8 Elementary Graphics

Overview

This chapter introduces graphics programming in the context of software for a flag manufacturer. You will see how to create applets using graphics commands, both static drawings and animated drawings. These applets can be part of a web page or part of a stand-alone application using a JFrame (which is introduced in the next chapter).

This chapter contains no new language features. The first six sections can be covered after finishing Section 4.8 (arrays are postponed until Section 8.7). You can even cover Sections 8.1-8.2 after Chapter One if you just ignore a few references.

We only discuss using an applet in a web page. If you wish to use graphics in a stand-alone application, all you need to know about JFrames is in Section 9.1. Using a JFrame is more complicated, which is why we only work with web pages in this chapter.

- Sections 8.1-8.2 explain basic graphics techniques for JApplets using the Graphics, Graphics2D, Shape, and Color classes.
- Sections 8.3-8.5 develop the logic for drawing a complete animated American flag.
- Sections 8.6-8.7 review basic principles of software development.
- Sections 8.8-8.9 discuss a large graphics program and implement the Turtle class.

8.1 The JApplet, Color, And Graphics2D Classes

You have been hired to create some graphical software by Flag-Maker Inc., a company that makes flags. Specifically, they want all of their web pages and application programs to have a small red-white-and-blue American flag dancing around on the screen when people are using them.

You probably think this is not the smartest idea in the world, but they are offering you $20,000 to design the software, so you keep your mouth shut and do it (this book is about real-life programming techniques, and doing what the client wants is one of the most profitable techniques, as long as it is ethical). You have probably heard of W. A. Mozart, a genius at developing software for the piano and other instruments. He had a financially rewarding job with a large corporation, but the kind of software he had to write was dictated by his district manager, the Archbishop of Salzburg. He quit after ten years on that boring job so he could write what he wanted; but then he nearly starved.

Imports and the Color class

Java has an enormous number of classes in its standard library for doing graphics; see `http://java.sun.com/docs` for details. One graphics package is `java.awt`; the Color class and Graphics class are in this package. Another package is `javax.swing`, which contains the JFrame and JApplet classes. JFrames are used for applications and JApplets are used for graphical parts of a web page.

The **Color** class contains several class variables named `red`, `blue`, `green`, etc. The complete way to mention e.g. the color `red` from the Color class in your program is `java.awt.Color.red`. It is of course much easier to simply use `Color.red`, which you can do if you have the following line at the top of your compilable file:

```
   import java.awt.Color;
```

The thirteen standard color constants available in the Color class are `black`, `blue`, `cyan`, `darkGray`, `gray`, `green`, `lightGray`, `magenta`, `orange`, `pink`, `yellow`, `red`, and `white`. Cyan is a greenish-blue and magenta is reddish-blue. You can also create your own color by supplying red/green/blue (RGB) values in the range from 0 to 255, inclusive, as in

```
   new Color(16, 255, 0) // green with a reddish tinge
```

The JApplet class

You begin developing an applet by defining a subclass of the JApplet class from the `javax.swing` package. Objects of the **JApplet** class represent rectangular panels within a web page or application frame. JApplet is a subclass of the Panel class which in turn is a subclass of the Component class.

The simplest applets contain nothing but a method with the heading `public void paint (Graphics g)`. Listing 8.1 shows an applet that writes two sentences, "stars goes here" in blue (lines 2-3) and "stripes go here" below it in red (lines 4-5). It then draws a line between the two sentences (line 6). The statements in the `paint` method are explained in detail in the few pages. We call this applet Dancer because a later version will produce the dancing flag. It imports from `java.awt.*` to use the Graphics, Graphics2D, and Color classes ("awt" stands for "Abstract Windowing Toolkit").

Listing 8.1 The Dancer applet, version 1

```
import javax.swing.JApplet;
import java.awt.*;  // import Graphics, Graphics2D, and Color

public class Dancer extends JApplet
{
   /** Draw things on the applet's drawing area. */

   public void paint (Graphics g)
   {  Graphics2D page = (Graphics2D) g;                  // 1
      page.setColor (Color.blue);                        // 2
      page.drawString ("stars goes here", 10, 45);       // 3
      page.setColor (Color.red);                         // 4
      page.drawString ("stripes go here", 10, 90);       // 5
      page.drawLine (10, 60, 30, 70);                    // 6
   }  //======================
}
```

HTML

To test an applet, you first need to compile it: Enter `javac Dancer.java` for the Dancer applet given in Listing 8.1 to produce the compiled form `Dancer.class`. Next, you need to have a web page that refers to it. The following is a very simple web page in HTML that you could use to test this Dancer applet:

```
   <HTML>
      <BODY> Your name goes here
         <APPLET CODE="Dancer.class" WIDTH=240 HEIGHT=120>
         </APPLET>
      </BODY>
   </HTML>
```

Each HTML tag is written inside angle braces < > as illustrated. HTML tags usually come in pairs, one to say where a certain kind of section starts and one to say where it ends. Usually the ending tag has the same first word as the beginning tag but with a division sign in front of it.

This particular HTML file has a body section. The body contains some words that will appear on the web page (put your own name in place of them) and an APPLET pair of tags. The APPLET tag tells the browser where to find the executable file that contains the applet, and it also tells the dimensions of the applet's rectangular area.

Appletviewer and web browsers

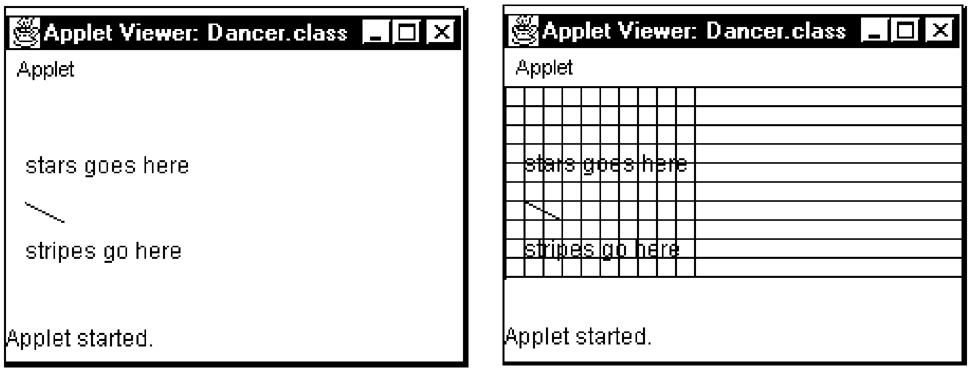
If you put that HTML material in a file named e.g. `Dancer.html` in your `C:\cs1` folder along with the applet Dancer.class, then enter `file:///C:/cs1/Dancer.html` in the URL locator area of your web browser, you should see the blue and red sentences (if your files are in a different folder from `C:\cs1`, make the obvious substitution).

For testing an applet, you may use the **appletviewer** program that comes with your Java installation: At the command line in the terminal window (for Windows), in response to the `C:\cs1` prompt, enter `appletviewer Dancer.html` to see just the applet code of the HTML file. Clicking on the word "Applet" at the top brings down the applet menu you can use to reload the applet (to run it again), print the applet, or quit the applet.

A web browser that loads a web page executes the logic of the `paint` method in any applets in the web page. The applet does not need a main method because the web browser has one. This `paint` method makes the drawing.

Each time the user moves the web page or returns to it after having linked elsewhere, the web browser calls the `repaint` method for the JApplet (inherited from the Component class). The `repaint` method does some initial preparation and then calls the applet's `paint` method.

The left side of Figure 8.1 shows the result of displaying the applet in Listing 8.1. The right side of the figure is the same except that lines have been added by hand ten pixels apart, so you can see the exact placement of the words on the screen. Only the starting point of the words is determined by the second and third values in the `drawString` method call; the size of the font determines how tall and how wide the words are.



```
stars goes here
stripes go here
Applet started.
```

```
stars goes here
stripes go here
Applet started.
```

Figure 8.1 Result of executing the Dancer applet

The Graphics2D class

Graphics2D is a class in the standard `java.awt` package; its objects represent drawing areas. The **drawing area** is measured in units of **pixels**. You specify points on the drawing area with two int values `<x, y>.` The first one tells how many pixels in from the left edge the drawing is to be made. The second one tells how many pixels down from the top edge the drawing is to be made.

Since the HTML specifies that the drawing area for the Dancer class is 240 pixels wide and 120 pixels tall, the center point is at <120,60>, the bottom-left corner is at <0,120>, and the top-right corner is at <240,0>. Anything you draw outside the boundaries of a drawing area does not appear (nor does it crash the program).

Graphics2D is a subclass of the Graphics class, which is also in the `java.awt` package. Graphics was used in earlier versions of Java. Graphics2D gives more control over geometric objects than does Graphics. To maintain compatibility with older browsers, the `paint` method continues to have its parameter be of Graphics type.

To write a simple applet, copy Listing 8.1 down to and including line 1 except replace Dancer by an appropriate name. Then change what comes after line 1 to do what you want. Line 1 declares a local Graphics2D variable `page` and assigns `(Graphics2D)g` to it, where `g` is the formal parameter. We then use `page` for everything.

Graphics methods

Each Graphics2D drawing area has a current **drawing Color** in which it draws all characters, lines, etc. As the Dancer applet shows, a call of an applet's `paint` method passes in a value for the Graphics parameter. You can then cast it to a Graphics2D object value and send the following messages to it:

- `setColor (someColor)` makes the drawing Color the specified value. All drawing will be done in this color thereafter, until the drawing Color is changed. Initially, when the applet begins execution, the drawing Color is `Color.black`.
- `getColor()` returns the current drawing Color.
- `drawString (messageString, xCorner, yCorner)` draws the given string of characters with the bottom-left corner of the first character at the pixel position given by the two int values `<xCorner, yCorner>`.
- `drawLine (xStart, yStart, xEnd, yEnd)` draws a line starting at the pixel position given by the two int values `<xStart, yStart>` and ending at the pixel position given by the two int values `<xEnd, yEnd>`.

Figure 8.1 shows that the first sentence that Dancer draws is at <10,45>, a point 10 pixels to the right and 45 pixels below the top-left corner of the drawing area. It is drawn in blue because of the `setColor(Color.blue)` command. The second sentence is drawn directly below it in red. A line is drawn starting from the point <10,60>, which is 10 pixels to the right and 60 pixels below the top-left corner of the drawing area. The line ends at the point <30,70>, which is 20 pixels to the right and 10 pixels down from the starting point <10,60>.

Exercise 8.1 Write a `paint` method for an applet that puts your name in the drawing area in three different colors, all on the same line.
Exercise 8.2 Write a `paint` method for an applet that draws a green rectangle, three times as tall as it is wide, using the `drawLine` method.
Exercise 8.3 Write a `paint` method for an applet that draws a 40x40 square divided into four 20x20 squares. Put the bottom-right corner of the square at <50, 70>.

8.2 Six Shape Classes: Rectangles, Lines, And Ellipses

Shape is an interface defined in the `java.awt` package. Concrete implementations of Shape include rectangles, ellipses, and lines. A Graphics2D object can send either of the following two messages to a Shape object:

- `draw (someShape)` draws the given Shape object.
- `fill (someShape)` draws the given Shape object plus fills in its inside portion. This is of course pointless for a line object.

You need to know how to create various shapes to use in the `draw` and `fill` messages. One basic Shape is a Rectangle, from the `java.awt` package. You can use the following for Rectangle objects. <u>All parameters here are int values</u>:

- `new Rectangle (w, h)` constructs a new Rectangle object whose top-left corner is at <`0,0`>, with width `w` and height `h` measured in pixels. If either `w` or `h` is negative, the Rectangle will not show in the drawing area.
- `setLocation (x, y)` move the top-left corner of the rectangle to the specified location <`x, y`>. The width and height remain unchanged.
- `setSize (w, h)` changes the width and height of the rectangle to the specified values. The top-left corner remains unchanged.
- `grow (dw, dh)` adds `dh` pixels above and below the rectangle, and it adds `dw` pixels to the left and right sides of the rectangle. So its width changes by `2*dw`, its height by `2*dh`, and its top-left corner changes by <`-dw,-dh`>.
- `translate (dx, dy)` moves the rectangle `dx` pixels to the right and `dy` pixels down from its current position. The width and height remain unchanged.
- `new Rectangle (x, y, w, h)` constructs a new Rectangle object whose top-left corner is at <`x,y`>, with width `w` and height `h` measured in pixels. If either `w` or `h` is negative, the Rectangle will not show in the drawing area.

