# Java Methods

## A & AB

Object-Oriented Programming
and
Data Structures

Maria Litvin
Phillips Academy, Andover, Massachusetts

Gary Litvin
Skylight Software, Inc.

Skylight Publishing
Andover, Massachusetts

Skylight Publishing
9 Bartlet Street, Suite 70
Andover, MA  01810

web: http://www.skylit.com
e-mail: sales@skylit.com
support@skylit.com

Library of Congress Control Number: 2005910949

ISBN-10: 0-9727055-7-0
ISBN-13: 978-0-9727055-7-8

**Copyright © 2006 by Maria Litvin, Gary Litvin, and Skylight Publishing**

This material is provided to you as a supplement to the book *Java Methods A&AB*. You may print out one copy for personal use and for face-to-face teaching for each copy of the *Java Methods A&AB* book that you own or receive from your school. You are not authorized to publish or distribute this document in any form without our permission. **You are not permitted to post this document on the Internet**. Feel free to create Internet links to this document's URL on our web site from your web pages, provided this document won't be displayed in a frame surrounded by advertisement or material unrelated to teaching AP* Computer Science or Java. You are not permitted to remove or modify this copyright notice.

* AP and Advanced Placement are registered trademarks of The College Board, which was not involved in the production of and does not endorse this book.

The names of commercially available software and products mentioned in this book are used for identification purposes only and may be trademarks or registered trademarks owned by corporations and other commercial entities. Skylight Publishing and the authors have no affiliation with and disclaim any sponsorship or endorsement by any of these products' manufacturers or trademarks' owners.

Sun, Sun Microsystems, Java, and Java logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

# Chapter 17

## Mouse, Keyboard, Sounds, and Images

17.1 Prologue 2
17.2 Mouse Events Handling 2
17.3 Keyboard Events Handling 4
17.4 *Lab:* Ramblecs Concluded 7
17.5 Sounds and Images 8
17.6 *Case Study and Lab:* Drawing Editor 12
17.7 Summary 17
Exercises 📖

Copyright © 2007 by Skylight Publishing

## 17.1 Prologue

JVM (Java Virtual Machine) has a "virtual" mouse that rolls on an *x-y* plane and has up to three buttons. Mouse coordinates are actually the graphics coordinates of the mouse cursor; they are in pixels, relative to the upper-left corner of the component that registers mouse events. Mouse events can be captured by any object designated as a `MouseListener` (that is, any object of a class that implements the `MouseListener` interface).

Keyboard events can be captured by any object designated as a `KeyListener`. Handling keyboard events in an object-oriented application is complicated by the fact that a computer has only one keyboard and different objects need to listen to it at different times. There is a fairly complicated system of passing the *focus* (the primary responsibility for processing keyboard events) from one component to another and of passing keyboard events from nested components up to their "parents." Handling mouse events is easier than handling keyboard events because the concept of "focus" does not apply.

In this chapter we will discuss the technical details of handling the mouse and the keyboard in a Java GUI application. We will also learn how to load and play audio clips and how to display images and icons.

## 17.2 Mouse Events Handling

The `MouseListener` interface defines five methods: `mousePressed`, `mouseReleased`, `mouseClicked`, `mouseEntered`, and `mouseExited`. Each of these methods receives one parameter, a `MouseEvent`. If `e` is a `MouseEvent`, `e.getX()` and `e.getY()` return the *x* and *y* coordinates of the event, relative to the upper-left corner of the component (usually a panel) whose listener captures the event. The `getButton` method returns an `int` value `MouseEvent.NOBUTTON`, `MouseEvent.BUTTON1`, `MouseEvent.BUTTON2`, or `MouseEvent.BUTTON3`, depending on which, if any, of the mouse buttons has changed state.

You add a mouse listener to a component by calling the component's `addMouseListener` method. It is often convenient to make a panel its own mouse listener. To do that, the panel's constructor can call `addMouseListener(this)`.. We've done that in `JM\Ch15\Puzzle\Puzzle.java`. A mouse listener can be also implemented as a private inner class.

