are two general types of components: heavyweight and lightweight. A heavyweight component has its own native window, which is called its *peer*. A lightweight component is implemented completely in Java code and uses the window provided by an ancestor. The AWT controls described and used in this chapter are all heavyweight. However, if a container holds any lightweight components (that is, has lightweight child components), your override of **paint( )** for that container must call **super.paint( )**. By calling **super.paint( )**, you ensure that any lightweight child components, such as lightweight controls, get properly painted. If you are unsure of a child component's type, you can call **isLightweight( )**, defined by **Component**, to find out. It returns **true** if the component is lightweight, and **false** otherwise.

This page has been intentionally left blank

CHAPTER

27 Images

This chapter examines the **Image** class and the **java.awt.image** package. Together, they provide support for *imaging* (the display and manipulation of graphical images). An *image* is simply a rectangular graphical object. Images are a key component of web design. In fact, the inclusion of the **<img>** tag in the Mosaic browser at NCSA (National Center for Supercomputer Applications) is what caused the Web to begin to grow explosively in 1993. This tag was used to include an image *inline* with the flow of hypertext. Java expands upon this basic concept, allowing images to be managed under program control. Because of its importance, Java provides extensive support for imaging.

Images are objects of the **Image** class, which is part of the **java.awt** package. Images are manipulated using the classes found in the **java.awt.image** package. There are a large number of imaging classes and interfaces defined by **java.awt.image**, and it is not possible to examine them all. Instead, we will focus on those that form the foundation of imaging. Here are the **java.awt.image** classes discussed in this chapter:

CropImageFilter	MemoryImageSource
FilteredImageSource	PixelGrabber
ImageFilter	RGBImageFilter

These are the interfaces that we will use:

ImageConsumer	ImageObserver	ImageProducer

Also examined is the **MediaTracker** class, which is part of **java.awt**.

File Formats

Originally, web images could only be in GIF format. The GIF image format was created by CompuServe in 1987 to make it possible for images to be viewed while online, so it was well suited to the Internet. GIF images can have only up to 256 colors each. This limitation

caused the major browser vendors to add support for JPEG images in 1995. The JPEG format was created by a group of photographic experts to store full-color-spectrum, continuous-tone images. These images, when properly created, can be of much higher fidelity as well as more highly compressed than a GIF encoding of the same source image. Another file format is PNG. It too is an alternative to GIF. In almost all cases, you will never care or notice which format is being used in your programs. The Java image classes abstract the differences behind a clean interface.

Image Fundamentals: Creating, Loading, and Displaying

There are three common operations that occur when you work with images: creating an image, loading an image, and displaying an image. In Java, the **Image** class is used to refer to images in memory and to images that must be loaded from external sources. Thus, Java provides ways for you to create a new image object and ways to load one. It also provides a means by which an image can be displayed. Let's look at each.

Creating an Image Object

You might expect that you create a memory image using something like the following:

```
Image test = new Image(200, 100); // Error -- won't work
```

Not so. Because images must eventually be painted on a window to be seen, the **Image** class doesn't have enough information about its environment to create the proper data format for the screen. Therefore, the **Component** class in **java.awt** has a factory method called **createImage( )** that is used to create **Image** objects. (Remember that all of the AWT components are subclasses of **Component**, so all support this method.)

The **createImage( )** method has the following two forms:

Image createImage(ImageProducer *imgProd*)
Image createImage(int *width*, int *height*)

The first form returns an image produced by *imgProd*, which is an object of a class that implements the **ImageProducer** interface. (We will look at image producers later.) The second form returns a blank (that is, empty) image that has the specified width and height. Here is an example:

```
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
```

This creates an instance of **Canvas** and then calls the **createImage( )** method to actually make an **Image** object. At this point, the image is blank. Later, you will see how to write data to it.

Loading an Image

The other way to obtain an image is to load one. One way to do this is to use the **getImage( )** method defined by the **Applet** class. It has the following forms:

Image getImage(URL *url*)
Image getImage(URL *url*, String *imageName*)

The first version returns an **Image** object that encapsulates the image found at the location specified by *url*. The second version returns an **Image** object that encapsulates the image found at the location specified by *url* and having the name specified by *imageName*.

Displaying an Image

Once you have an image, you can display it by using **drawImage( )**, which is a member of the **Graphics** class. It has several forms. The one we will be using is shown here:

boolean drawImage(Image *imgObj*, int *left*, int *top*, ImageObserver *imgOb*)

This displays the image passed in *imgObj* with its upper-left corner specified by *left* and *top*. *imgOb* is a reference to a class that implements the **ImageObserver** interface. This interface is implemented by all AWT (and Swing) components. An *image observer* is an object that can monitor an image while it loads. **ImageObserver** is described in the next section.

With **getImage( )** and **drawImage( )**, it is actually quite easy to load and display an image. Here is a sample applet that loads and displays a single image. The file **Lilies.jpg** is loaded, but you can substitute any GIF, JPG, or PNG file you like (just make sure it is available in the same directory with the HTML file that contains the applet).

```
/*
 * <applet code="SimpleImageLoad" width=400 height=345>
 *  <param name="img" value="Lilies.jpg">
 * </applet>
 */
import java.awt.*;
import java.applet.*;

public class SimpleImageLoad extends Applet
{
  Image img;

  public void init() {
    img = getImage(getDocumentBase(), getParameter("img"));
  }

  public void paint(Graphics g) {
    g.drawImage(img, 0, 0, this);
  }
}
```

In the **init( )** method, the **img** variable is assigned to the image returned by **getImage( )**. The **getImage( )** method uses the string returned by **getParameter("img")** as the filename for the image. This image is loaded from a **URL** that is relative to the result of **getDocumentBase( )**, which is the **URL** of the HTML page this applet tag was in. The filename returned by **getParameter("img")** comes from the applet tag **<param name= "img" value="Lilies.jpg">**. This is the equivalent, if a little slower, of using the HTML tag **<img src="Lilies.jpg" width=400 height=345>**. Figure 27-1 shows what it looks like when you run the program.

