The ACM Java Task Force

Tutorial

Eric Roberts (chair), Stanford University, `eroberts@cs.stanford.edu`
Kim Bruce, Pomona College, `kim@cs.pomona.edu`
Robb Cutler, The Harker School, `robbc@harker.org`
James Cross, Auburn University, `cross@eng.auburn.edu`
Scott Grissom, Grand Valley State University, `grissom@gvsu.edu`
Karl Klee, Alfred State College, `kleekj@alfredstate.edu`
Susan Rodger, Duke University, `rodger@cs.duke.edu`
Fran Trees, Drew University, `fran@ftrees.com`
Ian Utting, University of Kent, `i.a.utting@kent.ac.uk`
Frank Yellin, Google, Inc., `fyellin@gmail.com`

August 25, 2006

This work is supported by grants from the ACM Education Board, the SIGCSE Special Projects Fund, and the National Science Foundation (grant DUE-0411905).

The ACM Java Task Force Tutorial

(August 20, 2006)

Table of Contents

Chapter 1. Introduction to the JTF packages 1
Chapter 2. Using the **`acm.graphics`** package 11
Chapter 3. Animation and interactivity 44
Chapter 4. Graphical user interfaces 65

Chapter 1
Introduction to the JTF Packages

Since its release in 1995, the Java programming language has become increasingly important as a vehicle for teaching introductory programming. Java has many significant advantages over earlier teaching languages and enables students to write exciting programs that capture their interest and imagination. At the same time, Java is far more sophisticated than languages that have traditionally filled that role, such as BASIC and Pascal. The complexity that comes with that sophistication can be a significant barrier to both teachers and students as they try to understand the structure of the language.

In early 2004, the ACM created the Java Task Force (JTF) and assigned it the following charge:

> To review the Java language, APIs, and tools from the perspective of introductory computing education and to develop a stable collection of pedagogical resources that will make it easier to teach Java to first-year computing students without having those students overwhelmed by its complexity.

After two preliminary releases in February 2005 and February 2006, the JTF released its final report in July 2006.

This tutorial is designed to give instructors a gentle introduction into how to use the JTF materials in the context of an introductory programming course. As a tutorial, this document does not attempt to cover every detail of the varstructurery to defend the decisions that went into the overall design. The complete description of the packages can be found in the online **javadoc**; a review of the design is available in the Java Task Force Rationale document.

1.1 Getting started

In their classic textbook *The C Programming Language*, Brian Kernighan and Dennis Ritchie offered the following observation at the beginning of Chapter 1:

> The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:
>
> *Print the words*
>
> **`hello, world`**
>
> This is the big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where the output went. With these mechanical details mastered, everything else is comparatively easy.

Even though C has given way to Java and our expectations of what constitutes an interesting first program have changed, the wisdom of getting the mechanical details out of the way remains as applicable as ever. During the review process for the JTF packages, the most common questions we received were from users who were unsure how to compile and run programs that made use of *any* libraries beyond the standard classes supplied with the Java distribution. Once that hurdle was cleared, using the JTF packages seemed to be straightforward.

Downloading the "hello, world" programs

The moral of the story, therefore, is that it is essential to get a simple program working before you move on to more complex examples. Taking our inspiration from Kernighan and Ritchie, our first example will be a simple program that displays the "hello, world" message. You can download the code for this program—along with a copy of the **acm.jar** library and a couple of more advanced versions of the program—from the following web address:

http://jtf.acm.org/downloads/Hello.zip

Please take a moment to download the **Hello.zip** file and unzip it on your computer. If you are reading this tutorial online, you should be able simply to click on the link. Most browser programs will automatically download and unzip the code with no further interaction on your part. When you have done so, you should see a directory named **Hello** that contains the following four files: **HelloConsole.java**, **HelloDialog.java**, **HelloGraphics.java**, and **acm.jar**. The first three are Java program files, and the last one is a library file that contains the compiled code for the JTF packages.

Once you have successfully downloaded these files, your next step is to compile and run the **HelloConsole** program, which is the simplest of the examples. The code for this version of the program appears in Figure 1-1. If you are using the traditional command-line interface supplied with Sun's Java Development Kit, this process requires two steps. The first step is to compile the **HelloConsole.java** file by issuing the command

```
javac -classpath acm.jar HelloConsole.java
```

You can then run the program by invoking the command

```
java -cp .:acm.jar HelloConsole
```

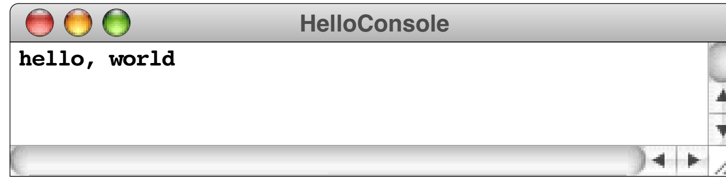
On Windows platforms, the colon in the classpath must be replaced with a semicolon (**;**).

Figure 1-1. Simple program to display "hello, world" on the screen

```
/*
 * File: HelloConsole.java
 * -----------------------
 * This program displays the message "hello, world" and is inspired
 * by the first program "The C Programming Language" by Brian
 * Kernighan and Dennis Ritchie.  This version displays its message
 * using a console window.
 */

import acm.program.*;

public class HelloConsole extends ConsoleProgram {

   public void run() {
      println("hello, world");
   }

/* Standard Java entry point */
/* This method can be eliminated in most Java environments */
   public static void main(String[] args) {
      new HelloConsole().start(args);
   }
}
```

Note that the **acm.jar** file must be specified as part of both the compilation and execution steps.

If everything is working, the computer should pop up a console window that looks something like this:



If you are using one of the many Integrated Development Environments (IDEs) available for Java—such as Microsoft Visual Studio™, Metrowerks CodeWarrior™, or the open-source Eclipse system—you will need to ensure that the **acm.jar** file is included as part of the **classpath**, which is the list of directories and JAR files that Java searches to find class definitions that are not part of the source files. The procedure for doing so varies from system to system. Please check the documentation for your own IDE to see how one goes about specifying the classpath.

Eliminating the static main method

As soon as you have the **HelloConsole** program working, it is useful to try one additional experiment. If you look at the code in Figure 1-1, you will see that there is a **main** method at the end of the class definition. As the comment indicates, it is possible to eliminate this method in many Java environments, but not all. Open the **HelloConsole.java** file in an editor, delete the **main** method and its associated comments, and then see if you can still compile and run the program. If so, you will be able to write shorter programs that will be much easier for novices to understand. If not, you will need to tell your students to include a standardized **main** method in their programs that always looks like

```
public static void main(String[] args) {
   new MainClass().start(args);
}
```

where *MainClass* is the name of the main class.

The examples available on the JTF web site include a static **main** method to ensure that these programs will run in as many environments as possible. For clarity of presentation, however, the programs in the remainder of this tutorial eliminate the **main** method to focus attention on the more substantive parts of these examples. The programs instead begin with a **run** method, which is called after the runtime libraries have created the necessary windows and arranged them on the screen.

Making the programs a little more interesting

Although getting the **HelloConsole** program working is a good first step, it isn't a particularly exciting example. If nothing else, the program seems terribly out of date. While printing a message on the console may have been a reasonable example in the 1970s, students today are used to much more sophisticated programs, with fancy graphics and interactive dialogs. Surely a modern object-oriented language like Java can do better than duplicating the kind of program students wrote a generation ago.

That's where the other two programs that you downloaded as part of the **Hello.zip** file come in. If you compile and run the **HelloDialog.java** program in precisely the same way that you ran **HelloConsole.java**, the **"hello, world"** message won't appear

in a console window. In fact, the program doesn't create a program frame at all. Instead the prorgam pops up an interactive dialog box that looks something like this, although the precise format of the display will vary depending on what operating system you are using and what "look and feel" it defines for Java applications:

The **HelloGraphics.java** file uses the facilities of the **acm.graphics** package to display the message in large, friendly letters across the window:

The code for each of these programs is similar in certain respects to that used in **HelloConsole**. The **HelloDialog** program is almost exactly the same. Other than changes in the comments, the only difference is the header line for the class, which now looks like this:

```
public class HelloDialog extends DialogProgram
```

The body of the class is exactly the same. The only difference—which is sufficient to cause the change in behavior—is that this version extends **DialogProgram** instead of **ConsoleProgram**.

The code for **HelloGraphics** appears in Figure 1-2. The details of the program are not important at this point, and will be covered in Chapter 2. Even so, the basic idea is likely to be clear, even if you could not have generated the code as it stands. The first line creates a **GLabel** object with the message text, the second line gives it a larger font, and the last three lines take care of adding the label so that it is centered in the window. What is important to notice is that the **HelloGraphics** class extends **GraphicsProgram**, which is yet another category of program. These three classes—**ConsoleProgram**, **DialogProgram**, and **GraphicsProgram**—are the building blocks for Java applications built using the **acm.program** package, which is introduced in the following section.

1.2 The Program class hierarchy

Each of the applications contained in the **Hello.zip** file represents a simple, paradigmatic example of one of three classes defined in the package called **acm.program**. The classes for the various versions of the **"hello, world"** program—taken together with the classes defined in the **acm.program** package—form the class hierarchy shown in

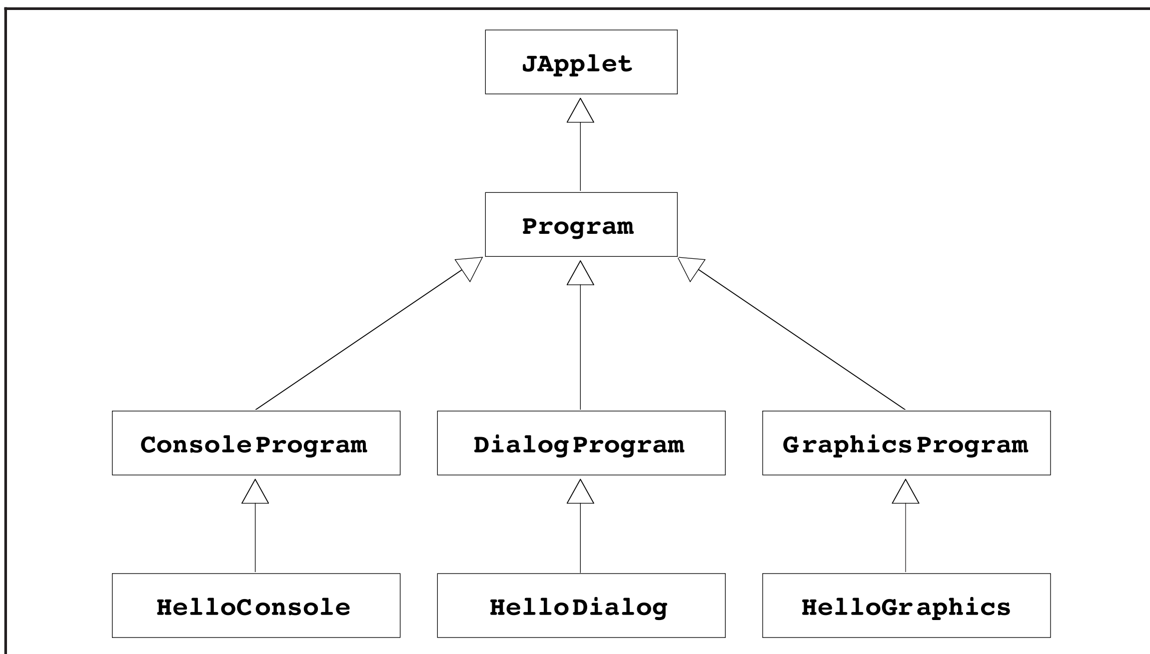
Figure 1-2. Program to display “hello, world” graphically

```
/*
 * File: HelloGraphics.java
 * ------------------------
 * This program displays the message "hello, world" and is inspired
 * by the first program "The C Programming Language" by Brian
 * Kernighan and Dennis Ritchie.  This version displays the message
 * graphically.
 */

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

public class HelloGraphics extends GraphicsProgram {

   public void run() {
      GLabel label = new GLabel("hello, world");
      label.setFont("SansSerif-100");
      double x = (getWidth() - label.getWidth()) / 2;
      double y = (getHeight() + label.getAscent()) / 2;
      add(label, x, y);
   }
}
```

Figure 1-3. As the diagram shows, each of the example applications is a subclass of a specific program type: **`HelloConsole`** is a subclass of **`ConsoleProgram`**, **`HelloDialog`** is a subclass of **`DialogProgram`**, and **`HelloGraphics`** is a subclass of **`GraphicsProgram`**. Each of these classes is a subclass of a more general **`Program`** class, which is in turn a subclass of Swing’s **`JApplet`** class.

The program class hierarchy in Figure 1-3 provides a straightforward introduction to the ideas of subclassing and inheritance that students seem to find compelling. After all,

Figure 1-3. The `Program` class hierarchy



the word *program* has the intuitive meaning of some kind of application that can be executed on a computer. It is clear, moreover, that there are different kinds of programs, which provides a motivation for subclassing. A particular program running on a machine—`HelloConsole`, for example—is an instance of a `ConsoleProgram`, but it is also clearly an instance of a more generic `Program` class. This inheritance structure therefore exemplifies the *is-a* relationship between a class and its superclass in a seemingly natural way.

Using the `Program` class offers several advantages beyond the pedagogical one of serving as an archetype for class hierarchies:

- The `Program` class encourages students to beyond the procedural paradigm implied by `public static void main` into a more object-oriented style in which all methods are executed in the context of an object.
- Because the `Program` class is a subclass of `JApplet`, a `Program` can run equally well as applications and web-based applets.
- The `Program` class includes several features to make instruction easier, such as menu bars with standard **File** and **Edit** menus.

Behavior common to all `Program` classes

Sitting as it does at the root of the program hierarchy, the `Program` class defines the behavior that all of its subclasses share, and it is therefore important to understand something of how the `Program` class works before moving on to its individual subclasses. The most important feature of the `Program` class is that it standardizes the process of program startup in a way that unifies the traditionally disparate models of applications and applets. The idea is that a program should work the same way if you run it as an application or as an applet in the context of a web browser. To achieve this goal, the `Program` class automatically executes several of the operations that a browser performs when running an applet. Thus, no matter whether you run a program as a standalone application or view it as an applet running inside a web browser, the startup process consists of the following steps:

1. Create a new instance of the main class.
2. Create a frame in which to run the program.
3. Install components in the frame as required by the program subclass. A `ConsoleProgram`, for example, installs a console in the frame; a `GraphicsProgram`, by contrast, installs a graphical canvas.
4. Call the program's `init` method to perform any application-specific initializion.
5. Ensure that the frame layout is up to date by calling `validate`.
6. Call the `run` method using a new thread of control.

For the most part, these steps are entirely automatic, and the student doesn't need to be aware of the details. From the student's perspective, the essential step in getting a program running is defining a `run` method that contains the code, as illustrated by each of the three implementations of the `"hello, world"` program. The code for each `run` method depends to some extent on the specific subclass, so that the code for a `ConsoleProgram` will include method calls for interacting with a console while a `GraphicsProgram` will include calls for displaying graphical objects on a canvas. Despite these differences, the startup operations for every program subclass remain the same.

The sections that follow offer a quick introduction to the **ConsoleProgram**, **DialogProgram**, and **GraphicsProgram** classes. For a more complete description of the methods available in each class, please see the **javadoc** documentation.

The ConsoleProgram class

A **ConsoleProgram** begins by creating a console window and installing it in the program frame. The code for the **ConsoleProgram** then communicates with the user through calls to methods that are passed on to the console, such as the

```
println("hello, world");
```

you saw in the the **HelloConsole** example.

Although the **ConsoleProgram** class exports a much larger set of input and output methods as defined by the **IOModel** interface in the **acm.io** package, you can easily get started using only the methods listed in Figure 1-4. This set includes the familiar **print** and **println** methods provided by the classes in the **java.io** package along with a set of methods like **readInt**, **readDouble**, and **readLine** for reading input of various types.

The code for the **Add2Console** program in Figure 1-5 offers an extremely simple illustration of how to use the **ConsoleProgram** class: a program that reads in two integers from the user and prints their sum. A sample run of the **Add2Console** program might look like this:

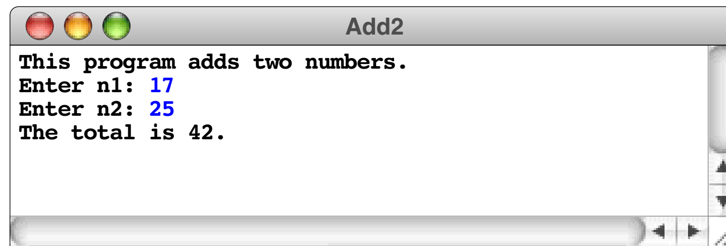


Figure 1-4. Useful methods in the ConsoleProgram class

Method	Description
Output methods	
void print(*any* value)	Writes the value to the console with no terminating newline.
void println(*any* value)	Writes the value to the console followed by a newline.
void println()	Returns the cursor on the console to the beginning of the next line.
void showErrorMessage(String msg)	Displays an error message on the console, which appears in red.
Input methods	
String readLine(String prompt)	Reads and returns a line of text from the console without the terminating newline.
int readInt(String prompt)	Reads and returns an **int** value from the console.
double readDouble(String prompt)	Reads and returns an **double** value from the console.
Additional methods	
void setFont(Font font) *or* **void setFont(String str)**	Sets the overall font for the console, which may also be specified as a string.
void clear()	Clears the console screen.

Figure 1-5. Program to add two numbers entered on the console

```
/*
 * File: Add2Console.java
 * ----------------------
 * This program adds two numbers and prints their sum.  Because
 * this version is a ConsoleProgram, the input and output appear
 * on the console.
 */

import acm.program.*;

public class Add2Console extends ConsoleProgram {

   public void run() {
      println("This program adds two numbers.");
      int n1 = readInt("Enter n1: ");
      int n2 = readInt("Enter n2: ");
      int total = n1 + n2;
      println("The total is " + total + ".");
   }

}
```

Given that the **ConsoleProgram** class derives its inspiration from the classical paradigm of text-based, synchronous interaction, using this model is generally quite straightforward for those who learned programming in that domain. If you use the **Add2Console** program as a template, you can easily write new versions of any of the traditional programs from the days of Pascal and C when consoles represented the primary style of interaction. Even though the underlying paradigm is familiar, there are nonetheless a few important features of the **ConsoleProgram** class that are worth highlighting:

- The **ConsoleProgram** class makes it possible for students to tell the difference between user input, program output, and error messages. By default, user input is shown in blue, and error messages appear in red. One of the principal advantages of making these distinctions is that the pattern of user interaction is obvious when the program is displayed on a classroom projection screen.
- The **setFont** method makes it possible to change the font used by the console. For classroom projection, it is useful to specify a larger font size using a line something like this:

```
setFont("Monospaced-bold-18");
```

- The **ConsoleProgram** class automatically installs a menu bar with standard **File** and **Edit** menus. These menus include facilities for printing or saving the console log, reading from an input script, and the standard cut/copy/paste operations.

Even though console-based interaction is comfortable for most teachers today, students who have grown up with modern graphical applications tend to find this style of interaction primitive and uninspiring. To avoid having them lose interest in computing altogether, it is important to introduce graphics and interactivity early in an introductory course. At the same time, the **ConsoleProgram** class has its place. Many instructors find that it is easier to illustrate how simple programming constructs work in a console-based environment because there aren't as many complicating details to distract the student from the essential character of the construct in question. Similarly, console-based

programs often provide a good framework for teaching problem-solving because students must focus on finding solution strategies instead of implementing the many graphical bells and whistles that can get in the way of fundamental ideas.

The `DialogProgram` class

The **DialogProgram** class is similar to **ConsoleProgram** except for one important detail. Instead of forwarding its input and output methods to a console, a **DialogProgram** implements those operations by popping up dialog boxes that deliver or request the same information. The **print** and **println** methods pop up a message dialog that contains the output line; the input methods like **readInt** and **readLine** pop up an input dialog that requests the information from the user. This difference is illustrated by the **Add2Dialog** program in Figure 1-6. Except for the fact that this version extends **DialogProgram** instead of **ConsoleProgram**, the code is identical to the **Add2Console** program from Figure 1-5. Running the **Add2Dialog** program produces a series of dialog boxes as shown in Figure 1-7.

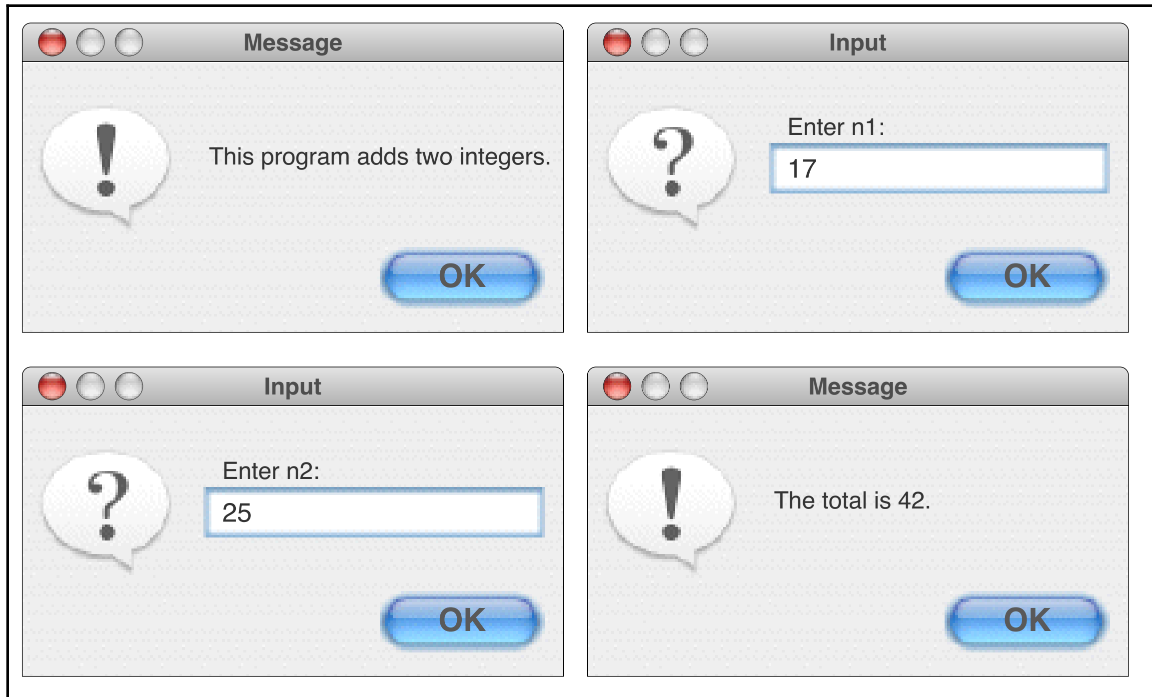
The **DialogProgram** class turns out to be valuable more for pedagogical than practical reasons. The advantage of having both the **ConsoleProgram** and the **DialogProgram** classes is that it emphasizes the nature of inheritance. The **Add2Console** program and the **add2Dialog** program have exactly the same run method. The difference in behavior comes from the fact that the two programs inherit operations from different **Program** subclasses.

