Although you have already seen several of the classes and methods in the **acm.graphics** package, you have not yet had the chance to consider that collection of classes as a whole. Rather than continuing to scratch the surface, this chapter examines the entire package as an integrated set of tools. Doing so has two main purposes. First, you will learn a lot more about the graphical capabilities that the package provides. That understanding, in turn, will help you to write more interesting graphics programs, which ought to be an exciting prospect. But there is a subtler purpose as well. The object-oriented approach to programming has less to do with styles of coding than it does with strategies of design. By focusing on the **acm.graphics** package, you can get a sense of what considerations go into the design of a package. By studying existing packages, you will gain a more detailed sense of how to approach the design of your own classes and packages that will make it easier to write them on your own.

8.1 The acm.graphics model

Before you can appreciate the classes and methods available in **acm.graphics**, you need to understand the assumptions, conventions, and metaphors on which the package is based. Collectively, these ideas that define the appropriate mental picture for a package are called a **model.** The model for a package allows you to answer various questions about how you should think about working in that domain. Before using a graphics package, for example, you need to be able to answer the following sorts of questions:

- What are the real-world analogies and metaphors that underlie the package design?
- How do you specify positions on the screen?
- What units do you use to specify lengths?

The collage metaphor

In the case of the **acm.graphics**, the first question—what analogies and metaphors are appropriate for the package—is perhaps the most important to resolve. There are many real-world analogies for computer graphics, because there are many different ways to create visual art. One possible metaphor is that of painting, in which the artist selects a paintbrush and a color and then draws images by moving the brush across a screen that represents a virtual canvas. If you instead imagined yourself drawing with a pencil, you might develop a different style of graphics in which line drawings predominated and in which it was possible to erase things you had previously drawn. You might conceivably design a graphics package around an engraving metaphor in which the artist uses a stylus to etch out a drawing; erasing would presumably not make sense in that metaphor. There are many other possible metaphors, just as there are many styles of art.

To support the notion of object-oriented design, the **acm.graphics** package uses the metaphor of a **collage.** A collage artist works by taking various objects and assembling them on a background canvas. Those objects might be geometrical shapes, words clipped from newspapers, lines formed from bits of string, or images taken from magazines. The **acm.graphics** package offers counterparts for all of these. The notion that the image is a collage has important implications for the way you describe the process of creating a design. If you are painting, you might talk about making a brush stroke in a particular position or filling an area with paint. In the collage model, the key operations are adding and removing objects, along with repositioning them on the background. Collages also have the property that some objects can be positioned on top of other objects, where they obscure whatever is behind them. Removing those objects reveals whatever used to be underneath.

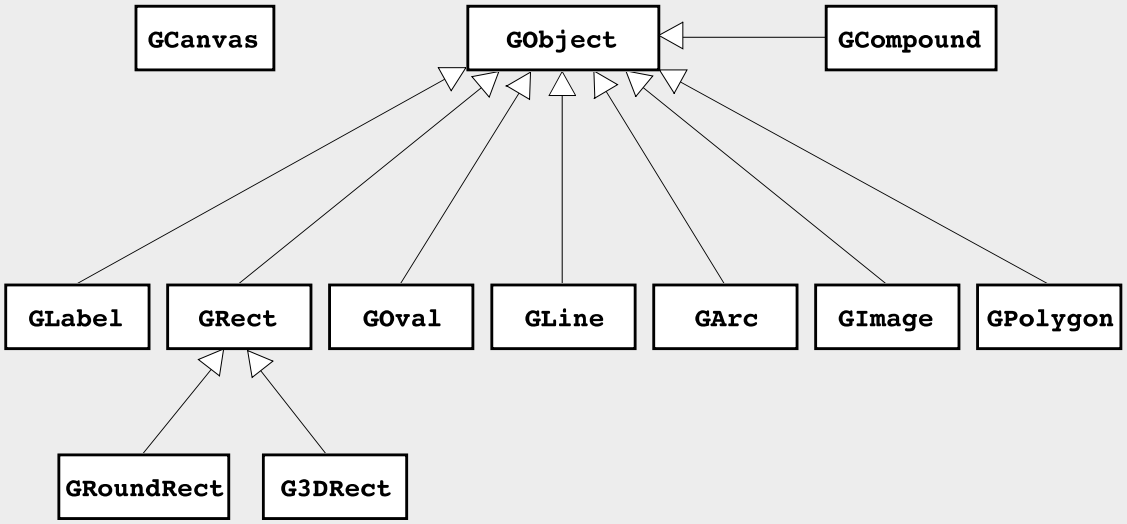
The coordinate system

You already know the answers to the questions about specifying positions on the screen and defining units for length from the discussion of the various **GraphicsProgram** examples presented in the preceding chapters. The fundamental rule is that the **acm.graphics** package uses precisely the coordinate model that traditional Java programs do, so that you will encounter less of a cognitive shift when you learn the more complex Java model later in this text. In the Java model, all lengths are expressed in terms of **pixels,** which are the individual dots that appear on the screen. Points on the screen are identified by specifying their coordinates in both the *x* and *y* directions. Those coordinates are also measured in pixels, with *x* values increasing as you move rightward across the canvas and *y* values increasing as you move down from the top. The interpretation of the *x* coordinate is therefore the same as in traditional Cartesian geometry, but the *y* coordinate is inverted. That inversion follows from the fact that Java uses a different starting point, or **origin,** for its coordinate system. Instead of placing the origin in the lower left corner as in the Cartesian world, Java puts it in the upper left corner. This reversal of the coordinate system will probably cause some confusion at first, but it doesn't take too long to become familiar with the new model.

8.2 The graphics class hierarchy

Once you understand the coordinate system, the next question you might reasonably ask is what sorts of objects are displayable in the collage. Again, you have some intuition for that answer from your holistic explorations of the **acm.graphics** package in the earlier chapters. In fact, you were introduced to most of the important classes in the package back in Chapter 2, in which Figure 2-7 offered a diagram of the shape classes in the **acm.graphics** package as an illustration of class hierarchies. That figure is reproduced as Figure 8-1, which shows the shape classes in the hierarchy along with two other classes—**GCanvas** and **GCompound**—that will be important in understanding the detailed operation of the package.

FIGURE 8-1 **Important classes in the acm.graphics package**



The `GObject` class

As the arrows in the Figure 8-1 emphasize, the central class in the **`acm.graphics`** package is **`GObject`**, which represents the universe of all graphical objects. You may have noticed that Figure 8-1 specifies the name of this class in italic type. In class diagrams, italics are used to identify **abstract classes,** which are classes that have no objects of their own but serve as a common template for a variety of concrete subclasses. Thus, there are no objects that have **`GObject`** as their primary class. Instead, every object you put on the screen is a **`GLabel`**, **`GRect`**, **`GOval`**, or one of the other subclasses that extends **`GObject`**. The rules of class formation, however, dictate that any of those objects is also a **`GObject`**, but you will never see a declaration of the form

```
GObject gobj = new GObject();
```

because you cannot construct a **`GObject`** itself. At the same time, it is perfectly legal to construct an instance of a specific **`GObject`** subclass and then assign it to a **`GObject`** variable, as in

```
GObject gobj = new GLabel("hello, world");
```

The label is a **`GObject`**, so the assignment works.

The **`GObject`** class itself defines the set of methods shown in Figure 8-2. As it happens, you've already seen most of these. Here are a few of the new ones:

- The **`movePolar(r, theta)`** method allows you to move an object using **polar coordinates,** which are expressed as a distance (**`r`**) and an angle (**`theta`**). The angle is measured in degrees counterclockwise from the +*x* axis, just as it is in classical geometry. Thus, the call

  ```
  gobj.movePolar(100, 45);
  ```

  moves the object stored in **`gobj`** 100 pixels along a line in the 45° direction, which is northeast.
- The **`contains(x, y)`** method allows you to determine whether an object contains a particular point. This is a predicate method and therefore returns **`true`** or **`false`**.
- The **`setVisible()`** method makes it possible to hide an object on the screen. If you call **`setVisible(false)`**, the object disappears until you call **`setVisible(true)`**. The predicate method **`isVisible()`** allows you to determine whether an object is visible.
- The various **`send`** methods allow you to change the layering arrangement in the collage. When you add a new object, it goes on top of the other objects and can therefore obscure the objects behind it. If you call **`sendToBack()`**, the object goes to the very back of the list. Conversely, **`sendToFront()`** brings it to the foreground. The **`sendForward()`** and **`sendBackward()`** methods move an object one step forward or backward in the stack. Because the layering axis corresponds to a third dimension at right angles to the *x*-*y* grid, this dimension is typically called the ***z*** **axis.**
- The mouse-handling methods at the bottom of Figure 8-2 make it possible for objects to respond to mouse events. Those capabilities are described in section 8.4.