When this applet runs, it starts loading **img** in the **init( )** method. Onscreen you can see the image as it loads from the network, because **Applet**'s implementation of the **ImageObserver** interface calls **paint( )** every time more image data arrives.

Figure 27-1 Sample output from **SimpleImageLoad**

Seeing the image load is somewhat informative, but it might be better if you use the time it takes to load the image to do other things in parallel. That way, the fully formed image can simply appear on the screen in an instant, once it is fully loaded. You can use **ImageObserver**, described next, to monitor loading an image while you paint the screen with other information.

ImageObserver

ImageObserver is an interface used to receive notification as an image is being generated, and it defines only one method: **imageUpdate( )**. Using an image observer allows you to perform other actions, such as show a progress indicator or an attract screen, as you are informed of the progress of the download. This kind of notification is very useful when an image is being loaded over a slow network.

The **imageUpdate( )** method has this general form:

```
boolean imageUpdate(Image imgObj, int flags, int left, int top,
                    int width, int height)
```

Here, *imgObj* is the image being loaded, and *flags* is an integer that communicates the status of the update report. The four integers *left*, *top*, *width*, and *height* represent a rectangle that contains different values depending on the values passed in *flags*. **imageUpdate( )** should return **false** if it has completed loading, and **true** if there is more image to process.

The *flags* parameter contains one or more bit flags defined as static variables inside the **ImageObserver** interface. These flags and the information they provide are listed in Table 27-1.

Flag	Meaning
WIDTH	The *width* parameter is valid and contains the width of the image.
HEIGHT	The *height* parameter is valid and contains the height of the image.
PROPERTIES	The properties associated with the image can now be obtained using **imgObj.getProperty( )**.
SOMEBITS	More pixels needed to draw the image have been received. The parameters *left*, *top*, *width*, and *height* define the rectangle containing the new pixels.
FRAMEBITS	A complete frame that is part of a multiframe image, which was previously drawn, has been received. This frame can be displayed. The *left*, *top*, *width*, and *height* parameters are not used.
ALLBITS	The image is now complete. The *left*, *top*, *width*, and *height* parameters are not used.
ERROR	An error has occurred to an image that was being tracked asynchronously. The image is incomplete and cannot be displayed. No further image information will be received. The ABORT flag will also be set to indicate that the image production was aborted.
ABORT	An image that was being tracked asynchronously was aborted before it was complete. However, if an error has not occurred, accessing any part of the image's data will restart the production of the image.

Table 27-1 Bit Flags of the **imageUpdate( )** *flags* Parameter

The **Applet** class has an implementation of the **imageUpdate( )** method for the **ImageObserver** interface that is used to repaint images as they are loaded. You can override this method in your class to change that behavior.

Here is a simple example of an **imageUpdate( )** method:

```
public boolean imageUpdate(Image img, int flags,
                           int x, int y, int w, int h) {
  if ((flags & ALLBITS) == 0) {
    System.out.println("Still processing the image.");
    return true;
  } else {
    System.out.println("Done processing the image.");
    return false;
  }
}
```

Double Buffering

Not only are images useful for storing pictures, as we've just shown, but you can also use them as offscreen drawing surfaces. This allows you to render any image, including text and graphics, to an offscreen buffer that you can display at a later time. The advantage to doing this is that the image is seen only when it is complete. Drawing a complicated image could take several milliseconds or more, which can be seen by the user as flashing or flickering. This flashing is distracting and causes the user to perceive your rendering as slower than it actually is. Use of an offscreen image to reduce flicker is called *double buffering*, because the

screen is considered a buffer for pixels, and the offscreen image is the second buffer, where you can prepare pixels for display.

Earlier in this chapter, you saw how to create a blank **Image** object. Now you will see how to draw on that image rather than the screen. As you recall from earlier chapters, you need a **Graphics** object in order to use any of Java's rendering methods. Conveniently, the **Graphics** object that you can use to draw on an **Image** is available via the **getGraphics( )** method. Here is a code fragment that creates a new image, obtains its graphics context, and fills the entire image with red pixels:

```
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
Graphics gc = test.getGraphics();
gc.setColor(Color.red);
gc.fillRect(0, 0, 200, 100);
```

Once you have constructed and filled an offscreen image, it will still not be visible. To actually display the image, call **drawImage( )**. Here is an example that draws a time-consuming image to demonstrate the difference that double buffering can make in perceived drawing time:

```
/*
 * <applet code=DoubleBuffer width=250 height=250>
 * </applet>
 */
 import java.awt.*;
 import java.awt.event.*;
 import java.applet.*;

 public class DoubleBuffer extends Applet {
   int gap = 3;
   int mx, my;
   boolean flicker = true;
   Image buffer = null;
   int w, h;

   public void init() {
     Dimension d = getSize();
     w = d.width;
     h = d.height;
     buffer = createImage(w, h);
     addMouseMotionListener(new MouseMotionAdapter() {
       public void mouseDragged(MouseEvent me) {
         mx = me.getX();
         my = me.getY();
         flicker = false;
         repaint();
       }
       public void mouseMoved(MouseEvent me) {
         mx = me.getX();
         my = me.getY();
         flicker = true;
```

```
        repaint();
      }
    });
  }

  public void paint(Graphics g) {
    Graphics screengc = null;

    if (!flicker) {
      screengc = g;
      g = buffer.getGraphics();
    }

    g.setColor(Color.blue);
    g.fillRect(0, 0, w, h);

    g.setColor(Color.red);
    for (int i=0; i<w; i+=gap)
      g.drawLine(i, 0, w-i, h);
    for (int i=0; i<h; i+=gap)
      g.drawLine(0, i, w, h-i);

    g.setColor(Color.black);
    g.drawString("Press mouse button to double buffer", 10, h/2);

    g.setColor(Color.yellow);
    g.fillOval(mx - gap, my - gap, gap*2+1, gap*2+1);

    if (!flicker) {
      screengc.drawImage(buffer, 0, 0, null);
    }
  }

  public void update(Graphics g) {
    paint(g);
  }
}
```