Figure 8.2 (see next page) shows the result of executing the Boxes applet in Listing 8.2. Study carefully the correspondence between the method and the figure.

Listing 8.2 An applet to demonstrate Graphics2D and Rectangle methods

```
import javax.swing.JApplet;
import java.awt.*;  // import Graphics, Graphics2D, and Rectangle

public class Boxes extends JApplet
{
   public void paint (Graphics g)
   {  Graphics2D page = (Graphics2D) g;
      Rectangle box = new Rectangle (50, 40);
      box.setLocation (30, 60);
      page.draw (box);           // the inner box in the figure
      box.grow (5, 5);
      page.draw (box);           // the outer box in the figure
      box.translate (70, 0);
      page.fill (box);           // the filled box on the right
   }  //======================
}
```

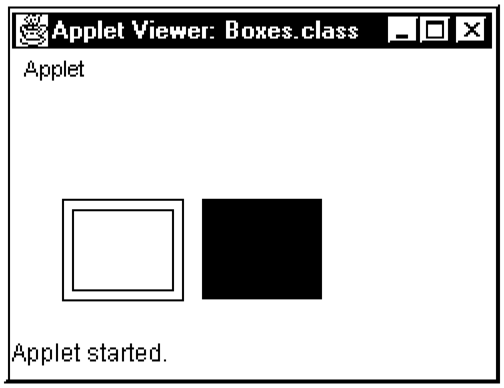


Figure 8.2 Illustration of Graphics2D and Rectangle methods

Line2D, Rectangle2D, Ellipse2D, and RoundRectangle2D

It is often useful to create geometric figures using decimal numbers for the parameters rather than int values. Then you can multiply or divide them by some fixed number without worrying about rounding them off, because the `draw` and `fill` methods use the parameter values rounded to the nearest int to calculate pixel positions. <u>All parameters here are decimal number values</u> (which java calls `double` values):

- `new Line2D.Double(x, y, x2, y2)` is a line object with one end-point at `<x,y>` and the other at `<x2,y2>`.
- `new Rectangle2D.Double(x, y, w, h)` is a rectangular object with top-left corner at `<x,y>` and width `w` and height `h`.
- `new Ellipse2D.Double(x, y, w, h)` is an elliptical object whose bounding box has upper-left corner `<x,y>` and the given width `w` and height `h`. The **bounding box** is the smallest rectangle with vertical and horizontal sides that contains the ellipse. The ellipse is a circle if `w` and `h` are equal.
- `new RoundRectangle2D.Double(x, y, w, h, arcWidth, arcHeight)` is a rectangular object whose first four parameters have the same meaning as for Rectangle2D. The last two parameters specify the width and height of the arc on each corner, so you get a rectangular shape with rounded corners.

These four classes are in the `java.awt.geom` package, and all are kinds of Shapes. The reference to such a class has a dot in it. This is because e.g. the Ellipse2D class contains a class named Double nested inside it. If say you want to draw an ellipse that fits exactly within the inner box on the left of Figure 8.2, use this statement:

```
   page.draw (new Ellipse2D.Double (30, 60, 50, 40));
```

<u>Caution</u> Check your drawing commands to be sure you do not draw outside the drawing area (e.g., all `x` values are 0 to 240 and all `y` values are 0 to 120). The program will not crash, but nothing will show up. An applet inherits two instance methods `getWidth()` and `getHeight()` from the Component class that you can use to find out the applet's current dimensions.

Older browsers require you to use the earlier **java.awt.Applet** class rather than JApplet. You also use the following Java 1.0 methods with the Graphics object and int values. You may of course use them with Graphics2D objects as well:

```
g.drawRect(x, y, w, h);       g.drawOval(x, y, w, h);
g.fillRect(x, y, w, h);       g.fillOval(x, y, w, h).
```

One significant difference between Applet and JApplet is that `repaint` clears previously drawn material before executing `paint` for an Applet, but not for a JApplet. In elementary cases, this only makes a difference if your drawing changes from one call of `paint` to the next, e.g., you randomly generate pixel positions. In such a case, it may be desirable to start the `paint` coding for a JApplet by erasing as follows:

```
   public void paint (Graphics g)
   {  Graphics2D page = (Graphics2D) g;
      page.setColor (Color.white);
      page.fillRect (0, 0, getWidth(), getHeight());
```

Turtlet drawings

The Turtle software described in Chapter One can be used for drawings on applets. You simply use the Turtlet class instead of Turtle (since you are making an applet rather than an application) with a different constructor. The Turtle class is a subclass of Turtlet; Turtlet provides the actual drawing commands for Turtle. The following constructor creates a Turtlet on a given Graphics page at a given starting position <xStart, yStart>:

```
   Turtlet sam = new Turtlet (page, xStart, yStart);
```

For instance, the TwoSquares application program in Listing 1.2 begins as follows:

```
   public class TwoSquares
   {  // Draw two 40x40 squares side by side, 10 pixels apart.
      public static void main (String[ ] args)
      {  Turtle sue = new Turtle();
         sue.paint (90, 40);  // draw the right side of square #1
```

You would instead begin as follows (the rest remains unchanged). This has the turtle start in the middle of an applet 760 pixels wide and 600 pixels tall. Just remember that all Turtlets have to be created within the `paint` method of an Applet or JApplet:

```
   public class TwoSquares extends JApplet
   {  // Draw two 40x40 squares side by side, 10 pixels apart.
      public void paint (Graphics g)
      {  Turtlet sue = new Turtlet (g, 380, 300);
         sue.paint (90, 40);  // draw the right side of square #1
```

Exercise 8.4 Write a `paint` method that draws two rectangles that form a Red Cross symbol. Put the word "give" in the left arm of the cross and the word "blood" in the right arm.
Exercise 8.5 Write a `paint` method for an applet that draws three rectangles, each inside the next, all with the same top-left corner. Use the `setSize` method twice appropriately.
Exercise 8.6 Revise the preceding exercise to fill the three rectangles with three different colors: `magenta`, `gray`, and `yellow` (Hint: Fill the outer square first).
Exercise 8.7 Write a `paint` method to draw three filled circles the same size all in a horizontal line, all of radius 10, with the middle one just barely touching each of the two side ones.
Exercise 8.8* Draw the Peace symbol.
Exercise 8.9* (Archery) Start a JApplet class named Archery (to be developed further in later exercises). Have its `paint` method draw a bullseye, a set of five concentric circles filled in with different colors. Write the Archery HTML, then compile and execute your applet. Hint: Fill a larger circle before filling a circle inside of it.
Exercise 8.10** Draw three filled circles with each barely touching each of the other two.

Part B Enrichment And Reinforcement

8.3 Analysis And Design Example: The Flag Software

Your client wants a dancing flag. Since a flag is a separate concept, you should have a separate class for Flag objects. Then the class can be reused in other software. The Dancer's `paint` method should pass to a Flag object the information it needs to do its job, which is to draw a flag at a particular point on a particular Graphics drawing area.

For the animation, you will repeatedly draw Flags at different points on the screen. Each time the browser refreshes a web page containing an applet, or the user moves to a different part of the web page not containing the applet and then back again, the browser executes a method in the applet with the heading `public void start()`. The animation commands will go in this `start` method.

The body of Dancer's `paint` method should have a single command such as `itsFlag.draw (page, 10, 20)` to draw the flag 10 pixels in and 20 pixels down on the drawing area. That gives the revision in Listing 8.3. In a later version of Dancer, to make the flag dance, the `start` method will change those two numeric values that the `paint` method uses.

Listing 8.3 The Dancer applet, version 2

```
import javax.swing.JApplet;
import java.awt.*;

public class Dancer extends JApplet
{
   private Flag itsFlag = new Flag();


   public void start()
   {  // leave empty for now; animation comes later
   }  //=======================


   public void paint (Graphics g)
   {  itsFlag.draw ((Graphics2D) g, 10, 20);
   }  //=======================
}
```

Analysis (clarifying what to do) You cannot draw a flag unless you know how it looks. So you do some research on the internet at `www.askjeeves.com` to find a site that gives the correct proportions. You might think you could just estimate it from a picture, but a basic principle of programming is, "Don't make stuff up unless you have to." You will come off looking better if you have the right information.

You find that the flag should be 91 pixels tall and 173 pixels wide, assuming each stripe is 7 pixels tall. The 13 stripes alternate red and white starting with red. The blue field of stars should be 49 pixels tall (corresponding to 7 stripes) and 69 pixels wide. The stars should be in 9 rows staggered 4.9 pixels apart vertically. In each row, each star should be 11.4 pixels to the right of the one before. The rows of stars have 6 and 5 stars alternating starting with the row of 6. That makes 50 stars altogether.

Right away you realize that fractions cannot be used here; only a whole number of pixels can be used in most commands. So you decide to make stars 12 pixels apart with 5 pixels between rows. That leaves 2 pixels extra space above and below the 50 stars. You also need to expand the size of the applet from 240x120 to at least 240x160, to allow room for the flag to move up and down on the drawing area.

Test Plan (seeing if you did it) You will run the program, check the placement in the upper-left corner of the window, and count the stripes and the stars.

Design (deciding how to do it) First you develop the overall plan for drawing the flag, in deference to a basic principle of programming, "If you don't know where you are going, you are not likely to get there." The accompanying design block is a good start.

```
STRUCTURED NATURAL LANGUAGE DESIGN for the main Flag logic
1.  Draw the 13 red and white stripes.
2.  Draw the blue field.
3.  Draw the 50 white stars.
```

The second step of this logic can be done with a simple `fill` command applied to a rectangle; the other two steps are complex enough to require their own methods. The initial design of the Flag class is implemented in Listing 8.4 (see next page), with the `drawStars` and `drawStripes` methods left for later development. The UML class diagram is in Figure 8.3.

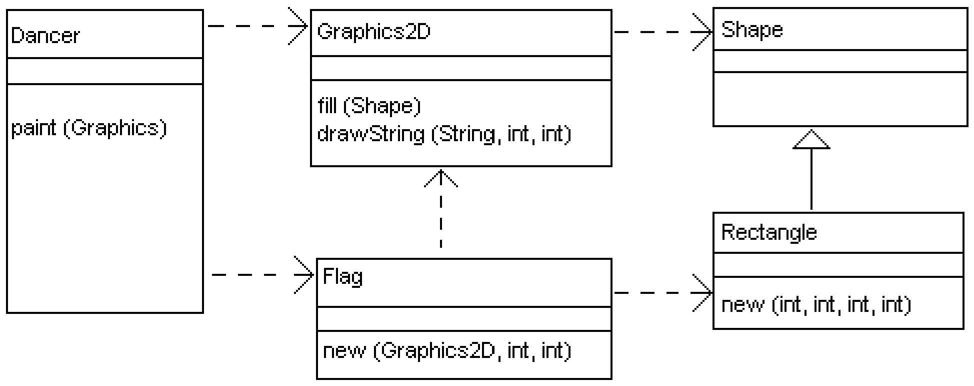


Figure 8.3 UML class diagram for Version 2

The Flag class defines six variables (names all in capital letters: `FLAG_WIDE`, `UNION_TALL`, etc.) that give the dimensions of the flag. These are defined as `final`, which just means that they can never be changed. That makes it safe to have them be public. In the further development of the logic, these constants are used in place of the actual numbers.

Exercise 8.11 Explain why it is not a good plan to do the three steps of the main Flag logic in the opposite order.
Exercise 8.12 What changes would you make in Listing 8.4 to make the flag twice as big?
Exercise 8.13* Find a picture of the Indiana state flag on the internet. Lay out a structured natural language design for drawing it. Or use some other picture of some complexity.
Exercise 8.14* **(Archery)** Reorganize your Archery applet to have the `paint` method call a `drawArrow` method to draw an arrow on the left side of the drawing area and a `drawBullseye` method to draw a bullseye on the right side of the drawing area.