Copyright © 2007 by Skylight Publishing

A class that implements the `MouseMotionListener` interface has `mouseMoved` and `mouseDragged` methods for processing events that report changes in the mouse coordinates without a change in the button state. These methods are used together with `MouseListener` methods. `mouseMoved` is called when the mouse is moved with its button up; `mouseDragged` is called when the mouse is moved with its button held down.

As you know, implementing a Java interface requires the programmer to implement each method in the interface, even those that are never used. In our *Puzzle* program, for example, only the `mousePressed` method of the mouse listener interface is used — the remaining four are empty. To eliminate these empty methods, Java designers came up with a `MouseAdapter` class. `MouseAdapter` implements all the `MouseListener` methods as empty methods. You can extend the adapter class, overriding only the methods you need in your `MouseListener` class. This is often done using an anonymous *inline class* (Figure 17-1).

```
public class MyPanel extends JPanel
{
  public MyPanel() // constructor
  {
    ...

    addMouseListener(new
      MouseAdapter()
      {
        public void mouseClicked(MouseEvent e)
        {
          ... // process click at e.getX(), e.getY()
        }
      } );
    ...
  }
}
```

**Figure 17-1. Adding a `MouseListener` to a panel using `MouseAdapter`**

An anonymous inline class is assumed to extend the class specified in the `new` operator. In this case, the anonymous class extends `MouseAdapter`, overriding its empty `mouseClicked` method. Here the whole expression `new MouseAdapter() {...}` is passed as a parameter to the `addMouseListener` method. Unfortunately, adapter classes undo the discipline of interfaces. If, for instance, you accidentally type `mouselicked` instead of `mouseClicked`, the code will compile but mouse events won't be processed.

Copyright © 2007 by Skylight Publishing

## 17.3 Keyboard Events Handling

Any object of a class that implements the KeyListener interface can capture keyboard events. A KeyListener can be attached to any Component by calling that component's addKeyListener method. However, before the component can process the keystrokes, it must request *focus* — the responsibility for handling the keyboard events. At different times different GUI components obtain focus. Some GUI components, such as buttons or text field objects, receive the focus automatically when they are clicked. Other components, such as JPanel objects, must be explicitly activated by passing the focus to them. The focus is passed to a component by calling the component's requestFocus method.

In the *Ramblecs* program from Chapter 16, for example, we attached a KeyListener to the control panel:

```
controlpanel.addKeyListener(new RamblecsKeyboard(...));
```

The RamblecsKeyboard class implements KeyListener, and controlpanel is an object of the class ControlPanel, which is a subclass of JPanel. However, our controlpanel will never "hear" any keystrokes unless we call its requestFocus method.

In *Ramblecs*, when you click on the button or move the slider, *Swing* automatically shifts keyboard focus to that component, because the button and the slider normally process keystrokes in their own way (for instance, a JButton is programmed to be activated by the spacebar, and a JSlider responds to cursor keys). If you want controlpanel get the focus back, the button listener's actionPerformed and the slider listener's stateChanged has to call controlpanel's requestFocus.

The KeyListener interface specifies three methods: keyPressed, keyReleased, and keyTyped. Each of these methods receives one parameter, KeyEvent e. The KeyEvent class distinguishes "character keys," such as letters, digits, and so on, from "action keys," such as cursor keys, function keys, <Enter>, and so on. Action keys do not have characters associated with them. These keys are identified by their "virtual codes." These virtual codes are defined as public static int constants in the KeyEvent class. Table 17-1 shows the names of several commonly used action keys. The VK prefix stands for "virtual key." For instance, KeyEvent.VK_LEFT refers to the left-arrow cursor key.