The similarity between the **ConsoleProgram** and **DialogProgram** classes underscores an important principle of object-oriented design. The input and output operations for these classes are specified by the **IOModel** interface in the **acm.io** package. This **IOModel** interface defines a set of methods that includes—along with several others—the methods **print**, **println**, **readInt**, **readDouble**, **readLine**, and **showErrorMessage** described in Figure 1-4. Because the code for the **Program** class declares that it implements **IOModel**, each of its subclasses will share that set of methods. Those methods, or course, are implemented in different ways, but they invariably have the same name and parameter structure.

Figure 1-6. Program to add two numbers entered via popup dialogs

```
/*
 * File: Add2Dialog.java
 * ---------------------
 * This program adds two numbers and prints their sum.  Because
 * this version is a DialogProgram, the input and output appear
 * as popup dialogs.
 */

import acm.program.*;

public class Add2Dialog extends DialogProgram {

   public void run() {
      println("This program adds two numbers.");
      int n1 = readInt("Enter n1: ");
      int n2 = readInt("Enter n2: ");
      int total = n1 + n2;
      println("The total is " + total + ".");
   }
}
```

Figure 1-7. Dialogs produced by the `Add2Dialog` program



The `GraphicsProgram` class

The **`GraphicsProgram`** class is by far the most interesting of the classes in the program hierarchy. As the **`HelloGraphics`** program made clear, a **`GraphicsProgram`** can present information on the screen in a way that holds at least a little more excitement than is possible with a **`ConsoleProgram`**. That example, however, merely scratched the surface of what is possible using the **`GraphicsProgram`** class. Because that class is far too powerful to cover in a single subsection, learning how to use the facilities provided by **`GraphicsProgram`** and the **`acm.graphics`** package on which it is based will take up the next two chapters in this tutorial. To unlock the power of graphics, read on.

Chapter 2
Using the **acm.graphics** Package

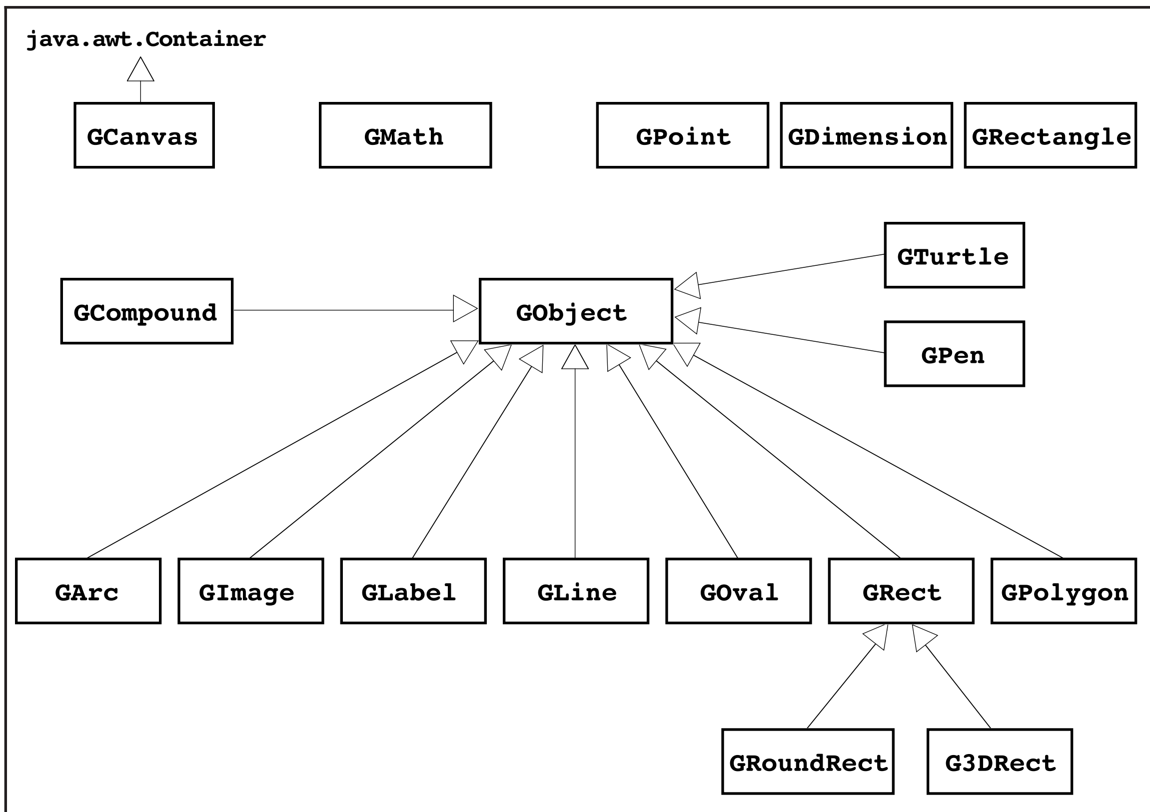
The **HelloGraphics** example in Chapter 1 offers a simple example of how to write graphical programs, but does not explain the details behind the methods it contains. The purpose of this chapter is to give you a working knowledge of the facilities available in the **acm.graphics** package and how to use them effectively.

The class structure of **acm.graphics** package appears in Figure 2-1. Most of the classes in the package are subclasses of the abstract class **GObject** at the center of the diagram. Conceptually, **GObject** represents the universal class of graphical objects that can be displayed. When you use **acm.graphics**, you assemble a picture by constructing various **GObject**s and adding them to a **GCanvas** at the appropriate locations. The following section describes the general model in more detail and the later sections offer a closer look at the individual classes in the package.

2.1 The acm.graphics model

When you create a picture using the **acm.graphics** package, you do so by arranging graphical objects at various positions on a background called a **canvas.** The underlying model is similar to that of a collage in which an artist creates a composition by taking various objects and assembling them on a background canvas. In the world of the collage artist, those objects might be geometrical shapes, words clipped from newspapers, lines formed from bits of string, or images taken from magazines. In the **acm.graphics** package, there are counterparts for each of these graphical objects.

Figure 2-1. Class diagram for the acm.graphics package



The "felt board" metaphor

Another metaphor that often helps students understand the conceptual model of the **`acm.graphics`** package is that of a felt board—the sort one might find in an elementary school classroom. A child creates pictures by taking shapes of colored felt and sticking them onto a large felt board that serves as the background canvas for the picture as a whole. The pieces stay where the child puts them because felt fibers interlock tightly enough for the pieces to stick together. The left side of Figure 2-2 shows a physical felt board with a red rectangle and a green oval attached. The right side of the figure is the virtual equivalent in the **`acm.graphics`** world. To create the picture, you would need to create two graphical objects—a red rectangle and a green oval—and add them to the graphical canvas that forms the background.

The code for the **`FeltBoard`** example appears in Figure 2-3. Even though you have not yet had a chance to learn the details of the various classes and methods used in the program, the overall framework should nonetheless make sense. The program first creates a rectangle, indicates that it should be filled rather than outlined, colors it red, and adds it to the canvas. It then uses almost the same operations to add a green oval. Because the oval is added after the rectangle, it appears to be in front, obscuring part of the rectangle underneath. This behavior, of course, is exactly what would happen with the physical felt board. Moreover, if you were to take the oval away by calling

```
remove(oval);
```

the parts of the underlying rectangle that had previously been obscured would reappear.

In this tutorial, the order in which objects are layered on the canvas will be called the **stacking order**. (In more mathematical descriptions, this ordering is often called ***z*-ordering**, because the *z*-axis is the one that projects outward from the screen.) Whenever a new object is added to a canvas, it appears at the front of the stack. Graphical objects are always drawn from back to front so that the frontmost objects overwrite those that are further back.

The coordinate system

The **`acm.graphics`** package uses the same basic coordinate system that traditional Java programs do. Coordinate values are expressed in terms of **pixels,** which are the individual dots that cover the face of the screen. Each pixel in a graphics window is identified by its x and y coordinates, with x values increasing as you move rightward across the window and y values increasing as you move down from the top. The point $(0, 0)$—which is called the **origin**—is in the upper left corner of the window. This

Figure 2-2. Physical felt board and its virtual equivalent

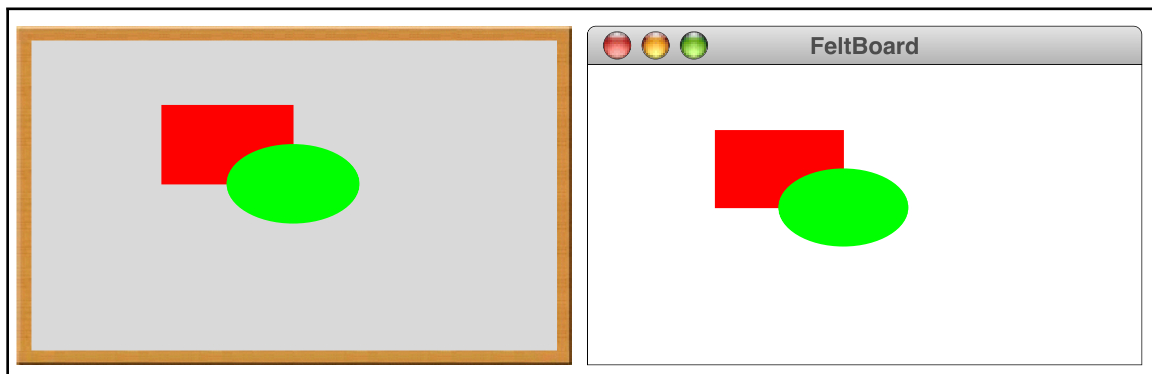
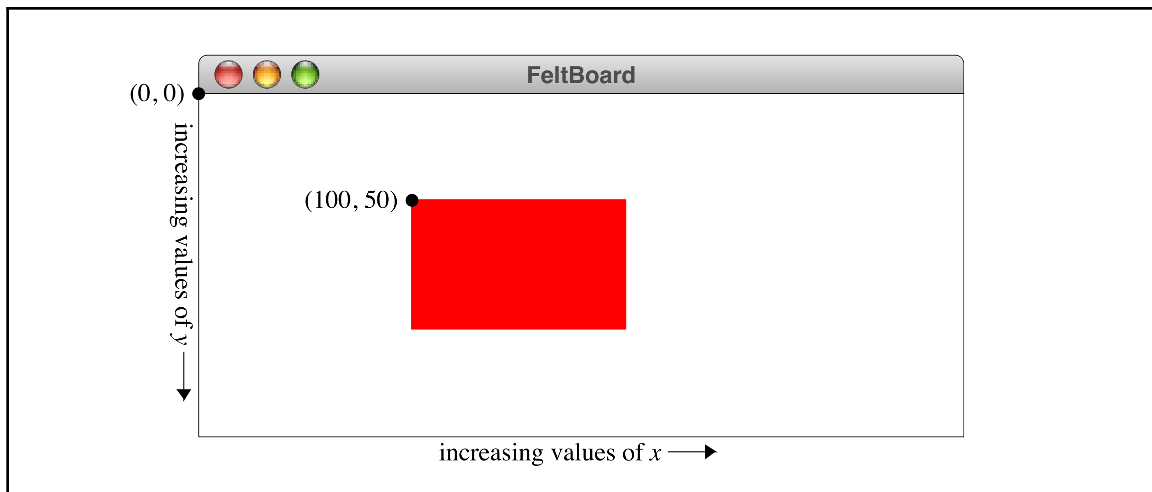


Figure 2-3. Code for the felt board example

```
/*
 * File: FeltBoard.java
 * --------------------
 * This program offers a simple example of the acm.graphics package
 * that draws a red rectangle and a green oval.  The dimensions of
 * the rectangle are chosen so that its sides are in proportion to
 * the "golden ratio" thought by the Greeks to represent the most
 * aesthetically pleasing geometry.
 */

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

public class FeltBoard extends GraphicsProgram {

/** Runs the program */
   public void run() {
      GRect rect = new GRect(100, 50, 100, 100 / PHI);
      rect.setFilled(true);
      rect.setColor(Color.RED);
      add(rect);
      GOval oval = new GOval(150, 50 + 50 / PHI, 100, 100 / PHI);
      oval.setFilled(true);
      oval.setColor(Color.GREEN);
      add(oval);
   }

/** Constant representing the golden ratio */
   public static final double PHI = 1.618;

}
```

coordinate system is illustrated by the diagram in Figure 2-4, which shows only the red rectangle from the **FeltBoard.java** program. The location of that rectangle is (100, 50), which means that its upper left corner is 100 pixels to the right and 50 pixels down from the origin of the graphics window.

Figure 2-4. The Java coordinate system



The only difference between the coordinate systems used in the **acm.graphics** package and Java's **Graphics** class is that the **acm.graphics** package uses **double**s to represent coordinate values instead of **int**s. This change makes it easier to create figures whose locations and dimensions are produced by mathematical calculations in which the results are typically not whole numbers. As a simple example, the dimensions of the red rectangle in Figure 2-4 are proportional to the *golden ratio*, which Greek mathematicians believed gave rise to the most pleasing aesthetic effect. The golden ratio is approximately equal to 1.618 and is usually denoted in mathematics by the symbol ϕ. Because the **acm.graphics** package uses **double**s to specify coordinates and dimensions, the code to generate the rectangle looks like this:

```
new GRect(100, 50, 100, 100 / PHI)
```

In the integer-based Java model, it would be necessary to include explicit code to convert the height parameter to an **int**. In addition to adding complexity to the code, forcing students to convert coordinates to integers can introduce rounding errors that distort the geometry of the displayed figures.

Judging from the experience of the instructors who tested the **acm.graphics** package while it was in development, the change from **int**s to **double**s causes no confusion but instead represents an important conceptual simplification. The only aspect of Java's coordinate system that students find problematic is the fact that the origin is in a different place from what they know from traditional Cartesian geometry. Fortunately, it doesn't take too long to become familiar with the Java model.

The GPoint, GDimension, and GRectangle classes

Although it is usually possible to specify individual values for coordinate values, it is often convenient to encapsulate an *x* and a *y* coordinate as a point, a *width* and a *height* value as a composite indication of the dimensions of an object, or all four values as the bounding rectangle for a figure. Because the coordinates are stored as **double**s in the **acm.graphics** package, using Java's integer-based **Point**, **Dimension**, and **Rectangle** classes would entail a loss of precision. To avoid this problem the **acm.graphics** package exports the classes **GPoint**, **GDimension**, and **GRectangle**, which have the same semantics as their standard counterparts except for the fact that their coordinates are **double**s.

As an example, the declaration

```
GDimension goldenSize = new GDimension(100, 100 / PHI);
```

introduces the variable **goldenSize** and initializes it to a **GDimension** object whose internal **width** and **height** fields are the dimensions of the golden rectangle illustrated in the earlier example. The advantage of encapsulating these values into objects is that they can then be passed from one method to another using a single variable.

The GMath class

Computing the coordinates of a graphical design can sometimes require the use of simple trigonometric functions. Although functions like **sin** and **cos** are defined in Java's standard **Math** class, students find them confusing in graphical applications because of inconsistencies in the way angles are represented. In Java's graphics libraries, angles are measured in degrees; in the **Math** class, angles must be given in radians. To minimize the confusion associated with this inconsistency of representation, the **acm.graphics** package includes a class called **GMath**, which exports the methods shown in Figure 2-5. Most of these methods are simply degree-based versions of the standard trigonometric functions, but the **distance**, **angle**, and **round** methods are also worth noting.

Figure 2-5. Static methods in the `GMath` class

Trigonometric methods in degrees

`static double sinDegrees(double angle)` Returns the trigonometric sine of an angle measured in degrees.
`static double cosDegrees(double angle)` Returns the trigonometric cosine of an angle measured in degrees.
`static double tanDegrees(double angle)` Returns the trigonometric tangent of an angle measured in degrees.
`static double toDegrees(double radians)` Converts an angle from radians to degrees.
`static double toRadians(double degrees)` Converts an angle from degrees to radians.

Conversion methods for polar coordinates

`double distance(double x, double y)` Returns the distance from the origin to the point (**`x`**, **`y`**).
`double distance(double x0, double y0, double x1, double y1)` Returns the distance between the points (**`x0`**, **`y0`**) and (**`x1`**, **`y1`**).
`double angle(double x, double y)` Returns the angle between the origin and the point (**`x`**, **`y`**), measured in degrees.

Convenience method for rounding to an integer

`static int round(double x)` Rounds a **`double`** to the nearest **`int`** (rather than to a **`long`** as in the **`Math`** class).

2.2 The `GCanvas` class

In the **`acm.graphics`** model, pictures are created by adding graphical objects—each of which is an instance of the **`GObject`** class hierarchy described in section 2.3—to a background canvas. That background—the analogue of the felt board in the physical world—is provided by the **`GCanvas`** class. The **`GCanvas`** class is a lightweight component and can be added to any Java container in either the **`java.awt`** or **`javax.swing`** packages, which makes it possible to use the graphics facilities in any Java application. For the most part, however, students in introductory courses won't use the **`GCanvas`** class directly but will instead use the **`GraphicsProgram`** class, which automatically creates a **`GCanvas`** and installs it in the program window, as illustrated in several preceding examples. The **`GraphicsProgram`** class forwards operations such as **`add`** and **`remove`** to the embedded **`GCanvas`** so that students don't need to be aware of the underlying implementation details.

The most important methods supported by the **`GCanvas`** class are shown in Figure 2-6. Many of these methods are concerned with adding and removing graphical objects. These methods are easy to understand, particularly if you keep in mind that a **`GCanvas`** is conceptually a container for **`GObject`** values. The container metaphor explains the functionality provided by the **`add`**, **`remove`**, and **`removeAll`** methods in Figure 2-6, which are analogous to the identically named methods in **`JComponent`** and **`Container`**.

The **`add`** method comes in two forms, one that preserves the internal location of the graphical object and one that takes an explicit *x* and *y* coordinate. Each method has its uses, and it is convenient to have both available. The first is useful particularly when the constructor for the **`GObject`** specifies the location, as it does, for example, in the case of the **`GRect`** class. If you wanted to create a 100 x 60 rectangle at the point (75, 50), you could do so by writing the following statement:

```
add(new GRect(75, 50, 100, 60));
```

Figure 2-6. Useful methods in the `GCanvas` class

Constructor
`new GCanvas()` Creates a new `GCanvas` containing no graphical objects.
Methods to add and remove graphical objects from a canvas
`void add(GObject gobj)` Adds a graphical object to the canvas at its internally stored location.
`void add(GObject gobj, double x, double y)` *or* **`add(GObject gobj, GPoint pt)`** Adds a graphical object to the canvas at the specified location.
`void remove(GObject gobj)` Removes the specified graphical object from the canvas.
`void removeAll()` Removes all graphical objects and components from the canvas.
Method to find the graphical object at a particular location
`GObject getElementAt(double x, double y)` *or* **`getElementAt(GPoint pt)`** Returns the topmost object containing the specified point, or `null` if no such object exists.
Useful methods inherited from superclasses
`int getWidth()` Return the width of the canvas, in pixels.
`int getHeight()` Return the height of the canvas, in pixels.
`void setBackground(Color bg)` Changes the background color of the canvas.

The second form is particularly useful when you want to choose the coordinates of the object in a way that depends on other properties of the object. For example, the following code taken from the `HelloGraphics` example in Chapter 1 centers a `GLabel` object in the window:

```
GLabel label = new GLabel("hello, world");
double x = (getWidth() - label.getWidth()) / 2;
double y = (getHeight() + label.getAscent()) / 2;
add(label, x, y);
```

Because the placement of the label depends on its dimensions, it is necessary to create the label first and then add it to a particular location on the canvas.

The `GCanvas` method `getElement(x, y)` returns the graphical object on the canvas that includes the point (`x`, `y`). If there is more than one such object, `getElement` returns the one that is in front of the others in the stacking order; if there is no object at that position, `getElement` returns `null`. This method is useful, for example, if you need to select an object using the mouse. Chapter 3 includes several examples of this technique.

Several of the most useful methods in the `GCanvas` class are those that are inherited from its superclasses in Java's component hierarchy. For example, if you need to determine how big the graphical canvas is, you can call the methods `getWidth` and `getHeight`. Thus, if you wanted to define a `GPoint` variable to mark the center of the canvas, you could do so with the following declaration:

```
GPoint center = new GPoint(getWidth() / 2.0, getHeight() / 2.0);
```

You can also change the background color by calling `setBackground(bg)`, where `bg` is the new background color for the canvas.

2.3 The `GObject` class

The **`GObject`** class represents the universe of graphical objects that can be displayed on a **`GCanvas`**. The **`GObject`** class itself is abstract, which means that programs never create instances of the **`GObject`** class directly. Instead, programs create instances of one of the **`GObject`** subclasses that represent specific graphical objects such as rectangles, ovals, and lines. The most important such classes are the ones that appear at the bottom of the class diagram from Figure 2-1, which are collectively called the **shape classes.** The shape classes are described in detail in section 2.4. Before going into those details, however, it makes sense to begin by describing the characteristics that are common to the **`GObject`** class as a whole.

Methods common to all `GObject` subclasses

All **`GObject`**s—no matter what type of graphical object they represent—share a set of common properties. For example, all graphical objects have a *location,* which is the *x* and *y* coordinates at which that object is drawn. Similarly, all graphical objects have a *size,* which is the width and height of the rectangle that includes the entire object. Other properties common to all **`GObject`**s include their color and how the objects are arranged in terms of their stacking order. Each of these properties is controlled by methods defined at the **`GObject`** level. The most important such methods are summarized in Figure 2-7.

Determining the location and size of a `GObject`

The first several methods in Figure 2-7 make it possible to determine the location and size of any **`GObject`**. The **`getX`**, **`getY`**, **`getWidth`**, and **`getHeight`** methods return these coordinate values individually, and the **`getLocation`**, **`getSize`**, and **`getBounds`** methods return composite values that encapsulate that information in a single object, as described in section 2.1.

Changing the location of a `GObject`

The next three methods in Figure 2-7 offer several techniques for changing the location of a graphical object. The **`setLocation(x, y)`** method sets the location to an absolute coordinate position on the screen. For example, in the **`FeltBoard`** example, executing the statement

```
rect.setLocation(0, 0);
```

would move the rectangle to the origin in the upper left corner of the window.

The **`move(dx, dy)`** method, by contrast, makes it possible to move an object relative to its current location. The effect of this call is to shift the location of the object by a specified number of pixels along each coordinate axis. For example, the statement

```
oval.move(10, 0);
```

would move the oval 10 pixels to the right. The **`dx`** and **`dy`** values can be negative. Calling

```
rect.move(0, -25);
```

would move the rectangle 25 pixels upward.

The **`movePolar(r, theta)`** method is useful in applications in which you need to move a graphical object in a particular direction. The name of the method comes from the concept of **polar coordinates** in mathematics, in which a displacement is defined by a

Figure 2-7. Useful methods common to all graphical objects

Methods to retrieve the location and size of a graphical object
`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.
`GPoint getLocation()` Returns the location of this object as a **`GPoint`**.
`GDimension getSize()` Returns the size of this object as a **`GDimension`**.
`GRectangle getBounds()` Returns the bounding box of this object (the smallest rectangle that covers the figure).
Methods to change the object's location
`void setLocation(double x, double y)` *or* **`setLocation(GPoint pt)`** 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.
Methods to set and retrieve the object's color
`void setColor(Color c)` Sets the color of the object.
`Color getColor()` Returns the object color. If this value is **`null`**, the package uses the color of the container.
Methods to change the stacking order
`void sendToFront()` *or* **`sendToBack()`** Moves this object to the front (or back) of the stacking order.
`void sendForward()` *or* **`sendBackward()`** Moves this object forward (or backward) one position in the stacking order.
Method to determine whether an object contains a particular point
`boolean contains(double x, double y)` *or* **`contains(GPoint pt)`** Checks to see whether a point is inside the object.

distance **`r`** and an angle **`theta`**. Just as it is in traditional geometry, the angle **`theta`** is measured in degrees counterclockwise from the $+x$ axis. Thus, the statement

```
rect.movePolar(10, 45);
```

would move the rectangle 10 pixels along a line in the 45° direction, which is northeast.