Listing 8.4 The Flag class, version 1

```
import java.awt.*;

public class Flag
{
   public static final int FLAG_WIDE  = 173;
   public static final int FLAG_TALL  = 91;
   public static final int UNION_WIDE = 69;
   public static final int UNION_TALL = 49;
   public static final int PIP = 2;
   public static final int STAR = 5;
   ////////////////////////////////////
   private Rectangle itsUnion = new Rectangle
                    (UNION_WIDE, UNION_TALL);  // width & height


/** Draw the flag with top-left corner at <x,y>. */

   public void draw (Graphics2D page, int x, int y)
   {  drawStripes (page, x, y);
      itsUnion.setLocation (x, y);
      page.fill (itsUnion);
      drawAllStars (page, x, y);
   }  //======================


   /** Draw 13 stripes in a 173x91 area, top-left at <x,y>. */

   private void drawStripes (Graphics2D page, int x, int y)
   {  page.drawString ("Here be stripes", x + 50, y + 70);
      // to be developed later
   }  //======================


   /** Draw 50 stars in a 69x49 area, top-left at <x,y>.*/

   private void drawAllStars (Graphics2D page, int x, int y)
   {  page.drawString ("Here be stars", x, y);
      // to be developed later
   }  //======================
}
```

8.4 Iterative Development Of The Flag Software

Version 1 of Dancer is compilable and runnable. It produces a result that is partial progress towards the desired end result. Now that we have Flag version 1 and Dancer version 2, we can run the applet again. This result is even closer to the desired end result.

Figure 8.4 (see next page) shows what Dancer v.2 produces. This is the result of version 2 of the iterative development. The result of version 1 was in the earlier Figure 8.1, obtained from Dancer v.1 and no Flag class at all.

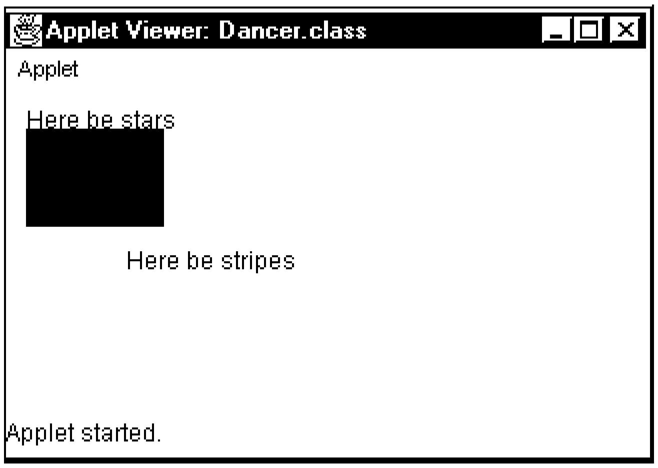


Figure 8.4 After iterative version 2 of the Flag software

In general, once we have a full analysis of a problem for which the software is quite complex, we divide the development of the software into a succession of several versions. Each version adds more functionality to the software, that is, either it does more of what the final version should do or it does better what the previous version already does.

In developing each of these versions, we design the next modifications in the existing software, then implement them in Java, then test them. This is one full cycle in **iterative development** of the software.

In larger software projects, it is typical to have an overall written plan of iterative development where each cycle is estimated to take from two weeks to two months (depending on the type of software). We decide what we want to have at the end of each cycle (the "deliverables").

Next we process one version at a time: First we design what we want to have when we finish the improvement. Then we develop that software in smaller steps of design/implement/test (sort of a sub-iterative development) as illustrated for this Flag software. If the initial analysis was not sufficiently detailed, additional analysis is required for the version before its design stage.

Experience has shown that iterative development of software is effective and efficient. Do not leap into the implementation stage in your programming -- follow this plan -- learn from others' experience. Remember, even Beethoven took lessons in music from more experienced people (although Joe Haydn was so irritated by his unorthodox ideas that he made Beethoven drop the class; but then Beethoven was a genius).

Ease of finding errors

In Figure 8.4, the filled rectangle is black. It was supposed to be blue. This indicates a bug in the software so far. The command to set the drawing Color to `Color.blue` was left out.

This sort of thing happens all the time, in accordance with the basic programming principle, "You're only human, you're supposed to make mistakes." It is easier to find and correct errors such as this when only an incremental change has been made to the software. This is a big advantage of iterative development.

Version 3 of iterative development

The next iterative step is to complete the entire flag, but without animation. The `drawStripes` method is simpler than the `drawStars` method, so we start with the `drawStripes` method. This is in accordance with the basic programming principle, "Always put the hard stuff off until later."

Development of the drawStripes method

You need to draw 13 stripes each 7 pixels tall. They alternate red with white, starting with the red at the top. So each stripe is done by setting the drawing Color to `Color.red` or `Color.white` and then drawing a rectangle. You need to have a field of stars in the top-left corner of the flag, but it is simplest to draw all of the stripes all the way across, since the later drawing of a rectangle puts the blue union over the stripes.

STRUCTURED NATURAL LANGUAGE DESIGN for drawStripes

```
1.  Create a stripe that is 173 pixels wide and 7 pixels tall.
2.  Repeat the following 13 times....
        2a.  Set the color to red or white alternately, starting with red.
        2b.  Draw the stripe.
        2c.  Move the stripe 7 pixels further down the page.
```

The top-left corner of the flag is to be at `<x,y>`. These `x` and `y` values are parameters of the `drawStripes` method. The rectangle should be created with 173 (the width) for its first parameter and 7 (the height) for its second parameter.

The problem is to get the color to switch back and forth between red and white. One way is to have an integer variable `k` that counts how many stripes you have made. If `k` is an even number, i.e., `k % 2 == 0`, make the next stripe red, otherwise make it white. The result is in Listing 8.5. The `page.setColor` statements use either `Color.red` or `Color.white` depending on whether `k` is even.

Listing 8.5 The drawStripes method in the Flag class

```java
   private Rectangle itsStripe = new Rectangle
                          (FLAG_WIDE, FLAG_TALL / 13);

   /** Draw 13 stripes in 173x91 area, top-left at <x,y>. */

   private void drawStripes (Graphics2D page, int x, int y)
   {  itsStripe.setLocation (x, y);
      for (int k = 0;  k < 13;  k++)
      {  if (k % 2 == 0)
            page.setColor (Color.red);
         else
            page.setColor (Color.white);
         page.fill (itsStripe);
         itsStripe.translate (0, TALL);
      }
   }  //======================
```

If you wanted to draw a flag with three colors of stripes that alternate, you could have an int variable that you test to see if it is a multiple of 3 (so draw a red stripe), 1 more than a multiple of 3 (so draw a white stripe), or 2 more than a multiple of 3 (so draw a blue stripe) as follows:

```
if (k % 3 == 0)
   page.setColor (Color.red);
else if (k % 3 == 1)
   page.setColor (Color.white);
else
   page.setColor (Color.blue);
```

Development of the drawStars method

The Flag specifications require you to draw 9 rows of 5 or 6 white stars within a 69x49 field of blue. Each star is to be within a bounding 5x5 square of pixels (but you do not draw the 5x5 square itself). The stars should be 12 pixels apart as you go across. So a row of 6 stars takes up 65 pixels, `(5 * 12 + 5)`, leaving 2 pixels on each side.

Each row starts 5 pixels below the one above it, so 9 rows require 45 pixels. Since you have 49 pixels of height, that again leaves 2 pixels on the top edge and 2 on the bottom edge. Therefore, for a blue union whose corner is at `<x,y>`, the top-left star should be at `<x+2,y+2>`. Once you make a drawing to clarify the positioning of the stars, you have enough information to create the overall plan to draw the 50 stars, as shown in the accompanying design block.

STRUCTURED NATURAL LANGUAGE DESIGN for drawStars

```
1. Set the drawing Color to white.
2. Repeat the following 9 times...
      3a. Draw a row of 6 or 5 stars indented 2 or 8 pixels, alternating.
      3b. Shift downwards by 5 pixels for the next row.
```

The actual star will just be five lines. So you make a 5x5 grid of squares and see just what lines have to be drawn to get a star. This requires some work, but eventually you decide on a nice-looking set of five lines that will form a star. That is enough to warrant `drawOneStar` as a separate method. Except for `drawStripes`, the completed Flag class is in Listing 8.6 (see next page), and the flag itself is in Figure 8.5. Reminder: `x += y` is preferable to `x = x + y`, and `x -= y` is preferable to `x = x - y`.

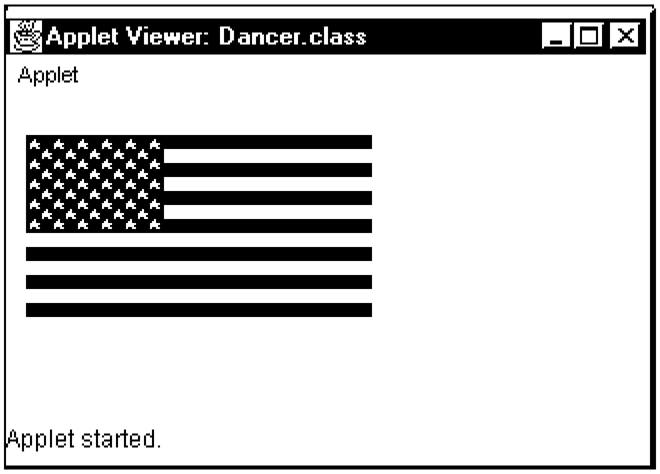


Figure 8.5 After Version 3 of iterative development of the Flag software

Listing 8.6 The Flag class, version 2 (final)

```
import java.awt.Graphics2D;
import java.awt.Color;
import java.awt.Rectangle;

public class Flag
{
   public static final int FLAG_WIDE  = 173;
   public static final int FLAG_TALL  = 91;
   public static final int UNION_WIDE = 69;
   public static final int UNION_TALL = 49;
   public static final int PIP = 2;
   public static final int STAR = 5;
   ////////////////////////////////////
   private Rectangle itsUnion = new Rectangle
                    (UNION_WIDE, UNION_TALL);  // width & height


   public void draw (Graphics2D page, int x, int y)
   {  drawStripes (page, x, y);
      page.setColor (Color.blue);
      itsUnion.setLocation (x, y);
      page.fill (itsUnion);
      drawAllStars (page, x, y);
   }  //======================


   private void drawAllStars (Graphics2D page, int x, int y)
   {  page.setColor (Color.white);
      for (int row = 0;  row < 9;  row++)
      {  int start = (row % 2 == 0)  ?  x + PIP : x + PIP + 6;
         drawRowOfStars (page, x + UNION_WIDE - STAR, start,
                         y + PIP + row * STAR);
      }
   }  //======================


   private void drawRowOfStars (Graphics2D page, int end,
                                int x, int y)
   {  for (;  x < end;  x += 12)
         drawOneStar (page, x, y);
   }  //======================


   private void drawOneStar (Graphics2D page, int x, int y)
   {  int twoPips = PIP * 2;
      page.drawLine (x + PIP, y, x, y + twoPips);
      page.drawLine (x, y + twoPips, x + twoPips, y + PIP);
      page.drawLine (x + twoPips, y + PIP, x, y + PIP);
      page.drawLine (x, y + PIP, x + twoPips, y + twoPips);
      page.drawLine (x + twoPips, y + twoPips, x + PIP, y);
   }  //======================
}
```

Exercise 8.15 How would you modify the `drawStripes` method to have 20 stripes each 5 pixels tall, alternating between green and orange?
Exercise 8.16 The star-drawing methods use a number that depends on the size of the flag, rather than a constant. If the flag were doubled in size, that number would not change and the flag would not look right. Define a new constant and rewrite those methods accordingly.
Exercise 8.17 Rewrite the `drawAllStars` method without the conditional operator.
Exercise 8.18 Revise the star-drawing methods to draw 6 rows of stars, alternating between 4 and 5 stars in each row, starting with 4. Use the same spacing; do not try to fill up the entire width of the union.
Exercise 8.19 (harder) Write a `drawStripes` method that has 20 stripes cycling among four different colors.
Exercise 8.20 (harder) Rewrite the `drawStripes` method to use a boolean variable named `isRed` instead of checking whether the counter is even.
Exercise 8.21* Rewrite the `drawStripes` method to have the body of the for-statement (a) set the color to white, (b) draw that stripe, (c) set the color to red, (d) draw that stripe. You need to draw the first red stripe before the for-statement. This executes faster than Listing 8.5 but is harder to develop.
Exercise 8.22* Does Listing 8.6 use too many constants, thereby obscuring the logic? Or should it have even more constants such as for the number of rows or number of stars?
Exercise 8.23* (Archery) Write the `drawArrow` method. Aim the arrow at the target.
Exercise 8.24* (Archery) Complete your `drawBullseye` method: Put the target on a stand and make the target look three-dimensional.
Exercise 8.25** Rewrite Listing 8.5 to not use the `translate` method. Instead, create a new Rectangle for each stripe.
Exercise 8.26** Create an Applet that places several dozen rectangles of randomly chosen positions, width, and height on the drawing area, using four different colors. Make it look like a Mondrian. Or would ovals look even better?

