**FIGURE 2-8** Useful methods to get you started with the acm.graphics package

**Constructors**

`new GLabel(`*string*`,` *x*`,` *y*`)`
Creates a new `GLabel` object containing the specified string that begins at the point (*x*, *y*).

`new GRect(`*x*`,` *y*`,` *width*`,` *height*`)`
Creates a new `GRect` object with the specified dimensions whose upper left corner is at (*x*, *y*).

`new GOval(`*x*`,` *y*`,` *width*`,` *height*`)`
Creates a new `GOval` object whose size is set to fit inside the `GRect` with the same arguments.

`new GLine(`$x_1$`,` $y_1$`,` $x_2$`,` $y_2$`)`
Creates a new `GLine` object connecting the points ($x_1$, $y_1$) and ($x_2$, $y_2$).

**Methods common to all graphical objects**

*object*`.setColor(`*color*`)`
Sets the color of the object to *color*, which is ordinarily a color name from `java.awt`.

*object*`.setLocation(`*x*`,` *y*`)`
Changes the location of the object to the point (*x*, *y*).

*object*`.move(`*dx*`,` *dy*`)`
Moves the object by adding *dx* to its *x* coordinate and *dy* to its *y* coordinate.

**Methods available for GRect and GOval only**

*object*`.setFilled(`*fill*`)`
Sets whether this object is filled (`true` means filled, `false` means outlined).

*object*`.setFillColor(`*color*`)`
Sets the color used to fill the interior of the object, which may be different from the border.

**Methods available for GLabel only**

*label*`.setFont(`*string*`)`
Sets the font for *label* as indicated by *string*, which gives the family name, style, and point size.

3. Use the `add` method in the `GraphicsProgram` class to add the new rectangle to the set of graphical objects displayed in the graphics window.

Figure 2-9 contains a simple program called `GRectExample` that produces the following output:

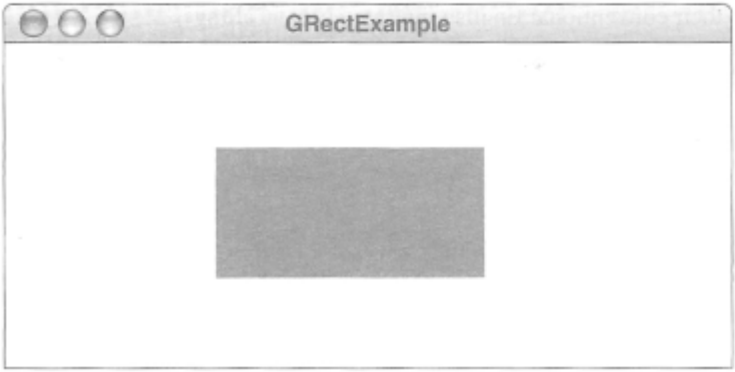


**FIGURE 3-1 Primitive types in Java**

<table>
<tr><th>Type</th><th>Domain</th><th>Common operations</th></tr>
<tr><td>byte</td><td>8-bit integers in the range –128 to 127</td><td rowspan="4">The arithmetic operators:<br>+ add * multiply<br>- subtract / divide<br>% remainder<br>The relational operators:<br>== equal != not equal<br>< less than <= less or equal<br>> greater than >= greater or equal</td></tr>
<tr><td>short</td><td>16-bit integers in the range –32768 to 32767</td></tr>
<tr><td>int</td><td>32-bit integers in the range<br>–2147483648 to 2147483647</td></tr>
<tr><td>long</td><td>64-bit integers in the range<br>–9223372036854775808 to 9223372036854775807</td></tr>
<tr><td>float</td><td>32-bit floating-point numbers in the range<br>$\pm 1.4 \times 10^{-45}$ to $\pm 3.4028235 \times 10^{38}$</td><td rowspan="2">The arithmetic operators except %<br>The relational operators</td></tr>
<tr><td>double</td><td>64-bit floating-point numbers in the range<br>$\pm 4.39 \times 10^{-322}$ to $\pm 1.7976931348623157 \times 10^{308}$</td></tr>
<tr><td>char</td><td>16-bit characters encoded using Unicode</td><td>The relational operators</td></tr>
<tr><td>boolean</td><td>the values true and false</td><td>The logical operators:<br>&& and || or ! not</td></tr>
</table>

maximum and minimum values that reflect the capacity of the memory cells in which they are stored. The next two—**float** and **double**—represent floating-point numbers, again with different dynamic ranges. Except in those rare cases in which a library method requires the use of one of the other types, this text will use **int** and **double** as its standard numeric types. The type **char** is used to represent character data and is covered in Chapter 8. The type **boolean**—which you encountered briefly in Chapter 2—is so important to programming that it merits an entire section of its own later in this chapter.