Setting the color of a `GObject`

The **`acm.graphics`** package does not define its own notion of color but instead relies on the **`Color`** class in the standard **`java.awt`** package. The predefined colors are:

```
Color.BLACK        Color.RED         Color.BLUE
Color.DARK_GRAY    Color.YELLOW      Color.MAGENTA
Color.GRAY         Color.GREEN       Color.ORANGE
Color.LIGHT_GRAY   Color.CYAN        Color.PINK
Color.WHITE
```

It is also possible to create additional colors using the constructors in the **`Color`** class. In either case, you need to include the import line

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

at the beginning of your program.

The **setColor** method sets the color of the graphical object to the specified value; the corresponding **getColor** method allows you to determine what color that object currently is. This facility allows you to make a temporary change to the color of a graphical object using code that looks something like this:

```
Color oldColor = gobj.getColor();
gobj.setColor(Color.RED);
. . . and then at some later time . . .
gobj.setColor(oldColor);
```

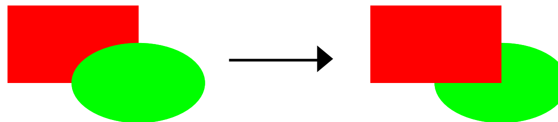
Controlling the stacking order

Figure 2-7 also lists a set of methods that make it possible to control the stacking order. The **sendToFront** and **sendToBack** methods move the object to the front or back of the stack, respectively. The **sendForward** and **sendBackward** methods move the object one step forward or backward in the stack so that it jumps ahead of or behind the adjacent object in the stack. Changing the stacking order also redraws the display to ensure that underlying objects are correctly redrawn.

For example, if you add the statement

```
oval.sendBackward();
```

to the end of the **FeltBoard** program, the picture on the display would change as follows:



Checking for containment

In many applications—particularly those that involve interactivity of the sort that you will see in Chapter 3—it is useful to be able to tell whether a graphical object contains a particular point. This facility is provided by the **contains(x, y)** method, which returns **true** if the point (**x**, **y**) is inside the figure. For example, given a standard Java **MouseEvent e**, you can determine whether the mouse is inside the rectangle **rect** using the following **if** statement:

```
if (rect.contains(e.getX(), e.getY()))
```

Even though every **GObject** subclass has a **contains** method, the precise definition of what it means for a point to be "inside" the object differs depending on the class. In the case of a **GOval**, for example, a point is considered to be inside the oval only if it is mathematically contained within the elliptical shape that the **GOval** draws. Points that are inside the bounding rectangle but outside of the oval are considered to be "outside." Thus, it is important to keep in mind that

```
gobj.contains(x, y)
```

and

```
gobj.getBounds().contains(x, y)
```

do not necessarily return the same answer.

The **GFillable**, **GResizable**, and **GScalable** interfaces

You have probably noticed that several of the examples you've already seen in this tutorial include methods that do not appear in the list in Figure 2-7. For example, the **FeltBoard** program includes calls to a **setFilled** method to mark the rectangle and oval as filled rather than outlined. Looking at Figure 2-7, it appears that the **GObject** class does not include a **setFilled** method, which is indeed the case.

As the caption for Figure 2-7 makes clear, the methods listed in that table are the ones that are common to *every* **GObject** subclass. While it is always possible to set the location of a graphical object, it is only possible to fill that object if the idea of "filling" makes sense for that class. Filling is easily defined for geometrical shapes such as ovals, rectangles, polygons, and arcs, but it is not clear what it might mean to fill a line, an image, or a label. Since there are subclasses that cannot give a meaningful interpretation to **setFilled**, that method is not defined at the **GObject** level but is instead implemented only for those subclasses for which filling is defined.

At the same time, it is important to define the **setFilled** method so that it works the same way for any class that implements it. If **setFilled**, for example, worked differently in the **GRect** and **GOval** classes, trying to keep track of the different styles would inevitably cause confusion. To ensure that the model for filled shapes remains consistent, the methods that support filling are defined in an interface called **GFillable**, which specifies the behavior of any fillable object. In addition to the **setFilled** method that you have already seen, the **GFillable** interface defines an **isFilled** method that tests whether the object is filled, a **setFillColor** method to set the color of the interior of the object, and a **getFillColor** method that retrieves the interior fill color. The **setFillColor** method makes it possible to set the color of an object's interior independently from the color of its border. For example, if you changed the code from the **FeltBoard** example so that the statements generating the rectangle were

```
GRect rect = new GRect(100, 50, 100, 100 / PHI);
rect.setFilled(true);
rect.setColor(Color.RED);
r.setFillColor(Color.MAGENTA);
```

you would see a rectangle whose border was red and whose interior was magenta.

In addition to the **GFillable** interface, the **acm.graphics** package includes two interfaces that make it possible to change the size of an object. Classes in which the dimensions are defined by a bounding rectangle—**GRect**, **GOval**, and **GImage**—implement the **GResizable** interface, which allows you to change the size of a resizable object **gobj** by calling

```
gobj.setSize(newWidth, newHeight);
```

A much larger set of classes implements the **GScalable** interface, which makes it possible to change the size of an object by multiplying its width and height by a scaling factor. In the common case in which you want to scale an object equally in both dimensions, you can call

```
gobj.scale(sf);
```

which multiplies the width and height by **sf**. For example, you could double the size of a scalable object by calling

```
gobj.scale(2);
```

The **scale** method has a two-argument form that allows you to scale a figure independently in the *x* and *y* directions. The statement

```
gobj.scale(1.0, 0.5);
```

leaves the width of the object unchanged but halves its height.

The methods specified by the **GFillable**, **GResizable**, and **GScalable** interfaces are summarized in Figure 2-8.

2.4 Descriptions of the individual shape classes

So far, this tutorial has looked only at methods that apply to all **GObject**s, along with a few interfaces that define methods shared by some subset of the **GObject** hierarchy. The most important classes in that hierarchy are the shape classes that appear at the bottom of Figure 2-1. The sections that follow provide additional background on each of the shape classes and include several simple examples that illustrate their use.

As you go through the descriptions of the individual shape classes, you are likely to conclude that some of them are designed in ways that are less than ideal for introductory students. In the abstract, this conclusion is almost certainly correct. For practical reasons that look beyond the introductory course, the Java Task Force decided to implement the shape classes so that they match their counterparts in Java's standard **Graphics** class. In particular, the set of shape classes corresponds precisely to the facilities that the **Graphics** class offers for drawing geometrical shapes, text strings, and images. Moreover, the constructors for each class take the same parameters and have the same semantics as the corresponding method in the **Graphics** class. Thus, the **GArc** constructor—which is arguably the most counterintuitive in many ways—has the structure it does, not because we thought that structure was perfect, but because that is the structure used by the **drawArc** method in the **Graphics** class. By keeping the semantics

Figure 2-8. Methods defined by interfaces

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

Method
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**)

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

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

Method
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.

consistent with its Java counterpart, the **acm.graphics** package makes it easier for students to move on to the standard packages as they learn more about programming.

The GRect class and its subclasses

The simplest and most intuitive of the shape classes is the **GRect** class, which represents a rectangular box. This class implements the **GFillable**, **GResizable**, and **GScalable** interfaces, but otherwise includes no other methods except its constructor, which comes in two forms. The most common form of the constructor is

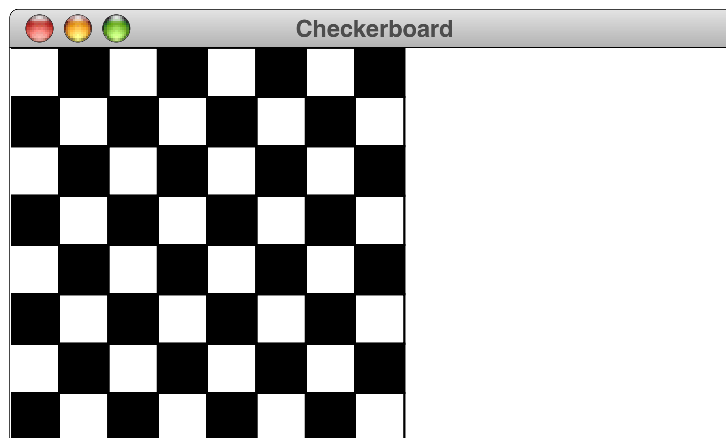
```
new GRect(x, y, width, height)
```

which defines both the location and size of the **GRect**. The second form of the constructor is

```
new GRect(width, height)
```

which defines a rectangle of the specified size whose upper left corner is at the origin. If you use this second form, you will typically add the **GRect** to the canvas at a specific (x, y) location as discussed in section 2.2.

You have already seen one example of the use of the **GRect** class in the simple **FeltBoard** example. A more substantive example is the **Checkerboard** program in Figure 2-9, which draws a checkerboard that looks like this:



As you can see from the diagram of the graphics class hierarchy in Figure 2-1, the **GRect** class has two subclasses—**GRoundRect** and **G3DRect**—that define shapes that are essentially rectangles but differ slightly in the way they are drawn on the screen. The **GRoundRect** class has rounded corners, and the **G3DRect** class has beveled edges that can be shadowed to make it appear raised or lowered. These classes extend **GRect** to change their visual appearance and to export additional method definitions that make it possible to adjust the properties of one of these objects. For **GRoundRect**, these properties specify the corner curvature; for **G3DRect**, the additional methods allow the client to indicate whether the rectangle should appear raised or lowered. Neither of these classes are used much in practice, but they are included in **acm.graphics** to ensure that it can support the full functionality of Java's **Graphics** class, which includes analogues for both.

The GOval class

The **GOval** class represents an elliptical shape and is defined so that the parameters of its constructor match the arguments to the **drawOval** method in the standard Java **Graphics** class. This design is easy to understand as long as you keep in mind the fact that Java

Figure 2-9. Code for the checkerboard example

```
/*
 * File: Checkerboard.java
 * -----------------------
 * This program draws a checkerboard.  The dimensions of the
 * checkerboard is specified by the constants NROWS and
 * NCOLUMNS, and the size of the squares is chosen so
 * that the checkerboard fills the available vertical space.
 */

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

public class Checkerboard extends GraphicsProgram {

/** Runs the program */
   public void run() {
      double sqSize = (double) getHeight() / NROWS;
      for (int i = 0; i < NROWS; i++) {
         for (int j = 0; j < NCOLUMNS; j++) {
            double x = j * sqSize;
            double y = i * sqSize;
            GRect sq = new GRect(x, y, sqSize, sqSize);
            sq.setFilled((i + j) % 2 != 0);
            add(sq);
         }
      }
   }

/* Private constants */
   private static final int NROWS = 8;     /* Number of rows    */
   private static final int NCOLUMNS = 8;  /* Number of columns */

}
```

defines the dimensions of an oval by specifying the rectangle that bounds it. Like **GRect**, the **GOval** class implements the **GFillable**, **GResizable**, and **GScalable** interfaces but otherwise includes no methods that are specific to the class.

The GLine class

The **GLine** class is used to display a straight line on the display. The standard **GLine** constructor takes the *x* and *y* coordinates of each end point. For example, to draw a line that extends diagonally from the origin of the canvas in the upper left to the opposite corner in the lower right, you could use the following code:

```
GLine diagonal = new GLine(0, 0, getWidth(), getHeight());
add(diagonal);
```

On the whole, the **GLine** class makes intuitive sense. There are, however, a few points that are worth remembering:

- Calling **setLocation(x, y)** or **move(dx, dy)** on a **GLine** object moves the line without changing its length or orientation. If you need to move one of the endpoints without affecting the other, you can do so by calling the methods **setStartPoint(x, y)** or **setEndPoint(x, y)**.
- The **GLine** class implements **GScalable**—which expands or contracts the line relative to its starting point—but not **GFillable** or **GResizable**.

- From a mathematical perspective, a line has no thickness and therefore does not actually any points. In practice, however, it is useful to define any point that is no more than a pixel away from the line segment as being part of the line. This definition makes it possible, for example, to select a line segment using the mouse by looking for points that are "close enough" to the line to be considered as being part of it.
- As with any other **`GObject`**, applying the **`getWidth`** method to a **`GLine`** returns its horizontal extent on the canvas. There is no way in **`acm.graphics`** to change the thickness of a line, which is always one pixel.

Even though the **`GLine`** class is conceptually simple, you can nonetheless create wonderfully compelling pictures with it. Figure 2-10, for example, shows a drawing made up entirely of **`GLine`** objects. The program to create this figure—which simulates the process of stringing colored yarn through a series of equally spaced pegs around the border—appears in Figure 2-11.

The `GArc` class

The **`GArc`** class—which is used to display elliptical arcs on the canvas—has proven to be somewhat more confusing to novices than the other shape classes in the **`acm.graphics`** package. As noted in the introduction to section 2.4, the Java Task Force chose to implement the **`GArc`** class so that its operation was consistent with the **`drawArc`** and **`fillArc`** methods in the standard **`Graphics`** class. This strategy has the advantage of making it easier for students to make the eventual transition to the standard tools at a cost of exposing some of Java's complexity earlier.

The **`GArc`** constructor takes six parameters: **`x`**, **`y`**, **`width`**, **`height`**, **`start`**, and **`sweep`**. The first four define a bounding rectangle exactly as they do for the **`GOval`** class. The **`start`** and **`sweep`** parameters—each of which is measured in degrees counterclockwise from the +*x* axis just as angles are in traditional geometry—indicate the angle at which the arc begins and how far it extends, respectively.

Figure 2-10. Pattern created by looping yarn around pegs spaced equally along the border

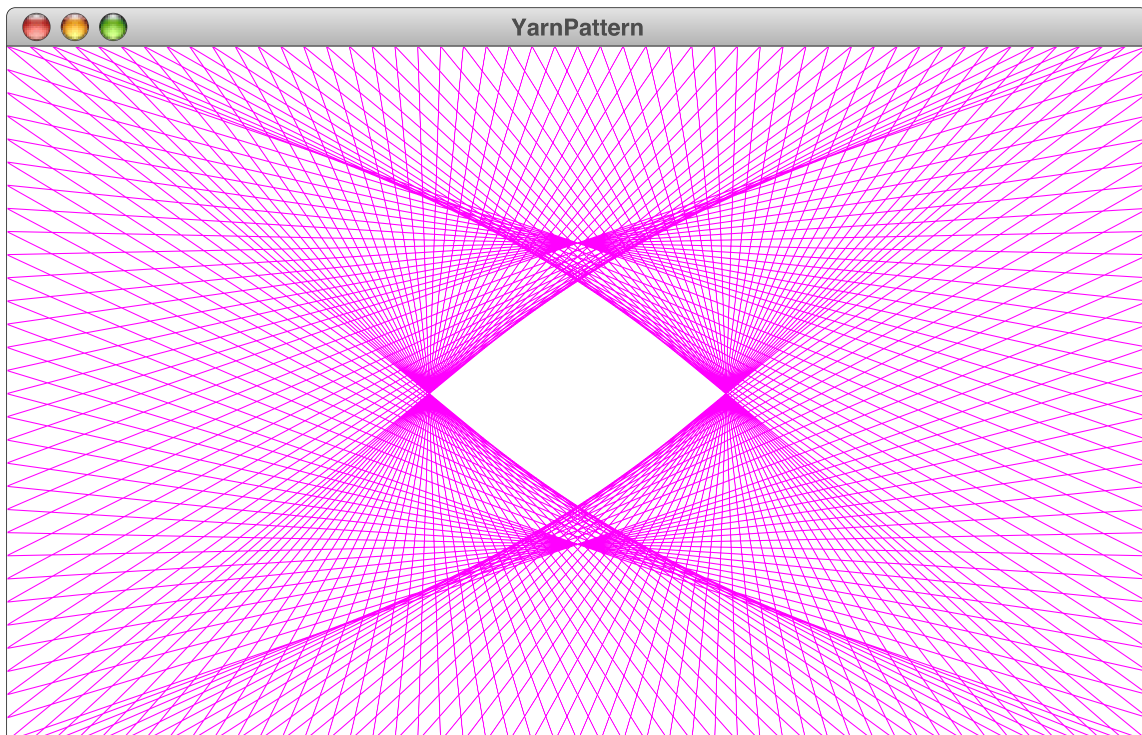


Figure 2-11. Code to generate the yarn pattern in Figure 2-10

```
/*
 * File: YarnPattern.java
 * ----------------------
 * This program illustrates the use of the GLine class to simulate
 * winding a piece of colored yarn around a set of pegs equally
 * spaced along the edges of the canvas.  At each step, the yarn is
 * stretched from its current peg to the one DELTA pegs further on.
 */

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

public class YarnPattern extends GraphicsProgram {

/** Runs the program */
   public void run() {
      ArrayList pegs = createPegList();
      int thisPeg = 0;
      int nextPeg = -1;
      while (thisPeg != 0 || nextPeg == -1) {
         nextPeg = (thisPeg + DELTA) % pegs.size();
         GPoint p0 = (GPoint) pegs.get(thisPeg);
         GPoint p1 = (GPoint) pegs.get(nextPeg);
         GLine line = new GLine(p0.getX(), p0.getY(),
                                p1.getX(), p1.getY());
         line.setColor(Color.MAGENTA);
         add(line);
         thisPeg = nextPeg;
      }
   }

/* Create an array list containing the locations of the pegs */
   private ArrayList createPegList() {
      ArrayList pegs = new ArrayList();
      for (int i = 0; i < N_ACROSS; i++) {
         pegs.add(new GPoint(i * PEG_SEP, 0));
      }
      for (int i = 0; i < N_DOWN; i++) {
         pegs.add(new GPoint(N_ACROSS * PEG_SEP, i * PEG_SEP));
      }
      for (int i = N_ACROSS; i > 0; i--) {
         pegs.add(new GPoint(i * PEG_SEP, N_DOWN * PEG_SEP));
      }
      for (int i = N_DOWN; i > 0; i--) {
         pegs.add(new GPoint(0, i * PEG_SEP));
      }
      return pegs;
   }

/* Private constants */
   private static final int N_ACROSS = 50;
   private static final int N_DOWN = 30;
   private static final int PEG_SEP = 10;
   private static final int DELTA = 67;

}
```

To get a sense of how these parameters work, it is easiest to look at a simple example, such as the following:

```
double size = 200;
double x = (getWidth() - size) / 2;
double y = (getHeight() - size) / 2;
GArc arc = new GArc(x, y, size, size, 45, 270);
add(arc);
```

The first three lines define the size of the bounding rectangle and calculate the *x* and *y* coordinates at its upper left corner. The constructor call itself then includes the dimensions of that rectangle along with a **start** parameter of 45 and a **sweep** parameter of 270. Together, these parameters define an arc that begins at 45˚ and then extends through 270˚ in the counterclockwise direction. The interpretation of these parameters is illustrated on the left side of Figure 2-12.

The two sample runs at the right of Figure 2-12 show how the arc appears on the canvas. The code example from the preceding paragraph creates an unfilled arc as shown in the upper diagram. The lower diagram shows what happens if you were to call

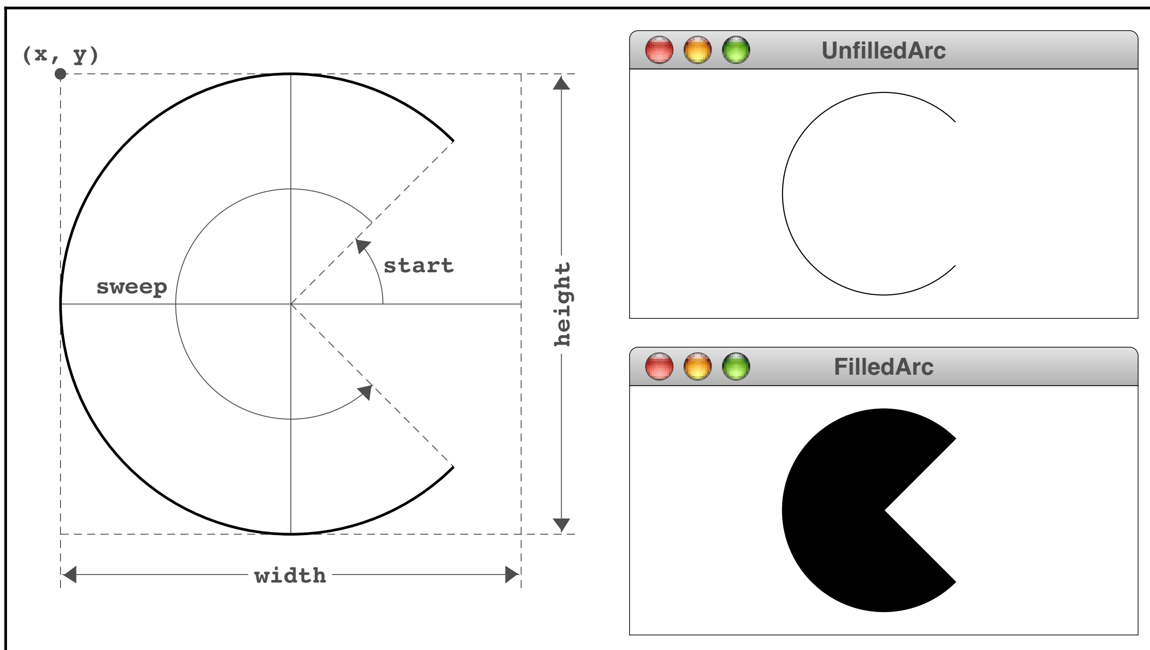
```
arc.setFilled(true);
```

In Java, a filled arc is drawn by connecting the endpoints of the arc to the center of the circle and the filling the interior. The result is a wedge shape of the sort you would find in a pie chart (or, as shown in the diagram, a Pac-Man game).

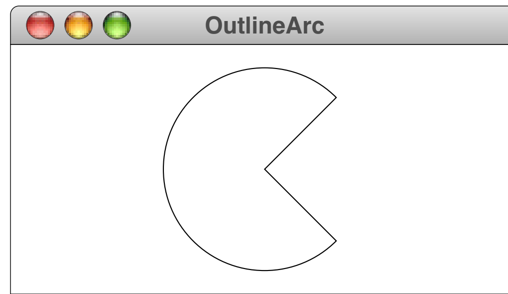
Java's interpretation of what it means to fill an arc can cause confusion because the unfilled arc does not include the complete outline of its filled counterpart. You can create the outline of the wedge by filling the **GArc** and then setting its fill color to match the background. Thus, if you were to add the line

```
arc.setFillColor(Color.WHITE);
```

Figure 2-12. The geometry of the GArc class



you would see a figure like this:

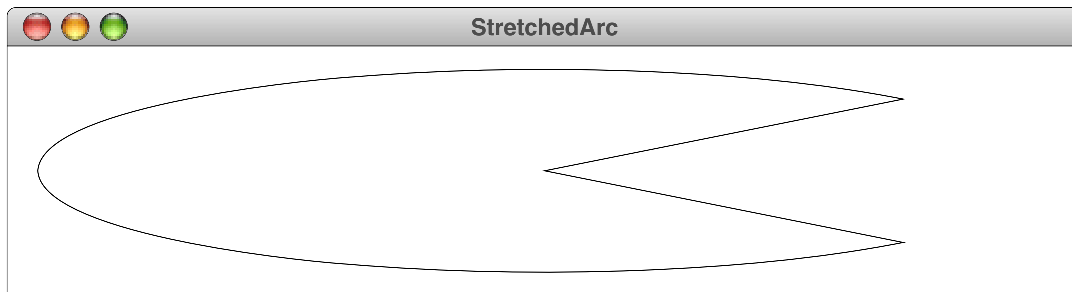


Java's interpretation of filling also has implications for the semantics of the **`contains`** method. For an unfilled arc, containment implies that the arc point is actually on the arc, subject to the same interpretation of "closeness" as described for lines in the preceding section. For a filled arc, containment implies inclusion in the wedge. This definition of containment is necessary to ensure that mouse events are transmitted to the arc in a way that matches the user's intuition.