8.5 Animation In A JApplet

The only thing left is to animate the flag, i.e., have it dance. One common way to do this to have the applet's `start` method repeatedly set a new location for the drawing and then call the applet's `paint` method. The following JApplet method is needed:

- `someJApplet.setVisible(true)` makes the drawing area visible. The browser automatically does this before it calls the applet's `paint` method, but you need to do it yourself if you call the `repaint` method from the `start` method. Otherwise, whatever the `start` method draws will not show up. Naturally, you could hide the drawing area by using a parameter value of `false` instead of `true`.

The browser executes an applet's `start` method when it returns to the web page that contains it after having browsed elsewhere. If your applet provides a method with the heading `public void stop()`, the browser will execute that method when it leaves the applet's web page. This is important for animations that go on for a long time, to keep them from taking up resources when the applet is not displayed. However, you will not need a `stop` method for this Flag applet.

If you only want an action performed when the browser first loads your applet, but not repeated every time the browser returns to the web page, you should put that action in a method with the heading `public void init()`.

Flag analysis and design

Analysis: You do not know just what the client considers a decent dance routine, so you decide to make a reasonable estimation and then see how the client likes it. After some thought, you decide to have the flag dance to the right, bobbing up and down, then bob back to the left, repeating this cycle twice and then stopping.

Design: To have the Flag dance to the right, you could have `itsLeftEdge` change from 10 to 50 one pixel at a time (since the Flag is 173 pixels wide and you have 320 pixels of space). To get the bobbing effect, you should change its placement below the top of the frame, which is determined by `itsTopEdge`. The first 10 times that `itsLeftEdge` increments, you could have `itsTopEdge` increase by 4s (from 20 up to 60); the next 10 times it decreases by 4s (from 60 back to 20); the next 10 it increases by 4s (from 20 up to 60), the next 10 it decreases by 4s, etc. Dancing to the left repeats this process except `itsLeftEdge` decrements each time. Much effort is required to get the dancing to the right into Java logic:

```
private void danceRight()  // called by Dancer's start method
{  itsTopEdge = 20;
   for (itsLeftEdge = 10;  itsLeftEdge <= 49;  itsLeftEdge++)
   {  repaint();
      itsTopEdge += (itsLeftEdge / 10 % 2 == 1) ?  4 : -4;
   }
}  //======================
```

When you try out the dancing flag, you see that it is looking good except that the flag moves way too fast. You need a way to slow it down. A handy method in the System class gives the current time, measured in milliseconds since January 1, 1970, as a long value (not int). So you add a `pause` method to the Dancer class to wait a given number of milliseconds, and you insert a `pause(50)` command after each call of the `repaint` method. This completes the final iterative Version 4 of the Flag software, shown in Listing 8.7 (see next page).

The `pause` method is declared as a class method to make it clear that it does not mention any instance variables or instance methods of the Dancer object. It calculates the `timeToQuit` as the specified number of milliseconds after the time at which the `pause` method is called. Then the while-statement loops, doing nothing except wastefully tying up the processor, until that time comes. This **busywait logic** is inferior to more advanced techniques you will learn to use later: a Timer, which is discussed in Chapter Ten, or Thread.sleep, which is discussed in Chapter Eleven. Actually, you could simply replace the two calls of `pause(50)` by `ThreadOp.pause(50)` from Chapter Eleven to get the same effect except it frees the processor to do other things.

Good programming style dictates that `itsLeftEdge` and `itsTopEdge` should be parameters of the call to `repaint` instead of instance variables, because of how they are used. But they cannot be, since `repaint()` is a method inherited from the JApplet superclass with only one parameter, of type Graphics. So they are instance variables.

Exercise 8.27 (harder) Explain why `itsTopEdge += 4 * (itsLeftEdge / 10 % 2 * 2 - 1)` gives the same result as the assignment statement in `danceRight`. Discuss which of the two expressions is clearer.
Exercise 8.28 (Archery)** Animate your Archery frame (or applet, as the case may be) by having the arrow fly into the target. Repeat the flight 15 times. Every fourth time, have the target jump up in the air just before the arrow hits, then come back down. Print the words "you missed!" each of those times.
Exercise 8.29* (Archery) Try out various pause periods for your Archery frame until you find ones you like. Make the pause between arrow launches far longer than the pauses between changes in position of the arrow as it flies.

Listing 8.7 The Dancer applet, Version 3 (final)

```
import javax.swing.JApplet;
import java.awt.*;

public class Dancer extends JApplet
{
   private Flag itsFlag = new Flag();
   private int itsLeftEdge;
   private int itsTopEdge;


   public void start()
   {  setVisible (true);
      for (int count = 0;  count < 2;  count++)
      {  danceRight();
         danceLeft();
      }
   }  //======================


   private void danceRight()
   {  itsTopEdge = 20;
      for (itsLeftEdge = 10;  itsLeftEdge <= 49;  itsLeftEdge++)
      {  repaint();
         pause (50);
         itsTopEdge += (itsLeftEdge / 10 % 2 == 1)  ?  4 : -4;
      }
   }  //======================


   private void danceLeft()
   {  itsTopEdge = 20;
      for (itsLeftEdge = 49;  itsLeftEdge >= 10;  itsLeftEdge--)
      {  repaint();
         pause (50);
         itsTopEdge += (itsLeftEdge / 10 % 2 == 1)  ?  -4 : 4;
      }
   }  //======================


   private static void pause (int wait)
   {  long timeToQuit = System.currentTimeMillis() + wait;
      while (System.currentTimeMillis() < timeToQuit)
      {  }  // take no action
   }  //======================


   public void paint (Graphics g)
   {  Graphics2D page = (Graphics2D) g;
      page.setColor (Color.white);
      page.fillRect (0, 0, 320, 160);  // clear the area
      itsFlag.draw (page, itsLeftEdge, itsTopEdge);
   }  //======================
}
```

8.6 Review Of The Software Development Paradigm

You have now seen all the basic elements of a workable and efficient process for developing software. This section reviews them and expands on them to be applicable to a very large range of problems. Some software situations will not call for all of the elements, but most complex ones will.

Stage 1: Analysis Study the problem to see exactly what is required of the software. See what parts of the description can be interpreted in more than one way, then find out from the client which is wanted. If you are working under contract with the client, write out the details of these specifications in a document that can be attached to the contract. The primary criteria for the specifications are that (a) they be understandable by the client and a judge (to enforce the contract so you get paid), and (b) they be precise enough that designers and implementers do not need to return to you for further clarification.

Make sure the specifications are **consistent** -- one part does not contradict another part. Make sure they are **complete** -- all possible cases have been covered, not just the ones you expect to occur. And make sure that the requirements are **feasible** -- they can be implemented in software without an unacceptable slowness of response.

Consider which sets of input data you will need to test each aspect of the software. Work out what the output or display will be in each case. Review the specifications to help you decide on good test sets. Use this opportunity to make sure that the specifications are clear and complete.

Stage 2: Logic Design Lay out the overall sequence of operations the software will perform, from beginning to end. Keep it to a 5- to 10-step description. For event-driven software, as you will see in Chapter Ten, it may be no more than the following two steps:

1. Prepare the frame or applet with buttons, textfields, and whatnot.
2. Wait for someone to click or take other action.

Stage 3: Object Design Decide on the major kinds of objects that will play a role in the software. Decide what operations each will carry out. For each major sub-task that the software has, you should have an object that performs that task, or a set of objects that cooperate in performing that task. Look for objects in your personal library or the standard library that can perform the tasks. If those are not sufficient, see if you can add to the capabilities of existing objects. And if that does not work, invent entirely new objects to perform the tasks.

Stage 4: Version 1 of Iterative Development Choose a subset and/or simplification of the specifications that looks like it can be implemented in a short period of time. Develop the logic for that subset as a Structured Natural Language Design. Run whichever of the test sets are appropriate to see if the software accomplishes the goal set for Version 1.

Stages 5, 6,...: Versions 2, 3,... of Iterative Development Choose a larger subset of the specifications each time, adding more functionality, until you eventually reach the end result. Quite often the top level of the software does not change as you progress through these versions; instead, the object classes become more effective. It is normal to discard 10% or more of what you had on the previous version to make it better (even Brahms threw out much of what he wrote; what is left of Brahms' work is of uniformly outstanding beauty -- a word to the wise).

Stage N: Run the tests Make sure that each test set produces the right output. For complex programs, this testing process should be automated. You can do this by e.g. replacing the bodies of the IO methods of Listing 6.2 by logic that reads input from a disk file (rather than the user) and writes output to a disk file. You can then run a program to compare the actual output file with another disk file that contains the expected output. That comparison program should report on any discrepancy between the two files so you know what tests found a bug.

Nota Bene: The technical definition of a **bug** is any disagreement with the specifications. If you do something your way instead of the client's way just because you think it looks better and performs better, it is still a bug. You should either (a) convince the client that your way is better than the client's way, or (b) do it the client's way, or (c) refuse to take the job. Remember who is paying for it. Remember Mozart. Nota Multum Bene: In the case of a homework assignment for a course, the instructor is your client.

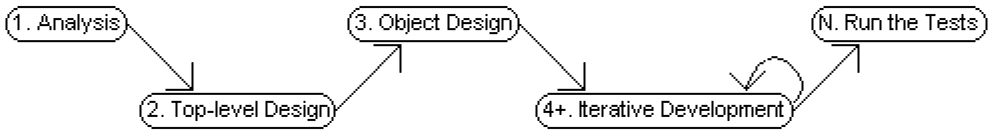


Figure 8.6 Five (or more) stages of software development

8.7 Common Looping Patterns

You have now seen many segments of logic that use looping statements. You have probably noticed that some patterns appear over and over. Probably more than half of all loops you run across fit one of the patterns discussed in this section.

Deciding when the loop stops

<u>Sentinel-controlled loop</u> Each iteration of the loop reads one value from the user or from a disk file. These are data values to be processed as they are read in or to be stored for later processing. When you reach the end of the data values to be processed, the next value obtained is special value that is recognizable as not suitable for processing. This **sentinel** signals the end of the sequence of values.

The while-statements in Listing 7.3 and Listing 7.4 have this logic, where you obtain a Worker value whose name is null after you have obtained all of the data values to be processed. So the sentinel value here is a Worker with a null name:

```
Worker data = new Worker (file.readLine());
while (data.getName() != null)
{  data.processThisDataValue();
   data = new Worker (file.readLine());
}
```

<u>Count-controlled loop</u> The number of times N you want to carry out a process is known when the loop begins. A counter keeps track of the number of iterations and the loop stops when you have done the process N times, or sooner if you accomplish the purpose of the loop before N times through the loop. An example is the following from Listing 8.6:

```
for (int row = 0;  row < 9;  row++)
{  // statements that draw one row of stars
}
```

<u>Iterator-controlled loop</u> You have a sequence of values to be processed. You keep track of a position in that sequence. Each iteration of the loop advances the position by one step. The loop terminates when you come to the end of the sequence, or sooner if you accomplish the purpose of the loop before you reach the end. If the sequence is stored in an array, the index in the array is the iterator. If it is a sequence of slots for a Vic object, you use `moveOn()` to advance the position indicator, as follows:

```
   for (int k = 0;  k < par.length();  k++)
   {  if (par.charAt (k) == ' ')
         return par.substring (0, k);
   }
   return par;
   for (Vic sam = new Vic();  sam.seesSlot();  sam.moveOn())
      sam.processThisSlot();
```

User-controlled loop You repeatedly ask for input from the user until he or she provides a value that indicates the loop is to terminate. One example is an attempt to get an acceptable input (the `IO.askInt` method gets an integer from the user):

```
   int choice = IO.askInt ("Enter a number from 10 to 20: ");
   while (choice < 10 || choice > 20)
   {  IO.say (choice + " is not in the range from 10 to 20.");
      choice = IO.askInt ("Enter a number from 10 to 20: ");
   }
```

Another example is a menu-driven loop, where you ask the user which of several choices he or she wishes to make. The various choices determine what is done inside the loop, except that one of the choices tells you that the user wishes to terminate the entire process. These examples are actually variants of the sentinel-controlled loop:

```
   String prompt = "Enter A to add, D to delete, P to print, "
                 + " or E to exit: ";
   char choice = IO.askLine (prompt).charAt (0);
   while (choice != 'E')
   {  processCurrentChoice (choice);
      choice = IO.askLine (prompt).charAt (0);
   }
```

Deciding what to do inside the loop

Those four patterns categorize how you decide when a loop should terminate. The other part of a loop is the action it performs at each of the items it processes. In each of the following common action patterns, the same iterator-controlled looping pattern is used for uniformity, namely, processing the elements of an array. All of these action patterns can occur in sentinel-controlled loops and count-controlled loops as well.