Copyright © 2007 by Skylight Publishing

| | |
|---|---|
| VK_F1 through VK_F24 | Function keys |
| VK_LEFT, VK_RIGHT, VK_UP, VK_DOWN | Cursor arrow keys |
| VK_KP_LEFT, VK_KP_RIGHT, VK_KP_UP, VK_KP_DOWN | Cursor arrow keys on the numeric keypad |
| VK_HOME, VK_END, VK_PAGE_UP, VK_PAGE_DOWN, etc. | As implied by the names |

**Table 17-1. KeyEvent's symbolic constants for virtual keys**

The details of what happens on the keyboard, which methods are called in response, and what value is passed to them in the KeyEvent are rather technical. For instance, if you press the shift key and then 'a', then release 'a' and release shift, this will result in two calls to keyPressed, two calls to keyReleased, and one call to keyTyped.

**To keep things simple, use the keyTyped method to capture characters and use either the keyPressed or keyReleased method to capture action keys. When an action key is pressed, keyTyped is not called.**

In the keyTyped method, e.getKeyChar() returns the typed character. In the keyPressed and keyReleased methods, e.getKeyCode() returns the virtual code.

A KeyEvent object also has the boolean methods isShiftDown, isAltDown, and isControlDown, which return true if the respective "modifier" key was held down when the key event occurred.

A more general getModifiers method returns an integer whose individual bits, when set, represent the pressed modifier keys: Shift, Ctrl, Alt, and so on. These bits can be tested by using bit masks defined as static constants in KeyEvent.

Copyright © 2007 by Skylight Publishing

Let's take this opportunity to review the use of Java's bit-wise logical operators. For example, we would use the bit-wise "and" operator to test whether a particular bit in an integer is set to 1:

```
if ((e.getModifiers() & KeyEvent.ALT_MASK) != 0)
  // if ALT is down ...
```

Here `&` is bit-wise "and." A bit in the result is set to 1 if both corresponding bits in the operands are 1. `KeyEvent.ALT_MASK` is an integer constant with only one bit set in it. The condition tests whether this bit is also set in the value returned by `getModifiers` (Figure 17-2).

```
getModifiers()           &  0010…0011001
KeyEvent.ALT_MASK           0000…0001000
                            ============
                            0000…0001000
```

**Figure 17-2. `&` operator used to test a particular bit in an integer**

You can use a combined mask to test whether two modifier keys are held down at once. For example:

```
int mask = KeyEvent.SHIFT_MASK | KeyEvent.CTRL_MASK;
```

Here | is the bit-wise "or" operator: a bit in the result is set to 1 if at least one of the corresponding bits in the operands is 1. So the above statement sets both the `Shift` and `Ctrl` bits in `mask` (Figure 17-3). Then

```
if ((e.getModifiers() & mask) == mask)
```

tests whether the two bits "cut out" by `mask` from the value returned by `getModifiers` are both set (Figure 17-4).

> **Be very careful not to misuse bit-wise operators instead of `&&` and `||` operators in `boolean` expressions. They are allowed, but they don't follow short-circuit evaluation rules.**

Copyright © 2007 by Skylight Publishing

```
KeyEvent.SHIFT_MASK   |  0000…0000001
KeyEvent.CTRL_MASK       0000…0000010
                         ============
                         0000…0000011
```

**Figure 17-3. | operator is used to combine bits in two integers**

```
getModifiers()        &  0010…0010110
mask                     0000…0000011
                         ============
getModifiers() & mask    0000…0000010
```

**Figure 17-4. `&` operator is used to "cut out" mask bits from an integer**

## 17.4 *Lab:* Ramblecs Concluded

In this lab we return to the *Ramblecs* program from the previous chapter. Your task here is to write the `RamblecsKeyListener` class. This class implements the `KeyListener` interface and handles keyboard input for *Ramblecs*.

The constructor for a `RamblecsKeyListener` takes two parameters: a `LetterPanel whiteboard` and a `ControlPanel controlpanel`. Table 17-2 lists the actions that `RamblecsKeyListener` takes in response to keystrokes. All other keyboard events should be ignored.