If the **`width`** and **`height`** parameters in the **`GArc`** constructor are different, the arc will be elliptical rather than circular. When this occurs, the arc segment is taken from the oval inscribed in the bounding rectangle, just as you would expect. The confusing thing is that the angles are always interpreted as if the arc were scaled to be circular. For example, if you were to increase the window size and then call

```
arc.scale(5, 1);
```

you would get the following elongated arc, which is five times as wide as it is high:



In this figure, the **`start`** and **`sweep`** angles are still defined to be 45˚ and 270˚, even though the missing wedge at the right is now clearly smaller than the 90˚ angle that appears in the unscaled figure.

Figure 2-13 offers a more substantive example of the use of arcs by generating the background curve for the Taoist yin-yang symbol:

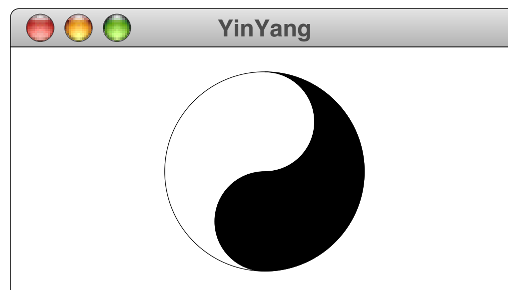


Figure 2-13. Code to create a yin-yang figure using arcs

```
/*
 * File: YinYang.java
 * ------------------
 * This program draws the Taoist yin-yang symbol at the center of
 * the graphics window.  The height and width of the entire figure
 * are both specified by the constant FIGURE_SIZE.
 */

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

public class YinYang extends GraphicsProgram {

/** Runs the program */
   public void run() {
      double x = getWidth() / 2;
      double y = getHeight() / 2;
      double r = FIGURE_SIZE / 2;
      GArc bigBlack = new GArc(x - r, y - r, 2 * r, 2 * r, -90, 180);
      bigBlack.setFilled(true);
      add(bigBlack);
      GArc smallWhite = new GArc(x - r / 2, y - r, r, r, -90, 180);
      smallWhite.setFilled(true);
      smallWhite.setColor(Color.WHITE);
      add(smallWhite);
      GArc smallBlack = new GArc(x - r / 2, y, r, r, 90, 180);
      smallBlack.setFilled(true);
      add(smallBlack);
      GArc outerCircle = new GArc(x - r, y - r, 2 * r, 2 * r, 0, 360);
      add(outerCircle);
   }

/* Private constants */
   private static final double FIGURE_SIZE = 150;
}
```

The **`GLabel`** class

The **`GLabel`** class is used to display text strings on the canvas. The **`GLabel`** class is different from the other shape classes because the operations one wants to perform on strings are different from those that are appropriate for geometrical figures. As a result, the **`GLabel`** class implements none of the standard **`GFillable`**, **`GResizable`**, and **`GScalable`** interfaces but instead has its own collection of methods, as shown in Figure 2-14. Despite the many differences, however, it is useful to include **`GLabel`** in the graphic hierarchy so that it is possible to mix geometric figures and text on the canvas.

Despite its lack of symmetry with the other shape classes, the **`GLabel`** class is easy to understand once you figure out the terminology used to define its geometry. The most important thing to realize is that the position of a **`GLabel`** is not defined by the upper left corner, but by the starting point of the **baseline,** which is the imaginary line on which the characters sit. The origin and baseline properties of the **`GLabel`** class are illustrated in the following diagram:

Figure 2-14. Useful methods in the `GLabel` class

Constructors

`new GLabel(String str, double x, double y)` Creates a new **`GLabel`** containing **`str`** whose baseline begins at the point (**`x`**, **`y`**).
`new GLabel(String str)` Creates a new **`GLabel`** containing **`str`** whose baseline begins at the point (0, 0).

Methods to set and retrieve the text string

`void setLabel(String str)` Changes the string displayed by the label to **`str`**. The size generally changes as a result.
`String getLabel()` Returns the string stored inside the label.

Methods to change and query the font used for the display

`void setFont(Font f)` *or* **`setFont(String description)`** Sets the font using a Java **`Font`** object or a string in the form **`"Family-style-size"`**.
`Font getFont()` Returns the current font.
`double getAscent()` Returns the maximum distance characters in the current font ascend above the baseline.
`double getDescent()` Returns the maximum distance characters in the current font descend below the baseline.

Given that the **`GLabel`** class does not implement either the **`GResizable`** or the **`GScalable`** interface, the only way to change the size of a **`GLabel`** is to change its font. The **`setFont`** method takes a standard Java **`Font`** object, but is overloaded so that it can also accept a string describing the font. That string is interpreted in the manner specified by **`Font.decode`** and consists of three parts—the font family, the style, and the point size—separated by hyphens. Thus, to set the font of a variable named **`title`** to be an 18-point, boldface, sans-serif font, you can simply write

```
label.setFont("SansSerif-bold-18");
```

When given a string as its argument, the **`setFont`** method interprets an asterisk in any of these positions as signifying the previous value. Thus, you can set the style of a label to italic without changing its family or size by writing

```
label.setFont("*-italic-*");
```

Several of the methods in Figure 2-14 are included to assist in the process of positioning the **`GLabel`**. The most important methods for controlling the position, however, are not specific to the **`GLabel`** class but are simply the **`getWidth`** and **`getHeight`** methods inherited from **`GObject`**. As they do for all graphical objects, these methods return the dimensions of the rectangle that bounds the figure. The **`getWidth`** method returns the horizontal extent of the label and will change if you change either the font or the internal string. The **`getHeight`** method returns the height of a single line of text in the current font, which is defined to be the distance between the baselines of successive lines. The **`getAscent`** and **`getDescent`** methods return the maximum distance the current font extends above and below the baseline, respectively.

Now that you know about these methods, you are finally in a position to understand the details of how the code in the **`HelloGraphics`** example from Chapter 1 centers the label on the canvas. The relevant lines look like this:

```
GLabel label = new GLabel("hello, world");
double x = (getWidth() - label.getWidth()) / 2;
double y = (getHeight() + label.getAscent()) / 2;
add(label, x, y);
```

The calculation of the *x* coordinate moves rightward half the width of the canvas as a whole, but subtracts away half the width of the **GLabel** to reach its origin point. The calculation of the *y* coordinate is similar, but this time the offset is half the distance the font extends above the baseline, which is given by **getAscent**. The plus sign in the calculation of the *y* coordinate value may initially seem confusing, but the confusion disappears when you remember that *y* values increase as you move down the screen.

If you work with a lot of text on the canvas, you will probably discover at some point that the strategy of using **getAscent** to center a **GLabel** vertically doesn't quite work. Most labels that you display on the canvas will appear to be a few pixels too low. The reason for this behavior is that **getAscent** returns the *maximum* ascent of the font and not the distance the text of this particular label happens to rise above the baseline. For most fonts, the parentheses and diacritical marks extend above the tops of the uppercase letters and therefore make the font ascent larger than it seems to be in practice. If you are a stickler for aesthetics, you may need to adjust the vertical centering by a pixel or two to have things look precisely right.

The GImage class

The **GImage** class is used to display an image on the canvas. That image must be presented in one of the standard formats for image data, of which the most common are the GIF (Graphics Interchange Format) and JPEG (Joint Photographic Experts Group) formats. To create an image, you will need to use an image processing tool (such as Adobe Photoshop™) or download the image from the web. Unless they are specifically marked as being in public domain, images you find on the web are usually subject to copyright protection, which means that you must ensure that any use you make of those images fits under the standard "fair use" guidelines.

Once you have created the image file, you need to put it in a place where your Java application can find it. The standard **GImage** constructor takes the name of the image and then looks for the image data in the following places:

1. Inside the JAR file for the application, looking first for an image resource in the top-level package and then for one in a package named **images**.
2. Inside the directory that contains the application, looking for an image file with the appropriate name.
3. Inside a subdirectory called **images** of the application directory.

If the image is found in any of those places, it is loaded automatically to create a Java **Image** object. If none of these places contains the image, the **GImage** constructor generates a runtime error. A more extensive discussion of the search strategy for images appears in the documentation for the **MediaTools** class in the **acm.util** package.

To offer a simple example, suppose that you wanted to display the logo for the Java Task Force in the center of a **GraphicsProgram** canvas. The first step would be to download the **JTFLogo.gif** file from our web site and store it on your own machine, either in the directory that contains the program or in a **images** subdirectory. You can then create the image and add it to the window like this:

```
GImage logo = new GImage("JTFLogo.gif");
double x = (getWidth() - logo.getWidth()) / 2;
double y = (getHeight() - logo.getHeight()) / 2;
add(logo, x, y);
```

This code generates the following display:

The **GImage** class implements both the **GResizable** and **GScalable** interfaces, but not **GFillable**. Scaling and resizing of images is performed automatically by Java's image-handling libraries. For example, you could stretch and recenter the logo by issuing the commands

```
logo.move(-logo.getWidth() / 2, 0);
logo.scale(2, 1);
```

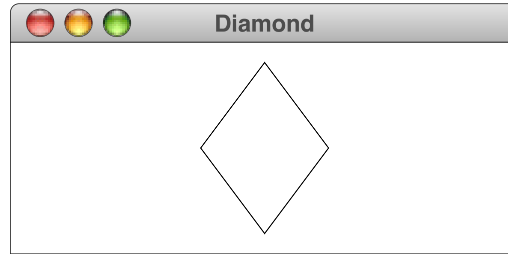
which would change the display to

The GPolygon class

The shape that represents the greatest deviation from the traditional Java model is the **GPolygon** class, which is used to draw closed polygonal shapes. Unlike the other shape classes, the constructor for the **GPolygon** does not create a fully formed object but instead creates a polygon with no vertices. Starting with that empty polygon, you can then add additional vertices using any of three different methods—**addVertex**, **addEdge**, and **addPolarEdge**—that are defined as part of the **GPolygon** class.

These methods are easiest to illustrate by example. Before doing so, however, it is important to note that the coordinates of each vertex are not expressed in the global coordinate space of the canvas but are instead defined in relation to a point chosen to represent the **origin** of the polygon. The origin need not be one of the vertices and is typically chosen to be the center, particularly for regular polygonal shapes. Setting the location of a **GPolygon** corresponds to specifying the location of its origin, and the polygon is then drawn relative to that point. The advantage of this strategy is that moving a polygon requires changing the coordinates of just one point and does not require adjusting the coordinates of each vertex.

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. Suppose, for example, that you wanted to draw a diamond shape, 80 pixels high and 60 pixels wide, that looks like this:



You can define the diamond **GPolygon** relative to its center using the following code:

```
GPolygon diamond = new GPolygon();
diamond.addVertex(-30, 0);
diamond.addVertex(0, 40);
diamond.addVertex(30, 0);
diamond.addVertex(0, -40);
```

If you then wanted to position the diamond in the center of the window as it appears in the example, you could add it to the canvas with the line

```
add(diamond, getWidth() / 2, getHeight() / 2);
```

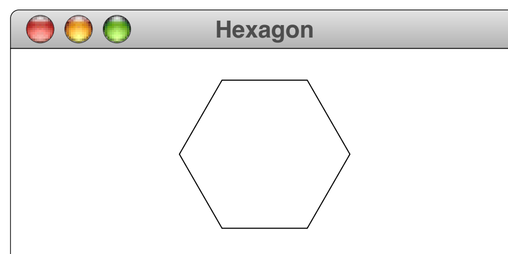
The diamond is then drawn so that its center is at the center of the canvas a whole. Each of the vertices is expressed relative to that point.

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

```
GPolygon diamond = new GPolygon();
diamond.addVertex(-30, 0);
diamond.addEdge(30, 40);
diamond.addEdge(30, -40);
diamond.addEdge(-30, -40);
diamond.addEdge(-30, 40);
```

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 terms of polar coordinates. This mode of specification is supported in the **GPolygon** class by the method **addPolarEdge(r, theta)**, which is identical to **addEdge(dx, dy)** except that its arguments are the length of the edge (**r**) and its direction (**theta**) expressed in degrees counterclockwise from the +*x* axis. This method makes it easy to create figures such as the hexagon



which can be generated using the following method, where **side** indicates the length of each edge:

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

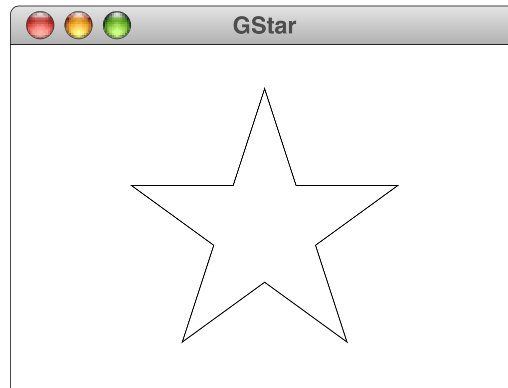
The **GPolygon** class implements the **GFillable** and **GScalable** interfaces, but not **GResizable**. It also supports the method **rotate(theta)**, which rotates the polygon **theta** degrees counterclockwise around its origin.

The **GPolygon** class is also useful as a base for new shape classes whose outlines are polygonal regions. This strategy is illustrated by the **GStar** class in Figure 2-15, which

Figure 2-15. Class definition for a five-pointed star

```
/*
 * File: GStar.java
 * ----------------
 * This file illustrates the strategy of subclassing GPolygon by
 * creating a new GObject class depicting a five-pointed star.
 */

import acm.graphics.*;

/**
 * Defines a new GObject class that appears as a five-pointed star.
 */

public class GStar extends GPolygon {

/**
 * Creates a new GStar centered at the origin that fits inside
 * a square of the specified size.
 */
   public GStar(double size) {
      double sinTheta = GMath.sinDegrees(18);
      double b = 0.5 * sinTheta / (1.0 + sinTheta);
      double edge = (0.5 - b) * size;
      addVertex(-size / 2, -b * size);
      int angle = 0;
      for (int i = 0; i < 5; i++) {
         addPolarEdge(edge, angle);
         addPolarEdge(edge, angle + 72);
         angle -= 72;
      }
      markAsComplete();
   }

/**
 * Creates a new GStar centered at the point (x, y) that fits inside
 * a square of the specified size.
 */
   public GStar(double x, double y, double size) {
      this(size);
      setLocation(x, y);
   }
}
```

draws the following five-pointed star:



The only complicated part of the `GStar` definition is the geometry required to compute the coordinates of the starting point of the figure.

The constructor for the `GStar` class ends with a call to the protected `GPolygon` method `markAsComplete`, which prohibits clients from adding more vertices to the polygon. This call protects the integrity of the class and makes it impossible for clients to change the shape of a `GStar` object into something else.

2.5 The GCompound class

The shape classes described in Section 2.4 are the basic building blocks that allow you to build more complicated structures. In a sense, these classes represent the "atoms" of the `acm.graphics` world. Particular as diagrams become more complex, it is useful to assemble several atomic shapes into a "molecule" that you can then manipulate as a unit. In the `acm.graphics` package, this facility is provided by the `GCompound` class.

The methods defined in `GCompound` are in some sense the union of those available to the `GObject` and `GCanvas` classes. As a `GObject`, a `GCompound` responds to method calls like `setLocation` and `move`; as an implementer (like `GCanvas`) of the `GContainer` interface, it supports methods like `add` and `remove`. A summary of the important methods available for `GCompound` appears in Figure 2-16.

A simple example of `GCompound`

To get a sense of 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:

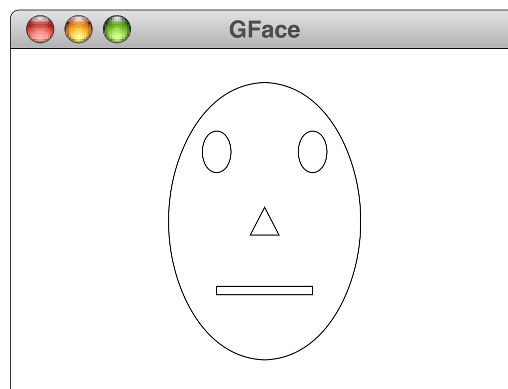


Figure 2-16. Important methods in the `GCompound` class (beyond those from `GObject`)

Constructor	
	`GCompound()` Creates a new **`GCompound`** that contains no objects.
Methods to add and remove graphical objects from a compound	
	`void add(GObject gobj)` Adds a graphical object to the compound.
	`void add(GObject gobj, double x, double y)` *or* **`add(GObject gobj, GPoint pt)`** Adds a graphical object to the compound at the specified location.
	`void remove(GObject gobj)` Removes the specified graphical object from the compound.
	`void removeAll()` Removes all graphical objects and components from the compound.
Methods to determine the contents of the compound	
	`Iterator iterator()` Returns an iterator that runs through the graphical objects from back to front.
	`GObject getElementAt(double x, double y)` *or* **`getElementAt(GPoint pt)`** Returns the topmost object containing the specified point, or **`null`** if no such object exists.
Miscellaneous methods	
	`void markAsComplete()` Marks this compound as complete to prohibit any further changes to its contents.
	`GPoint getLocalPoint(double x, double y)` *or* **`getLocalPoint(GPoint pt)`** Returns the point in the local coordinate space corresponding to **`pt`** in the canvas.
	`GPoint getCanvasPoint(double x, double y)` *or* **`getCanvasPoint(GPoint pt)`** Returns the point on the canvas corresponding to **`pt`** in the local coordinate space.

For the most part, this figure is 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 the face as a unit. Suppose, for example, that you wanted to move the entire face to some new position on the screen. As things stand, doing so would require moving the various graphical objects independently. It would be better simply to tell the entire face to move.

The code in Figure 2-17 uses **`GCompound`** to define a **`GFace`** class that contains the necessary components. These components are created and then added in the appropriate places as part of the **`GFace`** constructor. Once you have defined this class, you can then construct a new **`GFace`** object and add it to the center of the canvas using the following code:

```
GFace face = new GFace(100, 150);
add(face, getWidth() / 2, getHeight() / 2);
```

The `GCompound` coordinate system

The general paradigm for using **`GCompound`** is to create an empty instance of the class and then to add other graphical objects to it. The coordinates at which these objects appear are expressed relative to the reference point of the **`GCompound`** itself, and not to the canvas in which the compound will eventually appear. This strategy means that you can add a compound object to a canvas and the move all its elements as a unit simply by setting the location of the compound. Thus, once you had created the **`GFace`** object described in the preceding section, you could move the entire face 20 pixels to the right by executing the following method:

```
face.move(20, 0);
```

Figure 2-17. Program to create a `GFace` class by extending `GCompound`

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

import acm.graphics.*;

/**
 * This code defines a new class called GFace, which is a compound
 * object consisting of an outline, two eyes, a nose, and a mouth.
 * The origin point for the face is the center of the figure.
 */
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, -width / 2, -height / 2);
      add(leftEye, -0.25 * width - EYE_WIDTH * width / 2,
                   -0.25 * height - EYE_HEIGHT * height / 2);
      add(rightEye, 0.25 * width - EYE_WIDTH * width / 2,
                    -0.25 * height - EYE_HEIGHT * height / 2);
      add(nose, 0, 0);
      add(mouth, -MOUTH_WIDTH * width / 2,
                 0.25 * 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;

/* Private instance variables */
   private GOval head;
   private GOval leftEye, rightEye;
   private GPolygon nose;
   private GRect mouth;
}
```

In some cases—most notably when you need to translate the coordinates of a mouse click, which are expressed in the global coordinate space of the canvas—it is useful to be able to convert coordinates from the local coordinate space provided by the **GCompound** to the coordinate space of the enclosing canvas, and vice versa. These conversions are implemented by the methods **getCanvasPoint** and **getLocalPoint**, as described in Figure 2-16.

Recentering objects using GCompound

The fact that the **GCompound** class maintains its own coordinate system has an additional advantage that may not immediately spring to mind: it allows you to change the reference point for a single graphical object, usually to make the interpretation of that reference point more closely correspond to the way that object might behave in the real world. As an example, consider how you might represent a bouncing ball on the canvas. The obvious solution is to use a **GOval** with equal width and height so that it appears as a circle. The only problem with that strategy is that the reference point of a **GOval** is in the upper left corner. If you want to perform any physical calculations involving the ball, it would be far better if the location of the ball were defined to be its center.

The simplest way to accomplish this shift in the reference point from the corner to the center is to use the **GCompound** class. If you add a **GOval** with radius *r* so that its location in the coordinate system of the compound is at $(-r, -r)$, then the **GCompound** will display itself on the canvas as a circle centered at the location of the **GCompound** object. The code in Figure 2-18 shows how to use this strategy to create a new **GBall** class whose location represents the center of the ball. You'll have a chance to see this class in action in

Figure 2-18. Using GCompound to create a ball defined by its center

```
/*
 * File: GBall.java
 * ----------------
 * This file defines a GObject class that represents a ball.
 */

import acm.graphics.*;

/**
 * This class defines a GObject subclass that represents a ball
 * whose reference point is the center rather than the upper
 * left corner.
 */
public class GBall extends GCompound {

/** Creates a new ball with radius r centered at the origin */
   public GBall(double r) {
      GOval ball = new GOval(2 * r, 2 * r);
      ball.setFilled(true);
      add(ball, -r, -r);
      markAsComplete();
   }