Count-cases looping action The problem is to count how many of the items to be processed have a certain value. To do this, you initialize a counter to zero before the loop, then you add 1 each time you see an item with the required property:

```
   int count = 0;
   for (int k = 0;  k < a.length;  k++)
   {  if (someProperty (a[k]))
         count++;
   }
   return count;
```

Some-A-are-B looping action The problem is to find out whether at least one of the items has a certain property. You do this by returning `true` when you see an item with the required property as you progress through the loop. If you exit the loop without having returned `true`, then return `false` as the answer:

```
   for (int k = 0;  k < a.length;  k++)
      if (someProperty (a[k]))
         return true;
   return false;
```

An alternative is to make the loop stop when you see some item with the required property. That puts the `return` statement only at the end of the method. The body of the for-statement has no processing, so we write it as a while-statement. Its only purpose is to find the right value of `k`.

```
int k = 0;
while (k < a.length && ! someProperty (a[k]))
   k++;
return k < a.length;
```

The condition in the while-statement takes advantage of short-circuit evaluation: Evaluating `someProperty(a[k])` when `k` is not less than `a.length` would throw an IndexOutOfBoundsException.

All-A-are-B looping action The problem is to find out whether all of the items have a certain property. This is sort of the opposite of the Some-A-are-B looping action:

```
for (int k = 0;  k < a.length;  k++)
{  if ( ! someProperty (a[k]))
      return false;
}
return true;
```

Process-some-cases looping action The problem is to apply a certain process to those items that have a certain property. An example of this is the while-statement in Listing 7.7, where the process is to store all items away for later use. Another example is the last for-statement in the same listing, where the process is to print certain items in the sequence. Processing could also be modifying the state of object items:

```
for (int k = 0;  k < a.length;  k++)
{  if (someProperty (a[k]))
      processCurrentItem (a[k]);
}
```

Find-max-or-min looping action The problem is to find the largest of the items, or sometimes to find the smallest of the items. The two cases are analogous, so only finding the largest is discussed here. The largest item you have seen so far is stored in say `largestSoFar`. To process a given item, you compare it with the `largestSoFar`. If the current item is larger, replace the `largestSoFar` by that item, otherwise make no change. But you cannot apply this logic to the first item, because you have no value for `largestSoFar` at that point. So you treat the first item differently:

```
Object largestSoFar = a[0];
for (int k = 1;  k < a.length;  k++)
{  if (largestSoFar.compareTo (a[k]) < 0)
      largestSoFar = a[k];
}
return largestSoFar;
```

In each of these looping action patterns, you need to find out what you are to do if there are zero items to process. The Count-cases action returns 0, the Some-A-are-B action says no one is, the All-A-are-B action says that they all are (which is vacuously true, since none of them are not). But the Find-max-or-min action returns an invalid value: `a[0]` is garbage or it throws an IndexOutOfBoundsException because the array has length zero. So you need to put `if (a.length == 0) return null` or some other crash-guard at the beginning of that logic, whatever is appropriate to the situation.

Exercise 8.30 Rewrite the All-A-are-B example to put the return after the for-statement, as illustrated for the Some-A-are-B action.
Exercise 8.31 Rewrite the Find-max-or-min example to find the minimum rather than the maximum.
Exercise 8.32 Would it crash-guard the Find-max-or-min example against a length of zero if you replaced the first statement by:
`Object largestSoFar = (a.length == 0)  ?  null  :  a[0];?`
Exercise 8.33* One of the while-statements in Listing 6.1 is a sentinel-controlled loop. Which one is it?
Exercise 8.34* Categorize the looping actions of all loops in Listing 7.8 and Listing 7.9.
Exercise 8.35** Categorize the looping actions of all loops in Chapter Six listings and answers to exercises.

8.8 Case Study In Animation: Colliding Balls

This section will give you additional experience with applets, arrays of objects, inheritance, and especially polymorphism. No important new material is presented in this section.

Problem See if you can create an applet with a great deal of action: Have about twenty red balls bouncing around within a rectangle. When one red ball hits another red ball, it "poisons" that other ball. The poisoned ball turns blue and stops moving. It sits in that one spot for a while. Then it disappears and is reborn along the bottom as a moving red ball. This action continues for a minute or two, then it stops.

Design of the main logic

This will not be too hard to design if you work up to it slowly. You will need one JApplet object and many Ball objects. The overall logic is (1) create 21 balls, then (2) start them bouncing, hitting, and dying. The first step can be implemented in just three lines:

```
   final static int SIZE = 21;
   for (int k = 0;  k < SIZE;  k++)
      itsItem[k] = new Ball();
```

The `start` method of the applet will execute for several thousand cycles, whatever turns out to be around two minutes (you will have to try out several numbers to see which works best). Each cycle will check out each of the 21 balls and take the appropriate action depending on its current status. See the accompanying design block.

DESIGN for the main logic of Colliding Balls

```
1. For several thousand cycles, do...
        1a. For each one  x  of the 21 Balls, do...
                1aa. If  x  is healthy, then...
                        1aaa. Move  x  and see if it hits any other Ball.
                        1aab. If so, and if that other Ball is healthy, then...
                                Poison that other healthy Ball.
                1ab. Otherwise, if  x  is "dying" then...
                        It deteriorates a little more.
                1ac. Otherwise,  x  is "dead" so...
                        It reappears as a new healthy Ball.
        1b. Paint the drawing area to show the changed position of the 21 Balls.
```

Painting the drawing area should begin by clearing out the existing drawing with a filled rectangle. Then it can just draw each Ball's image, perhaps as follows:

```
for (int k = 0;  k < SIZE;  k++)
   itsItem[k].drawImage (page);
```

Both the `start` method and the `paint` method have to refer to the `itsItem` array, so it must be an instance variable of the applet.

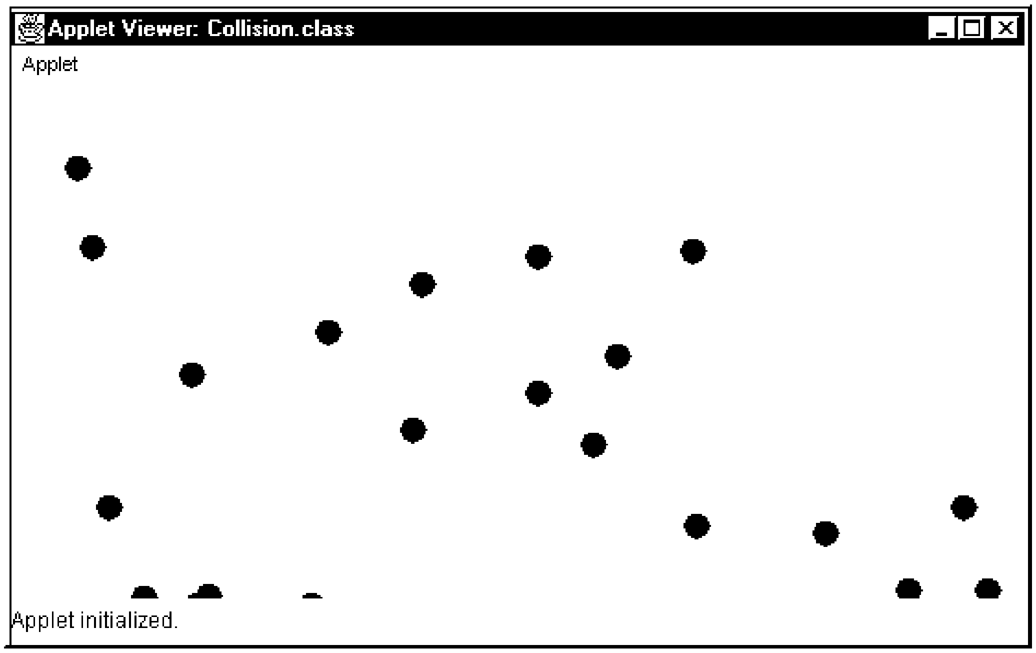


Figure 8.7 The Colliding Balls applet

The logic for telling whether a certain Ball `guy` hits anyone else is to go through the array looking at all the other Balls one at a time until you see one that `guy` does not miss by much. If that one is healthy, then it gets poison.

Listing 8.8 (see next page) expresses this Collision applet logic in Java. It is clear that the Ball class has to be told the dimensions of the rectangle the Balls can move within. This must be done before any Balls can be created, so a special initializing class method is called. The parameters are `getWidth()` and `getHeight()`, two methods that can be used with any applet to find its width and height.

The purpose of the while statement at the end of the `start` method is to pause for a fraction of a second at the end of each cycle to keep the Balls from moving too fast on the screen. This is the same logic as used in the Flag software. The purpose of the while statement in the `moveAndCheckHits` method is to go through all 21 Balls, adding 1 to the index each time, until you see a Ball that you do not miss (by more than 2 or 3 pixels) or you come to the end of the sequence of Balls.

The Ball class

If an object is to move on the screen, it needs to keep track of its current position. These are the instance variables `<itsXspot, itsYspot>` in the Ball class. It also keeps track of the amount to change `x` by, and the amount to change `y` by, each time it moves. These are the instance variables `<itsXmove, itsYmove>` in the Ball class.

Listing 8.8 The Collision JApplet

```
import java.awt.*;

public class Collision extends javax.swing.JApplet
{
   public static final int NUM_CYCLES = 4000;
   public static final int SIZE = 21;      // number of Balls
   private Ball[] itsItem = new Ball [SIZE];  // the 21 Balls

   public void start()
   {  this.setVisible (true);
      Ball.initialize (this.getWidth(), this.getHeight());
      for (int k = 0;  k < SIZE;  k++)
         itsItem[k] = new Ball();
      for (int cycle = 1;  cycle <= NUM_CYCLES;  cycle++)
      {  updateEachBall();
         repaint();
         long timeToQuit = System.currentTimeMillis() + 90;
         while (System.currentTimeMillis() < timeToQuit)
         {  }  // take no action
      }
   }  //======================

   private void updateEachBall()
   {  for (int k = 0;  k < SIZE;  k++)
      {  if (itsItem[k].isHealthy())
            moveAndCheckHits (itsItem[k]);
         else if (itsItem[k].isDying())
            itsItem[k].deteriorates();
         else
            itsItem[k] = new Ball();
      }
   }  //======================

   private void moveAndCheckHits (Ball guy)
   {  guy.move();
      int c = 0;
      while (c < SIZE && itsItem[c].misses (guy))
         c++;
      if (c < SIZE && itsItem[c].isHealthy())
         itsItem[c].getsPoison();
   }  //======================

   public void paint (Graphics g)
   {  Graphics2D page = (Graphics2D) g;
      page.setColor (Color.white);
      page.fill (new Rectangle (getWidth(), getHeight()));
      for (int k = 0;  k < SIZE;  k++)
         itsItem[k].drawImage (page);
   }  //======================
}
```

To get the Ball rolling, you can pick `<itsXspot, itsYspot>` at random along the bottom of the rectangle where the Balls will be moving, though not in the first or last ten percent of the area (so it will look better). Some experimentation came up with a change in the x-direction of up to plus or minus 2 pixels per cycle, and an initial change in the y-direction of -1 to -3 pixels per cycle. This is because you subtract from `y` each time to have it move up from the bottom of the rectangle.

One way to have the Ball stand still for say 100 cycles before it is reborn is to subtract 1 from a counter on each cycle, starting from 100, and call it dead when the counter reaches zero. You can start a new Ball with the counter (called `itsStatus`) at 100, and keep it there until the Ball is hit. This gives you a way of telling whether a particular Ball is healthy. When it is hit, change `itsStatus` to 99, which signals that the Ball is not healthy. Then deterioration simply amounts to subtracting 1 from `itsStatus` each time. It is "dying" as long as `itsStatus` remains positive but less than 100. Listing 8.9 contains this much of the Ball class.