Copyright © 2007 by Skylight Publishing

| Key | Action |
|---|---|
| ENTER | Calls whiteboard's enterWord() |
| Spacebar | Calls whiteboard's dropWord() |
| Left/right cursor keys or arrows on the numeric keypad | Calls whiteboard's moveCubeLeft() or moveCubeRight(), respectively |
| Up cursor key or arrow on the numeric keypad | Calls whiteboard's rotateCube(-1) |
| Down cursor key or arrow on the numeric keypad | Calls whiteboard's rotateCube(1) |
| Ctrl-F | Calls controlpanel's speedUp() |
| Ctrl-S | Calls controlpanel's slowDown() |

**Table 17-2. Keyboard events in *Ramblecs***

Copyright © 2007 by Skylight Publishing

Set up a project with your RamblecsKeyListener and our Ramblecs.java and Ramblecs.jar from JM\Ch16\Ramblecs. Test your class.

## 17.5 Sounds and Images

JApplet's getAudioClip method can be used to load an audio file into your applet. The file can be a .wav file; some other popular formats (.au, .mid, etc.) are supported, too. The file is actually loaded only when your applet attempts to play it for the first time. You need to import Java's AudioClip class to use this method:

```
import java.applet.AudioClip;
```

One overloaded form of getAudioClip takes one parameter: the URL (or pathname) of a file. The URL must be an absolute URL. Use the forward slash to separate directories in the string that represents a path — it works on all systems.

A more convenient form of getAudioClip takes two parameters: the path and file names separately. In this form, you can use as the first parameter the URL returned

by the applet's method `getDocumentBase()`, which returns the path of the HTML file that runs the applet. The second parameter is usually a literal string, the file name. For example:

```
AudioClip bells = getAudioClip(getDocumentBase(), "bells.wav");
```

Alternatively, you can use as the first parameter `getCodeBase()`, which returns the URL or path of the applet's compiled code.

Once loaded, the clip can be played using its `play` method. For example:

```
bells.play();
```

An application (not an applet) can use a static method `newAudioClip(URL url)` of the `Applet` class to load an audio clip. To use this method, you first need to create a URL object from the audio clip file's name. See Sun's tutorial for details [1].

We find this implementation counterintuitive and inelegant: Why use `Applet`'s methods and URLs when you are working on an application? We also came across some technical problems with `AudioClip`'s `play` method: it causes occasional delays when a short sound clip is played frequently. That's why we created our own class, `EasySound`, for loading and playing sound clips. This class is adapted from one of the more advanced examples found on the Sun Developer Network (SDN) pages devoted to sound [1]. As you have seen in several projects and exercises, our `EasySound` is easy to use. For example:

```
EasySound chomp = new EasySound("drop.wav");
chomp.play();
```

`EasySound.java` can be found in `JM\EasyClasses`; `EasySound.class` is included in `JM\EasyClasses\Easy.jar`.

Java has two types of objects that represent images: `Image` and `ImageIcon`. Either of these objects can be created by loading a picture from an image file. The two most common formats for image files are `.gif` (*Graphics Interchange Format*, pronounced "giff") and `.jpg` (*Joint Photographic Experts Group*, pronounced "jay-peg") format.

You can use an applet's `getImage` method to load an `Image`. This method is analogous to the `getAudioClip` method discussed above. There are two forms of `getImage` that take the same kinds of parameters as `getAudioClip`: the absolute

Copyright © 2007 by Skylight Publishing

URL (or pathname) of a file, or a path and a file name as two separate parameters. In the latter form, the first parameter is often `getDocumentBase()`.

The image file is actually loaded only when your program attempts to show it for the first time. You need to import `awt.Image` to use `getImage`.