This simple applet has a complicated **paint( )** method. It fills the background with blue and then draws a red moiré pattern on top of that. It paints some black text on top of that and then paints a yellow circle centered at the coordinates **mx**, **my**. The **mouseMoved( )** and **mouseDragged( )** methods are overridden to track the mouse position. These methods are identical, except for the setting of the **flicker** Boolean variable. **mouseMoved( )** sets **flicker** to **true**, and **mouseDragged( )** sets it to **false**. This has the effect of calling **repaint( )** with **flicker** set to **true** when the mouse is moved (but no button is pressed) and set to **false** when the mouse is dragged with any button pressed.

When **paint( )** gets called with **flicker** set to **true**, we see each drawing operation as it is executed on the screen. In the case where a mouse button is pressed and **paint( )** is called with **flicker** set to **false**, we see quite a different picture. The **paint( )** method swaps the **Graphics** reference **g** with the graphics context that refers to the offscreen canvas, **buffer**, which we created in **init( )**. Then all of the drawing operations are invisible. At the end of **paint( )**, we simply call **drawImage( )** to show the results of these drawing methods all at once.

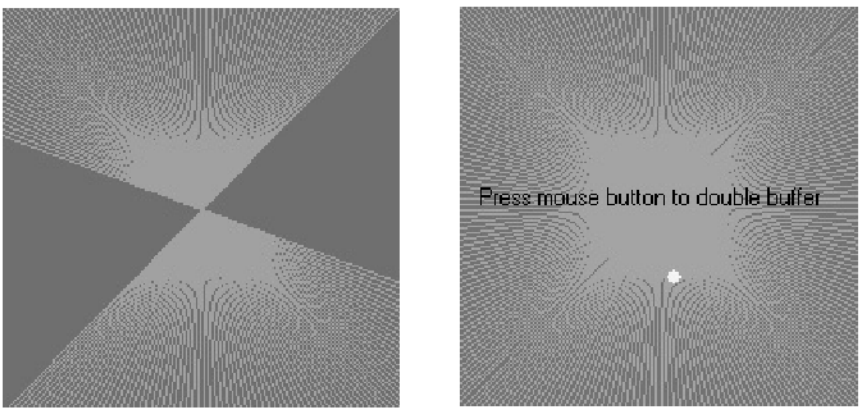


Figure 27-2 Output from **DoubleBuffer** without (left) and with (right) double buffering

Notice that it is okay to pass in a **null** as the fourth parameter to **drawImage( )**. This is the parameter used to pass an **ImageObserver** object that receives notification of image events. Since this is an image that is not being produced from a network stream, we have no need for notification. The left snapshot in Figure 27-2 is what the applet looks like with the mouse button not pressed. As you can see, the image was in the middle of repainting when this snapshot was taken. The right snapshot shows how, when a mouse button is pressed, the image is always complete and clean due to double buffering.

MediaTracker

A **MediaTracker** is an object that will check the status of an arbitrary number of images in parallel. To use **MediaTracker**, you create a new instance and use its **addImage( )** method to track the loading status of an image. **addImage( )** has the following general forms:

void addImage(Image *imgObj*, int *imgID*)
void addImage(Image *imgObj*, int *imgID*, int *width*, int *height*)

Here, *imgObj* is the image being tracked. Its identification number is passed in *imgID*. ID numbers do not need to be unique. You can use the same number with several images as a means of identifying them as part of a group. Furthermore, images with lower IDs are given priority over those with higher IDs when loading. In the second form, *width* and *height* specify the dimensions of the object when it is displayed.

Once you've registered an image, you can check whether it's loaded, or you can wait for it to completely load. To check the status of an image, call **checkID( )**. The version used in this chapter is shown here:

boolean checkID(int *imgID*)

Here, *imgID* specifies the ID of the image you want to check. The method returns **true** if all images that have the specified ID have been loaded (or if an error or user-abort has terminated loading). Otherwise, it returns **false**. You can use the **checkAll( )** method to see if all images being tracked have been loaded.

You should use **MediaTracker** when loading a group of images. If all of the images that you're interested in aren't downloaded, you can display something else to entertain the user until they all arrive.

CAUTION If you use **MediaTracker** once you've called **addImage( )** on an image, a reference in **MediaTracker** will prevent the system from garbage collecting it. If you want the system to be able to garbage collect images that were being tracked, make sure it can collect the **MediaTracker** instance as well.