Listing 8.9 The Ball class except for move, misses, and drawImage

```
import java.util.Random;
import java.awt.geom.Ellipse2D;
import java.awt.*;

public class Ball
{
   private static final int HEALTHY = 100;  // cycles to languish
   private static Random randy = new Random();
   private static int theRight;    // right side of the screen
   private static int theBottom;   // bottom edge of the screen
   //////////////////////////////////
   private int itsXspot;           // x-value of current location
   private int itsYspot;           // y-value of current location
   private int itsXmove;           // change in x each cycle
   private int itsYmove;           // change in y each cycle
   private int itsStatus;          // tells how healthy it is


   public static void initialize (int rightSide, int bottom)
   {  theRight = rightSide;
      theBottom = bottom;
   }  //======================

   public Ball()
   {  itsXspot = (theRight * (1 + randy.nextInt (9))) / 10;
      itsYspot = theBottom - randy.nextInt (40);
      itsXmove = randy.nextInt (5) - 2; // range -2 to 2
      itsYmove = randy.nextInt (3) - 3; // range -1 to -3
      itsStatus = HEALTHY;
   }  //======================

   public boolean isHealthy()
   {  return itsStatus == HEALTHY;
   }  //======================

   public void getsPoison()
   {  itsStatus = HEALTHY - 1;
   }  //======================

   public boolean isDying()
   {  return itsStatus < HEALTHY && itsStatus > 0;
   }  //======================

   public void deteriorates()
   {  itsStatus--;
   }  //======================
}
```

Movement for one cycle is mainly a matter of adding `itsXmove` to `itsXspot` and adding `itsYmove` to `itsYspot`. However, you have to check whether the Ball has reached one of the boundaries of the rectangle within which it is to move. If it reaches the left or right side, you only need replace `itsXmove` by the negative of `itsXmove`. If it reaches the top or bottom, you replace `itsYmove` by the negative of `itsYmove`. For instance, `itsXmove = -itsXmove` changes 2 to -2 but changes -2 to 2.

The image you draw is just a ball shape, using an Ellipse2D.Double object from the `java.awt.geom` package. You have to choose a diameter; call it UNIT. Since the ball is drawn to the right and below `<itsXspot, itsYspot>`, you want to have the ball "bounce" off the right edge when it is within UNIT pixels of the right edge, and to bounce off the bottom when it is within UNIT pixels of the bottom. At the left edge, you have it bounce when `itsXspot` is 1, and similarly at the top edge.

A test run showed that this causes a problem. Balls are initially "born" in the area just above the lower boundary. If a Ball is born less than UNIT pixels above the lower boundary and moving upwards, and thus `itsYmove` is negative, it is treated as if it had just arrived there from above, and `itsYmove` is changed to be positive. That makes it move further down, so `itsYmove` is changed to be negative again. The result is that the Ball is stuck in that lower area. Fixing this problem is left as an exercise.

How do you tell when the executor misses the other guy? Of course, you count it as a miss if the guy is the executor itself. Otherwise, you can count it as a miss if the distance between them (looking at `<itsXspot, itsYspot>`) is more than 2. This logic is in Listing 8.10 (see next page), where the standard library `Math.abs` method is used to find the absolute value of the difference of two numbers: `Math.abs(x)` is the absolute value of `x`.

The `drawImage` method just sets the drawing Color to red or blue depending on whether the ball is healthy and then draws the circle.

Major programming project

Define three subclasses of the Ball class called Scissors, Paper, and Rock. At the beginning of the applet's `start` method, create SIZE/3 objects from each subclass of Ball instead of creating SIZE ordinary Balls. Have the `drawImage` method in each subclass produce a shape that actually looks like Scissors or Paper or Rock. To make the Rock, look up the Arc2D.Double constructor with PIE type. Use various colors.

The challenging part is what happens when two of these subclass types collide. Whether the Scissors hits the Rock or the Rock hits the Scissors, the Scissors should die and come back as a Rock. Similarly, if Scissors and Paper collide, the Paper should die and come back as Scissors. And if Paper and Rock collide, the Rock should die and come back as Paper.

If two objects of the same type collide, they should both die and come back as one each of the other two types. For instance, if two Scissors collide, one comes back as a Rock and the other as a Paper. Use the `instanceof` operator that you have not seen before, e.g., `someBall instanceof Paper` will be `true` if `someBall` is an object from the Paper class but not if it is an object from the Rock or Scissors class.

You will need to have a `hitsOther` method in each of the four classes and replace the call of `getsPoison` in `moveAndCheckHits` by `itsItem[c].hitsOther(guy)`. This polymorphic `hitsOther` method will take the appropriate action in each case, poisoning the right object. And you need to tell each Ball when it is poisoned what kind of thing it is to be reborn as when the 100 cycles are up. You should also display a running count of each kind of object on the right side using `drawString`.

Listing 8.10 The Ball class, the three remaining methods

```
// the Ball class, completed

   public static final int UNIT = 15;       // width of figure


   public void move()
   {  itsXspot += itsXmove;
      itsYspot += itsYmove;
      if (itsXspot <= 1 || itsXspot >= theRight - UNIT)
         itsXmove = - itsXmove;
      if (itsYspot <= 1 || itsYspot >= theBottom - UNIT)
         itsYmove = - itsYmove;
   }  //======================


   public boolean misses (Ball guy)
   {  return this == guy || Math.abs (itsXspot - guy.itsXspot)
                         + Math.abs (itsYspot - guy.itsYspot) > 2;
   }  //======================


   public void drawImage (Graphics2D page)
   {  if (isHealthy())
         page.setColor (Color.red);
      else
         page.setColor (Color.blue);
      page.fillOval (itsXspot, itsYspot, UNIT, UNIT);
   }  //======================
```

Exercise 8.36 Fix the `move` method so that the just-born balls do not get stuck at the bottom of the drawing area.
Exercise 8.37* What would be the formula for the `misses` method to see if one object is within a circle of radius 2 of the other? Is the result the same? For what radii is the result different?
Exercise 8.38* Revise the `drawImage` method so that a dying Ball flickers between blue and gray, half-and-half. Hint: use the expression `itsStatus % 10 < 5`.
Exercise 8.39* Revise the Collision class so that the action continue for exactly two minutes rather than 4000 cycles.

8.9 Implementing The Turtle And Turtlet Classes

Chapter One describes software that lets you use do graphics programming with statements such as the following within a main method of an application. These statements make a black circle of radius 200 and then a red circle of radius 100:

```
Turtle sue = new Turtle();  // initial drawing Color is black
sue.swingAround (200);   // circle radius 200 centered at sue
sue.switchTo (Turtle.RED);
sue.swingAround (100);   // circle radius 100 centered at sue
```

The Turtle class is a subclass of the Turtlet class. Turtlets are used for drawing on applets. A Turtlet keeps track of its position and heading (instance variables `xcor`, `ycor`, and `heading`). The Turtlet class keeps track of the Graphics page on which all the Turtlets are drawing (the rules require that, at any given time, all Turtlets are on the same page). This Graphics object, a class variable named `thePage`, is the one that is the parameter of the `paint` method within which Turtlets are created.

Turtles leave all the drawing to the Turtlet methods. However, Turtles are created in a main method of an application, or at least outside of a graphical object's `paint` method. So they have to create their own frame to display the drawing in. All Turtle drawings go to the same graphical image. So the function of the Turtle class is to keep track of a class variable `theWorld`, which is the JFrame object where the drawings appear.

This variable `theWorld` is an instance of TurtleWorld, which is our own subclass of JFrame. We discuss JFrames in the next chapter, so for now all you need to know is that JFrames have a `repaint` method just like applets. The Turtle instance methods mainly call the corresponding Turtlet method and then tell `theWorld` to repaint itself. This updates the display. For instance, here is a method from the Turtle class:

```
public void swingAround (double radius)
{  super.swingAround (radius);
   theWorld.repaint();
}  //=====================
```

Coding the Turtlet class

Listing 8.11 (see next page) gives half of the Turtlet class needed for this software. It should all be completely understandable to you except for the definition of DEGREE, which we discuss shortly. Study the rest for now.

The constructor sets the Turtlet's position (`xcor` and `ycor`), but the `heading` is always due east to start. Internally we represent a due east heading by zero. For accuracy, we keep track of the Turtlet's current position as a decimal number (`double`). The `drawString` method requires int values, however, so we round them off to the nearest int value. The private `round` method uses a standard method for rounding: Since an int cast of a positive number rounds down, we first add 0.5 and then round down. That in effect rounds up if the number is closer to the higher int value.

The `switchTo` method changes the drawing color. The color is a property of the entire page on which all Turtlets draw, not a property of an individual Turtlet. But `thePage` is null until some Turtlet is created. If `switchTo` were a class method, it could be called before any Turtlet is created, which would crash. This is why it is an instance method.

The `swingAround` and `fillCircle` methods draw a circle with the Turtlet in the middle.

Listing 8.11 The Turtlet class, some methods postponed

```
import java.awt.Color;

public class Turtlet
{  public static final double DEGREE = Math.PI / 180;
   public static final Color RED = Color.red, BLUE = Color.blue,
                  BLACK = Color.black,      GRAY = Color.gray,
                  YELLOW = Color.yellow,    PINK = Color.pink,
                  ORANGE = Color.orange,    GREEN = Color.green,
                  MAGENTA = Color.magenta,  WHITE = Color.white;
   private static java.awt.Graphics thePage;
   //////////////////////////////////
   private double heading = 0;       // heading initially east
   private double xcor, ycor;        // current position of Turtle


   /** Write words without changing the Turtle's state.  */

   public void say (String message)
   {  thePage.drawString (message, round (xcor), round (ycor));
   }  //======================


   /** Supply the nearest int value to methods requiring ints. */

   private int round (double x)
   {  return (int) (x + 0.5);
   }  //======================


   /** Set the drawing Color to the given value.  */

   public void switchTo (Color given)
   {  thePage.setColor (given);
   }  //======================


   /** Make a circle of the given radius, Turtle at center. */

   public void swingAround (double radius)
   {  if (radius > 0.0)
      {  int rad = round (2 * radius);
         thePage.drawOval (round (xcor - radius),
                  round (ycor - radius), rad, rad);
      }
   }  //======================


   /** Fill a circle of the given radius, Turtle at center. */

   public void fillCircle (double radius)
   {  if (radius > 0.0)
      {  int rad = round (2 * radius);
         thePage.fillOval (round (xcor - radius),
                 round (ycor - radius), rad, rad);
      }
   }  //======================
}
```

Trigonometry

The Turtle's current position is tracked by the two variables `<xcor, ycor>`. Suppose the Turtle is oriented 30 degrees counterclockwise from a due-east heading, at the far left of Figure 8.8. If told to move say 100 Turtle steps, it has to calculate its new `<x,y>` coordinates from its original ones. The hypotenuse of the triangle is 100 pixels, so the vertical side of the triangle is 50 pixels (half the hypotenuse) and the base of the triangle is about 86.6 pixels (multiplying the length of the hypotenuse by half of the square root of 3). So the new `<x,y>` coordinates are 50 pixels above and 86.6 pixels to the right of its original position.

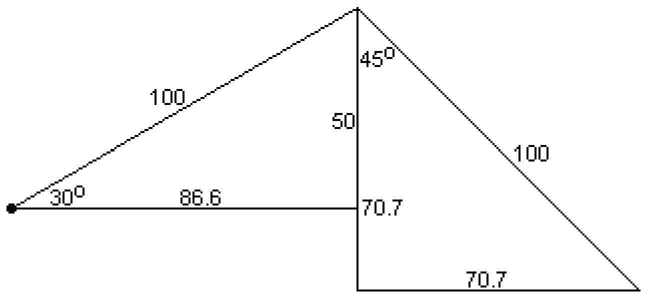


Figure 8.8 Distances and degrees

If the Turtle is then turned to the right 75 degrees, so it is facing southeast, and is then asked to move 100 Turtle steps, its new `<x,y>` coordinates are 70.7 pixels below and 70.7 pixels to the right of the original position (because half the square root of 2 is 0.707). In general, the Turtle needs to add the cosine of its heading to its current x-position `xcor` and the negative of the sine of its heading to its current y-position `ycor`. The reason for subtracting for the y-position is that y-values on a Graphics page are increasing in the downward direction, not the upward direction as is normally done in trigonometry.

Trigonometric methods from the Math class

The Math class in `java.lang` has many useful class methods. In particular, `Math.cos (heading)` is the cosine of the heading and `Math.sin (heading)` is the sine of the heading. So the Turtle needs to add the former to `xcor` and subtract the latter from `ycor`. However, these trigonometric functions are calculated from the number of radians, not the number of degrees. `Math.PI` is defined to be the number of radians in 180 degrees, so one degree is `Math.PI / 180` radians.

You can reduce the amount of calculation the Turtle has to do if you keep track of the heading in terms of radians instead of degrees. So each time the heading changes by the amount `left`, you add `left * DEGREES` to the `heading`. The initial value of the `heading` is zero, which conventionally indicates an orientation due east.