In addition to the primitive data types listed in Figure 3-1, it is often useful to think of the Java type **String** as if it were a primitive type, even though it is actually a class defined in the package **java.lang**. One reason for treating it as primitive is that the **String** class is built into the Java language in much the same ways that the primitive types are. For example, Java specifies a special syntax for string constants, just as it does for **int** or **double** or **boolean**. But a more important reason for regarding **String** as a primitive type is that doing so helps you think about strings in a more holistic way. As you will discover in Chapter 8, the **String** class defines a large number of methods that perform a variety of useful operations. Although understanding the details of those methods and how to use them will become important at some point, you can usually get away with imagining that string values are pretty much like integer values except that the two types have different domains. Just as you can declare a variable to be of type **int** and assign it an integer value, you can declare a variable to be of type **String** and assign it a string value. It is only when you need to use the methods provided by the **String** class that it makes any difference whether **String** is a class or a primitive type.

FIGURE 5-1 Selected methods from the Math class

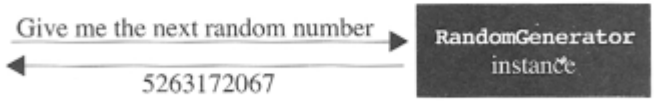
| | |
|---|---|
| `Math.abs(x)` | Returns the absolute value of **x**, which can be of any numeric type. |
| `Math.min(x, y)` | Returns the smaller of **x** and **y**. |
| `Math.max(x, y)` | Returns the larger of **x** and **y**. |
| `Math.sqrt(x)` | Returns the square root of the value **x**. |
| `Math.log(x)` | Returns the natural logarithm of **x**, which uses the mathematical constant *e* as its base. |
| `Math.exp(x)` | Returns the inverse logarithm of **x**, which is $e^{\mathbf{x}}$. |
| `Math.pow(x, y)` | Returns the value **x** raised to the **y** power. |
| `Math.sin(theta)` | Returns the trigonometric sine of the angle **theta**, which is measured in radians. |
| `Math.cos(theta)` | Returns the cosine of the angle **theta**. |
| `Math.tan(theta)` | Returns the tangent of the angle **theta**. |
| `Math.asin(x)` | Returns the angle whose sine is **x**. |
| `Math.acos(x)` | Returns the angle whose cosine is **x**. |
| `Math.atan(x)` | Returns the angle whose tangent is **x**. |
| `Math.toRadians(degrees)` | Converts an angle from degrees to radians. |
| `Math.toDegrees(radians)` | Converts an angle from radians to degrees. |

name of the class along with the method name. Thus, the static **sqrt** method in the **Math** class must be written as **Math.sqrt** whenever it is called.

## Method calls as messages

Static methods of the sort provided by the **Math** class are something of an anachronism in languages like Java, throwbacks to a style of programming that existed before object-oriented programming arrived on the scene. Object-oriented languages use methods in a different way than traditional languages do, mostly because the object-oriented paradigm depends on a different conceptual model of the underlying process. Once you understand that conceptual model, the structure of method calls in Java and the terminology used to describe it will make a great deal of intuitive sense.

The random generator object then replies with the next random number, computed by some algorithmic process, the details of which are hidden inside the black box. For example, given a request for a random number, the random generator object might give the following reply:




As is usually the case in object-oriented programming, the message takes the form of a method call. The reply is simply the value that method returns. As you will learn in the next section, the random generator object responds to several different methods, depending on the type of random value your application needs.

## Using the `RandomGenerator` class

The most important public methods in the **`RandomGenerator`** class are listed in Figure 6-1. For the most part, these methods are easy to understand. The only method that requires much explanation is the static **`getInstance`** method, which returns an new instance of the **`RandomGenerator`** class. The important thing to remember is that a **`RandomGenerator`** is not itself a random value but rather an object that generates random values. Although typically you will want to generate many different random values in the course of a program, you only need to have one generator. Moreover, because you will probably want to use that generator in several methods within your program, it is easiest to declare it as an instance variable, as follows:

**FIGURE 6-1 Useful methods in the RandomGenerator class**

| |
|---|
| **`static RandomGenerator getInstance()`**<br>Returns an instance of the **`RandomGenerator`** class, which is shared throughout the program. |
| **`int nextInt(int n)`**<br>Returns a random integer chosen from the **`n`** values in the range 0 to **`n`** - 1, inclusive. |
| **`int nextInt(int low, int high)`**<br>Returns a random integer in the range **`low`** to **`high`**, inclusive. |
| **`double nextDouble()`**<br>Returns a random double $d$ in the range $0 \le d < 1$; a range excluding one end is called **half-open.** |
| **`double nextDouble(double low, double high)`**<br>Returns a random double $d$ in the half-open range **`low`** $\le d <$ **`high`**. |
| **`boolean nextBoolean()`**<br>Returns a **`boolean`** that is **`true`** roughly 50 percent of the time. |
| **`boolean nextBoolean(double p)`**<br>Returns a **`boolean`** that is **`true`** with probability **`p`**, which must be between 0 and 1. |
| **`Color nextColor()`**<br>Returns a random Java color. |
| **`void setSeed(long seed)`**<br>Sets a "seed" to indicate a starting point for the pseudorandom sequence. |