FIGURE 8-2 Methods supported by all GObject subclasses

`void setLocation(double x, double y)` Sets the location of this object to the specified point.
`void move(double dx, double dy)` Moves the object using the displacements **`dx`** and **`dy`**.
`void movePolar(double r, double theta)` Moves the object **`r`** units in direction **`theta`**, measured in degrees.
`double getX()` Returns the x-coordinate of the object.
`double getY()` Returns the y-coordinate of the object.
`double getWidth()` Returns the width of the object.
`double getHeight()` Returns the height of the object.
`boolean contains(double x, double y)` Checks to see whether a point is inside the object.
`void setColor(Color c)` Sets the color of the object.
`Color getColor()` Returns the object color.
`void setVisible(boolean visible)` Sets whether this object is visible.
`boolean isVisible()` Returns **`true`** if this object is visible.
`void sendToFront()` Sends this object to the front of the canvas, where it may obscure objects further back.
`void sendToBack()` Sends this object to the back of the canvas, where it may be obscured by objects in front.
`void sendForward()` Sends this object forward one position in the *z* ordering.
`void sendBackward()` Sends this object backward one position in the *z* ordering.
`void addMouseListener(MouseListener listener)` Specifies a listener to process mouse events for this graphical object.
`void removeMouseListener(MouseListener listener)` Removes the specified mouse listener from this graphical object.
`void addMouseMotionListener(MouseMotionListener listener)` Specifies a listener to process mouse motion events for this graphical object.
`void removeMouseMotionListener(MouseMotionListener listener)` Removes the specified mouse motion listener from this graphical object.

Although it is nice to know about these new methods, it may be momentarily disturbing to discover that some of the old methods you've been using don't appear in Figure 8-2. There is no **`setFilled`** method, even though you've been filling **`GOval`**s and **`GRect`**s all along. Similarly, there is no **`setFont`** method, but you've been using that in the context of **`GLabel`**s. As the caption on the figure makes clear, the methods defined at the **`GObject`** level are the ones that apply to all the subclasses, and not just some of them. The **`setFont`** method, for example, only makes sense for the **`GLabel`** class and is therefore defined there. The **`setFilled`** method is more interesting. It makes sense to fill an oval, a rectangle, a polygon, and an arc, so there are four classes (plus the extended **`GRoundRect`** and **`G3DRect`** subclasses) for which **`setFilled`** would be appropriate. It is not at all clear, however, what one might mean by filling a line, an image, or a label. Since there are classes that cannot give a meaningful interpretation to **`setFilled`**, it is not

defined at the **GObject** level. At the same time, it doesn't seem ideal to define **setFilled** independently in each of the subclasses for which it is defined. You certainly would like **setFilled** to work the same way for each of the fillable classes, for consistency if nothing else.

In Java, the best way to define a suite of methods that are shared by some but not all subclasses in a particular hierarchy is to define what Java calls an **interface,** which is really just a listing of the method headers that all classes implementing that interface share. In the **acm.graphics** package, the classes **GOval**, **GRect**, **GPolygon**, and **GArc** all implement an interface called **GFillable**, which specifies the behavior of any fillable object. In addition to **GFillable**, there is an interface called **GResizable** that allows you to reset the bounds of an object and a somewhat broader interface called **GScalable** that allows you to scale an object by a scaling factor.

The methods specified by these three interfaces appear in Figure 8-3. The most interesting new method in that list is **setFillColor(c)**, which makes it possible to give the interior of a fillable shape a different color than its boundary. For example, if you execute the code

```
GRect r = new GRect(70, 20);
r.setColor(Color.RED);
r.setFillColor(Color.GREEN);
r.setFilled(true);
```

you get a filled rectangle whose interior is green and whose border is red.

The GLabel class

The **GLabel** class is the first class you encountered in this text, even though it is in some respects the most idiosyncratic entry in the hierarchy of shape classes. The most important difference between **GLabel** and everything else is that the position of a **GLabel** is not defined by the upper left corner, but rather by the starting point of the **baseline,**

FIGURE 8-3 Methods specified by interfaces

GFillable (implemented by **GArc**, **GOval**, **GPolygon**, and **GRect**)

void setFilled(boolean fill) Sets whether this object is filled (**true** means filled, **false** means outlined).
boolean isFilled() Returns **true** if the object is filled.
void setFillColor(Color c) Sets the color used to fill this object. If the color is **null**, filling uses the color of the object.
Color getFillColor() Returns the color used to fill this object.

GResizable (implemented by **GImage**, **GOval**, and **GRect**)

void setSize(double width, double height) Changes the size of this object to the specified width and height.
void setBounds(double x, double y, double width, double height) Changes the bounds of this object as specified by the individual parameters.

GScalable (implemented by **GArc**, **GCompound**, **GImage**, **GLine**, **GOval**, **GPolygon**, and **GRect**)

void scale(double sf) Resizes the object by applying the scale factor in each dimension, leaving the location fixed.
void scale(double sx, double sy) Scales the object independently in the *x* and *y* dimensions by the specified scale factors.

which is an imaginary line on which characters sit. The origin and baseline properties of the **GLabel** class are illustrated in the following diagram:

To set the size, style, and general appearance of a **GLabel**, you need to specify its font, using the **setFont** method as described in Chapter 5. The structure of this method and the other methods specific to the **GLabel** class appear in Figure 8-4.

The GRect class and its subclasses (GRoundRect and G3DRect)

If **GLabel** is the most idiosyncratic class in the **acm.graphics** package, **GRect** is the most conventional. It implements all three of the special interfaces—**GFillable**, **GResizable**, and **GScalable**—but otherwise defines nothing that is not in **GObject**.

The two specialized forms of rectangles, **GRoundRect** and **G3DRect**, differ only in their appearance on the display. The **GRoundRect** class has rounded corners, and the **G3DRect** class supports a raised appearance. Both of these display styles are included in the standard Java graphics package, and are therefore included in **acm.graphics** as well.

The GOval class

The **GOval** class is almost as straightforward as the **GRect** and operates in much the same way. The only thing that causes confusion about **GOval** is that its origin is the upper left corner and not the center. This convention makes sense when you see that ovals are specified in terms of their bounding rectangle, but can seem somewhat unnatural, particularly when the oval is a circle. Circles in mathematics are conventionally defined in terms of their center and their radius. In Java, you need to think of them in terms of their upper left corner and their diameter.

The GLine class

The **GLine** class makes it possible to construct line drawings in the **acm.graphics** package. The **GLine** class implements **GScalable** (which is performed relative to the starting point of the line), but not **GFillable** or **GResizable**. It is, moreover, important to modify the notion of containment for lines, since the idea of being within the boundary of the line is not well defined. In the abstract, of course, a line is infinitely thin and therefore contains no points in its interior. In practice, however, it makes sense to define

FIGURE 8-4 Additional methods in the GLabel class

void setFont(Font f) *or* **setFont(String description)** Sets the font to a Java **Font** object or a string in the form **"Family-style-size"**
Font getFont() Returns the current font.
double getAscent() Returns the distance the characters in the current font extend above the baseline.
double getDescent() Returns the distance the characters in the current font extend below the baseline.
void setLabel(String str) Changes the string used to display the label.
String getLabel() Returns the string that this label displays.

a point as being contained within a line if it is "close enough" to be considered as part of that line. In the **acm.graphics** package, that distance is specified by the constant **LINE_TOLERANCE** in the **GLine** class, which is defined to be a pixel and a half.