Here's an example that loads a three-image slide show and displays a nice bar chart of the loading progress:


```
/*
 * <applet code="TrackedImageLoad" width=400 height=345>
 * <param name="img"
 * value="Lilies+SunFlower+ConeFlowers">
 * </applet>
 */
import java.util.*;
import java.applet.*;
import java.awt.*;

public class TrackedImageLoad extends Applet implements Runnable {
  MediaTracker tracker;
  int tracked;
  int frame_rate = 5;
  int current_img = 0;
  Thread motor;
  static final int MAXIMAGES = 10;
  Image img[] = new Image[MAXIMAGES];
  String name[] = new String[MAXIMAGES];
  volatile boolean stopFlag;

  public void init() {
    tracker = new MediaTracker(this);
    StringTokenizer st = new StringTokenizer(getParameter("img"),
                                             "+");

    while(st.hasMoreTokens() && tracked <= MAXIMAGES) {
      name[tracked] = st.nextToken();
      img[tracked] = getImage(getDocumentBase(),
                              name[tracked] + ".jpg");
      tracker.addImage(img[tracked], tracked);
      tracked++;
    }
  }

  public void paint(Graphics g) {
    String loaded = "";
    int donecount = 0;

    for(int i=0; i<tracked; i++) {
      if (tracker.checkID(i, true)) {
```

```
            donecount++;
            loaded += name[i] + " ";
          }
        }

        Dimension d = getSize();
        int w = d.width;
        int h = d.height;

        if (donecount == tracked) {
          frame_rate = 1;
          Image i = img[current_img++];
          int iw = i.getWidth(null);
          int ih = i.getHeight(null);
          g.drawImage(i, (w - iw)/2, (h - ih)/2, null);
          if (current_img >= tracked)
            current_img = 0;
        } else {
          int x = w * donecount / tracked;
          g.setColor(Color.black);
          g.fillRect(0, h/3, x, 16);
          g.setColor(Color.white);
          g.fillRect(x, h/3, w-x, 16);
          g.setColor(Color.black);
          g.drawString(loaded, 10, h/2);
        }
      }

      public void start() {
        motor = new Thread(this);
        stopFlag = false;
        motor.start();
      }

      public void stop() {
        stopFlag = true;
      }

      public void run() {
        motor.setPriority(Thread.MIN_PRIORITY);
        while (true) {
          repaint();
          try {
            Thread.sleep(1000/frame_rate);
          } catch (InterruptedException e) {
            System.out.println("Interrupted");
            return;
          }
          if(stopFlag)
            return;
        }
      }
    }
```

This example creates a new **MediaTracker** in the **init( )** method and then adds each of the named images as a tracked image with **addImage( )**. In the **paint( )** method, it calls **checkID( )** on each of the images that we're tracking. If all of the images are loaded, they are displayed. If not, a simple bar chart of the number of images loaded is shown, with the names of the fully loaded images displayed underneath the bar.

ImageProducer

ImageProducer is an interface for objects that want to produce data for images. An object that implements the **ImageProducer** interface will supply integer or byte arrays that represent image data and produce **Image** objects. As you saw earlier, one form of the **createImage( )** method takes an **ImageProducer** object as its argument. There are two image producers contained in **java.awt.image**: **MemoryImageSource** and **FilteredImageSource**. Here, we will examine **MemoryImageSource** and create a new **Image** object from data generated in an applet.

MemoryImageSource

MemoryImageSource is a class that creates a new **Image** from an array of data. It defines several constructors. Here is the one we will be using:

MemoryImageSource(int *width*, int *height*, int *pixel*[ ], int *offset*,
int *scanLineWidth*)

The **MemoryImageSource** object is constructed out of the array of integers specified by *pixel*, in the default RGB color model to produce data for an **Image** object. In the default color model, a pixel is an integer with Alpha, Red, Green, and Blue (0xAARRGGBB). The Alpha value represents a degree of transparency for the pixel. Fully transparent is 0 and fully opaque is 255. The width and height of the resulting image are passed in *width* and *height*. The starting point in the pixel array to begin reading data is passed in *offset*. The width of a scan line (which is often the same as the width of the image) is passed in *scanLineWidth*.

The following short example generates a **MemoryImageSource** object using a variation on a simple algorithm (a bitwise-exclusive-OR of the x and y address of each pixel) from the book *Beyond Photography, The Digital Darkroom* by Gerard J. Holzmann (Prentice Hall, 1988).

```
/*
 * <applet code="MemoryImageGenerator" width=256 height=256>
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class MemoryImageGenerator extends Applet {
  Image img;
  public void init() {
    Dimension d = getSize();
    int w = d.width;
```

```
    int h = d.height;
    int pixels[] = new int[w * h];
    int i = 0;

    for(int y=0; y<h; y++) {
      for(int x=0; x<w; x++) {
        int r = (x^y)&0xff;
        int g = (x*2^y*2)&0xff;
        int b = (x*4^y*4)&0xff;
        pixels[i++] = (255 << 24) | (r << 16) | (g << 8) | b;
      }
    }
    img = createImage(new MemoryImageSource(w, h, pixels, 0, w));
  }

  public void paint(Graphics g) {
    g.drawImage(img, 0, 0, this);
  }
}
```

The data for the new **MemoryImageSource** is created in the **init( )** method. An array of integers is created to hold the pixel values; the data is generated in the nested **for** loops where the **r**, **g**, and **b** values get shifted into a pixel in the **pixels** array. Finally, **createImage( )** is called with a new instance of a **MemoryImageSource** created from the raw pixel data as its parameter. Figure 27-3 shows the image when we run the applet. (It looks much nicer in color.)

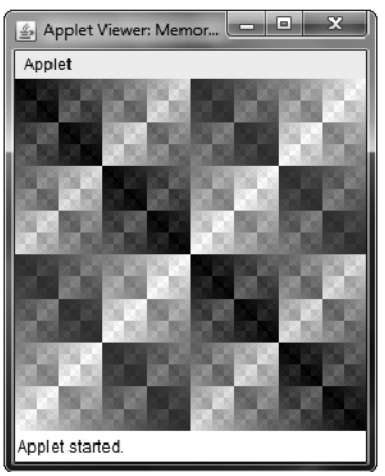


Figure 27-3 Sample output from **MemoryImageGenerator**

ImageConsumer

ImageConsumer is an interface for objects that want to take pixel data from images and supply it as another kind of data. This, obviously, is the opposite of **ImageProducer**, described earlier. An object that implements the **ImageConsumer** interface is going to create **int** or **byte** arrays that represent pixels from an **Image** object. We will examine the **PixelGrabber** class, which is a simple implementation of the **ImageConsumer** interface.

PixelGrabber

The **PixelGrabber** class is defined within **java.lang.image**. It is the inverse of the **MemoryImageSource** class. Rather than constructing an image from an array of pixel values, it takes an existing image and *grabs* the pixel array from it. To use **PixelGrabber**, you first create an array of **int**s big enough to hold the pixel data, and then you create a **PixelGrabber** instance passing in the rectangle that you want to grab. Finally, you call **grabPixels( )** on that instance.