/** Creates a new ball with radius r centered at (x, y) */
   public GBall(double r, double x, double y) {
      this(r);
      setLocation(x, y);
   }
}
```

Chapter 3 which includes code to animate a **GBall** object so that it bounces around inside the boundaries of the canvas.

2.6 The GPen and GTurtle classes

The remaining two classes in the **acm.graphics** package are the **GPen** and **GTurtle** classes, which don't really fit into the shape class framework. Their purpose is to provide students with a simple mechanism for drawing figures using a paradigm that is more aligned with the pen-on-paper model that the felt-board model used in the rest of the package.

The GPen class

The **GPen** class models a pen that remembers its current location on the **GCanvas** on which it is installed. The most important methods for **GPen** are **setLocation** (or **move** to specify relative motion) and **drawLine**. The former corresponds to picking up the pen and moving it to a new location; the latter represents motion with the pen against the canvas, thereby drawing a line. Each subsequent line begins where the last one ended, which makes it very easy to draw connected figures. The **GPen** object also remembers the path it has drawn, making it possible to redraw the path when repaint requests occur. The most important methods for the **GPen** class are shown in Figure 2-19.

Figure 2-19. Useful methods in the GPen class

Constructors

GPen() Creates a new **GPen** object with an empty path.
GPen(double x, double y) Creates a new **GPen** object whose initial location is the point (**x**, **y**).

Methods to reposition and draw lines with the pen

void setLocation(double x, double y) *or* **setLocation(GPoint pt)** Moves the pen to the specified absolute location.
void move(double dx, double dy) Moves the pen using the displacements **dx** and **dy**.
void movePolar(double r, double theta) Moves the pen **r** units in the direction **theta**, measured in degrees.
void drawLine(double dx, double dy) Draws a line with the specified displacements, leaving the pen at the end of that line.
void drawPolarLine(double r, double theta) Draws a line **r** units in the direction **theta**, measured in degrees.

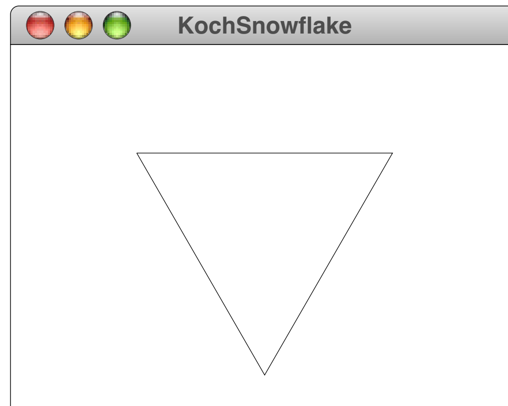
Methods to define a filled region bounded by pen strokes

void startFilledRegion() Fills the polygon formed by lines drawn between here and the next **endFilledRegion**.
void endFilledRegion() Closes and fills the region begun by **startFilledRegion**.

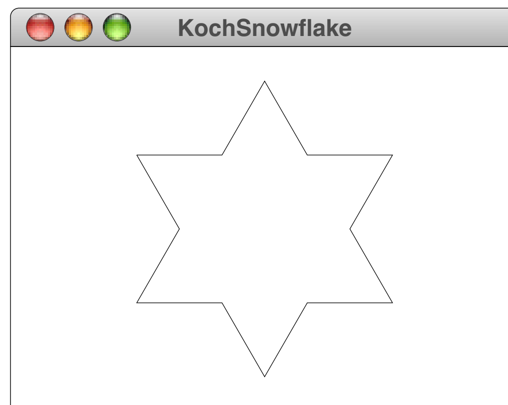
Miscellaneous methods

void showPen() Makes the pen itself visible, making it possible to see where the pen moves.
void hidePen() Makes the pen invisible.
void setSpeed(double speed) Sets the speed of the pen, which must be a number between 0 (slow) and 1 (fast).
double getSpeed() Returns the speed last set by **setSpeed**.
void erasePath() Removes all lines from this pen's path.

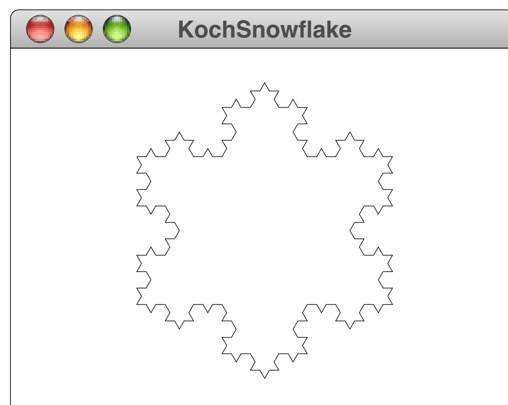
The graphics model provided by the `GPen` class is particularly well suited for generating recursive figures, such as the Koch fractal or "snowflake" curve. In its simplest form, the Koch fractal is simply an equilateral triangle that looks like this:



This figure is called the **order 0 Koch fractal.** To construct the Koch fractal of the next higher order, all you do is replace each of the lines in the current figure by a new line in which the center third is replace by a triangular wedge pointing to the outside of the figure. If you do this for each of the three line segments in the order 0 fractal, you get the order 1 fractal:



You can continue this process to obtain Koch fractals of successively higher orders. The order 3 Koch fractal, for example, looks like this:



A recursive program to display a Koch fractal appears in Figure 2-20.

Figure 2-20. Using `GPen` to draw a Koch fractal snowflake

```
/*
 * File: KochSnowflake.java
 * ------------------------
 * This program demonstrates the use of the GPen class by drawing
 * a Koch fractal snowflake.
 */

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

public class KochSnowflake extends GraphicsProgram {

/** Runs the program to create the snowflake display */
   public void run() {
      double width = getWidth();
      double height = getHeight();
      pen = new GPen();
      add(pen, width / 2, height / 2);
      drawKochFractal(EDGE_FRACTION * Math.min(width, height), ORDER);
   }

/*
 * Draws a snowflake fractal centered at the current pen position.
 * The edge parameter indicates the length of any of an edge on the
 * order 0 fractal, which is simply a triangle.  The order parameter
 * specifies the number of levels of recursive decomposition.
 */
   private void drawKochFractal(double edge, int order) {
      pen.move(-edge / 2, -edge / (2 * Math.sqrt(3)));
      drawFractalLine(edge, 0, order);
      drawFractalLine(edge, -120, order);
      drawFractalLine(edge, +120, order);
   }

/*
 * Draws a fractal line that extends r pixels in the direction theta.
 */
   private void drawFractalLine(double r, int theta, int order) {
      if (order == 0) {
         pen.drawPolarLine(r, theta);
      } else {
         drawFractalLine(r / 3, theta, order - 1);
         drawFractalLine(r / 3, theta + 60, order - 1);
         drawFractalLine(r / 3, theta - 60, order - 1);
         drawFractalLine(r / 3, theta, order - 1);
      }
   }

/* Private constants */
   private static final double EDGE_FRACTION = 0.75;
   private static final int ORDER = 3;

/* Private instance variables */
   private GPen pen;

}
```

Both the **GPen** class and the **GTurtle** class described in the following section are often used to create animated displays. To provide clients with some control over the speed of the animation, both classes include a **setSpeed** method, which takes a number between 0.0 and 1.0 as its argument. Calling **setSpeed(0.0)** means that the animation crawls along at a very slow pace; calling **setSpeed(1.0)** makes it proceed as fast as the system allows. Intermediate values are interpreted so as to provide a smoothly varying speed of operation. Thus, if the speed value is associated with a scrollbar whose ends represent the values 0.0 and 1.0, adjusting the scrollbar will cause the animation to speed up or slow down in a way that seems reasonably natural to users.

The GTurtle class

The **GTurtle** class is similar to **GPen** but uses a "turtle graphics" model derived from the Project Logo turtle described in Seymour Papert's book *Mindstorms*. In the turtle graphics world, the conceptual model is that of a turtle moving on a large piece of paper. A **GTurtle** object maintains its current location just as a **GPen** does, but also maintains a current direction.

The most common methods in the **GTurtle** class are shown in Figure 2-21. Of these, the ones that implement the essential semantics for **GTurtle** are **forward(***distance***)**, which moves the turtle forward the specified distance, and the directional methods

Figure 2-21. Useful methods in the GTurtle class

Constructors

Method	Description
GTurtle()	Creates a new .**GTurtle** object with an empty path.
GTurtle(double x, double y)	Creates a new .**GTurtle** object whose initial location is the point (**x**, **y**).

Methods to move and rotate the turtle

Method	Description
void setLocation(double x, double y) *or* **setLocation(GPoint pt)**	Moves the turtle to the specified absolute location without drawing a line.
void forward(double distance)	Moves the turtle **distance** units in the current direction, drawing a line if the pen is down.
void setDirection(double direction)	Sets the direction (in degrees counterclockwise from the *x*-axis) in which the turtle is moving.
double getDirection()	Returns the current direction in which the turtle is moving.
void right(double angle) *or* **right()**	Turns the turtle direction the specified number of degrees to the right (default is 90).
void left(double angle) *or* **left()**	Turns the turtle direction the specified number of degrees to the left (default is 90).

Miscellaneous methods

Method	Description
void penDown()	Tells the turtle to lower its pen so that it draws a track. The pen is initially up.
void penUp()	Tells the turtle to raise its pen so that it stops drawing a track
void showTurtle()	Makes the turtle visible. The turtle itself is initially visible.
void hideTurtle()	Makes the turtle invisible.
void setSpeed(double speed)	Sets the speed of the turtle, which must be a number between 0 (slow) and 1 (fast).
double getSpeed()	Returns the speed last set by **setSpeed**.
void erasePath()	Removes all lines from the turtle's path.

which moves the turtle forward the specified distance, and the directional methods **left(***angle***)** and **right(***angle***)**, which rotate the turtle the indicated number of degrees in the appropriate direction. The path is created by a pen located at the center of the turtle. If the pen is down, calls to **forward** generate a line; if the pen is up, such calls simply move the turtle without drawing a line.

Although **GTurtle** and **GPen** have similar capabilities, they are likely to be used in different ways. The **GTurtle** class is designed to be used at the very beginning of a course and must be both simple and evocative as intuitive model. The **GTurtle** therefore has somewhat different defaults than its **GPen** counterpart does. The image of the **GTurtle**, for example, is initially visible, while the **GPen** image is hidden. Moreover, the **GTurtle** does not actually draw lines until the pen is lowered. The **GPen** offers no option; the pen is always down. These defaults make the **GTurtle** consistent with the Logo model, in which students learn to move the turtle first and then start drawing pictures with it.

2.7 The GObjectTrace demonstration program

In order for students to get a good sense of how the **acm.graphics** model works, they need to see it in action. Although writing graphical programs is essential to gaining an understanding, there is a sense in which the resulting programs have lost some immediacy. Students need to understand that whenever they call a method in a graphical object—or, in the language of object-oriented programming, send it a message—that object will respond by repainting itself on the canvas. In a running application, those repaint requests are typically coalesced into a single operation, which makes it difficult to see how an object responds to the individual messages.

To make it easier for students to see precisely how graphical objects work and what effect each type of message has on that object, the **demos** section of the Java Task Force web site contains a program called **GObjectTrace**, which you can find at the following URL:

http://jtf.acm.org/demos/demos/GObjectTrace.html

This applet brings up a program window that has a **GCanvas** in the upper portion and an interactive console along the bottom. This console is running a stripped-down version of a Java interpreter that includes definitions for the entire **GObject** hierarchy. By typing Java statements into this window, you can create arbitrary objects, add them to the canvas, and then send those objects messages that are immediately reflected in the canvas display.

As an example, the three screen snapshots in Figure 2-22 show a series of steps in a **GObjectTrace** session. The line

```
GRect r = new GRect(25, 25, 100, 50);
```

allocates a new **GRect** object whose upper left corner is at the point (25, 25) and whose dimensions are 100×50. Nothing appears on the canvas, however, until the user calls

```
add(r);
```

at which point an unfilled rectangle appears. The next two lines of input show what happens if you send messages—in this case, **setFilled** and **move**—to the object, which responds as shown in the display.

Figure 2-22. Using the `GObjectTrace` application

Chapter 3
Animation and Interactivity

Even though the programs in Chapter 2 offer a reasonably complete survey to the classes in the **acm.graphics** package, they do so using examples that are entirely static. Running those programs causes a picture to appear in its final form. For students to get excited about graphics, it is essential to add *animation* so that the pictures evolve as the program runs and *interactivity* to give the user control over the program. This chapter introduces several strategies for implementing each of those capabilities.

As with almost every programming task, however, there are many different ways to animate a program or to get it to respond to mouse events. Some instructors will strongly prefer one style, while others will argue equally strongly for a different approach. To reach the widest possible audience, the Java Task Force chose to support several of the most popular styles and allow individual instructors to make their own choices.

Although the decision to support multiple styles seems appropriate in terms of the overall package design, it carries with it some pedagogical risks. Giving students several options for accomplishing the same task often confuses them to the point that they learn none of the strategies well. In general, it is more successful to teach one approach in detail, bringing up the possibility of alternative strategies only when students have mastered a particular approach. To avoid the same pitfalls for readers of this tutorial, we have chosen to foreground one strategy for animation and one for mouse-based interaction and to place the discussion of alternative strategies in an optional section. As you read this chapter for the first time, it probably makes sense to focus on sections 3.1 and 3.2, leaving the discussion of alternative strategies in section 3.3 for a subsequent rereading.

3.1 Graphical animation

In computer graphics, the process of updating a graphical display so that it changes over time is called **animation.** Implementing animation typically involves displaying an initial version of the picture and then changing it slightly over time so that the individual changes appear continuous from one version of the picture to the next. This strategy is analogous to classical film animation in which cartoonists break up the motion of the scene into a series of separate frames. The difference in time between each frame is called a **time step** and is typically very short. Movies, for example, typically run at 30 frames a second, which makes the time step approximately 33 milliseconds. If you want to obtain smooth motion in Java, you need to use a time step around this scale or even faster.

A simple example of animation

The easiest way to animate graphical programs is to include a loop in your **run** method that updates the picture from one frame to the next and then pauses for the duration of the time step. An example of this style of animation appears in Figure 3-1, 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 **TimesSquare** program in Figure 3-1 begins by creating a **GLabel** object and positioning it so that it is centered vertically in the window. Its starting point in the horizontal dimension, however, is just at the right edge of the canvas, which means that

Figure 3-1. Code to move text across the screen

```
/*
 * 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("Serif-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 int DELTA_X = 2;

/* 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 . . .";

}
```

the entire label is outside the visible area of the canvas. The animation is accomplished by the following lines:

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

This code loops until the label has moved entirely past the left edge of the display, shifting it **DELTA_X** pixels to the left on every time step. The call to **pause(PAUSE_TIME)** inside the loop causes the program to suspend operation for **PAUSE_TIME** milliseconds. This call is necessary to achieve the effect of animation, because computers run so quickly that the label would instantly zip off the left side of the window if you didn't slow things down.

Bouncing a ball

A slightly more sophisticated application of animation appears in Figure 3-2. This program bounces a ball around the walls of the graphics window and forms the foundation for such classic video games as Pong or Breakout. Because a static picture in

Figure 3-2. Program to bounce a ball off the boundaries of the canvas

```
/*
 * File: BouncingBall.java
 * -----------------------
 * This file implements a simple bouncing ball using the run method
 * to drive the animation.
 */

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

public class BouncingBall extends GraphicsProgram {

/** Initialize the ball and its velocity components */
   public void init() {
      ball = new GBall(BALL_RADIUS);
      add(ball, getWidth() / 2, getHeight() / 2);
      dx = 2;
      dy = 1;
   }

/** Run forever bouncing the ball */
   public void run() {
      waitForClick();
      while (true) {
         advanceOneTimeStep();
         pause(PAUSE_TIME);
      }
   }

/* Check for bounces and advance the ball */
   private void advanceOneTimeStep() {
      double bx = ball.getX();
      double by = ball.getY();
      if (bx < BALL_RADIUS || bx > getWidth() - BALL_RADIUS) dx = -dx;
      if (by < BALL_RADIUS || by > getHeight() - BALL_RADIUS) dy = -dy;
      ball.move(dx, dy);
   }

/* Private constants */
   private static final double BALL_RADIUS = 10;
   private static final int PAUSE_TIME = 20;

/* Private instance variables */
   private GBall ball;      /* The ball object               */
   private double dx;       /* Velocity delta in the x direction */
   private double dy;       /* Velocity delta in the y direction */

}
```

this text would offer little insight into how such an animated program works, it is useful to run this as an applet. If you are reading this tutorial on the JTF web site, you can bring up the applet in a separate window by clicking on the applet marker in the caption, but you can also run any of the applets from the demo site at

http://jtf.acm.org/demos/index.html

The code in Figure 3-2 uses the **GBall** class presented in Figure 2-18 to create a ball whose reference point is at the center. Doing so makes the geometric calculation simpler when checking whether a bounce occurs because all four edges can be treated symmetrically. The program code is also divided between the **init** method, which creates the ball and adds it to the window, and the **run** method, which runs the animation. The code for the **run** method is

```
public void run() {
   waitForClick();
   while (true) {
      advanceOneTimeStep();
      pause(PAUSE_TIME);
   }
}
```

which is almost precisely the paradigmatic for an animation loop. The new statement is the call to the **waitForClick** method, which is implemented by **GraphicsProgram** and suspends the program until a mouse click occurs in the graphics canvas. This call means that the program does not start up immediately, but instead waits for a mouse click before proceeding.

The code that implements the underlying physics of the animation appears in the private method **advanceOneTimeStep**. This method checks to see whether the ball has reached one of the edges of the canvas, in which case it changes the sign of the appropriate component of the ball's velocity, which is stored in the variables **dx** and **dy**. It then moves the ball by those displacements to update its position on the display.

Simulating randomness in animations

As written, the bouncing ball program from the preceding section is altogether too predictable. The ball begins with a constant velocity and then makes perfectly reflective bounces off the edges of the canvas, tracing the same trajectory each time. Many animated programs will involve some kind of random behavior, and students will quickly want to know how they can implement random processes in their own code.

Although it is certainly possible to use either the **Math.random** method or the **Random** class in **java.util** for this purpose, there are pedagogical advantages to using the **RandomGenerator** class in the **acm.util** package instead. Most importantly, he name of the class emphasizes that a **RandomGenerator** object is a *generator* for random values and not a random value in itself. When students use the **Random** class, they are much more likely to create a new **Random** instance for each value they wish to generate. In addition, the **RandomGenerator** class offers several additional methods that are often much easier to use than those in the base class. These extended methods are listed in Figure 3-5.

The conventional pattern for using the **RandomGenerator** class is to declare and initialize an instance variable to hold the generator using the line

```
private RandomGenerator rgen = RandomGenerator.getInstance();
```

Once this declaration is made, every method in this class can then generate new random values by invoking the appropriate method on the **rgen** variable. For example, you could use this strategy in the **BouncingBall** program to initialize each velocity component of the ball to a random value between –3 and 3:

```
dx = rgen.nextDouble(-3, 3);
dy = rgen.nextDouble(-3, 3);
```

Figure 3-5. Useful methods in the `RandomGenerator` class

Factory method
`static RandomGenerator getInstance()` Returns a standard random generator.
Methods inherited from the `Random` class in `java.util`
`int nextInt(int n)` Returns a random integer chosen from the **`n`** values in the range 0 to **`n`** - 1, inclusive.
`double nextDouble()` Returns a random **`double`** *d* in the range 0 ≤ *d*< 1.
`void nextBoolean()` Returns a random **`boolean`** that is **`true`** approximately 50% of the time.
`void setSeed(long seed)` Sets a "seed" to indicate a starting point for the pseudorandom sequence.
Additional methods defined by `RandomGenerator`
`int nextInt(int low, int high)` Returns a random integer in the specified range (inclusive).
`double nextDouble(double low, double high)` Returns a random **`double`** in the specified range.
`boolean nextBoolean(double p)` Returns a random **`boolean`** that is **`true`** with probability **`p`** (0 = never, 1 = always).
`Color nextColor()` Returns a random opaque color.

The **`RandomShapes`** program in Figure 3-6 makes more extensive use of the facilities of the **`RandomGenerator`** class. The program generates ten shapes and positions them on the canvas using randomness in each of the following ways:

- The shapes are randomly chosen to be rectangles, ovals, or stars. The stars are represented internally using the **`GStar`** class defined in Figure 2-15 from Chapter 2.
- The shapes are given a random size that ranges between **`MIN_SIZE`** and **`MAX_SIZE`** in each dimension.
- The shapes are positioned randomly on the canvas subject to the condition that the entire shape must fit inside the boundaries.
- The shape is filled in a random color.

A sample run of the **`RandomShapes`** program might look like this:

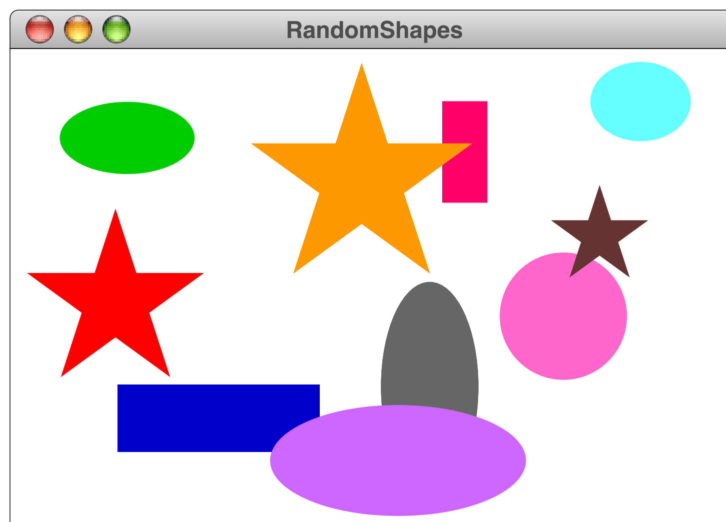


Figure 3-6. Program to generate random shapes

```
/*
 * File: RandomShapes.java
 * -----------------------
 * This file creates ten boxes, ovals, and stars at random locations
 * on the screen, pausing for a suitable interval between each one.
 */

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

public class RandomShapes extends GraphicsProgram {

/** Runs the program */
   public void run() {
      while (true) {
         for (int i = 0; i < NOBJECTS; i++) {
            addOneRandomShape();
            pause(PAUSE_TIME);
         }
         waitForClick();
         removeAll();
      }
   }

/* Adds a random shape to the canvas */
   private void addOneRandomShape() {
      GObject gobj = createRandomShape();
      gobj.setColor(rgen.nextColor());
      if (gobj instanceof GFillable) ((GFillable) gobj).setFilled(true);
      double x = rgen.nextDouble(0, getWidth() - gobj.getWidth())
                    - gobj.getBounds().getX();
      double y = rgen.nextDouble(0, getHeight() - gobj.getHeight())
                    - gobj.getBounds().getY();
      add(gobj, x, y);
   }

/* Generates a random shape whose reference point is the origin */
   private GObject createRandomShape() {
      double width = rgen.nextDouble(MIN_SIZE, MAX_SIZE);
      double height = rgen.nextDouble(MIN_SIZE, MAX_SIZE);
      switch (rgen.nextInt(3)) {
        case 0: return new GRect(width, height);
        case 1: return new GOval(width, height);
        case 2: return new GStar(width);
        default: throw new ErrorException("Illegal shape index");
      }
   }

/* Private constants */
   private static final int NOBJECTS = 10;
   private static final int PAUSE_TIME = 1000;
   private static final double MIN_SIZE = 25;
   private static final double MAX_SIZE = 150;

/* Private instance variables */
   private RandomGenerator rgen = RandomInteger.getInstance();
}
```

Most of **RandomShapes** program in Figure 3-6 is reasonably straightforward, but there are nonetheless a few aspects of the code that are easier to understand with some additional explanation:

- The code for the **run** method includes a **while** loop that allows the user to generate a new set of shapes by clicking the mouse. The **waitForClick** method was introduced earlier in the chapter in the discussion of the bouncing ball programs and simply waits for a mouse click.
- The calculation of the random coordinate positions seems slightly more complex that necessary. At first glance, it would seem as if one could ensure that the entire figure was inside the canvas by writing