The **GLine** class also contains several methods for determining and changing the endpoints of the line. The **setStartPoint** method allows clients to change the first endpoint of the line without changing the second; conversely, **setEndPoint** gives clients access to the second endpoint without affecting the first. These methods are therefore different in their operation from **setLocation**, which moves the entire line without changing its length and orientation.

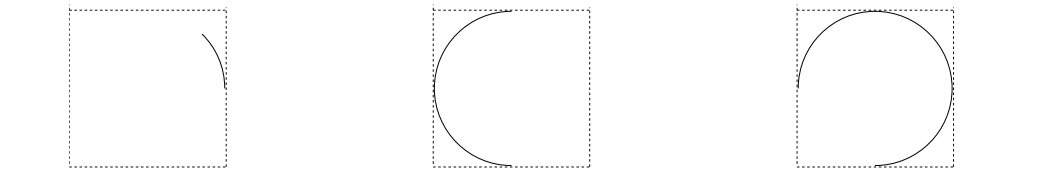
The GArc class

Arcs are implemented in many different ways in different graphics systems. The model used in **acm.graphics** is chosen—as always—to be consistent with the standard Java interpretation of arcs. The bad news is that this interpretation takes some getting used to. The good news is that you won't have to use **GArc**s very often.

In Java, an arc is defined in more or less the same way that an oval is. The usual form for calling the constructor is

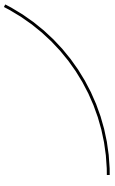
new GArc(*width***,** *height***,** *start***,** *sweep***)**

The *width* and *height* parameters are exactly the same as those for a **GOval** and specify the dimensions of the bounding rectangle. The *start* parameter specifies the angle at which the arc starts, and *sweep* parameter indicates the number of degrees through which it extends. As is true throughout Java, angles are measured in degrees counterclockwise from the +*x* axis. The effect of these parameters is illustrated by the following diagrams, which show where various arcs appear inside their bounding rectangle, which is indicated here by the dotted line but would not appear on the screen:



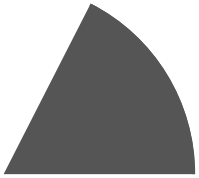
new GArc(60, 60, 0, 45) **new GArc(60, 60, 90, 180)** **new GArc(60, 60, -90, 270)**

Java defines filling for arcs in a way that at first seems a little odd. If you create an unfilled **GArc**, only the arc is shown. If you then fill it by calling **setFilled(true)**, Java fills the wedge that extends to the center of the circle as shown in the diagram on the right. Adopting the standard Java interpretation of arc filling has implications for the design. Most notably, the **contains** method for the **GArc** class returns a result that depends on whether the arc is filled. For an unfilled arc, containment implies that the arc point is actually on the arc, subject to the same interpretation of "closeness" as described for lines in the preceding section. For a filled arc, containment implies inclusion in the wedge.



Unfilled arc

The **GArc** class includes methods that enable clients to manipulate the angles defining the arc (**setStartAngle**, **getStartAngle**, **setSweepAngle**, and **getSweepAngle**) as well as methods to return the points at the beginning and end of the arc (**getStartPoint** and **getEndPoint**).



Filled arc

The `GImage` class

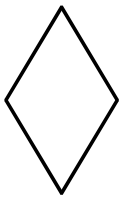
The **`GImage`** class is used to display an image stored in a recognized format for image data such as GIF (Graphics Interchange Format) or JPEG (Joint Photographic Experts Group). The way you use it is simply to put an image file in your workspace and then call

```
new GImage("filename")
```

where *filename* is the name of that file. This creates for you a **`GImage`** object that you can position on the screen just like any other **`GObject`**. A **`GImage`** object is both **`GResizable`** and **`GScalable`**, but not **`GFillable`**. Resizing or scaling an image has the effect of stretching or compressing the pixels in the image and is implemented by the standard graphics tools in Java.

The `GPolygon` class

The **`GPolygon`** class is the most complex of the shape classes, but it is not that hard to use. The primary difference in the design is that the constructor for a **`GPolygon`** does not create the whole thing. Rather, what you get is an empty **`GPolygon`** to which you can then add vertices or edges using the methods **`addVertex`**, **`addEdge`**, and **`addPolarEdge`**.



These methods are easiest to illustrate by example. The simplest method to explain is **`addVertex(x, y)`**, which adds a vertex at the point (**`x`**,**`y`**) relative to the location of the polygon. For example, the following code defines a diamond-shaped polygon in terms of its vertices, as shown in the diagram at the right:

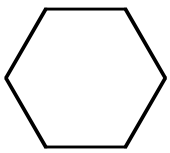
```
GPolygon diamond = new GPolygon();
diamond.addVertex(-22, 0);
diamond.addVertex(0, 36);
diamond.addVertex(22, 0);
diamond.addVertex(0, -36);
```

The diamond is drawn so that its center is at the point (0, 0) in the coordinate space of the polygon. Thus, if you were to add **`diamond`** to a **`GCanvas`** and set its location to the point (300, 200), the diamond would be centered at that location.

The **`addEdge(dx, dy)`** method is similar to **`addVertex`**, except that the parameters specify the displacement from the previous vertex to the current one. One could therefore draw the same diamond by making the following sequence of calls:

```
GPolygon diamond = new GPolygon();
diamond.addVertex(-22, 0);
diamond.addEdge(22, 36);
diamond.addEdge(22, -36);
diamond.addEdge(-22, -36);
diamond.addEdge(-22, 36);
```

Note that the first vertex must still be added using **`addVertex`**, but that subsequent ones can be defined by specifying the edge displacements. Moreover, the final edge is not explicitly necessary because the polygon is automatically closed before it is drawn.



Some polygons are easier to define by specifying vertices; others are more easily represented by edges. For many polygonal figures, however, it is even more convenient to express edges in polar coordinates. This mode of specification is supported in the **`GPolygon`** class by the method **`addPolarEdge`**, which is identical to **`addEdge`** except that its arguments are the length of the edge and its direction

expressed in degrees counterclockwise from the +x axis. This method makes it easy to create figures with more sophisticated structure, such as the centered hexagon generated by the following method (as shown on the right using 36 pixels as the value of **side**):

```
GPolygon createHexagon(double side) {
   GPolygon hex = new GPolygon();
   hex.addVertex(-side, 0);
   int angle = 60;
   for (int i = 0; i < 6; i++) {
      hex.addPolarEdge(side, angle);
      angle -= 60;
   }
   return hex;
}
```

The **GPolygon** class implements the **GFillable** and **GScalable** interfaces, but not **GResizable**. It also supports the method **rotate(theta)**, which rotates the polygon theta degrees counterclockwise around its origin.

8.3 Facilities available in the GraphicsProgram class

The preceding sections described the various shape classes available in the **GObject** hierarchy but did not discuss what you can do with those objects once you have them. Internally, graphical objects are added to a graphical canvas, which is implemented as the **GCanvas** class. Most of the time, however, you will not need to use **GCanvas** explicitly because the **GraphicsProgram** class you have already seen creates a **GCanvas** and fills the program window with it. It then intercepts all the messages you might send to a **GCanvas** and forwards them along. Those methods are listed in Figure 8-5.

Most of these methods are completely straightforward. The only one that deserves special mention is the method

```
GObject getElementAt(double x, double y)
```

which returns the object, if any, at the coordinates (**x**, **y**). If there is more than one such

FIGURE 8-5 Methods supported by the GraphicsProgram class (and by GCanvas)

void add(GObject gobj) Adds a graphical object to the canvas at its internally stored location.
void add(GObject gobj, double x, double y) Adds a graphical object to the canvas and sets its location as indicated.
void remove(GObject gobj) Removes the specified graphical object from the canvas.
void removeAll() Removes all graphical objects from the canvas.
int getWidth() Returns the width of the canvas in pixels.
int getHeight() Returns the height of the canvas in pixels.
int getElementCount() Returns the number of graphical objects contained on the canvas.
GObject getElement(int i) Returns the graphical object at the specified index, numbering from back to front.
GObject getElementAt(double x, double y) Returns the frontmost object containing the specified point, or **null** if no such object exists.

object, **getElementAt** returns the one that is in front according to the *z* ordering established by the **GObject** methods like **sendToBack** and **sendToFront**. But what if there is no object there? In that case, **getElementAt** returns the special constant **null**, which is a reserved word in Java that indicates that there is in fact no object corresponding to that value.