The **PixelGrabber** constructor that is used in this chapter is shown here:

PixelGrabber(Image *imgObj*, int *left*, int *top*, int *width*, int *height*, int *pixel* [ ],
int *offset*, int *scanLineWidth*)

Here, *imgObj* is the object whose pixels are being grabbed. The values of *left* and *top* specify the upper-left corner of the rectangle, and *width* and *height* specify the dimensions of the rectangle from which the pixels will be obtained. The pixels will be stored in *pixel* beginning at *offset*. The width of a scan line (which is often the same as the width of the image) is passed in *scanLineWidth*.

grabPixels( ) is defined like this:

boolean grabPixels( )
throws InterruptedException

boolean grabPixels(long *milliseconds*)
throws InterruptedException

Both methods return **true** if successful and **false** otherwise. In the second form, *milliseconds* specifies how long the method will wait for the pixels. Both throw **InterruptedException** if execution is interrupted by another thread.

Here is an example that grabs the pixels from an image and then creates a histogram of pixel brightness. The *histogram* is simply a count of pixels that are a certain brightness for all brightness settings between 0 and 255. After the applet paints the image, it draws the histogram over the top.

```
/*
 * <applet code=HistoGrab width=400 height=345>
 * <param name=img value=Lilies.jpg>
 * </applet> */
import java.applet.*;
import java.awt.* ;
import java.awt.image.* ;
```

```
public class HistoGrab extends Applet {
  Dimension d;
  Image img;
  int iw, ih;
  int pixels[];
  int w, h;
  int hist[] = new int[256];
  int max_hist = 0;

  public void init() {
    d = getSize();
    w = d.width;
    h = d.height;

    try {
      img = getImage(getDocumentBase(), getParameter("img"));
      MediaTracker t = new MediaTracker(this);

      t.addImage(img, 0);
      t.waitForID(0);
      iw = img.getWidth(null);
      ih = img.getHeight(null);
      pixels = new int[iw * ih];
      PixelGrabber pg = new PixelGrabber(img, 0, 0, iw, ih,
                                         pixels, 0, iw);
      pg.grabPixels();
    } catch (InterruptedException e) {
      System.out.println("Interrupted");
      return;
    }

    for (int i=0; i<iw*ih; i++) {
      int p = pixels[i];
      int r = 0xff & (p >> 16);
      int g = 0xff & (p >> 8);
      int b = 0xff & (p);
      int y = (int) (.33 * r + .56 * g + .11 * b);
      hist[y]++;
    }
    for (int i=0; i<256; i++) {
      if (hist[i] > max_hist)
        max_hist = hist[i];
    }
  }

  public void update() {}

  public void paint(Graphics g) {
    g.drawImage(img, 0, 0, null);
    int x = (w - 256) / 2;
    int lasty = h - h * hist[0] / max_hist;
```

```
    for (int i=0; i<256; i++, x++) {
      int y = h - h * hist[i] / max_hist;
      g.setColor(new Color(i, i, i));
      g.fillRect(x, y, 1, h);
      g.setColor(Color.red);
      g.drawLine(x-1,lasty,x,y);
      lasty = y;
    }
  }
}
```

Figure 27-4 shows an example image and its histogram.

ImageFilter

Given the **ImageProducer** and **ImageConsumer** interface pair—and their concrete classes **MemoryImageSource** and **PixelGrabber**—you can create an arbitrary set of translation filters that takes a source of pixels, modifies them, and passes them on to an arbitrary consumer. This mechanism is analogous to the way concrete classes are created from the abstract I/O classes **InputStream**, **OutputStream**, **Reader**, and **Writer** (described in Chapter 20). This stream model for images is completed by the introduction of the **ImageFilter** class. Some subclasses of **ImageFilter** in the **java.awt.image** package are **AreaAveragingScaleFilter**, **CropImageFilter**, **ReplicateScaleFilter**, and **RGBImageFilter**. There is also an implementation of **ImageProducer** called **FilteredImageSource**, which takes an arbitrary **ImageFilter** and wraps it around an **ImageProducer** to filter the pixels it produces. An instance of **FilteredImageSource**

Figure 27-4 Sample output from **HistoGrab**

can be used as an **ImageProducer** in calls to **createImage( )**, in much the same way that **BufferedInputStream**s can be passed off as **InputStream**s.

In this chapter, we examine two filters: **CropImageFilter** and **RGBImageFilter**.

CropImageFilter

CropImageFilter filters an image source to extract a rectangular region. One situation in which this filter is valuable is where you want to use several small images from a single, larger source image. Loading twenty 2K images takes much longer than loading a single 40K image that has many frames of an animation tiled into it. If every subimage is the same size, then you can easily extract these images by using **CropImageFilter** to disassemble the block once your program starts. Here is an example that creates 16 images taken from a single image. The tiles are then scrambled by swapping a random pair from the 16 images 32 times.

```
/*
 * <applet code=TileImage width=400 height=345>
 * <param name=img value=Lilies.jpg>
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class TileImage extends Applet {
  Image img;
  Image cell[] = new Image[4*4];
  int iw, ih;
  int tw, th;

  public void init() {
    try {
      img = getImage(getDocumentBase(), getParameter("img"));
      MediaTracker t = new MediaTracker(this);
      t.addImage(img, 0);
      t.waitForID(0);
      iw = img.getWidth(null);
      ih = img.getHeight(null);
      tw = iw / 4;
      th = ih / 4;
      CropImageFilter f;
      FilteredImageSource fis;
      t = new MediaTracker(this);
      for (int y=0; y<4; y++) {
        for (int x=0; x<4; x++) {
          f = new CropImageFilter(tw*x, th*y, tw, th);
          fis = new FilteredImageSource(img.getSource(), f);
          int i = y*4+x;
          cell[i] = createImage(fis);
          t.addImage(cell[i], i);
        }
```

```
        }
        t.waitForAll();
        for (int i=0; i<32; i++) {
          int si = (int)(Math.random() * 16);
          int di = (int)(Math.random() * 16);
          Image tmp = cell[si];
          cell[si] = cell[di];
          cell[di] = tmp;
        }
      } catch (InterruptedException e) {
        System.out.println("Interrupted");
      }
    }

    public void update(Graphics g) {
      paint(g);
    }

    public void paint(Graphics g) {
      for (int y=0; y<4; y++) {
        for (int x=0; x<4; x++) {
          g.drawImage(cell[y*4+x], x * tw, y * th, null);
        }
      }
    }
}
```

Figure 27-5 shows the flowers image scrambled by the **TileImage** applet.