You display an `image` object by calling `drawImage` from any `paint` or `paintComponent` method. Figure 17-5 offers an example.

```
import java.awt.Image;
import java.awt.Graphics;
import javax.swing.JApplet;

public class ImageTest extends JApplet
{
  private Image picture;

  public void init()
  {
    picture = getImage(getDocumentBase(), "teamPhoto.jpg");
    if (picture == null)
    {
      showStatus("Can't load teamPhoto.jpg");
    }
  }

  public void paint(Graphics g)
  {
    int x = 5, y = 10;
    if (picture != null)
    {
      g.drawImage(picture, x, y, null);
    }
  }
}
```

**Figure 17-5. Drawing an image in an applet**

In the `drawImage` call, the first parameter is a reference to the image; the second and third are the *x* and *y* coordinates of the upper-left corner of the displayed image, relative to the applet area or the component on which the image is drawn; and the fourth is a reference to an `ImageObserver` object, usually `null` or `this`.

Copyright © 2007 by Skylight Publishing

**`image.getWidth(null)` and `image.getHeight(null)` calls return the raw dimensions of `image`.**

Another way to load and show an image uses the `ImageIcon` class defined in the *Swing* package. This works well in both applets and applications. An `ImageIcon` object can be constructed directly from a file or a URL. For example:

```
ImageIcon coin = new ImageIcon("coin.gif");
```

In this example, the `coin.gif` file is located in the same folder as the program's code.

An `ImageIcon` can be also constructed from an `Image` object. For example:

```
private Image coinImage;
< ... load this image, etc. >
ImageIcon coin = new ImageIcon(coinImage);
```

You can display an `ImageIcon` object by calling its `paintIcon` method from any `paint` or `paintComponent` method. See Figure 17-6 for an example.

```
import java.awt.Graphics;
import javax.swing.JPanel;
import javax.swing.ImageIcon;

public class IconTest extends JPanel
{
  private ImageIcon coin;

  public IconTest()
  {
    coin = new ImageIcon("coin.gif");
  }

  public void paintComponent(Graphics g)
  {
    int x = 5, y = 10;
    if (coin != null)
      coin.paintIcon(this, g, x, y);
  }
}
```

**Figure 17-6. Drawing an `ImageIcon`**

Copyright © 2007 by Skylight Publishing

In the `coin.paintIcon` call, the first parameter is a reference to the component on which the icon is displayed; the second is a reference to the `Graphics` context; and the third and fourth are the *x* and *y* coordinates of the upper-left corner of the image, relative to the upper-left corner of the component on which the icon is painted.

**The `getIconWidth()` and `getIconHeight()` calls return the dimensions of the icon; you can also get the icon's image by calling its `getImage` method.**

Many *Swing* components, including `JButton`, `JCheckBox`, `JRadioButton`, and `JLabel`, have convenient constructors and a `setIcon` method that allow you to add an icon to these components (see Appendix D).

## 17.6 *Case Study and Lab:* Drawing Editor

In this lab we will create a *Drawing Editor* program in which the user can add several filled circles of different colors and sizes to the picture and drag and stretch or squeeze them with the mouse or cursor keys. Figure 17-7 shows a snapshot from the program. The program's control panel has two buttons: one for choosing a color, another for adding a "balloon" (a filled circle) to the picture. The third disabled small button in the middle shows the currently selected color.

Copyright © 2007 by Skylight Publishing

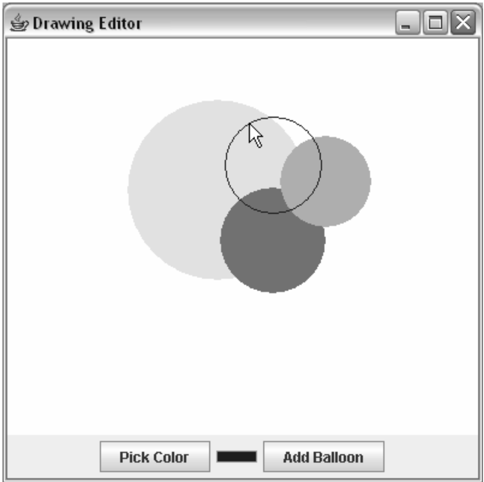


**Figure 17-7. The *BalloonDraw* program**

The user can "grab" any balloon by pressing and holding the mouse on it. The grabbed balloon changes from a solid (filled) shape to the outline only. If the user grabs the balloon somewhere inside it, then he can drag it with the mouse to a new location (while its size remains unchanged); if he grabs the balloon in the vicinity of its border, then he can stretch or squeeze the balloon while its center remains in the same location. When the mouse button is released, the balloon goes back to its solid shape.