```
double x = rgen.nextDouble(0, getWidth() - gobj.getWidth());
double y = rgen.nextDouble(0, getHeight() - gobj.getHeight());
```

  While that code would be sufficient for the **GRect** and **GOval** objects that have their reference point in the upper left corner, it doesn't work for figures like **GStar** for which the reference point is inside the figure. The **getBounds** method returns the actual bounding box of the figure, which means that **gobj.getBounds().getX()** returns the actual *x* coordinate of the left edge of the figure. You can make sure that the figure fits on the screen by adjusting the coordinates to compensate for the shift in origin.

The **RandomGenerator** class from the **java.util** class has applications in a wide variety of contexts beyond graphical animation. In our experience it far and away the most widely used class in the **java.util** package.

3.2 Interactivity

The animation capability presented in the preceding section certainly helps to make graphical programs more exciting, but it is not in itself sufficient to implement the kind of interactive graphical applications that today's students have come to expect. Interactive programs must also respond to actions taken by the user. The sections that follow outline the Java event model and describe one strategy for responding to those events. Several other paradigms for event handling are described in section 3.3.

The Java event model

Programs like **Add2Console** that request input from the user are interactive programs of a sort. Console programs, however, ask the user for input only at certain well-defined points in the program's execution history when the program makes an explicit call to an input method like **readInt**. This style of interaction is called **synchronous,** because it is always 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.

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.** In Java, objects that listen for user-interface events do so by implementing the methods in a specific **listener interface,** which is typically 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 other asynchronous events. The examples in the next several examples concentrate on the interfaces that define how programs respond to mouse events, which are described in the following section.

Responding to mouse events

The **java.awt.event** package defines two separate interfaces—**MouseListener** and **MouseMotionListener**—that specify how a program responds to mouse events. The **MouseListener** methods are called in response to actions that occur relatively infrequently, such as pressing a mouse button or moving the mouse entirely outside the boundary in which the listener is active. The **MouseMotionListener** methods are called whenever the mouse moves, which happens much more frequently. Moving the mouse without pressing the button results in calls to **mouseMoved**; dragging the mouse with the button down results in calls to **mouseDragged**. The methods in each interface are listed in Figure 3-3.

Each of the methods listed in Figure 3-3 takes as its argument an object of type **MouseEvent**, which is defined in the package **java.awt.event**, just as the listener interfaces are. The **MouseEvent** class includes a rich set of methods for designing sophisticated user interfaces. For most applications, however, you can get away with using only two of those methods. Given a **MouseEvent** stored in a variable named **e**, you can determine the location at which the mouse even occurred by calling **e.getX()** and **e.getY()**.

The **GraphicsProgram** class declares itself to be both a **MouseListener** and a **MouseMotionListener** by defining implementations for each of the listener methods in those interfaces. Those implementations, however, do nothing at all. For example, the default definition of **mouseClicked** is simply

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

Thus, unless you override the definition of **mouseClicked** in your **GraphicsProgram** subclass, it will simply ignore mouse clicks, just as it ignores all the other mouse events. If, however, you define a new **mouseClicked** method, the event handling system will call your version instead of the empty one. Because any methods that you don't override continue to do what they did by default (i.e., nothing), you only have to override the listener methods you need.

Whenever you write event-handling code in Java, it is important to remember that defining the listener methods is not sufficient in itself to establish the listener

Figure 3-3. Methods in the MouseListener and MouseMotionListener interfaces

The MouseListener interface

Method	Description
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.

The MouseMotionListener interface

Method	Description
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.

relationship. You also need to make sure that the object that is listening for events adds itself as a listener to the object that is generating the events. In the case of a **GraphicsProgram**, the program is doing the listening, and the embedded **GCanvas** is generating the events. You therefore need to have the program register its interest in events generated by the canvas by executing the following lines in the context of the program:

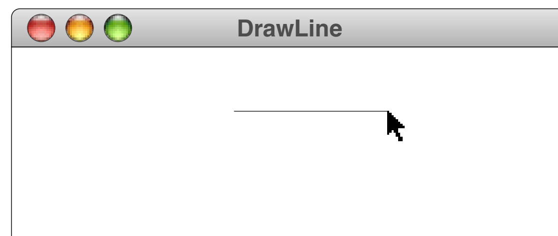
```
getCanvas().addMouseListener(this);
getCanvas().addMouseMotionListener(this);
```

To make this operation just a little bit simpler—and to avoid having to explain the **getCanvas** method and the keyword **this**—the **GraphicsProgram** class includes a method **addMouseListeners** that performs precisely these two steps. The examples in the subsections that follow make use of this simplified form.

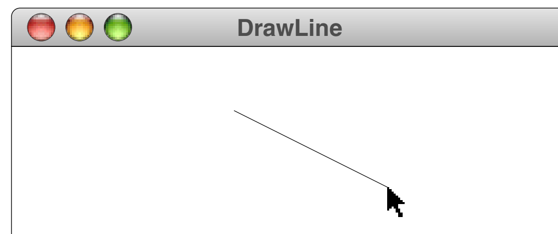
A line-drawing program

The first example of mouse interaction is a simple line-drawing program that operates—at least for straight lines—in the way that painting programs like Adobe Illustrator™ does. To create a line on the canvas, you press the mouse at its starting point. From there, you hold the mouse button down and drag it to the other endpoint. As you do so, the line keeps itself updated on the canvas so that it connects the starting point with the current position of the mouse.

As an example, suppose that you press the mouse button somewhere on the screen and then drag it rightward an inch, holding the button down. What you'd like to see is the following picture:



If you then move the mouse downward without releasing the button, the displayed line will track the mouse, so that you might see the following picture:



When you release the mouse, the line stays where it is, and you can go ahead and draw additional lines using the same sequence of operations.

Because the line joining the initial point and the mouse stretches and contracts as you move the mouse, this technique is called **rubber-banding.** The code for a line-drawing program that uses rubber-banding appears in Figure 3-4. Despite the fact that the program seems to perform a reasonably interesting task, the code is surprisingly short. The bodies of the three methods in the program contain a grand total of four lines. Even so, it is worth going through each of the methods in turn.

Figure 3-4. Program to create a line drawing on the screen

```
/*
 * File: DrawLine.java
 * -------------------
 * This program allows users to create lines on the graphics
 * canvas by clicking and dragging with the mouse.  The line
 * is redrawn from the original point to the new endpoint, which
 * makes it look as if it is connected with a rubber band.
 */

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

/** This class allows users to draw lines on the canvas */
public class DrawLine extends GraphicsProgram {

/** Initializes the program by enabling the mouse listeners */
   public void init() {
      addMouseListeners();
   }

/** Called on mouse press to create a new line */
   public void mousePressed(MouseEvent e) {
      line = new GLine(e.getX(), e.getY(), e.getX(), e.getY());
      add(line);
   }

/** Called on mouse drag to reset the endpoint */
   public void mouseDragged(MouseEvent e) {
      line.setEndPoint(e.getX(), e.getY());
   }

/* Private instance variables */
   private GLine line;

}
```

The first thing to notice is that this program contains an **init** method rather than a **run** method. In this particular case, you could call the method by either name and have the program run in exactly the same way, but you will soon encounter situations in which you need to be clear about the role of these two methods. Even though both are called as part of the program startup process, the two methods serve different conceptual purposes. The **init** method is intended for startup operations that are executed before the program starts; the **run** method is executed as part of the program operation. In the examples of animation earlier in the chapter, the **run** method implemented the animation. In this program, nothing is actually running after the program starts up. The only time things happen is when the user presses the mouse button and begins to drag it across the screen. Such programs are said to be **event-driven.** Event-driven programs tend to operate by performing some amount of initialization and then waiting for events to occur. In this case, the only initialization necessary is to enable the program as a listener for events, which is accomplished through the call to **addMouseListeners**.

The **mousePressed** method is called whenever the user presses the mouse button and overrides the empty definition implemented by the **GraphicsProgram** class itself. In the

line-drawing program, the body of the **mousePressed** method simply creates a new **GLine** object that starts and ends at the current mouse position. This **GLine** appears on the canvas as a dot.

The **GLine** is stored in the private instance variable **line**, which means that other methods in the class have access to it. In particular, dragging the mouse with the button down calls the **mouseDragged** method, which resets the endpoint of the line to the current mouse position.

Dragging objects on the canvas

The second example is a bit more sophisticated but still fits easily on a single page. The **DragObjects** program in Figure 3-5 illustrates how to use mouse listeners to support dragging graphical objects around on the canvas. The code in the **init** method should seem familiar, given that its effect is to create two graphical objects—the red rectangle and the green oval from the **FeltBoard** program in Chapter 2—and then add them to the canvas.

The code in Figure 3-5 overrides three of the event methods. The first of these 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 instance variables **lastX** and **lastY**. The final 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 where 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 stacking order. If there are no objects at all at the specified location, **getElementAt** returns the value **null**.

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**, no object is being dragged, so the rest of the method is skipped. If an object has been selected by a previous call to **mousePressed**, the **mouseDragged** method needs to move that object by some displacement in each direction. That displacement, however, does not depend on the absolute location of the mouse but rather in how far it has moved from the point at which you last updated the location of the object. 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 updated the location of the

Figure 3-5. Object-dragging program using the `addMouseListeners` method

```
/*
 * File: DragObjects.java
 * ----------------------
 * This implementation illustrates the technique of using the
 * addMouseListeners method to register the program itself as
 * a listeners for events in the underlying GCanvas.
 */

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 {

/** Initializes the program */
   public void init() {
      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) {
      last = new GPoint(e.getPoint());
      gobj = getElementAt(last);
   }

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

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

/* Private instance variables */
   private GObject gobj;           /* The object being dragged */
   private GPoint last;            /* The last mouse position  */
}
```

object being dragged, you 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 3-5 is **mouseClicked**, which looks like this:

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

The effect 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 only subtlety in this method is the question of whether it is appropriate to rely on the proper initialization of the variable **gobj**, which holds the current object. As it happens, 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.

3.3 Alternative strategies for animation and interactivity (optional)

One of the interesting discoveries that we made during the period of review and comment on the intermediate Java Task Force designs was that people teaching introductory programming courses have strongly held beliefs about how Java programs should be coded and how those programs should be presented to students. To the extent that our approach differs from the style that someone has grown accustomed to, our designs are often seen as being contrary to the spirit of Java—at least in that person's mind. Unfortunately, those reactions did not point in a single direction because those strongly held views diverge widely. For example, some people argue that the only appropriate way to declare a listener method is to use an anonymous inner class, while others have held that exposing Java's listener mechanism at all will be too confusing for students.

From these reactions, it became clear that the Java Task Force packages had to support multiple coding styles and allow individual instructors to choose the strategy that seems most closely aligned with their overall pedagogical approach. The purpose of this section is to describe several different approaches to animation and interactivity so that you can have a better sense of the range of options. Those alternative strategies are illustrated by recoding two of the example programs presented earlier in this chapter—the **BouncingBall** program from Figure 3-2 and the **DragObjects** program from Figure 3-5—using several different strategies. The code for each of these version is available on the JTF website.

Although we believe that it is important for the Java Task Force packages to support a range of coding strategies, it is probably not a good idea to try to cover all of these strategies in an introductory course. Many students find that having multiple strategies to accomplish the same task is more confusing than liberating. Thus, it is probably best to choose a particular approach to animation or event handling and then stick with that model until students gain enough experience to appreciate the strengths and weaknesses of the alternative styles.

Alternative strategies for implementing animation

The animated applications in section 3.1 use the **run** defined in the **Program** class to drive the animation. The task of dividing the animation into discrete time steps is accomplished by making periodic calls to **pause**. Because the **run** method runs in a thread of its own, calling **pause** does not disable system tasks that run, for example, on Java's event-handling thread. The remainder of this section describes two alternative

animation strategies using the **BouncingBall** program from Figure 3-2 as a common point of departure.

The first of these strategies involves giving the ball a thread of its own. The pedagogical foundation for this approach lies in the belief that students of modern programming need to learn about concurrency at a much earlier stage. Giving the ball its own thread makes it easy to see the ball as an active entity in a concurrent world. In this conceptual model, the ball is moving of its own accord rather than being moved by the program.

One possible implementation of this strategy appears in Figures 3-8 and 3-9. Figure 3-8 shows the main program, but all the real work takes place in the **RunnableGBall** class shown in Figure 3-9, which extends the **GBall** class. The **RunnableGBall** class implements Java's **Runnable** interface so that it can serve as the basis for an independent thread of control. The code for that **run** method has the same steps as in the original implementation:

```
public void run() {
   while (true) {
      advanceOneTimeStep();
      pause(PAUSE_TIME);
   }
}
```

Figure 3-8. Ball bouncing program using a separate thread

```
/*
 * File: BouncingBallUsingThreads.java
 * -----------------------------------
 * This file implements a simple bouncing ball by creating
 * a RunnableBall class and executing it in its own thread.
 */

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

public class BouncingBallUsingThreads extends GraphicsProgram {

/** Initialize the ball and its velocity components */
   public void init() {
      ball = new RunnableGBall(BALL_RADIUS);
      ball.setEnclosureSize(getWidth(), getHeight());
      ball.setVelocity(2, 1);
      add(ball, getWidth() / 2, getHeight() / 2);
   }

/** Create a thread to bounce the ball */
   public void run() {
      waitForClick();
      new Thread(ball).start();
   }

/* Private constants */
   private static final double BALL_RADIUS = 10;

/* Private instance variables */
   private RunnableGBall ball;
}
```

Figure 3-9. The `RunnableGBall` class

```
/*
 * File: RunnableGBall.java
 * ------------------------
 * This file defines an extension to the GBall class that is
 * designed to run as a separate thread of control.
 */

import acm.graphics.*;

public class RunnableGBall extends GBall implements Runnable {

/** Creates a new ball with radius r centered at the origin */
   public RunnableGBall(double r) {
      super(r);
   }

/** Sets the size of the enclosure */
   public void setEnclosureSize(double width, double height) {
      enclosureWidth = width;
      enclosureHeight = height;
   }

/** Sets the velocity of the ball */
   public void setVelocity(double vx, double vy) {
      dx = vx;
      dy = vy;
   }

/** Run forever bouncing the ball */
   public void run() {
      while (true) {
         advanceOneTimeStep();
         pause(PAUSE_TIME);
      }
   }

/* Check for bounces and advance the ball */
   private void advanceOneTimeStep() {
      double bx = getX();
      double by = getY();
      double r = getWidth() / 2;
      if (bx < r || bx > enclosureWidth - r) dx = -dx;
      if (by < r || by > enclosureHeight - r) dy = -dy;
      move(dx, dy);
   }

/* Private constants */
   private static final int PAUSE_TIME = 20;

/* Private instance variables */
   private double enclosureWidth;
   private double enclosureHeight;
   private double dx;
   private double dy;

}
```

In this case, however, the thread that executes this method is associated with the ball as opposed to being part of the main program. All the **BouncingBallUsingThreads** program does on its own behalf is to create the runnable ball, initialize various properties such as the speed and dimensions of the boundary enclosure, and then start up a separate thread for the ball by calling

```
new Thread(ball).start();
```

At first glance, it would seem that this strategy is better for applications in which there is more than one animated object. Given that any **RunnableBall** object can have a thread of its own, it would be simple to create a second ball, add that to the canvas, and start it running as well. As it happens, however, that strategy is difficult to manage because there is no way to ensure that the balls move at the same rate. The **pause** method is only approximate in its timing. Depending on the system load, it would be possible for one ball to advance through several time steps before the other had a chance to move at all. To avoid this problem, it is often preferable to have a single animation thread that updates the position of all moving objects during each time step.

The second alternative strategy for animation abandons the idea of pausing a thread altogether. The code for the **BouncingBallUsingTimerCode** in Figure 3-10 uses Swing's **Timer** class to alert the main program at regular intervals. When the timer goes off, the program can advance the ball's position by one time step. Although the idea behind this strategy is simple enough, a couple of aspects of the code are worth noting:

- The version of the **Timer** class used here is called **SwingTimer**, which is defined in the **acm.util** package. **SwingTimer** is a simple extension of **javax.swing.Timer** with absolutely no additional features beyond those provided by the base class. The reason for including the **SwingTimer** class in the JTF package collection is to avoid the unfortunate ambiguity that was introduced into Java in JDK 1.3. There are now *two* publicly accessible classes named **Timer**, one in **javax.swing** and the other in **java.util**. If a Java program imports both of these packages, the compiler cannot resolve the identity of the **Timer** class unless it is specifically imported from one package or the other. Using the class name **SwingTimer** eliminates the ambiguity and makes it obvious that the timers in question are of the **javax.swing** variety.
- The code for responding to the events generated by **SwingTimer** specifies the necessary **ActionListener** using an anonymous inner class. The definition of both the listener and its response appear in the lines

```
ActionListener listener = new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      advanceOneTimeStep();
   }
};
```

  Using anonymous inner classes to define listeners has become standard in modern Java code. Because the definition of the anonymous listener class is nested within the body of **BouncingBallUsingTimer**, it has access to the methods and fields defined in the public class, making it possible to invoke **advanceOneTimeStep** from inside the listener object. At the same time, there is considerable disagreement within the Java education community over when to introduce such classes to new students. Understanding the semantics of inner classes can be difficult for many students. Although it is essential to cover this capability eventually, the Task Force felt it was important to support at least some models that enabled instructors to avoid the use of inner classes during the early weeks of an introductory course.

Figure 3-10. Ball bouncing program using timer events

```
/*
 * File: BouncingBallUsingTimer.java
 * ---------------------------------
 * This file implements a simple bouncing ball using a Timer to
 * implement the animation.
 */

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

public class BouncingBallUsingTimer extends GraphicsProgram {

/** Initialize the ball and its velocity components */
   public void init() {
      ball = new GBall(BALL_RADIUS);
      add(ball, getWidth() / 2, getHeight() / 2);
      dx = 2;
      dy = 1;
   }

/** Create a timer to advance the ball */
   public void run() {
      waitForClick();
      ActionListener listener = new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            advanceOneTimeStep();
         }
      };
      SwingTimer timer = new SwingTimer(TIMER_RATE, listener);
      timer.start();
   }

/* Check for bounces and advance the ball */
   private void advanceOneTimeStep() {
      double bx = ball.getX();
      double by = ball.getY();
      if (bx < BALL_RADIUS || bx > getWidth() - BALL_RADIUS) dx = -dx;
      if (by < BALL_RADIUS || by > getHeight() - BALL_RADIUS) dy = -dy;
      ball.move(dx, dy);
   }

/* Private constants */
   private static final double BALL_RADIUS = 10;
   private static final int TIMER_RATE = 20;

/* Private instance variables */
   private GBall ball;
   private double dx;
   private double dy;

}
```

Alternative strategies for responding to mouse events

Just as there is more than one way to implement animation, there are also multiple approaches that one can take to respond to mouse events. In addition to the strategy of calling the **addMouseListeners** method to register the program itself as a listener, the Java Task Force packages support several additional coding styles, each of which has its own strengths and weaknesses. The next few paragraphs describe three additional approaches in the context of the **DragObjects** example from Figure 3-7

The **DragUsingInnerClasses** program shown in Figure 3-11 offers the most straightforward rewrite of the original version. The only change is that the mouse listeners are now supplied using anonymous inner classes instead of having the program itself assume that role. The advantage of this structure is that it corresponds most closely to the style that has become standard in the Java community. The disadvantage is the additional conceptual overhead involved in presenting inner classes to students. In a way, the situation is even more problematic here than it was in the case of the **BouncingBallUsingTimer** program presented in the preceding section. In that model, it was possible to use **ActionListener** as the base class for the listener, because the one method the interface specifies is defined in the body of the inner class. In the object-dragging example, the base classes need to be **MouseAdapter** and **MouseMotionAdapter** to ensure that all the methods in the corresponding interfaces are defined.

The **DragUsingGObjectEvents** program in Figure 3-12 offers a model that initially seems similar to the original implementation but that actually represents an important change in point of view. In this implementation, the listeners are attached to the individual **GObject**s and not to the canvas. When a mouse event occurs in the screen area of a **GObject**, the code for the **acm.graphics** package generates mouse events that use the **GObject** itself as the source of the event and which are then forwarded to any listeners registered for that object. The advantage here is that the model supports the notion that objects are active entities that can both generate and accept messages from other objects. The disadvantage lies in the fact that many applications will also need to assign a listener to the canvas to respond to events that occur outside the context of any of the graphical objects currently being displayed. If the canvas listener is required in any case, it seems easiest to use it for all event handling rather than to adopt two separate models.

The final version of the object-dragging program appears in Figure 3-13. This strategy is derived from the **objectdraw** package developed at Williams College and uses a simpler model in which the **acm.graphics** code forwards events to a set of specialized event handlers defined specifically for this purpose. If a **GraphicsProgram** subclass defines any of the methods

```
mousePressed(GPoint pt)
mouseReleased(GPoint pt)
mouseClicked(GPoint pt)
mouseMoved(GPoint pt)
mouseDragged(GPoint pt)
```

then that method is called whenever the appropriate event occurs in the **GCanvas**. The parameter **pt** in each of these methods is the point at which the mouse event occurred, already translated into the real-valued coordinate space of the **acm.graphics** package. This model completely hides the details of mouse events and mouse listeners, so that the student need not, for example, import the **java.awt.event** package or take any special steps to register the program as a listener. All of that comes for free. The primary disadvantage is that students who learn this strategy for event handling will have to learn how standard Java listeners work at some later point.

Figure 3-11. Object-dragging program using inner classes to specify the listeners

```
/*
 * File: DragUsingInnerClasses.java
 * --------------------------------
 * This implementation illustrates the technique of defining
 * listeners as anonymous inner classes.
 */

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

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

/** Initializes the program */
   public void init() {
      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);
      GCanvas canvas = getGCanvas();
      canvas.addMouseListener(new MouseAdapter() {
         public void mousePressed(MouseEvent e) {
            last = new GPoint(e.getPoint());
            gobj = getElementAt(last);
         }

         public void mouseClicked(MouseEvent e) {
            if (gobj != null) gobj.sendToFront();
         }
      });
      canvas.addMouseMotionListener(new MouseMotionAdapter() {
         public void mouseDragged(MouseEvent e) {
            if (gobj != null) {
               gobj.move(e.getX() - last.getX(),
                         e.getY() - last.getY());
               last = new GPoint(e.getPoint());
            }
         }
      });
   }

/* Private instance variables */
   private GObject gobj;          /* The object being dragged */
   private GPoint last;           /* The last mouse position  */
}
```

Figure 3-12. Object-dragging program that listens to the `GObjects`

```
/*
 * File: DragUsingGObjectEvents.java
 * ---------------------------------
 * This implementation illustrates the technique of assigning
 * listeners to GObjects.
 */

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

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

/** Initializes the program */
   public void init() {
      GRect rect = new GRect(100, 100, 150, 100);
      rect.setFilled(true);
      rect.setColor(Color.RED);
      rect.addMouseListener(this);
      rect.addMouseMotionListener(this);
      add(rect);
      GOval oval = new GOval(300, 115, 100, 70);
      oval.setFilled(true);
      oval.setColor(Color.GREEN);
      oval.addMouseListener(this);
      oval.addMouseMotionListener(this);
      add(oval);
   }

/** Called on mouse press to record the coordinates of the click */
   public void mousePressed(MouseEvent e) {
      last = new GPoint(e.getPoint());
   }

/** Called on mouse drag to reposition the object */
   public void mouseDragged(MouseEvent e) {
      GObject gobj = (GObject) e.getSource();
      gobj.move(e.getX() - last.getX(), e.getY() - last.getY());
      last = new GPoint(e.getPoint());
   }