Figure 27-5 Sample output from **TileImage**

RGBImageFilter

The **RGBImageFilter** is used to convert one image to another, pixel by pixel, transforming the colors along the way. This filter could be used to brighten an image, to increase its contrast, or even to convert it to grayscale.

To demonstrate **RGBImageFilter**, we have developed a somewhat complicated example that employs a dynamic plug-in strategy for image-processing filters. We've created an interface for generalized image filtering so that an applet can simply load these filters based on **<param>** tags without having to know about all of the **ImageFilter**s in advance. This example consists of the main applet class called **ImageFilterDemo**, the interface called **PlugInFilter**, and a utility class called **LoadedImage**, which encapsulates some of the **MediaTracker** methods we've been using in this chapter. Also included are three filters—**Grayscale**, **Invert**, and **Contrast**—which simply manipulate the color space of the source image using **RGBImageFilter**s, and two more classes—**Blur** and **Sharpen**—which do more complicated "convolution" filters that change pixel data based on the pixels surrounding each pixel of source data. **Blur** and **Sharpen** are subclasses of an abstract helper class called **Convolver**. Let's look at each part of our example.

ImageFilterDemo.java

The **ImageFilterDemo** class is the applet framework for our sample image filters. It employs a simple **BorderLayout**, with a **Panel** at the *South* position to hold the buttons that will represent each filter. A **Label** object occupies the *North* slot for informational messages about filter progress. The *Center* is where the image (which is encapsulated in the **LoadedImage Canvas** subclass, described later) is put. We parse the buttons/filters out of the **filters <param>** tag, separating them with +'s using a **StringTokenizer**.

The **actionPerformed( )** method is interesting because it uses the label from a button as the name of a filter class that it tries to load with **(PlugInFilter) Class.forName(a).newInstance( )**. This method is robust and takes appropriate action if the button does not correspond to a proper class that implements **PlugInFilter**.

```
/*
 * <applet code=ImageFilterDemo width=400 height=345>
 * <param name=img value=Lilies.jpg>
 * <param name=filters value="Grayscale+Invert+Contrast+Blur+Sharpen">
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class ImageFilterDemo extends Applet implements ActionListener {
  Image img;
  PlugInFilter pif;
  Image fimg;
  Image curImg;
  LoadedImage lim;
  Label lab;
  Button reset;
```

```
  public void init() {
    setLayout(new BorderLayout());
    Panel p = new Panel();

    add(p, BorderLayout.SOUTH);
    reset = new Button("Reset");
    reset.addActionListener(this);
    p.add(reset);
    StringTokenizer st = new StringTokenizer(getParameter("filters"), "+");

    while(st.hasMoreTokens()) {
      Button b = new Button(st.nextToken());
      b.addActionListener(this);
      p.add(b);
    }

    lab = new Label("");
    add(lab, BorderLayout.NORTH);

    img = getImage(getDocumentBase(), getParameter("img"));
    lim = new LoadedImage(img);
    add(lim, BorderLayout.CENTER);
  }

  public void actionPerformed(ActionEvent ae) {
    String a = "";

    try {
      a = ae.getActionCommand();
      if (a.equals("Reset")) {
        lim.set(img);
        lab.setText("Normal");
      }
      else {
        pif = (PlugInFilter) Class.forName(a).newInstance();
        fimg = pif.filter(this, img);
        lim.set(fimg);
        lab.setText("Filtered: " + a);
      }
      repaint();
    } catch (ClassNotFoundException e) {
      lab.setText(a + " not found");
      lim.set(img);
      repaint();
    } catch (InstantiationException e) {
      lab.setText("couldn't new " + a);
    } catch (IllegalAccessException e) {
      lab.setText("no access: " + a);
    }
  }
}
```

Figure 27-6 shows what the applet looks like when it is first loaded using the applet tag shown at the top of this source file.

Figure 27-6 Sample normal output from **ImageFilterDemo**

PlugInFilter.java

PlugInFilter is a simple interface used to abstract image filtering. It has only one method, **filter( )**, which takes the applet and the source image and returns a new image that has been filtered in some way.

```
interface PlugInFilter {
  java.awt.Image filter(java.applet.Applet a, java.awt.Image in);
}
```

LoadedImage.java

LoadedImage is a convenient subclass of **Canvas**, which takes an image at construction time and synchronously loads it using **MediaTracker**. **LoadedImage** then behaves properly inside of **LayoutManager** control, because it overrides the **getPreferredSize( )** and **getMinimumSize( )** methods. Also, it has a method called **set( )** that can be used to set a new **Image** to be displayed in this **Canvas**. That is how the filtered image is displayed after the plug-in is finished.

```
import java.awt.*;

public class LoadedImage extends Canvas {
  Image img;

  public LoadedImage(Image i) {
    set(i);
  }

  void set(Image i) {
    MediaTracker mt = new MediaTracker(this);
```

```
    mt.addImage(i, 0);
    try {
      mt.waitForAll();
    } catch (InterruptedException e) {
      System.out.println("Interrupted");
      return;
    }
    img = i;
    repaint();
  }

  public void paint(Graphics g) {
    if (img == null) {
      g.drawString("no image", 10, 30);
    } else {
      g.drawImage(img, 0, 0, this);
    }
  }

  public Dimension getPreferredSize() {
    return new Dimension(img.getWidth(this), img.getHeight(this));
  }

  public Dimension getMinimumSize() {
    return getPreferredSize();
  }
}
```


Grayscale.java

The **Grayscale** filter is a subclass of **RGBImageFilter**, which means that **Grayscale** can use itself as the **ImageFilter** parameter to **FilteredImageSource**'s constructor. Then all it needs to do is override **filterRGB( )** to change the incoming color values. It takes the red, green, and blue values and computes the brightness of the pixel, using the NTSC (National Television Standards Committee) color-to-brightness conversion factor. It then simply returns a gray pixel that is the same brightness as the color source.