**FIGURE 6-4** The javadoc page for RandomGenerator

Overview Package **Student** Complete Tree Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

---

acm.util

## Class RandomGenerator

```
java.lang.Object
  |
  +--java.util.Random
        |
        +--acm.util.RandomGenerator
```

---

`public class RandomGenerator extends Random`

This class implements a simple random number generator that allows clients to generate pseudorandom integers, doubles, booleans, and colors. To use it, the first step is to declare an instance variable to hold the random generator as follows:

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

By default, this `RandomGenerator` object is initialized to begin at an unpredictable point in a pseudorandom sequence. During debugging, it is often useful to set the internal seed for the random generator explicitly so that it always returns the same sequence. To do so, you need to invoke the `setSeed` method.

The `RandomGenerator` object returned by `getInstance` is shared across all classes in an application. Using this shared instance of the generator is preferable to allocating new instances of `RandomGenerator`. If you create several random generators in succession, they will typically generate the same sequence of values.

---

| Constructor Summary |
|---|
| `RandomGenerator()`<br>Creates a new random generator. |

| Method Summary | |
|---|---|
| `RandomGenerator` | `getInstance()`<br>This method returns a `RandomGenerator` instance that can be shared among several classes. |
| `boolean` | `nextBoolean(double p)`<br>Returns a random `boolean` value with the specified probability. |
| `Color` | `nextColor()`<br>Returns a random opaque `Color` whose components are chosen uniformly in the 0-255 range. |
| `double` | `nextDouble(double low, double high)`<br>Returns the next random real number in the specified range. |
| `int` | `nextInt(int low, int high)`<br>Returns the next random integer in the specified range. |

| Inherited Method Summary | |
|---|---|
| `boolean` | `nextBoolean()`<br>Returns a random boolean that is true 50 percent of the time. |
| `double` | `nextDouble()`<br>Returns a random double $d$ in the range $0 \leq d < 1$. |
| `int` | `nextInt(int n)`<br>Returns a random integer $k$ in the range $0 \leq k < n$. |
| `void` | `setSeed(long seed)`<br>Sets a new starting point for the random generator sequence. |

**FIGURE 6-4** The javadoc page for RandomGenerator (continued)

## Constructor Detail

`public RandomGenerator()`

Creates a new random generator. Most clients will not use the constructor directly but will instead call `getInstance` to obtain a `RandomGenerator` object that is shared by all classes in the application.

**Usage:** `RandomGenerator rgen = new RandomGenerator();`

## Method Detail

`public static RandomGenerator getInstance()`

This method returns a `RandomGenerator` instance that can be shared among several classes.

**Usage:** `RandomGenerator rgen = RandomGenerator.getInstance();`
**Returns:** A shared `RandomGenerator` object

---

`public boolean nextBoolean(double p)`

Returns a random `boolean` value with the specified probability. You can use this method to simulate an event that occurs with a particular probability. For example, you could simulate the result of tossing a coin like this:

```
String coinFlip = rgen.nextBoolean(0.5) ? "HEADS" : "TAILS";
```

**Usage:** `if (rgen.nextBoolean(p)) . . .`
**Parameter:** `p` A value between 0 (impossible) and 1 (certain) indicating the probability
**Returns:** The value `true` with probability `p`

---

`public Color nextColor()`

Returns a random opaque `Color` whose components are chosen uniformly in the 0-255 range.

**Usage:** `Color color = rgen.newColor()`
**Returns:** A random opaque `Color`

---

`public double nextDouble(double low, double high)`

Returns the next random real number in the specified range. The resulting value is always at least `low` but always strictly less than `high`. You can use this method to generate continuous random values. For example, you can set the variables `x` and `y` to specify a random point inside the unit square as follows:

```
double x = rgen.nextDouble(0.0, 1.0);
double y = rgen.nextDouble(0.0, 1.0);
```

**Usage:** `double d = rgen.nextDouble(low, high)`
**Parameters:** `low` The low end of the range
`high` The high end of the range
**Returns:** A random `double` value $d$ in the range `low` $\leq d <$ `high`

---

`public int nextInt(int low, int high)`

Returns the next random integer in the specified range. For example, you can generate the roll of a six-sided die by calling

```
rgen.nextInt(1, 6);
```

or a random decimal digit by calling

```
rgen.nextInt(0, 9);
```

**Usage:** `int k = rgen.nextInt(low, high)`
**Parameters:** `low` The low end of the range
`high` The high end of the range
**Returns:** The next random `int` between `low` and `high`, inclusive