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

/* Private instance variables */
   private GPoint last;             /* The last mouse position */
}
```

Figure 3-13. Object-dragging program using callback methods in the style of objectdraw

```
/*
 * File: DragUsingObjectDrawModel.java
 * -----------------------------------
 * This implementation illustrates the technique of using callback
 * methods in the style of the objectdraw package.
 */

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

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

/** Initializes the program */
   public void init() {
      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);
   }

/** Called on mouse press to record the coordinates of the click */
   public void mousePressed(GPoint pt) {
      last = pt;
      gobj = getElementAt(last);
   }

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

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

/* Private instance variables */
   private GObject gobj;           /* The object being dragged */
   private GPoint last;            /* The last mouse position  */
}
```

Chapter 4
Graphical User Interfaces

One of the most exciting things about coding in Java is that the standard libraries include a large number of tools for creating applications with sophisticated graphical user interfaces, usually referred to as **GUI**s. The Swing package, for example, offers a large set of interactor classes that support buttons, text fields, selectable lists, sliders, and much more. Many instructors who have taught Java at the introductory level, however, report that GUI programming is difficult for beginners, which makes it harder to take advantage of the many attractive features that Java offers.

4.1 Adding interactors to the borders of a program

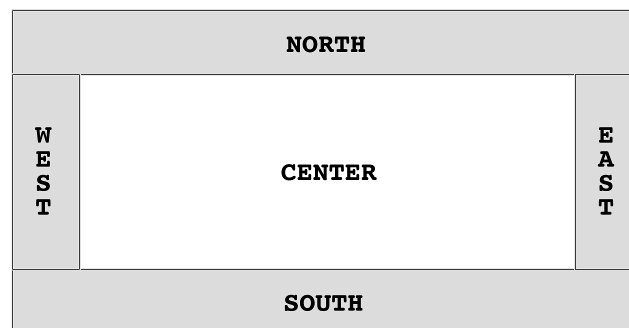
To make it possible for students to create simple GUI applications with a minimum of conceptual overhead, the **acm.program** package makes it easy to add Java interactors along the borders of any **Program** subclass. The usual approach is to pick one of the borders and add several interactors there, creating a control strip that allows the user to control the operation of the program.

Layout strategy for border interactors

As an example, suppose that you want to write a program that displays two buttons—**Start** and **Stop**—at the bottom of a program window. Let's ignore for the moment what those buttons actually do and concentrate instead on how to make them appear. If you use the standard layout management tools provided by the **Program** class, all you have to do is include the following code as part of the **init** method:

```
add(new JButton("Start"), SOUTH);
add(new JButton("Stop"), SOUTH);
```

The constant **SOUTH** indicates the bottom of the window and represents one of four border regions that are automatically created as part of the initialization of any **Program** subclass. Those four regions are the ones defined in the standard **BorderLayout** class and are arranged like this:



Each border region is initially empty. Empty regions take up no space, so that a particular region does not actually appear until you add an interactor or some other Java component to it. The **NORTH** and **SOUTH** regions arrange the interactors horizontally; the **WEST** and **EAST** regions arrange them vertically.

Assigning action listeners to the buttons

Creating the buttons, however, accomplishes only part of the task. To make the buttons active, you need to give each one an action listener so that pressing the button performs

the appropriate action. These days, the most common programming style among experienced Java programmers is to assign an individual action listener to each button in the form of an anonymous inner class. Suppose, for example, that you want the **Start** and **Stop** buttons to invoke methods called **startAction** and **stopAction**, respectively. You could do so by changing the initialization code as follows:

```
JButton startButton = new JButton("Start");
startButton.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      startAction();
   }
});
add(startButton, SOUTH);
JButton stopButton = new JButton("Start");
stopButton.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      stopAction();
   }
});
add(stopButton, SOUTH);
```

Although there are instructors who favor this style even at the level of introductory courses, the members of the Java Task Force remain concerned that this coding style introduces too many unfamiliar concepts for novice programmers to comprehend. To simplify the structure and eliminate the use of inner classes, the Task Force chose to designate the **Program** class as an **ActionListener**. Adopting this strategy means that the initialization of the buttons can be shortened to

```
JButton startButton = new JButton("Start");
startButton.addActionListener(this);
add(startButton, SOUTH);
JButton stopButton = new JButton("Start");
stopButton.addActionListener(this);
add(stopButton, SOUTH);
addActionListeners();
```

When using the program as an action listener, every button in the application triggers the same **actionPerformed** method, which must now look at the event to determine what button triggered the action. One approach is to call **e.getActionCommand()** method, which returns an "action command" string that, by default, is the label that appears on the button. This strategy is illustrated in the following **actionPerformed** implementation:

```
public void actionPerformed(ActionEvent e) {
   if (e.getActionCommand().equals("Start")) {
      startAction();
   } else if (e.getActionCommand().equals("Stop")) {
      stopAction();
   } else
}
```

An alternative strategy would be to call **e.getSource()** to obtain the identity of the button that triggered the event. That strategy, however, would be useful only if you had chosen to store the button objects in instance variables so that you could compare them against the source of the event.

As a further simplification, the **Program** class includes an **addActionListeners** method that recursively traverses the components displayed on the screen and adds the program as an action listener for every **JButton** it encounters. This style means that the

initialization of the buttons can be shortened even more dramatically so that the code requires only the following lines:

```
add(new JButton("Start"), SOUTH);
add(new JButton("Stop"), SOUTH);
```

A simple example

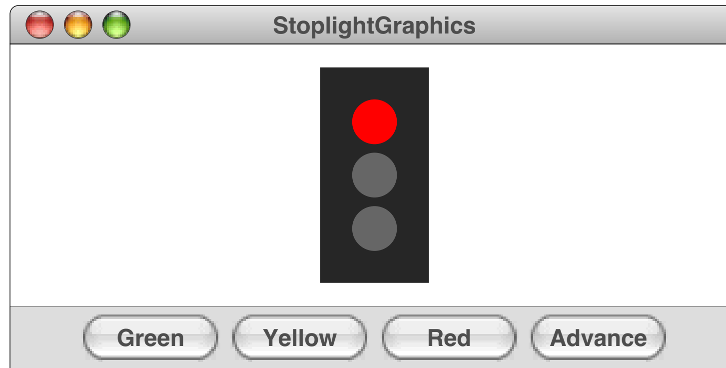
As an illustration of how to place interactors along the program border, take a look at the **StoplightConsole** program in Figure 4-1. This program creates three buttons—**Green**, **Yellow**, and **Red**—and places them along the bottom of a console window. In this version of the program, pressing a button simply prints out the label of the button. For example, pressing the three buttons in order from left to right would generate the following output on the display:

Although it is perfectly fine as an illustration of how to create a control strip along the bottom edge of the program window, the **StoplightConsole** program isn't particularly

Figure 4-1. Code for the console-based stoplight

```
/*
 * File: StoplightConsole.java
 * ---------------------------
 * This program illustrates the construction of a simple GUI.
 */

import acm.program.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * This class displays three buttons at the south edge of the window.
 * The name of the button is echoed on the console each time a button
 * is pressed.
 */
public class StoplightConsole extends ConsoleProgram {

/** Initialize the GUI */
   public void init() {
      add(new JButton("Green"), SOUTH);
      add(new JButton("Yellow"), SOUTH);
      add(new JButton("Red"), SOUTH);
      addActionListeners();
   }

/** Listen for a button action */
   public void actionPerformed(ActionEvent e) {
      println(e.getActionCommand());
   }
}
```

exciting as an application, largely because it is console based. The ability to place interactors around the border of a program is even more equally useful with the other **Program** subclasses. The code in Figures 4-2 and 4-3 shows a similar application redesigned as a **GraphicsProgram** in which the stoplight is represented graphically on the display, like this:



The **Stoplight** class shown in Figure 4-3 extends **GCompound** to create an object that responds to the messages **setState(color)** and **advance()**.

Figure 4-2. A GraphicsProgram version of a stoplight

```
/*
 * File: StoplightGraphics.java
 * ----------------------------
 * This program illustrates the construction of a simple GUI using a
 * GraphicsProgram as the main class.
 */

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

/**
 * This class displays four buttons at the south edge of the window.
 * Pressing a button lights the indicated lamp in the stoplight or
 * advances the stoplight to its next configuration.
 */

public class StoplightGraphics extends GraphicsProgram {

/** Initialize the buttons and create the stoplight */
   public void init() {
      add(new JButton("Green"), SOUTH);
      add(new JButton("Yellow"), SOUTH);
      add(new JButton("Red"), SOUTH);
      add(new JButton("Advance"), SOUTH);
      signal = new Stoplight();
      add(signal, getWidth() / 2, getHeight() / 2);
      addActionListeners();
   }
```

Figure 4-2. A `GraphicsProgram` version of a stoplight (continued)

```
/** Listen for a button action */
   public void actionPerformed(ActionEvent e) {
      String command = e.getActionCommand();
      if (command.equals("Advance")) {
         signal.advance();
      } else if (command.equals("Red")) {
         signal.setState(Stoplight.RED);
      } else if (command.equals("Yellow")) {
         signal.setState(Stoplight.YELLOW);
      } else if (command.equals("Green")) {
         signal.setState(Stoplight.GREEN);
      }
   }

/* Private instance variables */
   private Stoplight signal;
}
```

Figure 4-3. Graphical implementation of the `Stoplight` class

```
/*
 * File: Stoplight.java
 * --------------------
 * This class implements a stoplight as a compound graphical object.
 */

import acm.graphics.*;
import acm.util.*;
import java.awt.*;

/**
 * This class represents a graphical stoplight with its origin point
 * at the center.
 */

public class Stoplight extends GCompound {

/* Public constants for the colors */
   public static final Color RED = Color.RED;
   public static final Color YELLOW = Color.YELLOW;
   public static final Color GREEN = Color.GREEN;

/** Creates a new Stoplight object, which is initially red */
   public Stoplight() {
      GRect frame = new GRect(STOPLIGHT_WIDTH, STOPLIGHT_HEIGHT);
      frame.setFilled(true);
      frame.setColor(Color.DARK_GRAY);
      add(frame, -STOPLIGHT_WIDTH / 2, -STOPLIGHT_HEIGHT / 2);
      redLamp = createLamp(0, -STOPLIGHT_HEIGHT / 4);
      yellowLamp = createLamp(0, 0);
      greenLamp = createLamp(0, STOPLIGHT_HEIGHT / 4);
      add(redLamp);
      add(yellowLamp);
      add(greenLamp);
      setState(RED);
   }
```

Figure 4-3. Graphical implementation of the `Stoplight` class (continued)

```
/** Changes the state of the stoplight to the indicated color */
   public void setState(Color color) {
      state = color;
      redLamp.setColor((state == RED) ? RED : Color.GRAY);
      yellowLamp.setColor((state == YELLOW) ? YELLOW : Color.GRAY);
      greenLamp.setColor((state == GREEN) ? GREEN : Color.GRAY);
   }

/** Returns the current state of the stoplight */
   public Color getState() {
      return state;
   }

/** Advances the stoplight to the next state */
   public void advance() {
      if (state == RED) {
         setState(GREEN);
      } else if (state == YELLOW) {
         setState(RED);
      } else if (state == GREEN) {
         setState(YELLOW);
      } else {
         throw new ErrorException("Illegal stoplight state");
      }
   }

/* Creates a new GOval to represent one of the three lamps */
   private GOval createLamp(double x, double y) {
      GOval lamp = new GOval(x - LAMP_RADIUS, y - LAMP_RADIUS,
                             2 * LAMP_RADIUS, 2 * LAMP_RADIUS);
      lamp.setFilled(true);
      return lamp;
   }

/* Private constants */
   private static final double STOPLIGHT_WIDTH = 50;
   private static final double STOPLIGHT_HEIGHT = 100;
   private static final double LAMP_RADIUS = 10;

/* Private instance variables */
   private Color state;
   private GOval redLamp;
   private GOval yellowLamp;
   private GOval greenLamp;
}
```

4.2 Numeric fields

The interactors that you can place in the border regions are by no means limited to the **`JButton`** class used in the preceding examples. The **`javax.swing`** package includes a variety of useful interactor classes including **`JCheckBox`**, **`JComboBox`**, **`JLabel`**, **`JScrollBar`**, **`JRadioButton`**, **`JSlider`**, **`JSpinner`**, **`JToggleButton`**, **`JTextField`**. None of these interactors are particularly hard to use, and the Java Task Force did not feel there was any need to extend the set of interactors except in one respect. Unfortunately, none of the existing classes is suitable for reading numeric data from the user. If students are required to use **`JTextField`** exclusively and perform their own numeric conversion, they must first master such difficult conceptual issues as the use of wrapper classes for

numeric types and the details of exception handling. Hiding that complexity simplifies such operations considerably.

To this end, the Task Force decided to add two new classes—**`IntField`** and **`DoubleField`**—to simplify the development of applications that require numeric input. Each of these classes extends **`JTextField`** but provides additional methods to hide the complexity involved in numeric conversion and exception handling. The most useful methods available for **`DoubleField`** appear in Figure 4-4; the methods for **`IntField`** are the same except for the expected changes in the argument and result types.

The format control methods at the end of Figure 4-4 turn out to be relatively important. In the absence of format control, the value of a **`DoubleField`** often displays so many digits that the number becomes unreadable. The **`setFormat`** and **`getFormat`** methods eliminate this problem by allowing you to specify the output format. The format itself is specified using a string as defined in the **`DecimalFormat`** class in **`java.text`**. The use of format codes is illustrated in the currency converter program shown in Figure 4-8 later in this chapter.

4.3 Using interactors to control animation

One of the most common uses of interactors in the border region is to control the state of an animation running in the primary window. The Java Task Force packages provides excellent support for this type of animation control through the **`Animator`** class in the **`acm.util`** package. At one level, the **`Animator`** class is simply an extension of **`Thread`**, so you can use it as the thread of control for an animation as described in section 3.1. The **`Animator`** class, however, exports several methods that are useful for writing simple animation code. These methods are listed in Figure 4-5.

These methods are most easily illustrated by example. For the last several years, the Computer Science Advanced Placement course has used a marine biology simulation as its case study. In that simulation, different species of fish inhabit an environment and evolve by breeding, moving, and dying as specified by various parameters of the simulation. A sample run of the applet version of the Marine Biology Simulation appears in Figure 4-6, which shows both the random initial state of a simulation and a control panel at the bottom of the window. The **`Start`** button starts the simulation, the **`Step`** button advances it a single step, the **`Stop`** button stops it, and the **`Reset`** button creates a new initial state. The slider at the right of the control bar sets the speed.

Figure 4-4. Methods defined in the `DoubleField` class

Constructors

`DoubleField()` Creates a **`DoubleField`** object with no initial value.	
`DoubleField(double value)` Creates a **`DoubleField`** object with the specified initial value.	

Methods to set and retrieve the value of the field

`void setValue(double value)` Sets the value of the field and updates the display.	
`double getValue()` Returns the value in the field. If the value is out of range, errors or retries occur here.	

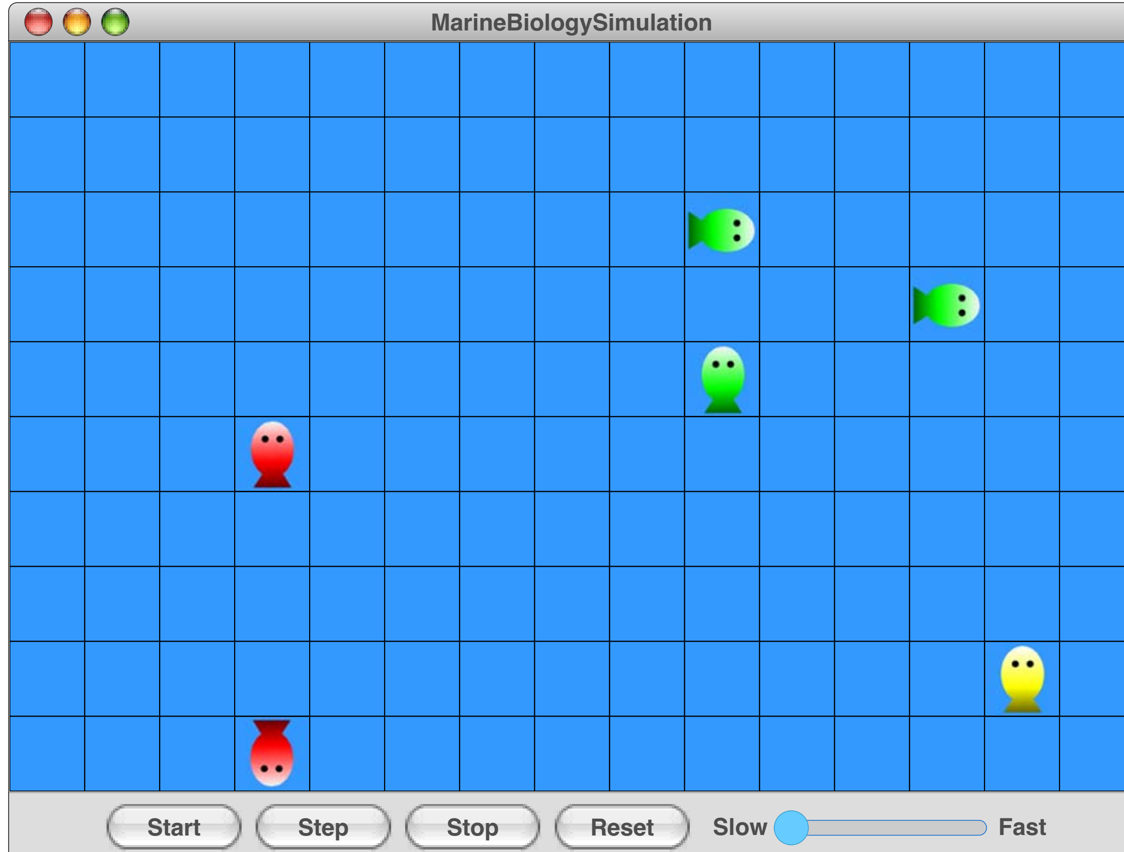
Methods to control formatting

`void setFormat(String format)` Sets the format string for the field as specified in the **`DecimalFormat`** class in **`java.text`**.	
`String getFormat()` Returns the current format string.	

Figure 4-5. Useful methods defined in the `Animator` class

Method	Description
Constructor	
`Animator()`	Creates a new `Animator` object.
Method to specify the code for the animation thread	
`void run()`	The code to animate the object goes in a `run` method specific to each subclass.
Methods to control the speed and flow of the animation	
`void pause(double milliseconds)`	Pauses the animation thread for the specified number of milliseconds.
`void setSpeed(double speed)`	Sets the speed of the animator to `speed`, which must be between 0.0 (slow) and 1.1 (fast).
`double getSpeed()`	Returns the speed of the animator set by the last call to `setSpeed`.
`void trace()`	Checks for tracing operations from buttons such as `Start`, `Stop`, and `Step`.
`void delay()`	Calls `trace` and then delays the animation by a time interval appropriate to the current speed.
Methods to support GUI controls	
`void buttonAction(String actionCommand)`	Invokes the action associated with the action command (`Start`, `Stop`, or `Step`)
`void registerSpeedBar(JSlider slider)`	Registers the specified slider as the speed bar for this animator.

Figure 4-6. Initial state of the Marine Biology Simulation



In the AP version of the case study, the details of the control panel are hidden from the student. When this example is recoded using the JTF tools, the code to create the control panel becomes quite short:

```
private void initControlPanel() {
   add(new JButton("Start"), SOUTH);
   add(new JButton("Step"), SOUTH);
   add(new JButton("Stop"), SOUTH);
   add(new JButton("Reset"), SOUTH);
   JSlider speedSlider = new JSlider(JSlider.HORIZONTAL);
   speedSlider.setValue(0);
   theSimulation.registerSpeedBar(speedSlider);
   add(new JLabel("  Slow"), SOUTH);
   add(speedSlider, SOUTH);
   add(new JLabel("Fast"), SOUTH);
   addActionListeners();
}
```

The code to handle the action events is equally manageable:

```
public void actionPerformed(ActionEvent e) {
   String cmd = e.getActionCommand();
   if (cmd.equals("Reset")) {
      theSimulation.buttonAction("Stop");
      createInitialPopulation();
   } else {
      theSimulation.buttonAction(cmd);
   }
}
```

The code in the simulator class—which is stored in the instance variable **theSimulation** in the above code fragments—is also straightforward. This class is a subclass of **Animator** and includes a **run** method with the following form:

```
public void run() {
   while (simulation is not complete) {
      Locatable[] theFishes = theEnv.allObjects();
      for (int index = 0; index < theFishes.length; index++) {
         ((Fish) theFishes[index]).act();
      }
      theDisplay.showEnv();
      Debug.println(theEnv.toString());
      Debug.println("---- End of Timestep ----");
      delay();
   }
}
```

This simplification should make it possible for students to see the code for the entire simulation, and possibly to write more of it themselves.

4.4 The TableLayout Class

As it happens, the biggest problems that students have in creating GUI-based applications don't come from the design of the Swing interactor classes themselves but rather from the problems involved in arranging those interactors inside a window. Java's traditional approach is to use a **layout manager,** which is responsible for managing the arrangement of the components within a **JPanel** or other form of container. Layout managers, however, can be difficult to teach. If nothing else, they introduce yet another source of complexity into programs that already seem to push the limits of student comprehension.

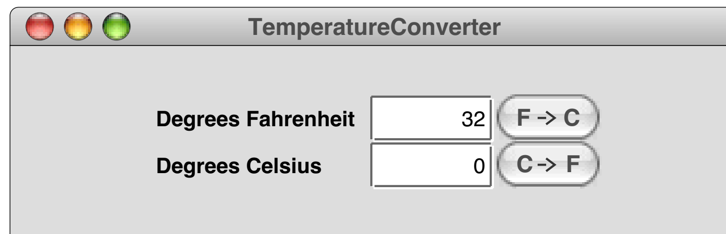
The more serious problem, however, is that the existing layout managers do not strike the right balance for teaching. On the one hand, simple layout managers like **`FlowLayout`** and **`BorderLayout`** are easy to learn, but do not provide enough flexibility to design many common layout configurations. On the other, "industrial strength" layout managers like **`GridBagLayout`** have all the power you might want, but are extremely hard for beginners to learn.

To address this problem, the Java Task Force developed the **`TableLayout`** class, which allows you to arrange components in a two-dimensional grid. The **`TableLayout`** class is a layout manager that has all the capabilities of Java's **`GridBagLayout`** manager, but is much easier to use.

Simple examples of the `TableLayout` class

The easiest way to understand how the **`TableLayout`** class works is to look at some simple examples. The basic structure of a **`TableLayout`** application is illustrated in Figure 4-7, which implements a simple temperature converter.

The user interface for the **`TemperatureConverter`** program looks like this:



The **`TemperatureConverter`** program provides a convenient template for creating GUI-based applications using **`TableLayout`**. The general strategy is to create a new class that extends the basic **`Program`** class and then define an **`init`** method that assembles the interactors into the desired arrangement. The first line of the **`init`** method is usually a call to **`setLayout`**, which creates the layout manager and assigns it to the program window. For **`TableLayout`** applications, the call to **`setLayout`** is simply