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

class Grayscale extends RGBImageFilter implements PlugInFilter {
  public Image filter(Applet a, Image in) {
    return a.createImage(new FilteredImageSource(in.getSource(), this));
  }

  public int filterRGB(int x, int y, int rgb) {
    int r = (rgb >> 16) & 0xff;
    int g = (rgb >> 8) & 0xff;
    int b = rgb & 0xff;
    int k = (int) (.56 * g + .33 * r + .11 * b);
    return (0xff000000 | k << 16 | k << 8 | k);
  }
}
```

Invert.java

The **Invert** filter is also quite simple. It takes apart the red, green, and blue channels and then inverts them by subtracting them from 255. These inverted values are packed back into a pixel value and returned.

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

class Invert extends RGBImageFilter implements PlugInFilter {
  public Image filter(Applet a, Image in) {
    return a.createImage(new FilteredImageSource(in.getSource(), this));
  }

  public int filterRGB(int x, int y, int rgb) {
    int r = 0xff - (rgb >> 16) & 0xff;
    int g = 0xff - (rgb >> 8) & 0xff;
    int b = 0xff - rgb & 0xff;
    return (0xff000000 | r << 16 | g << 8 | b);
  }
}
```

Figure 27-7 shows the image after it has been run through the **Invert** filter.

Figure 27-7 Using the **Invert** filter with **ImageFilterDemo**

Contrast.java

The **Contrast** filter is very similar to **Grayscale**, except its override of **filterRGB( )** is slightly more complicated. The algorithm it uses for contrast enhancement takes the red, green, and blue values separately and boosts them by 1.2 times if they are already brighter than 128. If they are below 128, then they are divided by 1.2. The boosted values are properly clamped at 255 by the **multclamp( )** method.

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

public class Contrast extends RGBImageFilter implements PlugInFilter {

  public Image filter(Applet a, Image in) {
    return a.createImage(new FilteredImageSource(in.getSource(), this));
  }

  private int multclamp(int in, double factor) {
    in = (int) (in * factor);
    return in > 255 ? 255 : in;
  }

  double gain = 1.2;
  private int cont(int in) {
    return (in < 128) ? (int)(in/gain) : multclamp(in, gain);
  }

  public int filterRGB(int x, int y, int rgb) {
    int r = cont((rgb >> 16) & 0xff);
    int g = cont((rgb >> 8) & 0xff);
    int b = cont(rgb & 0xff);
    return (0xff000000 | r << 16 | g << 8 | b);
  }
}
```

Figure 27-8 shows the image after **Contrast** is pressed.

Convolver.java

The abstract class **Convolver** handles the basics of a convolution filter by implementing the **ImageConsumer** interface to move the source pixels into an array called **imgpixels**. It also creates a second array called **newimgpixels** for the filtered data. Convolution filters sample a small rectangle of pixels around each pixel in an image, called the *convolution kernel*. This area, 3 x 3 pixels in this demo, is used to decide how to change the center pixel in the area.

NOTE The reason that the filter can't modify the **imgpixels** array in place is that the next pixel on a scan line would try to use the original value for the previous pixel, which would have just been filtered away.

Figure 27-8 Using the **Contrast** filter with **ImageFilterDemo**

The two concrete subclasses, shown in the next section, simply implement the **convolve( )** method, using **imgpixels** for source data and **newimgpixels** to store the result.

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

abstract class Convolver implements ImageConsumer, PlugInFilter {
  int width, height;
  int imgpixels[], newimgpixels[];
  boolean imageReady = false;

  abstract void convolve(); // filter goes here...

  public Image filter(Applet a, Image in) {
    imageReady = false;
    in.getSource().startProduction(this);

    waitForImage();
    newimgpixels = new int[width*height];

    try {
      convolve();
    } catch (Exception e) {
      System.out.println("Convolver failed: " + e);
      e.printStackTrace();
    }
```

```
    return a.createImage(
      new MemoryImageSource(width, height, newimgpixels, 0, width));
  }

  synchronized void waitForImage() {
    try {
      while(!imageReady) wait();
    } catch (Exception e) {
      System.out.println("Interrupted");
    }
  }

  public void setProperties(java.util.Hashtable<?,?> dummy) { }
  public void setColorModel(ColorModel dummy) { }
  public void setHints(int dummy) { }

  public synchronized void imageComplete(int dummy) {
  imageReady = true;
    notifyAll();
  }

  public void setDimensions(int x, int y) {
    width = x;
    height = y;
    imgpixels = new int[x*y];
  }

  public void setPixels(int x1, int y1, int w, int h,
    ColorModel model, byte pixels[], int off, int scansize) {
    int pix, x, y, x2, y2, sx, sy;

    x2 = x1+w;
    y2 = y1+h;
    sy = off;
    for(y=y1; y<y2; y++) {
      sx = sy;
      for(x=x1; x<x2; x++) {
        pix = model.getRGB(pixels[sx++]);
        if((pix & 0xff000000) == 0)
          pix = 0x00ffffff;
        imgpixels[y*width+x] = pix;
      }
      sy += scansize;
    }
  }

  public void setPixels(int x1, int y1, int w, int h,
    ColorModel model, int pixels[], int off, int scansize) {
      int pix, x, y, x2, y2, sx, sy;

      x2 = x1+w;
      y2 = y1+h;
      sy = off;
```

```
      for(y=y1; y<y2; y++) {
        sx = sy;
        for(x=x1; x<x2; x++) {
          pix = model.getRGB(pixels[sx++]);
        if((pix & 0xff000000) == 0)
          pix = 0x00ffffff;
        imgpixels[y*width+x] = pix;
      }
      sy += scansize;
    }
  }
}
```

NOTE A built-in convolution filter called **ConvolveOp** is provided by **java.awt.image**. You may want to explore its capabilities on your own.