The **getWidth** and **getHeight** methods make it possible to determine the size of the graphics canvas. For example, the constructor

```
new GLine(0, getHeight(), getWidth(), 0)
```

creates a **GLine** object that extends from the lower left corner of the canvas to the upper right corner. These methods also make it possible to center graphical objects in the window because the *x*- and *y*-coordinates of the center of the window can be expressed as **getWidth() / 2** and **getHeight() / 2**, respectively.

8.4 Animation and interactivity

So far, the graphical programs you have seen in this text produce a fixed picture on the screen that does not change after you have put it together. Although the facilities in the graphics library allow you to create wonderful pictures in this way, the programs that you write won't be nearly as exciting until you learn how to change those pictures as the program runs. You might, for example, want to have particular graphical objects move across the screen or have them change their size and color. In computer graphics, the process of updating a displayed image so that it changes over time is called **animation.**

Time-based animation

The easiest way to animate graphical programs is to write a loop in your program that makes a small change to the image and then suspends the program for a very short time interval. A very simple example of this style of graphics is shown in Figure 8-6, which moves a **GLabel** across the screen from right to left, just the way the headline displays in New York's Times Square do.

The first part of the **run** method is very much like the "hello, world" program from Chapter 2. The first three lines are

```
GLabel label = new GLabel(HEADLINE);
label.setFont("Times-72");
add(label, getWidth(), (getHeight() + label.getAscent()) / 2);
```

which create a new **GLabel**, set its font to something appropriately large for a headline, and then add the label to the window. The coordinates, however, might seem a little surprising at first. The expression for the *y*-coordinate of the label—cumbersome as it seems to be—is actually the idiomatic expression for centering a label vertically on the screen. The baseline of the label needs to be at the vertical center of the window, but shifted further towards the bottom by half the vertical ascent of the characters. It is the expression for the *x*-coordinate that is more surprising, because positioning the label so that its *x*-coordinate is the width of the window means that the entire label is outside the visible part of the window.

If the display stayed that way, the **TimesSquare** program would simply display an empty canvas. The rest of the **run** method, however, has the effect of changing the display so that the label gradually moves across the window. The code looks like this:

FIGURE 8-6 Program to animate a text label

```
/*
 * File: TimesSquare.java
 * ----------------------
 * This program displays the text of the string HEADLINE
 * on the screen in an animated way that moves it across
 * the display from left-to-right.
 */

import acm.graphics.*;
import acm.program.*;

public class TimesSquare extends GraphicsProgram {

/** Runs the program */
   public void run() {
      GLabel label = new GLabel(HEADLINE);
      label.setFont("Times-72");
      add(label, getWidth(), (getHeight() + label.getAscent()) / 2);
      while (label.getX() + label.getWidth() > 0) {
         label.move(-DELTA_X, 0);
         pause(PAUSE_TIME);
      }
   }

/** The number of pixels to shift the label on each cycle */
   private static final double DELTA_X = 2.0;

/** The number of milliseconds to pause on each cycle */
   private static final int PAUSE_TIME = 20;

/** The string to use as the value of the label */
   private static final String HEADLINE =
     "When in the course of human events it becomes necessary " +
     "for one people to dissolve the political bands which " +
     "connected them with another . . .";
}
```

```
while (label.getX() + label.getWidth() > 0) {
   label.move(-DELTA_X, 0);
   pause(PAUSE_TIME);
}
```

For the moment, forget about the condition in the while loop and focus on the two statements that make up the body. The first is

```
label.move(-DELTA_X, 0);
```

which adjusts the *x*-coordinate of the label to move it **DELTA_X** pixels to the left, where **DELTA_X** is defined to have the value 2. Thus, on each iteration of the **while** loop, the label moves two pixels leftward. The second line is

```
pause(PAUSE_TIME);
```

which causes the program to suspend its operation for the specified number of milliseconds. Sleeping for 20 milliseconds means that the display will be updated 50 times a second, which is well below the threshold at which the eye perceives motion as continuous.

The call to **pause** in the **TimesSquare** program is necessary to achieve the effect of animation. Computers today run so quickly that the label would instantly zip off the left side of the window if you didn't slow things down enough to bring the operation back to human speed. The fact that the label will eventually—even at this slower speed—move off the left edge makes it easier to understand the condition in the **while** loop. The expression **label.getX() + label.getWidth()** indicates the *x*-coordinate of the rightmost part of the label. As long as that is still greater than 0, some part of the label will be visible on the screen. When the last character shifts off the left edge, the **while** loop stops.

You can change the speed of an animation in either of two ways. First, you can change the distance that the position of each object is updated on each time step. For example, setting **DELTA_X** to 4 would double the apparent speed of the motion, although it would probably also begin to appear more jerky. The second approach is that you can change the delay time. Intuitively, you could double the speed of the display by halving the delay time on each cycle, although that strategy has limits in practice. The delay that a program experiences when it calls **pause** is not guaranteed to be precise. Moreover, there is always some overhead associated with pausing a program, and it is impossible to run the animation with a cycle time that is less than the overhead. Thus, there is always some point at which reducing the value of **PAUSE_TIME** in this example will no longer have any noticeable effect.

The general pattern for programs that animate the display can be expressed in the following pseudocode form:

```
public void run() {
    Initialize the graphical objects in the display.
    while (as long as you want to run the animation) {
        Update the properties of the objects that you wish to animate.
        pause(delay time);
}
```

The **TimesSquare** program from Figure 8-6 fits this paradigm, as does the code in Figure 8-7, which displays a square whose color changes randomly once a second.

Responding to mouse events

In the preceding section, you learned a simple strategy through which you can animate a graphical program so that the display changes with time. If you want to write programs that operate like the computing applications you use every day, you will also need to learn how to make those programs interactive by having them respond to actions taken by the user. In some ways, the **ConsoleProgram** examples you have been writing all along are interactive. Most of these programs display messages on the console and wait for a response, which is certainly one style of interaction. In those examples, however, the user was asked for input only at certain well-defined points in the program's execution history, such as when the program called a console method like **readInt** and waited for a response. This style of interaction is called **synchronous,** because it occurs in sync with the program operation. Modern user interfaces, however, are **asynchronous** in that they allow the user to intercede at any point, typically by using the mouse or the keyboard to trigger a particular action.

FIGURE 8-7 Program to display a square whose color changes once a second

```
/*
 * File: ColorChangingSquare.java
 * ------------------------------
 * This program puts up a square in the center of the window
 * and randomly changes its color every second.
 */

import acm.graphics.*;
import acm.program.*;
import acm.util.*;

public class ColorChangingSquare extends GraphicsProgram {

/** Runs the program */
   public void run() {
      GRect square = new GRect(SQUARE_SIZE, SQUARE_SIZE);
      square.setFilled(true);
      add(square, (getWidth() - SQUARE_SIZE) / 2,
                  (getHeight() - SQUARE_SIZE) / 2);
      while (true) {
         square.setColor(rgen.nextColor());
         pause(PAUSE_TIME);
      }
   }

/** Size of the square in pixels */
   private static final double SQUARE_SIZE = 100;

/** Pause time in milliseconds */
   private static final double PAUSE_TIME = 1000;

/** Random number generator */
   private RandomGenerator rgen = new RandomGenerator();
}
```

Events that occur asynchronously with respect to the program operation—mouse clicks, key strokes, and the like—are represented using a structure called an **event.** When an event occurs, the response is always the invocation of a method in some object that is waiting to hear about that event. Such an object is called a **listener.** You'll have the opportunity to see several more examples of listeners later in the text, but it is wonderful to be able to use the mouse, even at this early stage in programming.

In Java, objects that listen for user-interface events do so by implementing the methods in a specific listener interface defined in the package **`java.awt.event`**. This package contains several interfaces that allow clients to respond to mouse clicks, button presses, keystrokes, changes in component sizes, and many more. For the moment, this text will concentrate only on mouse events, which involve two types of listeners. The first such interface is **`MouseListener`**, which makes it possible to listen for user actions that primarily involve clicking the mouse; the second is **`MouseMotionListener`**, which is used to track the mouse as it moves. The reason for separating these two types of listeners is that mouse motions generate many more events than click-type actions do. If your application is driven only by mouse clicks and never has to track the mouse itself,

that application will run more efficiently if it does not have to respond to frequent motion events in which it has no interest.