```
setLayout(new TableLayout(rows, columns));
```

where *rows* and *columns* are integers indicating the dimensions of the table. For example, to create a 2×3 table (two rows running horizontally and three columns running vertically), you would write

```
setLayout(new TableLayout(2, 3));
```

You can also use 0 in place of the number of rows or the number of columns to indicate an unbounded value. For example, the call

```
setLayout(new TableLayout(0, 7));
```

indicates a table with seven columns and as many rows as needed to display the components in the table. That layout will form the basis for a calendar application in section 4.6.

Once the layout manager is in place, the rest of the **`init`** method then creates the necessary interactors and adds them to the table, filling each row from left to right and then each row from top to bottom. In the **`TemperatureConverter`** example, the calls to **`add`** create the Fahrenheit row of the table using the lines

Figure 4-7. Temperature conversion program

```java
/*
 * File: TemperatureConverter.java
 * -------------------------------
 * This program allows users to convert temperatures
 * back and forth from Fahrenheit to Celsius.
 */

import acm.gui.*;
import acm.program.*;
import java.awt.event.*;
import javax.swing.*;

public class TemperatureConverter extends Program {

/** Initialize the graphical user interface */
   public void init() {
      setLayout(new TableLayout(2, 3));
      fahrenheitField = new IntField(32);
      fahrenheitField.setActionCommand("F -> C");
      fahrenheitField.addActionListener(this);
      celsiusField = new IntField(0);
      celsiusField.setActionCommand("C -> F");
      celsiusField.addActionListener(this);
      add(new JLabel("Degrees Fahrenheit"));
      add(fahrenheitField);
      add(new JButton("F -> C"));
      add(new JLabel("Degrees Celsius"));
      add(celsiusField);
      add(new JButton("C -> F"));
      addActionListeners();
   }

/** Listen for a button action */
   public void actionPerformed(ActionEvent e) {
      String cmd = e.getActionCommand();
      if (cmd.equals("F -> C")) {
         int f = fahrenheitField.getValue();
         int c = (int) Math.round((5.0 / 9.0) * (f - 32));
         celsiusField.setValue(c);
      } else if (cmd.equals("C -> F")) {
         int c = celsiusField.getValue();
         int f = (int) Math.round((9.0 / 5.0) * c + 32);
         fahrenheitField.setValue(f);
      }
   }

/* Private instance variables */
   private IntField fahrenheitField;
   private IntField celsiusField;

}
```

```
add(new JLabel("Degrees Fahrenheit"));
add(fahrenheitField);
add(new JButton("F -> C"));
```

and the corresponding Celsius row using the lines

```
add(new JLabel("Degrees Celsius"));
add(celsiusField);
add(new JButton("C -> F"));
```

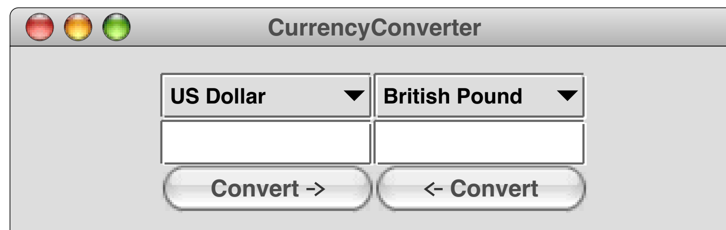
If you look at the sample run diagram that this code produces, you will quickly see that the sizes of the various interactors in the table have been adjusted according to their preferred sizes and the constraints imposed by the grid. The **JLabel** objects are of different sizes, but the implementation of **TableLayout** makes sure that there is enough space in the first column to hold the longer of the two labels. By default, each component added to a **TableLayout** container is expanded to fill its grid cell.

The code for the **TemperatureConverter** example calls the **addActionListeners** method to designate the program as an action listener for all buttons within it. This strategy of using **addActionListeners** was described in the preceding chapter. This style gives rise to relatively compact programs that introductory students find easy to understand, but you could just as well use any of the other styles of event detection described in section 3.3.

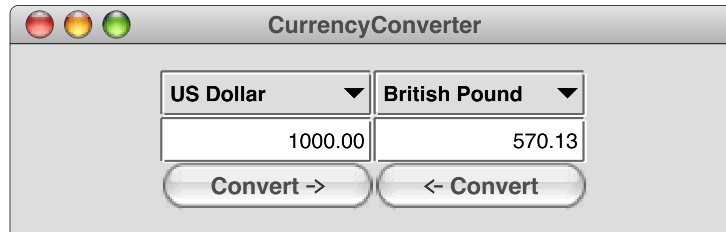
The calls to **addActionListener** and **setActionCommand** make it possible for the user to trigger a conversion either by hitting the appropriate button or by hitting the ENTER key in the interactor itself. Each of these actions generates an **ActionEvent** whose action command is either the string **"F -> C"** or **"C -> F"** depending on which button or interactor generated the event. These events are fielded by the **actionPerformed** method in the class, which performs the necessary conversion and then updates the value of the corresponding field.

The code for the **CurrencyConverter** program in Figure 4-8 implements a simple GUI-based program for converting currencies. When it first comes up on the screen, the application looks like this:

The interactors at the top are instances of the Swing **JComboBox** class, which makes it possible to choose an item from a list. If, for example, someone wanted to convert from U.S. dollars to British pounds, that user could click on the right **JComboBox** and then use the mouse to select the appropriate entry, as follows:



The user could then enter a currency amount into either of the two numeric entry boxes and hit the corresponding conversion button. If, for example, the user entered 1000 in the left entry area and then hit the button below it, the program would compute the corresponding number of pounds and display the result like this:



The user interface code for the **CurrencyConverter** program is quite similar to that used in **TemperatureConverter**, but there are a couple of interesting new features:

- The user interface uses a **JComboBox** to specify the currencies, thereby illustrating that the **TableLayout** strategy can be used with a wide range of Java interactors.
- The code calls **setFormat("0.00")** on each of the **DoubleField** values to ensure that two decimal places are included in these displays.

The **CurrencyConverter** program depends on a class called **CurrencyTable** that encapsulates the information about exchange rates. The **CurrencyTable** class exports two methods. The first is

```
public String[] getCurrencyNames()
```

which returns an array of the defined currency names. This list is used to create the array of options for the two **JComboBox** choosers. The second is

```
public double getExchangeRate(String name)
```

which returns a conversion rate for the specified currency. The implementation on the web site simply provides historical exchange rate values for a small sampling of currencies. A more ambitious implementation could go out to the web and find the current rates.

Specifying constraints

Although the ability to assign components to table cells is useful in its own right, you will soon discover that you often want to exercise more fine-grained control over the formatting of tables. One of the strengths of Java's **GridBagLayout** class is that it offers considerable flexibility in terms of specifying the format of a table. Unfortunately, **GridBagLayout** exposes that complexity from the very beginning, which means that novices are quickly overwhelmed by the mass of details. The **TableLayout** class, by contrast, offers the same flexibility, but in a way that hides the details unless you actually need them. Using **TableLayout** makes it possible for students to learn the simple features of the model quickly but still have access to the more advanced features later on.

The most important factor in terms of simplifying the conceptual model is that **TableLayout** allows you to specify constraints for each cell using strings instead of a **GridBagConstraints** structure. When you add a component to a **TableLayout** grid, you can specify a constraint string that has the following form:

constraint=*value*

where *constraint* is the name of one of the **GridBagConstraints** fields and *value* is a value appropriate for that field. For example, to duplicate the effect of setting the

Figure 4-8. Currency conversion program

```
/*
 * File: CurrencyConverter.java
 * ----------------------------
 * This program implements a simple currency converter.
 */

import acm.gui.*;
import acm.program.*;
import java.awt.event.*;
import javax.swing.*;

public class CurrencyConverter extends Program {

/** Initialize the graphical user interface */
   public void init() {
      setLayout(new TableLayout(3, 2));
      currencyTable = new CurrencyTable();
      leftChooser = new JComboBox(currencyTable.getCurrencyNames());
      leftChooser.setSelectedItem("US Dollar");
      rightChooser = new JComboBox(currencyTable.getCurrencyNames());
      rightChooser.setSelectedItem("Euro");
      leftField = new DoubleField();
      leftField.setFormat("0.00");
      leftField.setActionCommand("Convert ->");
      leftField.addActionListener(this);
      rightField = new DoubleField();
      rightField.setFormat("0.00");
      rightField.setActionCommand("<- Convert");
      rightField.addActionListener(this);
      add(leftChooser);
      add(rightChooser);
      add(leftField);
      add(rightField);
      add(new JButton("Convert ->"));
      add(new JButton("<- Convert"));
      addActionListeners();
   }

/** Listen for a button action */
   public void actionPerformed(ActionEvent e) {
      String cmd = e.getActionCommand();
      if (cmd.equals("Convert ->")) {
         double fromValue = leftField.getValue();
         double fromRate = getRateFromChooser(leftChooser);
         double toRate = getRateFromChooser(rightChooser);
         double toValue = fromValue * fromRate / toRate;
         rightField.setValue(toValue);
      } else if (cmd.equals("<- Convert")) {
         double fromValue = rightField.getValue();
         double fromRate = getRateFromChooser(rightChooser);
         double toRate = getRateFromChooser(leftChooser);
         double toValue = fromValue * fromRate / toRate;
         leftField.setValue(toValue);
      }
   }
```

Figure 4-8. Currency conversion program (continued)

```
/* Gets a rate from the specified chooser */
   private double getRateFromChooser(JComboBox chooser) {
      String currencyName = (String) chooser.getSelectedItem();
      return currencyTable.getExchangeRate(currencyName);
   }

/* Private instance variables */
   private CurrencyTable currencyTable;
   private JComboBox leftChooser;
   private JComboBox rightChooser;
   private DoubleField leftField;
   private DoubleField rightField;
}
```

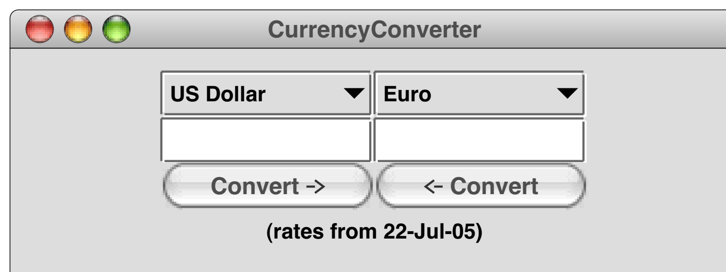
gridwidth field of a constraints object to 2 (thereby specifying a two-column entry), adopters of the **acm.gui** package can simply specify the constraint string

```
"gridwidth=2"
```

As an example, you could use this feature to add a notice to the currency converter program indicating the date at which the rates are calculated. That information is provided by the **CurrencyTable** class using the **getDate** method. To display that information at the bottom of the window, you could add a fourth row to the table by changing the dimensions in the constructor and then adding the following code to the end of the **init** method:

```
String date = "(rates from " + currencyTable.getDate() + ")";
add(new JLabel(date, JLabel.CENTER), "gridwidth=2");
```

The effect of this call is to add the **JLabel** reporting the date of the currency data to the **TableLayout** model for the program. However, instead of taking up a single column in the way that the other components do, the constraint string **"gridwidth=2"** tells the layout manager that this component should take up two columns in the grid, which ends up generating the following display:



The strings used as constraint objects can set several constraints at once by including multiple constraint/value pairs separated by spaces. Moreover, for those constraints whose values are defined by named constants in the **GridBagConstraints** class, **TableLayout** allows that name to be used as the value field of the constraint string. For example, the following string indicates that a table cell should span two columns but that the component should fill space only in the *y* direction:

```
"gridwidth=2 fill=VERTICAL"
```

Constraint strings are checked at run time to make sure that the constraints and values are defined and are consistent. The case of letters, however, is ignored, which makes it

Figure 4-9. Constraints supported by the `TableLayout` class

Constraint	Description
gridwidth=*columns* *or* **gridheight=***rows*	Indicates that this table cell should span the indicated number of columns or rows.
width=*pixels* *or* **height=***pixels*	The **width** specification indicates that the width of this column should be the specified number of pixels. If different widths are specified for cells in the same column, the column width is defined to be the maximum. In the absence of any **width** specification, the column width is the largest of the preferred widths. The **height** specification is interpreted symmetrically for row heights.
weightx=*weight* *or* **weighty=***weight*	If the total size of the table is less than the size of its enclosure, **TableLayout** will ordinarily center the table in the available space. If any of the cells, however, are given nonzero **weightx** or **weighty** values, the extra space is distributed along that axis in proportion to the weights specified. As in the **GridBagLayout** model, the weights are floating-point values and may therefore contain a decimal point.
fill=*fill*	Indicates how the component in this cell should be resized if its preferred size is smaller than the cell size. The legal values are **NONE**, **HORIZONTAL**, **VERTICAL**, and **BOTH**, indicating the axes along which stretching should occur. The default is **BOTH**.
anchor=*anchor*	If a component is not being filled along a particular axis, the **anchor** specification indicates where the component should be placed in its cell. The default value is **CENTER**, but any of the standard compass directions (**NORTH**, **SOUTH**, **EAST**, **WEST**, **NORTHEAST**, **NORTHWEST**, **SOUTHEAST**, or **SOUTHWEST**) may also be used.

possible to name the constraints in a way that is consistent with Java's conventions. Thus, if you want to emphasize the case convention that has each word within a multiword identifier begin with an uppercase letter, it is equally effective to write

```
"gridWidth=2 fill=VERTICAL"
```

The **TableLayout** class accepts all of the constraints supported by **GridBagLayout**, but the ones students are most likely to find useful are shown in Figure 4-9.

In addition to the standard **GridBagLayout** constraints, the **TableLayout** class uses two additional parameters—**hgap** and **vgap**—that apply to the layout as a whole rather than the individual cell. These parameters have the same interpretation as in other standard Java layout managers such as **FlowLayout** and **BorderLayout**. When the table is formatted, **hgap** pixels are left blank at the left and right edges and between each column; symmetrically, the layout manager leaves **vgap** blank pixels at the top and bottom edges and between each row. Although these values are typically positive, the implementation supports negative gaps, in which cells overlap by the specified number of pixels. The most common application for negative gaps occurs in displaying bordered components, which is illustrated in the **CalendarDemo** program described in section 4.6.

4.5 The `TablePanel` Classes

The examples presented so far in this chapter use **TableLayout** as the layout manager for the central region of a program, which is likely to be its most common application in the introductory curriculum. The **TableLayout** manager, however, can be used with any container and is extremely useful in assembling patterns of interactors.

To make it easier to assemble nested containers hierarchically, the **acm.gui** package includes three convenience classes that extend **JPanel** but install an appropriate **TableLayout** manager. These classes and their constructor patterns appear in Figure 4-10. The **HPanel** and **VPanel** classes make it easy to create complex assemblages of

Figure 4-10. Convenience classes based on `TableLayout`

`TablePanel` constructors

`public TablePanel(int rows, int columns)` Creates a **`JPanel`** with the indicated number of rows and columns.
`public TablePanel(int rows, int columns, int hgap, int vgap)` Creates a **`JPanel`** with the specified dimensions and gaps.

`HPanel` constructors

`public HPanel()` Creates a **`JPanel`** consisting of a single horizontal row.
`public HPanel(int hgap, int vgap)` Creates an **`HPanel`** with the specified gaps (**`vgap`** applies above and below the row).

`VPanel` constructors

`public VPanel()` Creates a **`JPanel`** consisting of a single vertical column.
`public VPanel(int hgap, int vgap)` Creates an **`VPanel`** with the specified gaps (**`hgap`** applies to the left and right of the column).

interactors by decomposing them hierarchically into rows and columns. In this respect, they have a common purpose with the **`BoxLayout`** manager introduced in the **`javax.swing`** package. The panel **`HPanel`** and **`VPanel`** classes, however, offer far more flexibility because they have the full power of the **`TableLayout`** class. The **`BoxLayout`** manager, by contrast, makes it difficult to do anything except to string together components in a linear form with no control over spacing or format.

4.6 Putting it all together: Creating a calendar display

To give you a sense of how the layout manager for the **`Program`** class and the **`TableLayout`** class can be used in a more sophisticated contexts, this section presents the complete implementation of a program called **`CalendarDemo`** program that displays a calendar page. The user interface for the program appears in Figure 4-11 and the code to create the display appears in Figure 4-12.

Figure 4-11. Sample run of the `CalendarDemo` application

Calendar

<- United States ->

June 2006

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

Figure 4-12. Calendar program

```
/*
 * File: CalendarDemo.java
 * -----------------------
 * This program uses the GUI layout mechanism to create a calendar
 * page.  The program uses the features of Java's Locale class to
 * internationalize the calendar.
 */

import acm.gui.*;
import acm.program.*;
import acm.util.*;
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;
import javax.swing.*;
import javax.swing.border.*;

public class CalendarDemo extends Program implements ItemListener {

/** Initialize the graphical user interface */
   public void init() {
      setBackground(Color.WHITE);
      initCountryList();
      localeChooser = new JComboBox(countries);
      String country = Locale.getDefault().getDisplayCountry();
      localeChooser.setSelectedItem(country);
      localeChooser.addItemListener(this);
      add(new JButton("<-"), NORTH);
      add(localeChooser, NORTH);
      add(new JButton("->"), NORTH);
      currentCalendar = Calendar.getInstance();
      itemStateChanged(null);
      addActionListeners();
   }

/** Respond to a button action */
   public void actionPerformed(ActionEvent e) {
      int delta = (e.getActionCommand().equals("<-")) ? -1 : +1;
      currentCalendar.add(Calendar.MONTH, delta);
      updateCalendarDisplay(currentCalendar);
   }

/** Respond to a change in the locale selection */
   public void itemStateChanged(ItemEvent e) {
      if (e == null || e.getStateChange() == ItemEvent.SELECTED) {
         Date time = currentCalendar.getTime();
         Locale locale = LOCALES[localeChooser.getSelectedIndex()];
         currentCalendar = Calendar.getInstance(locale);
         currentCalendar.setTime(time);
         symbols = new DateFormatSymbols(locale);
         weekdayNames = symbols.getWeekdays();
         monthNames = symbols.getMonths();
         firstDayOfWeek = currentCalendar.getFirstDayOfWeek();
         updateCalendarDisplay(currentCalendar);
      }
   }
```

Figure 4-12. Calendar program (continued)

```
/* Update the calendar display when a new month is selected */
   private void updateCalendarDisplay(Calendar calendar) {
      removeAll();
      setLayout(new TableLayout(0, 7, -1, -1));
      add(createMonthLabel(calendar), "gridwidth=7 bottom=3");
      for (int i = 0; i < 7; i++) {
         add(createWeekdayLabel(i), "weightx=1 width=1 bottom=2");
      }
      int weekday = getFirstWeekdayIndex(calendar);
      for (int i = 0; i < weekday; i++) {
         add(createDayBox(null), "weighty=1");
      }
      int nDays = getDaysInMonth(calendar);
      for (int day = 1; day <= nDays; day++) {
         add(createDayBox("" + day), "weighty=1");
         weekday = (weekday + 1) % 7;
      }
      while (weekday != 0) {
         add(createDayBox(null), "weighty=1");
         weekday = (weekday + 1) % 7;
      }
      validate();
   }

/* Generate the header label for a particular month */
   private JLabel createMonthLabel(Calendar calendar) {
      int month = calendar.get(Calendar.MONTH);
      int year = calendar.get(Calendar.YEAR);
      String monthName = capitalize(monthNames[month]);
      JLabel label = new JLabel(monthName + " " + year);
      label.setFont(JTFTools.decodeFont(TITLE_FONT));
      label.setHorizontalAlignment(JLabel.CENTER);
      return label;
   }

/* Create a label for the weekday header at the specified index */
   private JLabel createWeekdayLabel(int index) {
      int weekday = (firstDayOfWeek + index + 6) % 7 + 1;
      JLabel label = new JLabel(capitalize(weekdayNames[weekday]));
      label.setFont(JTFTools.decodeFont(LABEL_FONT));
      label.setHorizontalAlignment(JLabel.CENTER);
      return label;
   }

/* Compute the number of days in the current month */
   private int getDaysInMonth(Calendar calendar) {
      calendar = (Calendar) calendar.clone();
      int current = calendar.get(Calendar.DAY_OF_MONTH);
      int next = current;
      while (next >= current) {
         current = next;
         calendar.add(Calendar.DAY_OF_MONTH, 1);
         next = calendar.get(Calendar.DAY_OF_MONTH);
      }
      return current;
   }
```

Figure 4-12 Calendar program (continued)

```
/* Compute the index of the first weekday for the current Locale */
   private int getFirstWeekdayIndex(Calendar calendar) {
      int day = calendar.get(Calendar.DAY_OF_MONTH);
      int weekday = calendar.get(Calendar.DAY_OF_WEEK);
      int weekdayIndex = (weekday + 7 - firstDayOfWeek) % 7;
      return ((5 * 7 + 1) + weekdayIndex - day) % 7;
   }

/* Create a box for a calendar day containing the specified text */
   private Component createDayBox(String text) {
      VPanel vbox = new VPanel();
      if (text== null) {
         vbox.setBackground(EMPTY_BACKGROUND);
      } else {
         JLabel label = new JLabel(text);
         label.setFont(JTFTools.decodeFont(DATE_FONT));
         vbox.add(label, "anchor=NORTHEAST top=2 right=2");
         vbox.setBackground(Color.WHITE);
      }
      vbox.setOpaque(true);
      vbox.setBorder(new LineBorder(Color.BLACK));
      return vbox;
   }

/* Create a list of country names from the list of Locales */
   private void initCountryList() {
      countries = new String[LOCALES.length];
      for (int i = 0; i < LOCALES.length; i++) {
         countries[i] = LOCALES[i].getDisplayCountry();
      }
   }

/* Capitalize the first letter of a word */
   private String capitalize(String word) {
      return word.substring(0, 1).toUpperCase() + word.substring(1);
   }

/* Private constants */
   private static final Color EMPTY_BACKGROUND = new Color(0xDDDDDD);
   private static final String TITLE_FONT = "Serif-36";
   private static final String LABEL_FONT = "Serif-bold-14";
   private static final String DATE_FONT = "Serif-18";
   private static final Locale[] LOCALES = {
      new Locale("fr", "FR", ""), new Locale("de", "DE", ""),
      new Locale("es", "MX", ""), new Locale("it", "IT", ""),
      new Locale("nl", "NL", ""), new Locale("es", "ES", ""),
      new Locale("en", "GB", ""), new Locale("en", "US", "")
   };

/* Private instance variables */
   private JComboBox localeChooser;
   private String[] countries;
   private Calendar currentCalendar;
   private DateFormatSymbols symbols;
   private String[] monthNames;
   private String[] weekdayNames;
   private int firstDayOfWeek;
}
```

The **CalendarDemo** program makes use of the full range of capabilities described in this chapter. The top row of controls uses the layout capabilities of the **Program** class to create controls for the calendar, including a **JComboBox** that can choose the language and style for a particular country. The main body of the calendar uses a **TableLayout** manager with seven columns. The rows and columns are also set to overlap by a pixel to ensure that there is only a single-pixel line dividing the individual cells. Finally, each day in the calendar is represented using a **VPanel** with a Swing border attached. The exciting thing about the program is that it offers some very sophisticated features—GUI-based control, tabular formatting, and internationalization—and still fits in three pages of code. Such is the power of Java in which you have so many resources on which to draw.