The upper part of Listing 8.12 (see next page) contains the two methods `move` and `paint`, which should now be understandable. The methods are defined to return the Turtle itself rather than being void methods, because it is sometimes convenient to be able to write chained statements such as the following:

```
    sam.paint(30, 100).paint(-75, 100).move(-135, 70.7);
```

The `sleep` method could be written using a busywait which you already understand. But for heavily-used software, it makes more sense to use the `Thread.sleep` method explained in Chapter Eleven. Just don't worry about this coding until Chapter Eleven. It is a class method so you can write `Turtlet.sleep(n)` without having any Turtlets.

Listing 8.12 The rest of the Turtle class

```
// the Turtle class, completed

   /** Rotate left by left degrees; MOVE for forward steps. */

   public Turtle move (double left, double forward)
   {  heading += left * DEGREE;
      xcor += forward * Math.cos (heading);
      ycor -= forward * Math.sin (heading);
      return this;
   }  //======================


   /** Rotate left by left degrees; PAINT for forward steps. */

   public Turtle paint (double left, double forward)
   {  heading += left * DEGREE;
      double x = xcor + forward * Math.cos (heading);
      double y = ycor - forward * Math.sin (heading);
      thePage.drawLine (round (xcor), round (ycor),
                        round (x), round (y));
      xcor = x;
      ycor = y;
      return this;
   }  //======================


   /** Fill rectangle of given width & height, Turtle at center.*/

   public void fillBox (double width, double height)
   {  if (width > 0.0 && height > 0.0)
         thePage.fillRect (round (xcor - width / 2),
                           round (ycor - height / 2),
                           round (width), round (height));
   }  //======================


   /** Pause the animation for wait milliseconds. A class method
    *  so an applet can pause an animation "between turtles". */

   public static void sleep (int wait)
   {  try
      {  Thread.sleep (wait);
      }catch (InterruptedException ex)
      { }
   }  //======================


   /** Create Turtle on a given Component at a given position.*/

   public Turtle (java.awt.Graphics page,
                  double xstart, double ystart)
   {  if (page == null)
         throw new NullPointerException ("You did not give a "
                          + "world where turtles can live!");
      thePage = page;
      xcor = xstart;
      ycor = ystart;
   }  //======================
```

The Turtlet constructor

A statement such as `new Turtlet (getGraphics(), xstart, ystart)` within the `paint` method of an applet gives you a Turtlet that draws on an applet. The last two parameters give the initial position of the Turtle. This was not mentioned in Chapter One because you had not learned to used applets yet. A subtle point is that a new Graphics2D object is created each time the applet is covered and then uncovered, so the Graphics2D `page` value keeps changing. You should therefore create the Turtle in the `paint` method rather than in the `start` method to establish the new page reference.

Exercise 8.40 What changes would be required in the entire Turtlet class if the user prefers to specify degrees of turn to the right in `move` and `paint` and prefers that the Turtlet initially be oriented due north?
Exercise 8.41* Turtle software could execute faster if it had three convenience methods, one to turn left a number of degrees without moving, the others to move or paint forward a certain distance without turning. Write the three without calling `move` or `paint`.
Exercise 8.42* Revise the FractalTurtle's `drawTree` method (in Listing 1.11) so that each of the two branches off the tree trunk is at a 45-degree angle instead of a 30-degree angle.

8.10 Elements Of JavaScript

You have probably heard of JavaScript. It is a language with strong similarities to Java that can be used in web pages, with a restricted use of objects (mostly built-in ones). The purpose of this section is to tell you just enough about JavaScript that you can talk about it intelligently, you can use it to some extent in your web pages, and you have a foundation for learning much more about it if you choose.

One built-in object is **`window`**, which denotes the current window in the browser. It has some methods similar in effect to JOptionPane methods:

```
   x = window.prompt ("Enter a number:", "0")
   window.alert ("you entered " + x)
```

`window.prompt``(someString, defaultString)` returns a String value, the user input in a dialog box. It is similar to `JOptionPane.showInputDialog ("Enter a number:")`. The second parameter of `window.prompt` is the default value that appears in the textfield, in case the user chooses to just press Enter rather than type a String value. For this example, `x` would have to be previously declared as a variable.

`window.alert``(someString)` displays its message on the screen until the user clicks the OK button, similar to `JOptionPane.showMessageDialog`.

`parseInt``(someString)` is a built-in class method that returns the int equivalent of the String value. If `someString` is badly-formed, it uses only the first part of `someString`. So `parseInt("3.01")` produces the number 3, and `parseInt("4xy3")` produces the number 4. Consequently, this is the same as Java's `Integer.parseInt` but without throwing the NumberFormatException.

A complete example of a JavaScript segment that could appear in a web page is shown in Listing 8.13 (see next page). It computes how long it would take an amount of money to double at a particular interest rate. The type of a variable does not have to be declared; the browser can figure it out from the context. So all variables can be declared using the same word `var`.

Listing 8.13 JavaScript in a web page

```
<SCRIPT LANGUAGE="JavaScript">

   /* Calculate time to double money at a given interest rate */

   var name = window.prompt ("What is your name?", "");
   var rate = parseInt (window.prompt ("Interest rate?", "6"));
   if (rate <= 0)
      window.alert (name + ", that does not make sense");
   else
   {  var balance = 1.0;
      var count = 0;
      while (balance < 2.0)
      {  balance *= 1 + rate / 100.0;
         count++;
      }
      window.alert ("Your money would double in " + count
                   + " years, " + name);
   }
</SCRIPT>
```

The first and last line of Listing 8.13 are required to notify the browser that the part within the two SCRIPT tags is JavaScript. As you may infer from this example, JavaScript has most of the operators (`+ - * / % < >= = != <= ==`) that Java has, as well as if-else-statements, while-statements, and for-statements.

HTML

Most of what you do with JavaScript is much more involved with HTML than Listing 8.13 indicates. For instance, the web page currently displayed in the browser is represented by the **`document`** object. You can write HTML to the `document` for rendering in accordance with its HTML tags. For instance:

```
   document.write ("<P><FONT SIZE=24,COLOR='green'>");
   document.writeln ("Hello world </FONT></P>");
```

This prints on the current document a new paragraph (<P>) in large (24-point) green font, containing just the two words "Hello world". The `write` and `writeln` methods are analogous to `System.out.print` and `System.out.println`.

You can define new "class methods" in JavaScript. The word **`function`** normally stands for `public static void` or `public static String` or whatever type you want to return (if any). For instance, a method that adds a given number (the parameter) to a running total `sum` that has been declared elsewhere could be as follows:

```
   function addIt (number)
   {  sum += number;
   }
```

In these simple cases, it is as if the JavaScript forms a single class with only class methods, and all variables mentioned other than parameters are class variables. But if `sum` were declared inside of that function (using the `var` keyword), `sum` would be a local variable.

You can declare an array variable as follows:

```
var item = new Array();
```

You need not say what the size of the array is to be; it will expand as needed. After executing the previous declaration, `item.length` is zero. If you execute `item[0]="Testing"` and `item[1]="JavaScript"`, then `item.length` is two.

Event-driven programming

Event-handling is much easier in JavaScript than in Java itself. The following segment of HTML creates a window in the `document`, called a **form**, that contains a textfield for input and a button for clicking:

```
<FORM   NAME="steve">
<INPUT  TYPE=TEXT    NAME="dave">
<INPUT  TYPE=BUTTON  NAME="ruth"   onClick="checkInput();">
</FORM>
```

INPUT means that it is a part of the form used for user input. Each part has a name: The form is named `steve`, the textfield is named `dave`, and the button is named `ruth`. You can refer to the form elsewhere in the HTML as `document.steve` and to the textfield as `document.steve.dave`, since the form is an attribute of the document and the textfield is an attribute of the form.

The event-handling is in the last phrase of the specification for the button (the second line). It means that, when the user clicks on the button, the JavaScript function named `checkInput` is to be called with no parameter values. That function could retrieve the value of the textfield and calculate twice that number as follows:

```
function checkInput()
{  window.alert ("Twice your input number is "
         + 2 * parseInt (document.steve.dave.value));
}
```

One attribute of any textfield is the `value` attribute, telling what is currently stored in it. You can also put a label on a button by specifying its `value` attribute inside the HTML <INPUT> tag that defines it, as in `VALUE="click here"`.

You can put a textarea on a web page to get multiple lines of input from the user. Put the following within the definition of the `steve` form shown previously:

```
<TEXTAREA  NAME="input"  ROWS=5  COLS=30 WRAP></TEXTAREA>
```

This puts a textarea named `input` in the next available position on the form, allowing five rows of characters to be typed, each with up to 30 characters. The lines will wrap around if they are too long. When you want to see what is written there, you refer to:

```
document.steve.input.value
```

If you want to have one of these components of the `steve` form "get the focus", that is, be the place where the user's next keystrokes will take effect, use one of the following:

```
document.steve.ruth.focus();
document.steve.input.focus();
```

Object classes in JavaScript

You can create your own objects in JavaScript. For instance, the Person class in Listing 4.4 has two instance variables `itsFirstName` and `itsLastName`, as well as two instance methods `getFirstName` and `getLastName`. You can set up the equivalent situation in JavaScript as follows. Note that `function` is used to indicate a constructor or an instance method as well as a class method:

```
function Person (first, last)  // constructor
{  this.itsFirstName = first;
   this.itsLastName = last;
   this.getFirstName = getFirstName;
   this.getLastName = getLastName;
}
function getFirstName()  // Person instance method
{  return this.itsFirstName;
}
function getLastName()  // Person instance method
{  return this.itsLastName;
}
```

The statements of the constructor specify the instance variables and instance methods of the object class. The first two statements of this Person constructor say that the two instance variables of Person objects are `itsFirstName` and `itsLastName`, and also initialize those two variables to the parameter values. The last two statements say that a Person has two associated instance methods named `getFirstName` and `getLastName`.

The runtime system tells the difference between instance variables and instance methods by the fact that instance methods are defined later as functions. That requires the two function definitions of `getFirstName` and `getLastName` in your JavaScript coding. The bodies of these two methods are allowed to use `this` because they are now Person instance methods, since the constructor said so. Then the rest of your JavaScript coding can include statements such as the following:

```
var somePerson = new Person ("Garth", "Brooks");
document.write (somePerson.getFirstName());
```

This introduction to JavaScript should make you think about how different computer languages can do the same thing differently, such as giving the type of a variable (though type-compatibility problems can occur). Also, JavaScript considers all integer values to be doubles, and semicolons at the ends of statements are optional if you do not have anything on the line after the statement.

Exercise 8.43 Write HTML to create two buttons on a form, each with a different value. Add JavaScript so that clicking either button calls a function that prints the value on that button.
Exercise 8.44* Write JavaScript similar to Listing 8.13 that reads in Strings of characters until an empty string is entered, then prints the one that is alphabetically first (using Unicode ordering).
Exercise 8.45* Revise the Time class in Listing 4.5 to be a JavaScript object class, similar to the Person class.

*8.11 About Font, Polygon, and Point2D.Double (*Sun Library)*

This section discusses three additional classes that may be useful in working with graphics.

Font objects (from java.awt)

You may control the shape and size of the letters and other characters you write on a Graphics object. The kinds of font allowed depend on your system, but they always include these five: "Serif" (like Times New Roman), "SansSerif" (like Ariel or Helvetica), "Monospaced" (like Courier), "Dialog", and "DialogInput" (both of which are fonts san serif).

- `new Font(someString, styleInt, sizeInt)` creates a new Font object. `someString` is a name such as "Serif". The `styleInt` can be Font.PLAIN, Font.BOLD, Font.ITALIC, or the sum Font.BOLD+Font.ITALIC. The `sizeInt` value should range from 8 (one-ninth of an inch tall) to 36 (one-half of an inch tall).
- `someGraphics.setFont(someFont)` causes all Strings of characters later drawn on the Graphics object to be drawn according to the given Font object.

Polygon objects (from java.awt, implements Shape)

A Polygon is a number of points connected by line segments, with the last point connected to the first. Rectangles and triangles are the simplest polygonal shapes. Since Polygon implements the Shape interface, you can use the `draw` and `fill` commands for Graphics2D objects.