The listener methods that can be called in response to mouse events are outlined in Figure 8-8. The **GraphicsProgram** class declares itself to be both a **MouseListener** and a **MouseMotionListener** by defining implementations for each of these listener methods; those implementations, however, do nothing at all. For example, the standard definition of **mouseClicked** is simply

```
public void mouseClicked(MouseEvent e) {
   /* Empty */
}
```

Thus, unless you take some action to the contrary, a **GraphicsProgram** will simply ignore mouse clicks, along with all the other mouse events. If you, however, want to change the behavior for a particular event, all you need to do is add a new definition for that method. This new definition supersedes the original definition and will be called instead of the empty one. Any methods that you don't override continue to do what they did by default, which was nothing. Thus, you only have to override the methods you need.

Each of the methods listed in Figure 8-8 takes as its argument an object of type **MouseEvent**, which is a class defined as part of Java's standard window system toolkit. Like the listener interfaces themselves, the **MouseEvent** class lives in the package **java.awt.event**, which means that you need to add the **import** statement

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

to the beginning of any program that uses mouse events.

The **MouseEvent** class includes a rich set of methods for designing sophisticated user interfaces. This text, however, uses only two of those methods. Given a **MouseEvent** stored in a variable named **e**, you can determine the location of the mouse by calling **e.getX()** and **e.getY()**. Being able to detect the location at which a mouse event

FIGURE 8-8 Standard listener methods for responding to mouse events

MouseListener interface

void mousePressed(MouseEvent e) Called whenever the mouse button is pressed.
void mouseReleased(MouseEvent e) Called whenever the mouse button is released.
void mouseClicked(MouseEvent e) Called when the mouse button is "clicked" (pressed and released within a short span of time).
void mouseEntered(MouseEvent e) Called whenever the mouse enters the canvas.
void mouseExited(MouseEvent e) Called whenever the mouse exits the canvas.

MouseMotionListener interface

void mouseMoved(MouseEvent e) Called whenever the mouse is moved with the button up.
void mouseDragged(MouseEvent e) Called whenever the mouse is moved with the button down.

occurred enables you to write many interesting mouse-driven applications, as illustrated by the examples in the two subsections that follow.

Dragging objects on the canvas

The first example of using the mouse is a program that puts up two objects on the screen and lets the user drag them around, which appears as Figure 8-9.

FIGURE 8-9 Program to drag objects on the canvas

```
import java.awt.*;
import java.awt.event.*;
import acm.graphics.*;
import acm.program.*;

/** This class displays a mouse-draggable rectangle and oval */
public class DragObjects extends GraphicsProgram {

/** Runs the program */
   public void run() {
      GRect rect = new GRect(100, 100, 150, 100);
      rect.setFilled(true);
      rect.setColor(Color.RED);
      add(rect);
      GOval oval = new GOval(300, 115, 100, 70);
      oval.setFilled(true);
      oval.setColor(Color.GREEN);
      add(oval);
      addMouseListeners();
   }

/** Called on mouse press to record the coordinates of the click */
   public void mousePressed(MouseEvent e) {
      lastX = e.getX();
      lastY = e.getY();
      gobj = getElementAt(lastX, lastY);
   }

/** Called on mouse drag to reposition the object */
   public void mouseDragged(MouseEvent e) {
      if (gobj != null) {
         gobj.move(e.getX() - lastX, e.getY() - lastY);
         lastX = e.getX();
         lastY = e.getY();
      }
   }

/** Called on mouse click to move this object to the front */
   public void mouseClicked(MouseEvent e) {
      if (gobj != null) gobj.sendToFront();
   }

/* Instance variables */
   private GObject gobj;          /* The object being dragged   */
   private double lastX;          /* The last mouse X position  */
   private double lastY;          /* The last mouse Y position  */
}
```

The first part of the **run** method in Figure 8-9 should be entirely familiar. It simply creates two graphical objects—a red rectangle and a green oval—and then adds them to the canvas. It's the line

```
addMouseListeners();
```

that introduces something new. The **addMouseListeners** method in **GraphicsProgram** hides a modest amount of internal complexity, but has the effect of enabling the program to watch the mouse and respond to its events. As described in the preceding section, you specify the actions to take when those events occur by overriding listener methods in your program. These listener methods are then called automatically when those events occur.

The first listener method defined in Figure 8-9 is **mousePressed**, which is called when the mouse button first goes down. That method looks like this:

```
public void mousePressed(MouseEvent e) {
   lastX = e.getX();
   lastY = e.getY();
   gobj = getElementAt(lastX, lastY);
}
```

The first two statements simply record the *x* and *y* coordinates of the mouse in the variables **lastX** and **lastY**. As you can see from the program, these variables are declared as instance variables for the object and not as local variables of the sort that you have seen in most methods. It turns out that you will need these values later when you try to drag the object. Because local variables disappear when a method returns, you have to hold onto these in the object itself.

The last statement in **mousePressed** checks to see what object on the canvas contains the current mouse position. Here, it is important to recognize that there are two possibilities. First, you could be pressing the mouse button on top of an object, which means that you want to start dragging it. Second, you could be pressing the mouse button somewhere else on the canvas at which there is no object to drag. The **getElementAt** method looks at the specified position and returns the object it finds there. If there is more than one object covering that space, it chooses the one that is in front of the others in the *z*-axis ordering. If there are no objects at that location, **getElementAt** returns the special value **null**, which signifies an object that does not exist. The other methods will check for this value to determine whether there is an object to drag.

The **mouseDragged** method consists of the following code:

```
public void mouseDragged(MouseEvent e) {
   if (gobj != null) {
      gobj.move(e.getX() - lastX, e.getY() - lastY);
      lastX = e.getX();
      lastY = e.getY();
   }
}
```

The **if** statement simply checks to see whether there is an object to drag. If the value of **gobj** is **null**, there is nothing to drag, and the rest of the method is simply skipped. If there is an object, you need to move it by some distance in each direction. That distance does not depend on where the mouse is in an absolute sense but rather in how far it has moved from where you last took stock of its position. Thus, the arguments to the **move**

method are—for both the *x* and *y* components—the location where the mouse is now minus where it used to be. Once you have moved it, you then have to record the mouse coordinates again so that the location will update correctly on the next **mouseDragged** call.

The final listener method specified in Figure 8-9 is **mouseClicked**, which looks like this:

```
public void mouseClicked(MouseEvent e) {
   if (gobj != null) gobj.sendToFront();
}
```

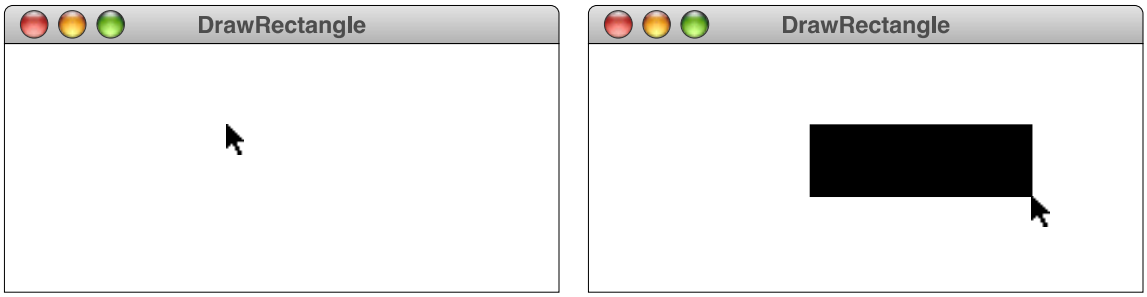
The intent of this method is to allow the user to move an object to the front by clicking on it, thereby bringing it out from under the other objects on the canvas. The code is almost readable as an English sentence

> *If there is a current object, send it to the front of the canvas display.*

The only question you might have is how the variable **gobj**, which holds the current object, is initialized in this case. The answer depends on the fact that the **mouseClicked** event is always generated in conjunction with a **mousePressed** and a **mouseReleased** event, both of which precede the **mouseClicked** event. The **gobj** variable is therefore set by **mousePressed**, just as if you were going to drag it.