**FIGURE 9-2** Useful methods in the `GCanvas` and `GraphicsProgram` classes

**Methods to add and remove graphical objects**

| |
|---|
| `void add(GObject gobj)`<br>Adds a graphical object to the canvas at its internally stored location. |
| `void add(GObject gobj, double x, double y)`<br>Adds a graphical object to the canvas at the specified location. |
| `void remove(GObject gobj)`<br>Removes the specified graphical object from the canvas. |
| `void removeAll()`<br>Removes all graphical objects from the canvas. |

**Method to find the graphical object at a particular location**

| |
|---|
| `GObject getElementAt(double x, double y)`<br>Returns the topmost object containing the specified point, or `null` if no such object exists. |

**Method to support graphical animation (available in `GraphicsProgram` only)**

| |
|---|
| `void pause(double milliseconds)`<br>Suspends program execution for the specified time interval, which is measured in milliseconds. |
| `void waitForClick()`<br>Suspends program execution until the user clicks the mouse anywhere in the canvas. |

**Useful methods inherited from superclasses**

| |
|---|
| `int getWidth()`<br>Returns the width of the canvas, in pixels. |
| `int getHeight()`<br>Returns the height of the canvas, in pixels. |
| `void setBackground(Color bg)`<br>Changes the background color of the canvas. |

new ones in the stacking order. Removing an object takes it off the canvas, revealing any objects that were behind it.

The **`add`** method comes in two forms. The first is useful when the constructor for the **`GObject`** has already established the location of the object, as it has in the examples you have seen. The second form is used when you want to place the object on the screen in a way that depends on its size or other properties. In that case, you typically need to create a **`GObject`** without specifying a location, figure out where it should go, and then add it to the canvas at that explicit *x* and *y* location. The most common example of this style of positioning occurs when you want to center a **`GLabel`** at some location; the section on "The **`GLabel`** class" later in this chapter describes how it works.

The **`getElement`** method in the **`GCanvas`** class returns the graphical object on the canvas that includes a specified point. This method is particularly useful when you want to select an object using the mouse. Because the description fits better in that context, the details of **`getElement`** are described in Chapter 10.

You have already seen the **`pause`** method, which delays the execution of the program for the number of milliseconds specified by the argument. As you learned in Chapter 4, you can animate a program by making small changes to the graphical objects on the canvas and then calling **`pause`** to ensure that the program doesn't run

Although the **GPoint**, **GDimension**, and **GRectangle** classes export a number of useful methods to simplify common geometric operations, listing them all in this book would provide much more detail than you need. The most important operations are the constructors that create the composite objects and the methods that retrieve the individual coordinate values. These methods are listed in Figure 9-4. To find out about the other methods available in these classes, you can explore the **javadoc** pages.

## The GMath class

Computing the positions and sizes of objects in a graphical figure can sometimes require the use of simple mathematical functions. Although Java's **Math** class defines methods to compute trigonometric functions such as **sin** and **cos**, these methods are often confusing because they adopt a coordinate model that is in some ways incompatible with the one Java uses for its graphics packages. In Java's graphics libraries, angles are measured in degrees; in the **Math** class, angles must be specified 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 in Figure 9-5. As with the methods in Java's math class, the methods

**FIGURE 9-4** Essential methods in the **GPoint**, **GDimension**, and **GRectangle** classes

**GPoint constructors and methods**

| |
|---|
| **new GPoint(double x, double y)**<br>Creates a new **GPoint** object containing the coordinate values **x** and **y**. |
| **double getX()**<br>Returns the **x** component of a **GPoint**. |
| **double getY()**<br>Returns the **y** component of a **GPoint**. |

**GDimension constructors and methods**

| |
|---|
| **new GDimension(double width, double height)**<br>Creates a new **GDimension** object containing the values **width** and **height**. |
| **double getWidth()**<br>Returns the **width** component of a **GDimension**. |
| **double getHeight()**<br>Returns the **height** component of a **GDimension**. |

**GRectangle constructors and methods**

| |
|---|
| **new GRectangle(double x, double y, double width, double height)**<br>Creates a new **GRectangle** object containing the four specified values. |
| **double getX()**<br>Returns the **x** component of a **GRectangle**. |
| **double getY()**<br>Returns the **y** component of a **GRectangle**. |
| **double getWidth()**<br>Returns the **width** component of a **GRectangle**. |
| **double getHeight()**<br>Returns the **height** component of a **GRectangle**. |

**FIGURE 9-5** Static methods in the `GMath` class

**Trigonometric methods in degrees**

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

**Methods to simplify the conversion to polar coordinates**

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

**Method to round a `double` to an `int`**

| |
|---|
| `int round(double x)`<br>Rounds a **double** to the nearest **int** (rather than to a **long** as in the **Math** class). |

exported by **GMath** are static. Calls to these methods therefore need to include the name of the class, as in

```
double cos45 = GMath.cosDegrees(45);
```

which sets the variable **cos45** to the cosine of 45 degrees.