Blur.java

The **Blur** filter is a subclass of **Convolver** and simply runs through every pixel in the source image array, **imgpixels**, and computes the average of the 3 x 3 box surrounding it. The corresponding output pixel in **newimgpixels** is that average value.

```
public class Blur extends Convolver {
  public void convolve() {
    for(int y=1; y<height-1; y++) {
      for(int x=1; x<width-1; x++) {
        int rs = 0;
        int gs = 0;
        int bs = 0;

        for(int k=-1; k<=1; k++) {
          for(int j=-1; j<=1; j++) {
            int rgb = imgpixels[(y+k)*width+x+j];
            int r = (rgb >> 16) & 0xff;
            int g = (rgb >> 8) & 0xff;
            int b = rgb & 0xff;
            rs += r;
            gs += g;
            bs += b;
          }
        }

        rs /= 9;
        gs /= 9;
        bs /= 9;

       newimgpixels[y*width+x] = (0xff000000 |
                                  rs << 16 | gs << 8 | bs);
      }
    }
  }
}
```

Figure 27-9 shows the applet after **Blur**.

Figure 27-9 Using the **Blur** filter with **ImageFilterDemo**

Sharpen.java

The **Sharpen** filter is also a subclass of **Convolver** and is (more or less) the inverse of **Blur**. It runs through every pixel in the source image array, **imgpixels**, and computes the average of the 3 x 3 box surrounding it, not counting the center. The corresponding output pixel in **newimgpixels** has the difference between the center pixel and the surrounding average added to it. This basically says that if a pixel is 30 brighter than its surroundings, make it another 30 brighter. If, however, it is 10 darker, then make it another 10 darker. This tends to accentuate edges while leaving smooth areas unchanged.

```
public class Sharpen extends Convolver {

  private final int clamp(int c) {
    return (c > 255 ? 255 : (c < 0 ? 0 : c));
  }

  public void convolve() {
    int r0=0, g0=0, b0=0;

    for(int y=1; y<height-1; y++) {
      for(int x=1; x<width-1; x++) {
        int rs = 0;
        int gs = 0;
        int bs = 0;

        for(int k=-1; k<=1; k++) {
          for(int j=-1; j<=1; j++) {
```

```
            int rgb = imgpixels[(y+k)*width+x+j];
            int r = (rgb >> 16) & 0xff;
            int g = (rgb >> 8) & 0xff;
            int b = rgb & 0xff;
            if (j == 0 && k == 0) {
              r0 = r;
              g0 = g;
              b0 = b;
            } else {
              rs += r;
              gs += g;
              bs += b;
            }
          }
        }

        rs >>= 3;
        gs >>= 3;
        bs >>= 3;
        newimgpixels[y*width+x] = (0xff000000 |
                                   clamp(r0+r0-rs) << 16 |
                                   clamp(g0+g0-gs) << 8 |
                                   clamp(b0+b0-bs));
      }
    }
  }
}
```

Figure 27-10 shows the applet after **Sharpen**.

Figure 27-10 Using the **Sharpen** filter with **ImageFilterDemo**

Additional Imaging Classes

In addition to the imaging classes described in this chapter, **java.awt.image** supplies several others that offer enhanced control over the imaging process and that support advanced imaging techniques. Also available is the imaging package called **javax.imageio**. This package supports plug-ins that handle various image formats. If sophisticated graphical output is of special interest to you, then you will want to explore the additional classes found in **java.awt.image** and **javax.imageio**.

This page has been intentionally left blank

CHAPTER

28 The Concurrency Utilities

From the start, Java has provided built-in support for multithreading and synchronization. For example, new threads can be created by implementing **Runnable** or by extending **Thread**; synchronization is available by use of the **synchronized** keyword; and interthread communication is supported by the **wait( )** and **notify( )** methods that are defined by **Object**. In general, this built-in support for multithreading was one of Java's most important innovations and is still one of its major strengths.

However, as conceptually pure as Java's original support for multithreading is, it is not ideal for all applications—especially those that make intensive use of multiple threads. For example, the original multithreading support does not provide several high-level features, such as semaphores, thread pools, and execution managers, that facilitate the creation of intensively concurrent programs.

It is important to explain at the outset that many Java programs make use of multithreading and are, therefore, "concurrent." For example, many applets and servlets use multithreading. However, as it is used in this chapter, the term *concurrent program* refers to a program that makes *extensive, integral* use of concurrently executing threads. An example of such a program is one that uses separate threads to simultaneously compute the partial results of a larger computation. Another example is a program that coordinates the activities of several threads, each of which seeks access to information in a database. In this case, read-only accesses might be handled differently from those that require read/write capabilities.

To begin to handle the needs of a concurrent program, JDK 5 added the *concurrency utilities,* also commonly referred to as the *concurrent API*. The original set of concurrency utilities supplied many features that had long been wanted by programmers who develop concurrent applications. For example, it offered synchronizers (such as the semaphore), thread pools, execution managers, locks, several concurrent collections, and a streamlined way to use threads to obtain computational results.

Although the original concurrent API was impressive in its own right, it was significantly expanded by JDK 7. The most important addition was the *Fork/Join Framework*. The Fork/Join Framework facilitates the creation of programs that make use of multiple processors (such as those found in multicore systems). Thus, it streamlines the development of programs in