A simple drawing program

Most of you have used drawing programs that allow you to draw shapes on a canvas. In a typical drawing program, you create a rectangle by pressing the mouse at one corner and then dragging it to the opposite corner. For example, if the user pressed the mouse at the location in the left diagram and then dragged it to the position in which you see it in the right diagram, the program would create the rectangle shown:



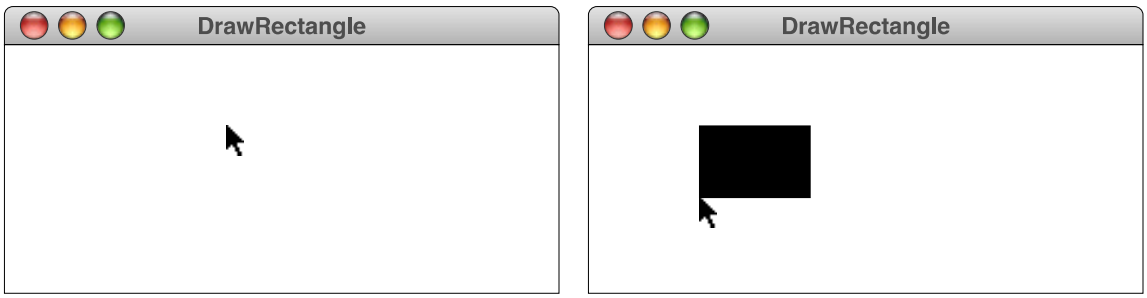
To make it easy for the user to see the shape as it is drawn, drawing programs typically update the coordinates of the rectangle on each call to **mouseDragged**. When the user releases the mouse button, the rectangle is complete and stays where it is. The user can then go back and add more rectangles to the screen by clicking and dragging in the same way.

Figure 8-10 shows a simple **GraphicsProgram** that allows the user to create rectangles by clicking and dragging as illustrated in the example. The only part of the program that might not be immediately obvious is in the calculation of the coordinates for the rectangle in the **mouseDragged** method. The example above illustrates only one possible dragging operation, in which the initial mouse click is in the upper left corner of the **GRect**. The program, however, has to work just as well if the user drags the mouse in some other direction besides to the right and down. For example, it should also be possible to draw a

FIGURE 8-10 Program to draw rectangles on the canvas

```
/*
 * File: DrawRectangle.java
 * ------------------------
 * This program allows users to create rectangles on the canvas
 * by clicking and dragging with the mouse.
 */

import java.awt.event.*;
import acm.graphics.*;
import acm.program.*;

/** This class allows users to drag rectangles on the canvas */
public class DrawRectangle extends GraphicsProgram {

/** Runs the program */
   public void run() {
      addMouseListeners();
   }

/** Called on mouse press to record the starting coordinates */
   public void mousePressed(MouseEvent e) {
      startX = e.getX();
      startY = e.getY();
      currentRect = new GRect(startX, startY, 0, 0);
      currentRect.setFilled(true);
      add(currentRect);
   }

/** Called on mouse drag to reshape the current rectangle */
   public void mouseDragged(MouseEvent e) {
      double x = Math.min(e.getX(), startX);
      double y = Math.min(e.getY(), startY);
      double width = Math.abs(e.getX() - startX);
      double height = Math.abs(e.getY() - startY);
      currentRect.setBounds(x, y, width, height);
   }

/* Private state */
   private GRect currentRect;     /* The current rectangle          */
   private double startX;         /* The initial mouse X position   */
   private double startY;         /* The initial mouse Y position   */
}
```

rectangle by dragging to the left, as shown in the following illustration:

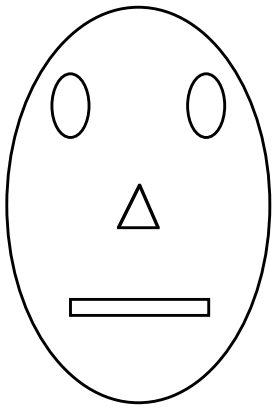


In this example, the origin of the **GRect** is no longer at the original position of the mouse. The width and height of a **GRect** cannot be negative, which means that the program needs to change the origin of the **GRect** whenever the mouse is moved to the left or upward.

8.5 Creating compound objects

The class from the **acm.graphics** hierarchy that has not yet been discussed is the **GCompound** class, which turns out to be extraordinarily handy. The **GCompound** class makes it possible to collect several **GObject**s together into a single unit, which is itself a **GObject**. This ability extends the notion of abstraction as discussed for methods into the domain of graphical objects. In much the same way that methods allow you to assemble many statements into a single unit, the **GCompound** class allows you to put together graphical objects into a single unit that has its own integrity as an graphical object.

To understand how the **GCompound** class works, it is easiest to start with a simple example. Imagine that you wanted to assemble the following face on the canvas:



For the most part, this figure would be easy to create. All you need to do is create a new **GOval** for the head, two **GOval**s for the eyes, a **GRect** for the mouth, and a **GPolygon** for the nose. If you put each of these objects on the canvas individually, however, it will be hard to manipulate it as a unit. Suppose, for example, that you wanted to move the face around as a unit. Doing so would require moving every piece independently. It would be better simply to tell the entire face to move.

The code in Figure 8-11 uses the **GCompound** class to do just that. Figure 8-11 contains the code for a **GFace** class that extends **GCompound** to create a face object that contains the necessary components. These components are created and then added in the appropriate places by the first entry in the **GFace** class. This entry is not quite the same as a traditional method in the following respects:

- Its name matches the name of the class.
- It does not specify a return type

In Java, such methods are called **constructors** and are used to create new instances of a particular class. Here, the **GFace** constructor makes it possible to create a new **GFace** object by specifying the width and height in a statement like

```
GFace face = new GFace(200, 300);
```

FIGURE 8-11 A "graphical face" class defined using GCompound

```
/*
 * File: GFace.java
 * ----------------
 * This file defines a compound GFace class.
 */

import acm.graphics.*;

public class GFace extends GCompound {

/** Construct a new GFace object with the specified dimensions. */
   public GFace(double width, double height) {
      head = new GOval(width, height);
      leftEye = new GOval(EYE_WIDTH * width, EYE_HEIGHT * height);
      rightEye = new GOval(EYE_WIDTH * width, EYE_HEIGHT * height);
      nose = createNose(NOSE_WIDTH * width, NOSE_HEIGHT * height);
      mouth = new GRect(MOUTH_WIDTH * width, MOUTH_HEIGHT * height);
      add(head, 0, 0);
      add(leftEye, 0.25 * width - EYE_WIDTH * width / 2,
                   0.25 * height - EYE_HEIGHT * height / 2);
      add(rightEye, 0.75 * width - EYE_WIDTH * width / 2,
                    0.25 * height - EYE_HEIGHT * height / 2);
      add(nose, 0.50 * width, 0.50 * height);
      add(mouth, 0.50 * width - MOUTH_WIDTH * width / 2,
                 0.75 * height - MOUTH_HEIGHT * height / 2);
   }

/** Creates a triangle for the nose */
   private GPolygon createNose(double width, double height) {
      GPolygon poly = new GPolygon();
      poly.addVertex(0, -height / 2);
      poly.addVertex(width / 2, height / 2);
      poly.addVertex(-width / 2, height / 2);
      return poly;
   }

/* Constants specifying feature size as a fraction of the head size */

   private static final double EYE_WIDTH    = 0.15;
   private static final double EYE_HEIGHT   = 0.15;
   private static final double NOSE_WIDTH   = 0.15;
   private static final double NOSE_HEIGHT  = 0.10;
   private static final double MOUTH_WIDTH  = 0.50;
   private static final double MOUTH_HEIGHT = 0.03;

/* Instance variables */

   private GOval head;
   private GOval leftEye, rightEye;
   private GPolygon nose;
   private GRect mouth;
}
```

which constructs a new **GFace** whose width is 200 and whose height is 300 pixels. Inside the constructor, the sizes of each component are expressed in terms of the width and height of the face as a whole, so that small faces have, for example, eyes of the appropriately small size. Once the constructor has created each of the features, it then adds each of them in turn, using calls like

```
add(nose, 0.50 * width, 0.50 * height);
```

which adds the nose object half the width and half the height from the upper left corner of the face, otherwise known as the center.