The first few methods in Figure 9-5 compute trigonometric functions of angles in degrees. The **distance** and **angle** methods make it easy to convert traditional $x$-$y$ coordinates into **polar coordinates,** in which points are defined in terms of their distance and direction from the origin. The distance is usually denoted by the letter $r$, which comes from the observation that a particular value of $r$ corresponds to the points lying on a circle with that radius. The angle—typically written using the Greek letter $\theta$ in mathematics and represented using the variable name **theta**—is measured in degrees counterclockwise from the $+x$ axis, just as it is in classical geometry.

The last method listed in Figure 9-5 is **round**, which rounds a **double** to the nearest **int**. Although Java's **Math** class also exports a method called **round**, that version returns a **long**, which makes it less convenient to use.

## The `GObject` class and its subclasses

If you look back at the arrows in Figure 9-1, it is immediately clear that the **GObject** class plays a central role in the **acm.graphics** package. Just as all roads led to

**FIGURE 9-6** Methods supported by all `GObject` subclasses

| |
|---|
| **`void setLocation(double x, double y)`**<br>Sets the location of this object to the specified point. |
| **`void move(double dx, double dy)`**<br>Moves the object using the displacements **`dx`** and **`dy`**. |
| **`void movePolar(double r, double theta)`**<br>Moves the object **`r`** units in direction **`theta`**, measured in degrees. |
| **`double getX()`**<br>Returns the x-coordinate of the object. |
| **`double getY()`**<br>Returns the y-coordinate of the object. |
| **`double getWidth()`**<br>Returns the width of the object. |
| **`double getHeight()`**<br>Returns the height of the object. |
| **`boolean contains(double x, double y)`**<br>Checks to see whether a point is inside the object. |
| **`void setColor(Color c)`**<br>Sets the color of the object. |
| **`Color getColor()`**<br>Returns the object color. |
| **`void setVisible(boolean visible)`**<br>Sets whether this object is visible. |
| **`boolean isVisible()`**<br>Returns **`true`** if this object is visible. |
| **`void sendToFront()`**<br>Sends this object to the front of the canvas, where it may obscure objects further back. |
| **`void sendToBack()`**<br>Sends this object to the back of the canvas, where it may be obscured by objects in front. |
| **`void sendForward()`**<br>Sends this object forward one position in the stacking order. |
| **`void sendBackward()`**<br>Sends this object backward one position in the stacking order. |

- The **`setVisible`** method makes it possible to hide an object on the screen. If you call **`setVisible(false)`**, the object disappears from the display until you call **`setVisible(true)`**. The predicate method **`isVisible`** allows you to determine whether an object is visible.
- The various **`send`** methods allow you to change the stacking order. When you add a new object to the canvas, it goes on top of the other objects and can therefore obscure the objects behind it. If you call **`sendToBack`**, the object goes to the back of the stack. Conversely, **`sendToFront`** brings it to the front. You can also change the order by calling **`sendForward`** and **`sendBackward`**, which move an object one step forward or backward, respectively, in the stack.

Although it is useful to know about these new methods, it may be momentarily disconcerting to discover that some of the methods you've been using don't appear

**FIGURE 9-7 Methods specified by the graphical interfaces**

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

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

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

| |
|---|
| **`void setSize(double width, double height)`**<br>Changes the size of this object to the specified width and height. |
| **`void setBounds(double x, double y, double width, double height)`**<br>Changes the bounds of this object as specified by the individual parameters. |

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

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

- The classes at the bottom of the diagram—**`GLabel`**, **`GRect`**, **`GOval`**, **`GLine`**, **`GArc`**, **`GImage`**, and **`GPolygon`**, along with the **`GRoundRect`** and **`G3DRect`** subclasses of **`GRect`**—are collectively known as the **shape classes.** These classes are the concrete realizations of the abstract **`GObject`** class, each of which defines one of the shapes that you can add to the collage. The shape classes are the ones that show up most often in graphical programming and are important enough to warrant a major section of their own.

## 9.3 Using the shape classes

As defined at the end of the preceding section, the shape classes are the concrete subclasses of **`GObject`** used to represent the actual objects on the screen. There are several different shape classes corresponding to the variety of graphical objects—labels, rectangles, ovals, lines, arcs, images, and polygons—that you might display on a canvas. Each of the shape classes inherits the methods from **`GObject`**, but usually defines additional methods that are specifically appropriate to that subclass. The sections that follow describe each of the shape classes in more detail.