The last balloon added or "grabbed" becomes the "active" balloon. The user can move it using cursor (arrow) keys and stretch or squeeze it using the cursor keys with the `Ctrl` key held down. Click on the `BalloonDraw.jar` file in `JM\Ch17\Draw` to see how it all works.

The program consists of four classes (Figure 17-8). `BalloonDraw` (derived from `JFrame`) represents the program window; `ControlPanel` (derived from `JPanel`) represents the panel that holds the buttons; `DrawingPanel` represents the canvas on which balloons are drawn, and `Balloon` represents a balloon object.

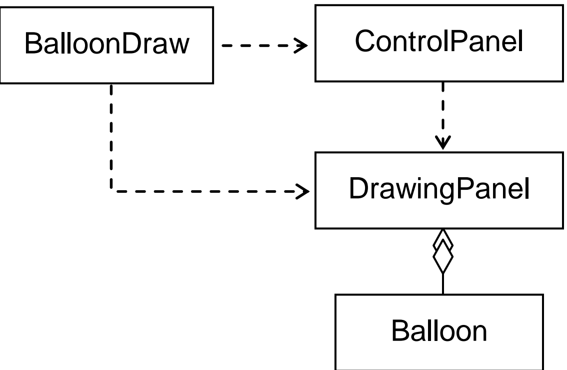


**Figure 17-8. Classes in *BalloonDraw***

Copyright © 2007 by Skylight Publishing

Our *Drawing Editor* program uses a `JColorChooser` for choosing a color. You can learn about it from the "How to Use Color Choosers" lesson [1] in Sun's Java tutorial.

In the previous labs, we provided the GUI front-end classes, while you implemented the "back-end" classes — the details of arithmetic or logic. This is your chance to write a complete program from scratch.

1. `BalloonDraw`

The `BalloonDraw` class extends `JFrame`. In the `BalloonDraw`'s constructor display "Drawing Editor" in the title bar. Create a `DrawingPanel canvas` and a `ControlPanel controls`, passing `canvas` to the `ControlPanel`'s constructor as a parameter (so that `controls` knows what it controls). Attach `canvas` and `controls` to the appropriate regions of the `JFrame`'s content pane. To complete the `BalloonDraw` class, provide a standard `main` method that opens a `BalloonDraw` window on the screen.

2. `ControlPanel`

`ControlPanel` extends `JPanel`. Add three buttons to it: "Pick Color," color display, and "Add Balloon." The color display button is a small disabled button in the middle; its purpose is to show the currently selected color. Call `canvas`'s `getColor` method to obtain the initial color. Attach the appropriate listener to the "Pick Color" and "Add Balloon" buttons, using the control panel itself as a listener, or, if you prefer, objects of two different inner action listener classes or anonymous inline classes. When "Pick Color" is clicked, call `canvas`'s `pickColor` method, then get the selected color back from `canvas` and show that color on the color display button. When "Add Balloon" is clicked, call `canvas`'s `addBalloon` method. Don't forget to return the keyboard focus to `canvas` in either event.

3. `DrawingPanel`

This is where all the work is done. This class is a subclass of `JPanel` and it implements the `MouseListener`, `MouseMotionListener`, and `KeyListener` interfaces. In the `DrawingPanel` constructor, add `this` panel to itself as these three listeners. (If you prefer, implement the three listeners in private inner classes and add them to this panel).
A `DrawingPanel` maintains a list of balloons (an `ArrayList<Balloon>`) and a reference to the "active balloon." The latter refers to the last added balloon or the balloon last picked with a mouse (a balloon is "picked" when the mouse clicks inside of it or on its border). `DrawingPanel` should also have an `enum` field or `boolean` fields that indicate whether a balloon is currently picked and, if so, whether it is being moved or stretched.