The first thing to recognize, however, is that this **add** call is not adding the objects to the canvas, but rather to the compound. The **add** method you've been using up to now is the one defined in **GraphicsProgram**, which does add objects to the **GCanvas** that it installs in the window. This version of the **add** method is defined in **GCompound**; the **GFace** class extends **GCompound** and therefore uses its add method. Adding things to a **GCompound** is intuitively analogous to the same operation at the canvas level except that the objects stay together as a unit. You can move everything in the **GCompound** simply by moving the **GCompound** itself.

Another important observation to make is that each **GCompound** has its own coordinate system. In that coordinate system, the point (0, 0) is the upper left corner of the compound and not of the entire canvas. This makes it possible for you to define a **GCompound** without having to know where it is going to appear on the canvas. When the components of a **GCompound** actually get drawn, they are shifted to the appropriate position.

Figure 8-12 contains the code necessary to create a **GFace** and allow the user to drag it as a unit, just as in the ObjectDrag example from the preceding section.

8.6 Principles of good object-oriented design

In contrast to the individual classes described in Chapter 6, the graphical facilities described in this chapter are part of a package—**acm.graphics**—that includes a large set of classes and interfaces. The contents of the package as a whole are summarized in the **javadoc** documentation, which also includes detailed descriptions of each class and method in the package. The package-level documentation appears in Figure 8-13; clicking on the various links shown on that page will take you to the more detailed documentation for the classes that comprise the **acm.graphics** package.

No matter at what level you are working—individual methods, classes that offer a suite of those methods, or packages that provide a set of classes—you need to think carefully about design. If you were programming entirely for your own amusement, design issues might seem relatively unimportant. But that situation rarely applies in the computing industry. Programs are developed cooperatively, with programmers on different parts of a large project agreeing to share a common set of stylistic conventions and to coordinate the ways in which their independent pieces fit together. In the absence of cooperative agreements, the programming process would descend—as it all too often does—into chaos. The important question, therefore, is how should one design methods, classes, and packages for *other* programmers to use. What are the principles that underlie well-chosen designs?

FIGURE 8-12 A program to drag a GFace object

```
/*
 * File: DragFace.java
 * -------------------
 * This program creates a GFace object and allows the user to drag
 * it around the canvas.
 */

import java.awt.event.*;

import acm.graphics.*;
import acm.program.*;

public class DragFace extends GraphicsProgram {

/** Width of the face */
   private static final double FACE_WIDTH = 200;

/** Height of the face */
   private static final double FACE_HEIGHT = 300;

/** Runs the program */
   public void run() {
      GFace face = new GFace(FACE_WIDTH, FACE_HEIGHT);
      double x = (getWidth() - FACE_WIDTH) / 2;
      double y = (getHeight() - FACE_HEIGHT) / 2;
      add(face, x, y);
      addMouseListeners();
   }

/** Called on mouse press to record the coordinates of the click */
   public void mousePressed(MouseEvent e) {
      lastX = e.getX();
      lastY = e.getY();
      gobj = getElementAt(lastX, lastY);
   }

/** Called on mouse drag to reposition the object */
   public void mouseDragged(MouseEvent e) {
      if (gobj != null) {
         gobj.move(e.getX() - lastX, e.getY() - lastY);
         lastX = e.getX();
         lastY = e.getY();
      }
   }

/** Called on mouse click to move this object to the front */
   public void mouseClicked(MouseEvent e) {
      if (gobj != null) gobj.sendToFront();
   }

/* Private state */
   private GObject gobj;          /* The object being dragged   */
   private double lastX;          /* The last mouse X position  */
   private double lastY;          /* The last mouse Y position  */
}
```

FIGURE 8-13 Documentation page for the acm.graphics package

Overview **Package** Class Tree Index Help

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

Package acm.graphics

This package provides a set of classes that support the creation of simple, object-oriented graphical displays.

See:
Description

Interface Summary	
GContainer	Defines the functionality of an object that can serve as the parent of a `GObject`.
GFillable	Specifies the characteristics of a graphical object that supports filling.
GResizable	Specifies the characteristics of a graphical object that supports the `setSize` and `setBounds` methods.
GScalable	Specifies the characteristics of a graphical object that supports the `scale` method.

Class Summary	
G3DRect	The `G3DRect` class is a graphical object whose appearance consists of a three-dimensional rectangle.
GArc	The `GArc` class is a graphical object whose appearance consists of an arc.
GCanvas	The `GCanvas` class is a lightweight component that also serves as a container for graphical objects.
GCanvasMenuBar	The `ConsoleMenuBar` class provides a standard menu bar for frames containing a console.
GCompound	This class defines a graphical object that consists of a collection of other graphical objects.
GDimension	This class is a double-precision version of the `Dimension` class in `java.awt`.
GImage	The `GImage` class is a graphical object whose appearance is defined by an image.
GLabel	The `GLabel` class is a graphical object whose appearance consists of a text string.
GLine	The `GLine` class is a graphical object whose appearance consists of a line segment.
GObject	This class is the common superclass of all graphical objects that can be displayed on a `GCanvas`.
GOval	The `GOval` class is a graphical object whose appearance consists of an oval.
GPen	The `GPen` class simulates a pen drawing on a canvas.
GPoint	This class is a double-precision version of the `Point` class in `java.awt`.
GPolygon	The `GPolygon` class is a graphical object whose appearance consists of a polygon.
GRect	The `GRect` class is a graphical object whose appearance consists of a rectangular box.
GRectangle	This class is a double-precision version of the `Rectangle` class in `java.awt`.
GRoundRect	The `GRoundRect` class is a graphical object whose appearance consists of a rounded rectangle.
GTurtle	The `GTurtle` class simulates a turtle moving on a canvas.

Developing a solid understanding of those principles requires you to understand that clients and implementors look at the facilities provided by a method, class, or package from different perspectives. Clients want to know what operations are available and are unconcerned about the details of the implementation. For implementors, the details of how those operations work are the fundamental issue.

It is, however, essential for implementors to keep the needs of clients in mind. If you are trying to design effective resources for others, you need to balance several criteria. Those criteria are described here at the level of an individual class, but these same considerations also apply at the lower level of individual methods and the higher level of complete packages:

- *Unified*. A class should define a consistent abstraction with a clear unifying theme. If a class does not fit within that theme, it should not be part of the class.
- *Simple*. The class design should try to simplify things for the client. To the extent that the underlying implementation is itself complex, the class must seek to hide that complexity.
- *Sufficient*. For clients to use a class, it must provide sufficient functionality to meet their needs. If some critical operation is missing from a class, clients may decide to abandon it and develop their own tools. As important as simplicity is, the designer must avoid simplifying a class to the point that it becomes useless.
- *General*. A well-designed class should be flexible enough to meet the needs of many different clients. A class that performs a narrowly defined set of operations for one client is not nearly as useful as one that can be used in many different situations.
- *Stable*. The methods defined in a class should continue to have precisely the same structure and effect, even as the package that includes it evolves. Making changes in the behavior of a class forces clients to change their programs, which reduces the utility of that class.

The sections that follow discuss each of these criteria in detail.

The importance of a unifying theme

Unity gives strength.

— Aesop, *The Bundle of Sticks,* 6th century BCE

A central feature of a well-designed class is that it presents a unified and consistent abstraction. In part, this criterion implies that the methods within a class should be chosen so that they reflect a coherent theme. For example, the **`Math`** class consists of mathematical methods, the **`ConsoleProgram`** class provides methods that make it easy to converse with a user typing at a console, and the various classes in the **`acm.graphics`** package provide methods for arranging graphical objects on a canvas. Each method exported by these classes fits the purpose of that class. For example, you would not expect to find **`sqrt`** in the **`GObject`** class, even though graphical applications will often call **`sqrt`** to compute the length of a diagonal line. The **`sqrt`** method fits much more naturally into the framework of the **`Math`** class.