### The `GLabel` class

The **`GLabel`** class is the first class you encountered in this text, even though it is in some respects the most idiosyncratic of the shape classes. For one thing, **`GLabel`** does not implement any of the graphical interfaces from Figure 9-7. For another, it

**FIGURE 9-9** Useful methods exported by the `GLabel` class

| | |
|---|---|
| **Constructor** | |
| `new GLabel(String str, double x, double y)`<br>Creates a new `GLabel` object containing the string `str` whose origin is the point (`x`, `y`). | |
| `new GLabel(String str)`<br>Creates a new `GLabel` object containing `str` at the point (0, 0); the actual location is set later. | |
| **Methods to get and set the text displayed by the label** | |
| `void setLabel(String str)`<br>Changes the string displayed by the label to `str`. | |
| `String getLabel()`<br>Returns the text string currently being dispayed. | |
| **Methods to get and set the font** | |
| `void setFont(Font f)` *or* `setFont(String description)`<br>Sets the font to a Java `Font` object or a string in the form `"Family-style-size"` | |
| `Font getFont()`<br>Returns the current font. | |
| **Methods to retrieve geometric properties of the label and its font** | |
| `double getWidth()`<br>Returns the horizontal extent of the label when displayed in its current font. | |
| `double getHeight()`<br>Returns the height of the `GLabel` object, which is defined to be the height of its font. | |
| `double getAscent()`<br>Returns the distance the characters in the current font extend above the baseline. | |
| `double getDescent()`<br>Returns the distance the characters in the current font extend below the baseline. | |

baseline would disappear off the top of the canvas. In fact, the second version is often considerably more convenient, particularly if you don't yet know exactly where the label should be placed at the time you create it.

Many of the shape classes, including `GLabel`, export one version of the constructor that includes the initial coordinates and another that does not. You use the constructor that includes the coordinates when you know the location at the time you create the object. If you instead need to perform some calculation to determine the location, the easiest approach is to create it without specifying a location, perform the necessary calculations to figure out where it should go, and then add it to the canvas using the version of the `add` method that includes the *x* and *y* coordinates of the object. You will see several examples of this approach later in the chapter.

The `getLabel` and `setLabel` methods allow you to retrieve or change the contents of a `GLabel` after it has been created. Suppose, for example, that you want to include a `GLabel` on the screen to keep track of the user's score in an interactive game. You might start things off by defining a `GLabel` called `scoreLabel` like this:

```
GLabel scoreLabel = new GLabel("Score: 0");
```

The only aspect of the **GOval** that tends to cause confusion is the fact that using the upper left corner as the location makes less sense for ovals than it does for rectangles, given that this point is not inside the figure. Somehow, the confusion seems most acute when the oval happens to be a circle. In mathematics, a circle is conventionally defined in terms of its center and radius and not by the dimensions of the square that encloses it. As you saw with the **createFilledCircle** method that appeared in Figure 5-3, you can define methods that restore the conventional mathematical interpretation.

Despite the confusion that sometimes arises from defining ovals in terms of their bounding rectangle, there is a compelling reason for adopting that design decision in the **acm.graphics** package. The classes in the standard **java.awt** package use the bounding-rectangle approach. Maintaining consistency with the standard Java model makes it easier to move back and forth between the two. Although Emerson may have been correct in his observation that "a foolish consistency is the hobgoblin of little minds," there is enough justification behind this particular consistency to move it beyond the foolish category.

## The GLine class

The **GLine** class is used to construct line drawings in the **acm.graphics** package. The **GLine** class implements **GScalable** (which is implemented so that the starting point of the line remains fixed and the line expands outward from there), but not **GFillable** or **GResizable**. The **GLine** class also defines several additional methods, as shown in Figure 9-10. The **setStartPoint** method allows clients to change the first endpoint of the line without changing the second; conversely, **setEndPoint** gives clients access to the second endpoint without affecting the first. These methods are therefore different in their operation from **setLocation**, which moves the entire line without changing its length or orientation. The corresponding **getStartPoint** and **getEndPoint** methods return the coordinates as a **GPoint**, which combines the individual *x* and *y* values into a single object.

**FIGURE 9-10 Useful methods exported by the GLine class**

**Constructor**

| |
|---|
| **new GLine(double x0, double y0, double x1, double y1)**<br>Creates a new **GLine** object connecting the points (**x0**, **y0**) and (**x1**, **y1**). |

**Methods to get and set the endpoints independently**

| |
|---|
| **void setStartPoint(double x, double y)**<br>Resets the coordinates of the initial point of the line to (**x**, **y**) without changing the end point. |
| **GPoint getStartPoint()**<br>Returns the coordinates of the initial point in the line. |
| **void setEndPoint(double x, double y)**<br>Resets the coordinates of the end point of the line to (**x**, **y**) without changing the starting point. |
| **GPoint getEndPoint()**<br>Returns the coordinates of the end point in the line. |

**FIGURE 9-13** Methods exported by the `GArc` class

**Constructor**

`new GArc(double x, double y, double width, double height, double start, double sweep)`
Creates a new **GArc** object as specified by the six parameters.

`new GArc(double width, double height, double start, double sweep)`
Creates a new **GArc** object with a default location of (0, 0).

**Methods to get and set the start and sweep angles**

`void setStartAngle(double theta)`
Resets the start angle used to define the arc.

`double getStartAngle()`
Returns the current value of the start angle.

`void setSweepAngle(double theta)`
Resets the sweep angle used to define the arc.