Copyright © 2007 by Skylight Publishing

`DrawingPanel`'s constructor and some of the methods are summarized in Table 17-3. The methods that implement the requirements of the `MouseListener`, `MouseMotionListener`, and `KeyListener` interfaces are not shown. Of these only `mousePressed`, `mouseReleased`, `mouseDragged`, and `keyPressed` are used.

| Constructor: `DrawingPanel()` | Sets the background color to white and the initial drawing color to blue. Adds `this` as the `MouseListener`, `MouseMotionListener`, and `KeyListener`. Creates an empty balloon list. |
|---|---|
| Methods: | |
| `Color getColor()` | Returns the current drawing color. |
| `void pickColor()` | Called from `ControlPanel` when the "Pick Color" button is clicked. Brings up a `JColorChooser` and sets the chosen color as the new drawing color. Leaves the drawing color unchanged if the user clicks "Cancel." |
| `void addBalloon()` | Called from `ControlPanel` when the "Add Balloon" button is clicked. Adds a new balloon to the list. The new balloon has its center at the center of the drawing panel, a random radius (within a reasonable range) and the current drawing color. The new balloon is designated as the "active balloon." |
| `void paintComponent(Graphics g)` | Draws all the balloons in the list. The balloons should be drawn in reverse from the order in which they were added to the list. However, if one of the balloons is "picked," then this "active balloon" should be drawn last, in outline only. |

**Table 17-3. The constructor and methods of the `DrawingPanel` class (except the listener methods)**

Copyright © 2007 by Skylight Publishing

When the mouse is pressed on balloons, make sure you pick the topmost balloon that contains the coordinates of the click. If you add balloons at the end of the list, then you need to scan the list from the end backward to achieve that.

The secret for smooth dragging action is to keep constant the *x-y* offsets from the current mouse position to the current balloon's center as the mouse moves. Save these offsets when the mouse button is first pressed down, then set the active balloon's coordinates in `mouseDragged` in such a way that these offsets remain the same.

If the initial click happens in the vicinity of the border of a balloon (that is, `isOnBorder(x,y)` returns `true`), then instead of dragging the balloon make the mouse motion stretch or squeeze it. You need a `boolean` field in `DrawingPanel` to mark whether you will be moving or stretching the picked balloon. The same principle as for moving applies for smooth stretching action, only this time the distance from the current mouse position to the balloon's border must remain constant. (This distance is equal to the distance from the balloon's center, returned by `balloon.distance(x,y)`, minus the balloon's radius.)

Enable cursor keys to move the "active balloon" (if not `null`) or to stretch or squeeze it when the `Ctrl` key is held down.

Don't forget to call `repaint` after adding a balloon and whenever one of the listener methods changes the appearance of the picture.

4. `Balloon`

This class implements a circle with a given center, radius, and color. Its constructor and methods are summarized in Table 17-4.

Thoroughly test all the mouse and keyboard action in your program.

> **Consider what it would take to support balloons of different shapes: round, elongated horizontally, and elongated vertically.**

Copyright © 2007 by Skylight Publishing

| Constructor: | |
|---|---|
| `Balloon(int x, int y, int radius, Color color)` | Creates a balloon with the center at (*x*, *y*) with the specified radius and color. |
| Methods: | |
| `int getX()` | Returns the *x*-coordinate of the center. |
| `int getY()` | Returns the *y*-coordinate of the center. |
| `int getRadius()` | Returns the radius. |
| `double distance(int x, int y)` | Returns the distance from the point (*x*, *y*) to the center of this balloon. |
| `void move(int x, int y)` | Moves the center of this balloon to (*x*, *y*). |
| `void setRadius(int r)` | Sets the radius of this balloon to *r*. |
| `boolean isInside(int x, int y)` | Returns `true` if the point (*x*, *y*) lies inside this balloon (and not on the border), `false` otherwise. |
| `boolean isOnBorder(int x, int y)` | Returns `true` if the point (*x*, *y*) lies approximately on the border of this balloon, `false` otherwise. |
| `void draw(Graphics g, boolean filled)` | Draws this balloon. Draws a filled circle if `filled` is `true`, and a hollow circle otherwise. |