The principle of a unifying theme also influences the design of the methods within a class. The methods within a class should behave in as consistent a way as possible. Differences in the ways its methods operate make using a class much harder for the client. For example, all the methods in the **`acm.graphics`** package use coordinates specified in pixels and angles specified in degrees. If the implementor of the class had decided to toss in a method that required a different unit of measurement, clients would have to remember what units to use for each method.

Simplicity and the principle of information hiding

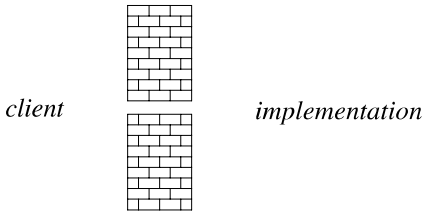
Embrace simplicity.

— Lao-tzu, *The Way of Lao-tzu,* ca. 550 BCE

Because a primary goal of using classes is to reduce the complexity of the programming process, it makes sense that simplicity is a desirable criterion in the design of a class. In general, a class should be as easy to use as possible. The underlying implementation may perform extremely intricate operations, but the client should nonetheless be able to think about those operations in a simple, more abstract way.

As noted earlier in this chapter, the documentation that you need to use Java classes and packages is usually presented on the web. If, for example, you want to know how to use the **`Math`** class, you go to the web page that contains its javadoc description. If that documentation is well-designed, it tells you precisely the information that you need to know as a client, but no more. For clients, getting too much information can be as bad as getting too little, because additional detail is likely to make the class more difficult to understand. Often, the real value of a class lies not in the information it *reveals* but rather in the information it *hides*.

When you design a class, you should try to protect the client from as many of the complicating details of the implementation as possible. In that respect, it is perhaps best to think of a class not primarily as a communication channel between the client and the implementation, but instead as a wall that divides them.



Like the wall that divided the lovers Pyramus and Thisbe in Greek mythology, the wall representing a class has a small chink that allows the client and the implementation to communicate. The main purpose of the wall, however, is to keep the two sides apart. Because it forms the border between the abstraction perspectives on each side, that wall is often called an **abstraction boundary**. Ideally, all the complexity involved in the realization of a class lies on the implementation side of the wall. The design is successful if it keeps that complexity away from the client side. Keeping details confined to the implementation domain is called **information hiding**.

The principle of information hiding has important practical implications for class design. When you write a class, you should be sure you don't reveal details of the implementation, even in the commentary. Especially if you are writing a class and an implementation at the same time, you may be tempted to document in your class all the clever ideas you used to write the implementation. Try to resist that temptation. The class is written for the benefit of the client and should contain only what the client needs to know.

Similarly, you should design the methods in a class so that they are as simple as possible. If you can reduce the number of arguments or find a way to eliminate confusing special cases, it will be easier for the client to understand how to use those methods. Moreover, it is usually good practice to limit the total number of methods exported by class, so that the client does not become lost in a mass of methods, unable to make sense of the whole.

Meeting the needs of your clients

Everything should be as simple as possible, but no simpler.

— attributed to Albert Einstein

Simplicity is only part of the story. You can easily make a class simple just by throwing away any parts of it that are hard or complicated. There is a good chance you will also make the class useless. Sometimes clients need to perform tasks that have some inherent complexity. Denying your clients the tools they require just to make the class simpler is

not an effective strategy. Your class must provide sufficient functionality to serve the clients' needs. Learning to strike the right balance between simplicity and completeness in class design is one of the fundamental challenges in programming.

In many cases, the clients of a class are concerned not only with whether a particular method is available but also with the efficiency of the underlying implementation. For example, if a programmer is developing a system for air-traffic control and needs to call methods provided by a class, those methods must return the correct answer quickly. Late answers may be just as devastating as wrong answers.

For the most part, efficiency is a concern for the implementation rather than the abstract design. Even so, you will often find it valuable to think about implementation strategies while you are designing the class itself. Suppose, for example, that you are faced with a choice of two designs. If you determine that one of them would be much easier to implement efficiently, it makes sense—assuming there are no compelling reasons to the contrary—to choose that design.

The advantages of general tools

Give us the tools and we will finish the job.

—Winston Churchill, radio address, 1941

A class that is perfectly adapted to a particular client's needs may not be useful to others. A good class abstraction serves the needs of many different clients. To do so, it must be general enough to solve a wide range of problems and not be limited to one highly specific purpose. By choosing a design that offers your clients flexibility in how they use the abstraction, you can create classes that are widely used.

The desire to ensure that a class remains general has an important practical implication. When you are writing a program, you will often discover that you need a particular tool. If you decide that the tool is important enough to go into a class, you then need to change your mode of thought. When you design the class for that class, you have to forget about the application that caused you to want the tool in the first place and instead design such a tool for the most general possible audience.

The value of stability

People change and forget to tell each other. Too bad—causes so many mistakes.

—Lillian Hellman, *Toys in the Attic,* 1959

Class and package designs have another property that makes them critically important to programming: they tend to be stable over long periods of time. Stable classes can dramatically simplify the problem of maintaining large programming systems by establishing clear boundaries of responsibility. As long as the client perspective on a class does not change, both implementors and clients are free to make changes on their own side of the abstraction boundary.

For example, suppose that you are the implementor of the **`Math`** class. In the course of your work, you discover a clever new algorithm for calculating the **`sqrt`** method that cuts in half the time required to calculate a square root. If you can say to your clients that you have a new implementation of **`sqrt`** that works just as it did before, only faster, they will probably be pleased. If, on the other hand, you were to say that the name of the method had changed or that its use involved certain new restrictions, your clients would be justifiably annoyed. To use your "improved" implementation of square root, they would

be forced to change their programs. Changing programs is a time-consuming, error-prone activity, and many clients would happily give up the extra efficiency for the convenience of being able to leave their programs alone.

Summary

In this chapter, you have had the chance to explore the **acm.graphics** package in more detail and to develop an appreciation of the entire package as an integrated collection of tools. Along the way, you have also had the opportunity to think holistically about the design of the graphics package and the assumptions, conventions, and metaphors on which the package is based.

Important points introduced in this chapter include:

- The set of assumptions, conventions, and metaphors that underlie the design of a package represent its conceptual *model*. Before you can use a package effectively, you must take the time to understand the model on which it is based.
- An essential part of the model for the **acm.graphics** package is the coordinate system that it uses. The **acm.graphics** package follows the conventions of the standard graphics packages in Java by specifying coordinates in *pixels* and placing the *origin* in the upper left corner of a canvas. This coordinate system is different from the Cartesian plane used in high-school geometry classes, which has its origin in the lower left.
- The foundation of the **acm.graphics** package is the **GObject** class, which is the common superclass of all objects that can be displayed on a canvas. The **GObject** class itself is an *abstract class,* which means that there are no objects whose primary class is **GObject**. When you create a graphical image on a canvas, the classes that you actually use are the subclasses of **GObject** called *shape classes:* **GArc**, **GImage**, **GLabel**, **GLine**, **GOval**, **GPolygon**, **GRect**, **GRoundRect**, and **G3DRect**.
- Each of the shape classes inherits a set of methods from **GObject** that all graphical objects share. In addition, each shape class includes additional methods that define its particular behavior. Several of the shape classes also share common behavior by virtue of implementing one or more of the interfaces **GFillable**, **GResizable**, and **GScalable**.
- To display a graphical object, you need to add it to a **GCanvas**, which serves as the background for a "collage" of **GObject**s. In most cases, that **GCanvas** will be provided automatically as part of a **GraphicsProgram**, although you can also create your own **GCanvas** objects and use them independently of the **acm.program** package.
- You can make your program respond to mouse events by implementing one or more of the following *listener methods:* **mousePressed**, **mouseReleased**, **mouseClicked**, **mouseMoved**, **mouseDragged**, **mouseEntered**, and **mouseExited**. In most cases, it makes sense to use the **GCanvas** as the source of these events; you can enable the event listeners in the **GCanvas** by calling **addMouseListeners** as part of the **run** method.
- You can use the **GCompound** type to assemble individual objects into larger structures that you can then manipulate as a unit.
- A well-designed package must be *unified*, *simple*, *sufficient*, *general*, and *stable*. Since these criteria sometimes conflict with each other, you must learn to strike an appropriate balance as you design your programs.