`double getSweepAngle()`
Returns the current value of the sweep angle.

**Methods to retrieve the endpoints of the arc**

`GPoint getStartPoint()`
Returns the coordinates of the initial point in the arc.

`GPoint getEndPoint()`
Returns the coordinates of the end point in the arc.

**Methods to retrieve or reset the framing rectangle that encloses the arc**

`GRectangle getFrameRectangle()`
Returns the rectangle that bounds the ellipse from which the arc is taken.

`void setFrameRectangle(GRectangle bounds)`
Resets the rectangle that bounds the arc.

If you wanted to center the image in the canvas instead, you could rewrite the **run** method as follows:

```
public void run() {
   GImage image = new GImage("MyImage.gif");
   double x = (getWidth() - image.getWidth()) / 2;
   double y = (getHeight() - image.getHeight()) / 2;
   add(image, x, y);
}
```

As these examples make clear, the mechanical details of using images are not particularly complicated, because Java takes care of the work necessary to display the actual image on the screen. All you have to do is put the image data into a file and then tell Java the name of that file.

A more interesting question is where these images come from in the first place. One possibility is to create images of your own. To do so, you need to use a digital camera or some kind of image-creation software. The second possibility is to download existing images from the web. Most web browsers make it possible for you to download the corresponding image file whenever an image appears on the screen.

**FIGURE 9-17** Methods exported by the `GPolygon` class

**Constructor**

| |
|---|
| `new GPolygon()`<br>Creates an empty `GPolygon` object with its reference point at (0, 0). |
| `new GPolygon(double x, double y)`<br>Creates an empty `GPolygon` object with its reference point at (`x`, `y`). |

**Methods to add edges to the polygon**

| |
|---|
| `void addVertex(double x, double y)`<br>Adds a vertex at (`x`, `y`) relative to the reference point of the polygon. |
| `void addEdge(double dx, double dy)`<br>Adds a new vertex whose coordinates are shifted by `dx` and `dy` from the previous vertex. |
| `void addPolarEdge(double r, double theta)`<br>Adds a new vertex whose location is expressed in polar coordinates relative to the previous one. |
| `void addArc(double arcWidth, double arcHeight, double start, double sweep)`<br>Adds a series of edges that simulates an arc specified in the style of the `GArc` constructor. |

**Other useful methods**

| |
|---|
| `void rotate(double theta)`<br>Rotates the polygon around its reference point by the angle `theta`, measured in degrees. |
| `void recenter()`<br>Adjusts the vertices of the polygon so that the reference point is at the geometric center. |
| `GPoint getCurrentPoint()`<br>Returns the coordinates of the last vertex added to the polygon, or `null` if the polygon is empty. |

information on the methods that are not described in this book, the best approach is to consult the **javadoc** pages.

## 9.4 Creating compound objects

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

The methods available in the `GCompound` class are listed in Figure 9-18. The sections that follow introduce a series of examples that illustrate the use of these methods.

### A simple GCompound example

To understand how the `GCompound` class works, it is easiest to start with a simple example. Imagine that you want to assemble an abstract face on the canvas that looks something like this:

**FIGURE 9-18 Methods exported by the `GCompound` class**

**Constructor**

`new 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)`
Adds a graphical object to the compound at the specified location.

`void remove(GObject gobj)`
Removes the specified graphical object from the compound.

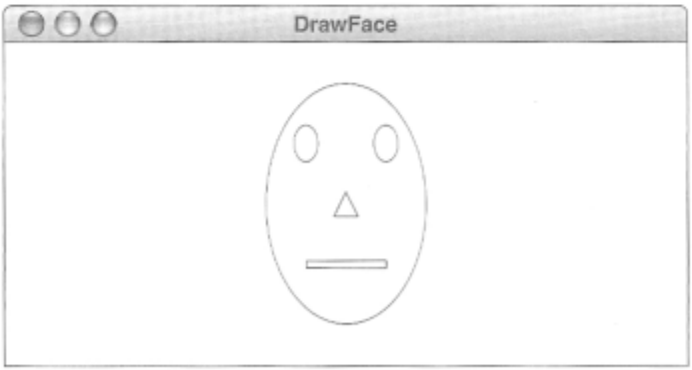
`void removeAll()`
Removes all graphical objects and components from the compound.

**Miscellaneous methods**

`GObject getElementAt(double x, double y)`
Returns the frontmost object containing the specified point, or **`null`** if no such object exists.

`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.




The face consists of several independent features—a **`GOval`** for the head, two **`GOval`**s for the eyes, a **`GRect`** for the mouth, and a **`GPolygon`** for the nose—which you then need to add in the appropriate places. You can, of course, add each of these objects to the canvas just as you have all along. If you do so, however, you will find it hard to manipulate the face as a unit. Suppose, for example, that you want to move the face to a different position on the canvas. If you added each of the features independently, moving the face would require then moving every feature. It would be much better if you could simply tell the entire face to move.