**Table 17-4. The constructor and public methods of the `Balloon` class**

Copyright © 2007 by Skylight Publishing

# 17.7 Summary

Mouse events on a component can be captured and processed by a `MouseListener` object attached to that component. A mouse listener is added to a component (usually a panel) by calling its `addMouseListener` method. It is not uncommon for a panel to be its own mouse listener or use a private inner class. A class that implements a mouse listener interface must have five methods: `mousePressed`, `mouseReleased`, and `mouseClicked`, called upon the corresponding button action,

and `mouseEntered` and `mouseExited`, called when the mouse cursor enters or leaves the component. Each of these methods takes one parameter, a `MouseEvent`. `e.getX()` and `e.getY()` return the *x* and *y* coordinates of the event in pixels, relative to the upper-left corner of the panel whose listener captures it.

For more detailed mouse tracking you can use a `MouseMotionListener` which has two more methods, `mouseMoved` and `mouseDragged`. These methods are called when the mouse moves with the button up or down, respectively.

Adapter classes allow a programmer to supply only those listener methods that the program actually uses, inheriting the other (empty) methods from the adapter class.

Keyboard events can be captured and processed by a `KeyListener` object: an object of a class that implements the `KeyListener` interface and defines the `keyPressed`, `keyReleased`, and `keyTyped` methods. You add a key listener to a component by calling its `addKeyListener` method. A component must obtain "focus" by calling the `requestFocus` method to enable processing of keyboard events.

The `keyPressed` and `keyReleased` methods are used to process "action" keys (such as, Enter, cursor keys, home, etc.). The `keyTyped` method is called for "typed" keys that represent a character. Each of these three methods takes one parameter of the `KeyEvent` type. `KeyEvent`'s `getKeyCode` method returns the virtual code of the key, such as `VK_ENTER`, `VK_LEFT`, `VK_HOME`, and so on. `KeyEvent`'s `getKeyChar` method returns the typed character. `KeyEvent`'s `boolean` methods `isShiftDown`, `isControlDown`, `isAltDown` return `true` if the corresponding modifier key, `Shift`, `Ctrl`, `Alt`, was held down when the event occurred. The `getModifiers` method returns an integer that holds a combination of bits representing the pressed modifier keys. `KeyEvent` has static constants defined for different modifier bits: `KeyEvent.SHIFT`, `KeyEvent.CTRL`, and so on.

You can load an audio file (a `.wav` file or a file in one of several other popular formats) into your applet by calling `JApplet`'s `getAudioClip` method. To play it, call `AudioClip`'s `play` method. In applications, use our `EasySound` class instead.

An `Image` object can be loaded from a `.gif` or `.jpg` file by calling `JApplet`'s `getImage` method; to display it, call `Graphics`'s `drawImage` method.

The `ImageIcon` class in *Swing* provides another way to represent an image in your program. `ImageIcon`'s constructor loads an icon from a file, and `ImageIcon`'s `paintIcon` method displays the image. An `ImageIcon` object can be added to any `JLabel`, `JButton`, `JCheckBox`, or `JRadioButton` object.

Copyright © 2007 by Skylight Publishing

## 🕮 Exercises 🕮

The exercises for this chapter are in the book (*Java Methods A and AB: Object-Oriented Programming and Data Structures, AP Edition*, ISBN 0-9727055-7-0, Skylight Publishing, 2006 [1]).

Copyright © 2007 by Skylight Publishing