The code in Figure 9-19 uses the **`GCompound`** class to do just that. The **`GFace`** class extends **`GCompound`** to create a face object containing the necessary features.

```
public void mouseClicked(MouseEvent e) {
   GStar star = new GStar(STAR_SIZE);
   star.setFilled(true);
   add(star, e.getX(), e.getY());
}
```

For the most part, the code is straightforward and uses nothing beyond the graphics methods you have already seen. The first statement creates a **GStar** object, relying on the definition of the **GStar** class presented in Figure 9-16. The next statement makes sure that the star is filled rather than outlined. The final statement then adds the star to the canvas.

The only new feature in this implementation of **mouseClicked** is the **MouseEvent** parameter, which provides information about where the click occurred. That information is stored inside the **MouseEvent** object **e**, and you can retrieve it by calling **e.getX()** and **e.getY()**. In this example, the goal is to have the star appear precisely where the mouse is pointing, so these values are precisely what you need to set the location of the star.

## 10.3 Responding to mouse events

The **mouseClicked** method in the **DrawStarMap** program is only one of several listener methods you can use to respond to mouse events. The complete set of listener methods called in response to mouse events appears in Figure 10-2. Each method allows you to respond to a specific type of action with the mouse, most of which will probably seem familiar from using your computer. Dragging the mouse, for example, consists of moving the mouse while holding the button down. Applications tend to use dragging when they want to move an object from one place

**FIGURE 10-2** Standard listener methods for responding to mouse events

**MouseListener** interface

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

**MouseMotionListener** interface

| |
|---|
| **void mouseMoved(MouseEvent e)**<br>Called whenever the mouse is moved with the button up. |
| **void mouseDragged(MouseEvent e)**<br>Called whenever the mouse is moved with the button down. |

**FIGURE 10-5** Standard listener methods for responding to keyboard events

| |
|---|
| **`void keyPressed(KeyEvent e)`**<br>Called whenever a key is pressed. |
| **`void keyReleased(KeyEvent e)`**<br>Called whenever a key is released. |
| **`void keyTyped(KeyEvent e)`**<br>Called when a key is "typed" (pressed and released). |

it matters how long you hold a key down. The **`keyTyped`** method provides a slightly higher level of control and makes sense for applications in which you use the keyboard to enter text.

The methods that you call to extract information from a **`KeyEvent`** depend on which of these styles you are using. In the **`keyPressed`** and **`keyReleased`** methods, you can find out which key was pressed by calling the **`getKeyCode`** method on the **`KeyEvent`**. The return value, however, is not a character but an integer code representing what the designers of Java's event model called a **virtual key.** The constant names defined by **`KeyEvent`** for the most common virtual key codes appear in Figure 10-6.

When you use the **`keyTyped`** method, you can determine the actual character entered on the keyboard by calling the **`getKeyChar`** method on the **`KeyEvent`**. In this case, the **`getKeyChar`** method automatically takes account of modifier keys like SHIFT, so that holding down the SHIFT key and typing the A key delivers the expected uppercase character **`'A'`**. The **`getKeyChar`** method is not available if you are using the **`keyPressed`** and **`keyReleased`** methods. If you need to take account of modifier keys, you need to call other methods in the **`KeyEvent`** class that are beyond the scope of this text.

So that you have a chance to see at least one illustration of key listeners, it is useful to extend the **`DragObjects`** program from Figure 10-4 so that you can move the currently selected object either by dragging it with the mouse or by using the

**FIGURE 10-6** Virtual key constants defined in the **`KeyEvent`** class

| | | |
|---|---|---|
| `VK_A` *through* `VK_Z` | `VK_F1` *through* `VK_F12` | `VK_UP` |
| `VK_0` *through* `VK_9` | `VK_NUMPAD0` *through* `VK_NUMPAD9` | `VK_DOWN` |
| `VK_COMMA` | `VK_BACK_SPACE` | `VK_LEFT` |
| `VK_PERIOD` | `VK_DELETE` | `VK_RIGHT` |
| `VK_SLASH` | `VK_ENTER` | `VK_PAGE_UP` |
| `VK_SEMICOLON` | `VK_TAB` | `VK_PAGE_DOWN` |
| `VK_EQUALS` | `VK_SHIFT` | `VK_HOME` |
| `VK_OPEN_BRACKET` | `VK_CONTROL` | `VK_END` |
| `VK_BACK_SLASH` | `VK_ALT` | `VK_ESCAPE` |
| `VK_CLOSE_BRACKET` | `VK_META` | `VK_PRINTSCREEN` |
| `VK_BACK_QUOTE` | `VK_NUM_LOCK` | `VK_INSERT` |
| `VK_QUOTE` | `VK_SCROLL_LOCK` | `VK_HELP` |
| `VK_SPACE` | `VK_CAPS_LOCK` | `VK_CLEAR` |