You specify a Polygon object by giving two arrays of int values, the first being the x-coordinates of the points and the second being the y-coordinates of the points. For example, if you want to draw the diamond-shaped figure with corners at <200,50>, <160,150>, <200,250>, and <240,150>, you could have the following four statements (where the third parameter of the constructor is the number of points):

```
int[4] xvals = {200, 160, 200, 240};  // the x-coordinates
int[4] yvals = {50, 150, 250, 150};   // the y-coordinates
Polygon diamond = new Polygon (xvals, yvals, 4);
page.draw (diamond);
```

- `new Polygon(xvals, yvals, sizeInt)` constructs a Polygon object from two arrays declared as `int[]`. For the `sizeInt` points given by `<x[k],y[k]>` for the various values of `k`, each point is connected to the next, except the figure is completed by connecting `<x[sizeInt-1],y[sizeInt-1]>` to `<x[0],y[0]>`.
- `new Polygon()` constructs a Polygon with zero points.
- `somePolygon.addPoint(xInt, yInt)` adds a new point to the end of the Polygon, so that what previously was the last point now connects to this added point and the added point connects to the first point.
- `somePolygon.translate(dxInt, dyInt)` adds `dxInt` to each x-coordinate and `dyInt` to each y-coordinate of the points in the Polygon.

You have to be sure to put the points in the right order; if the third and fourth points of the diamond shape were switched, you would not have a diamond, you would have two triangles that meet at a point.

Point2D.Double objects (from java.awt.geom)

You may want to use the Point2D.Double class to make your coding more compact and more understandable. The Point2D.Double class has several useful methods:

- `new Point2D.Double(x, y)` constructs a Point at double coordinates `x` and `y`.
- `somePoint2D.getX()` returns the current x-value of the point.
- `somePoint2D.getY()` returns the current x-value of the point.
- `somePoint2D.setLocation(x, y)` re-assigns the coordinates as indicated.

The following illustrate how you can use these objects (`p1` and `p2` denote instances of the Point2D.Double class):

- `new Line2D.Double(p1, p2)` is an alternate constructor using points.
- `someShape.contains(p1)` tells whether p1 is inside the Shape object.
- `someLine2D.ptLineDist(p1)` tells how far `p1` is from the line (a double).

8.12 Review Of Chapter Eight

About the java.awt.Graphics2D class (a subclass of Graphics):

- The **drawing area** is measured in **pixels**, counting in from the left and down from the top. The point `<0,0>` is in the top-left corner. At any given time, one particular color is the **drawing Color** for the Graphics object, used for all drawings.
- `someGraphics.getColor()` returns the current drawing Color.
- `someGraphics.setColor(someColor)` makes it the current drawing Color.
- `someGraphics.drawString(someString, x, y)` prints the characters in the String starting at `<x,y>` where `x` and `y` are both int values.
- `someGraphics2D.draw(someShape)` draws the outline of the rectangle, ellipse, polygon, or whatever shape you supply as the parameter.
- `someGraphics2D.fill(someShape)` draws the shape and fills in its interior with the drawing Color.

About the java.awt.Rectangle class (implementing the java.awt.Shape interface):

- `new Rectangle(width, height)` constructs a new Rectangle object whose top-left corner is at `<0,0>`, with the given `width` and `height` measured in pixels. If either the `width` or the `height` is negative, the Rectangle will not show up. All parameters are int values for this constructor and the following four methods.
- `someRectangle.setLocation(x, y)` moves the top-left corner of the rectangle to the point `<x,y>`. The width and height remain unchanged.
- `someRectangle.setSize(width, height)` changes the width and height of the rectangle to the specified values. The top-left corner remains unmoved.
- `someRectangle.grow(width, height)` adds `width` pixels to the left and right sides of the rectangle, and adds `height` pixels to top and bottom of the rectangle. So its width changes by `2*width` and its height changes by `2*height`.
- `someRectangle.translate(x, y)` moves the rectangle `x` pixels to the right and `y` pixels down from its current position. The width and height remain unchanged.

About some java.awt.geom classes (all implementing java.awt.Shape):

- `new Line2D.Double(x, y, xend, yend)` constructs a new Line object with one end at `<x,y>` and the other at `<xend,yend>`. All parameters are doubles.

- `new Rectangle2D.Double(x, y, width, height)` constructs a new rectangular object with its top-left corner at `<x,y>`, and of the given `width` and `height`. All parameters are doubles.
- `new Ellipse2D.Double(x, y, width, height)` constructs a new elliptical object whose bounding box has its top-left corner at `<x,y>` and has the given `width` and `height`. All parameters are doubles.
- `new RoundRectangle2D.Double(x, y, width, height, arcWidth, arcHeight)` constructs a new rectangular object with rounded corners and its top-left corner at `<x,y>`, of the given `width` and `height`, where the width and height of the arcs at the corners are given by the last two parameters. All parameters are doubles.

About the javax.swing.JApplet and java.awt.Component class:

- The `JApplet` class is a subclass of Applet, which has these three methods called by a browser: `init()` when the applet is loaded; `start()` when the applet is to start execution, and `stop()` when the applet is to stop execution. Applet is a subclass of the Panel class, which is a subclass of the Container class, which is a subclass of the Component class. The Component class contains the following six methods.
- `someComponent.paint(someGraphics)` paints this Component. The browser calls it whenever the window is uncovered after having been minimized or covered.
- `someComponent.getGraphics()` returns the Component's Graphics object.
- `someComponent.setVisible(aBoolean)` makes the Component visible or not.
- `someComponent.setSize(width, height)` resizes the Component to the given width and height, both of which must be int values.
- `someComponent.getWidth()` returns the int width of the drawing area in pixels.
- `someComponent.getHeight()` returns the int height of the drawing area in pixels.

About other Sun standard library classes:

- The `Color` class (in `java.awt`) defines thirteen final class variables: `black`, `blue`, `cyan`, `darkGray`, `gray`, `green`, `lightGray`, `magenta`, `orange`, `pink`, `yellow`, `red`, and `white`. The constructor `new Color(r, g, b)` with int parameters in the range 0 to 255 creates an opaque Color with the corresponding red/green/blue values; the higher the number, the more you have of that color.
- Read through the documentation for the `java.awt` and `javax.swing` classes partially described in this chapter. Look at the API documentation at `http://java.sun.com/docs` or on your hard disk.

Basic programming principles mentioned in this chapter:

- If you don't know where you are going, you are not likely to get there.
- Always put the hard stuff off until later.
- You're only human, you're supposed to make mistakes (per Billy Joel).
- Don't tell the client his/her ideas are dumb if it is both profitable and ethical to create the software he/she wants.
- Don't make stuff up unless you have to.
- Don't be afraid to ask for help from someone who knows.

Common looping patterns and looping actions:

- Common looping patterns include sentinel-controlled logic, count-controlled logic, iterator-controlled logic, and user-controlled logic.
- Common looping actions (actions to take inside the loop) include Count-cases, Some-A-are-B, All-A-are-B, Process-some-cases, and Find-min-or-max.

Other vocabulary to remember:

- **Iterative development** of a large program is several cycles of the following: Develop a complete working program that does more of what the final program is to do than what you had at the end of the previous cycle, making sure that most of what you add will not have to be discarded in a later cycle.
- A **busywait** is a segment of coding that keeps the runtime system busy without accomplishing anything useful except to pass the time. It is sometimes used in animation. A better technique using Thread.sleep is described in Section 11.10.
- A **bug** in software is any disagreement with the specifications for that software. In particular, it is a bug to have a method do something more than it is supposed to do. For instance, a method that is specified to print two particular lines has a bug if it prints the wrong two lines, or if it prints less than two lines, or even if it prints the right two lines plus another line.

Answers to Selected Exercises

```
8.1     public void paint (Graphics g)    // one of many possible solutions
        {   Graphics2D page = (Graphics2D) g;
            page.setColor (Color.yellow);
            page.drawString ("YourName", 10, 50);
            page.setColor (Color.cyan);
            page.drawString ("YourName", 70, 50);
            page.setColor (Color.pink);
            page.drawString ("YourName", 130, 50);
        }
8.2     public void paint (Graphics g)    // one of many possible solutions
        {   Graphics2D page = (Graphics2D) g;
            page.setColor (Color.green);
            page.drawLine (10, 50, 40, 50);  // top, 30 pixels wide
            page.drawLine (40, 50, 40, 140);  // right side, 90 pixels tall
            page.drawLine (40, 140, 10, 140);  // bottom
            page.drawLine (10, 140, 10, 50);    // left side
        }
8.3     public void paint (Graphics g)  // not much choice here
        {   Graphics2D page = (Graphics2D) g;
            // THE OUTER SQUARE
            page.drawLine (10, 30, 50, 30);
            page.drawLine (50, 30, 50, 70);
            page.drawLine (50, 70, 10, 70);
            page.drawLine (10, 70, 10, 30);
            // THE TWO CENTER DIVIDERS
            page.drawLine (30, 30, 30, 70);
            page.drawLine (10, 50, 50, 50);
        }
8.4     public void paint (Graphics g)    // one of many possible solutions
        {   Graphics2D page = (Graphics2D) g;
            Rectangle box = new Rectangle (120, 40);
            box.setLocation (10, 50);
            page.draw (box);
            box.setSize (40, 120);
            box.setLocation (50, 10);
            page.draw (box);
            page.drawString ("give", 11, 45);
            page.drawString ("blood", 91, 45);
        }
8.5     public void paint (Graphics g)   // one of many possible solutions
        {   Graphics2D page = (Graphics2D) g;
            Rectangle box = new Rectangle (80, 40);
            box.setLocation (10, 30);
            page.draw (box);
            box.setSize (70, 30);
            page.draw (box);
            box.setSize (60, 20);
            page.draw (box);
        }
```

8.6 Replace each "page.draw" by "page.fill" in the preceding answer, and put these three statements just before those page.fill commands, in order:

```
page.setColor (Color.magenta);
page.setColor (Color.gray);
page.setColor (Color.yellow);
```

8.7

```
public void paint (Graphics g)
{       Graphics2D page = (Graphics2D) g;
        page.draw (new Ellipse2D.Double (30, 40, 20, 20));
        page.draw (new Ellipse2D.Double (50, 40, 20, 20));
        page.draw (new Ellipse2D.Double (70, 40, 20, 20));
}
```

8.11 As given, you draw a completely blue rectangle and then put white stars on it. If you drew the white stars first, how would you fill in the blue part around them?

8.12 Just double the four constants to 346, 182, 138, 98, 4 and 10. The rest of the Flag class should be written so that all other dimensions are doubled as a consequence.

8.15 Replace FLAG_TALL by 100, 13 by 20, "red" by "green", and "white" by "orange".

8.16 It is not the 9 or the 2; the flag will have 9 rows alternating among 2 possibilities no matter what the size. Define public static final int WIDTH = 12; then replace PIP + 6 by PIP + WIDTH / 2 and replace x += 12 by x += WIDTH.

8.17 Replace the body of the for-statement by the following:

```
if (row % 2 == 0)
        drawRowOfStars (page, x + UNION_WIDE - STAR, x + PIP,  y + PIP + row * STAR);
else
        drawRowOfStars (page, x + UNION_WIDE - STAR, x + PIP + 6,  y + PIP + row * STAR);
```

8.18 Replace row < 9 by row < 6 in the drawAllStars method, and UNION_WIDE - STAR by UNION_WIDE - 2 * STAR in the call of the drawRowOfStars method.

8.19 In the drawStripes method, replace 13 by 20 and replace the if-statement by the following:

```
if (k % 4 == 0)
        page.setColor (Color.green);
else if (k % 4 == 1)
        page.setColor (Color.orange);
else if (k % 4 == 2)
        page.setColor (Color.red);
else // (k % 4 == 3)
        page.setColor (Color.white);
```

8.20 Insert this statement before the for-statement:

```
boolean isRed = true;
```

Replace the if-else-statement by the following five lines:

```
if (isRed)
        page.setColor (Color.red);
else
        page.setColor (Color.white);
isRed = ! isRed;
```

8.27 The expression itsLeftEdge / 10 % 2 is tricky enough, being 1 or 0 depending on whether it is an odd or an even multiple of 10. When you double 1 or 0 and subtract 1, you get either 1 or -1, and multiplying that result by 4 means you are either adding or subtracting 4. But that takes far longer for the human reader of your logic to understand. You do that only if there is a strongly compensating advantage to doing so, and there is not; execution speed is not materially affected.

8.30

```
int k = 0;
while (k < a.length && someProperty (a[k]))
        k++;
return k == a.length;
```

8.31 Change the less-than operator < to the greater-than operator > for the call of compareTo.

8.32 It would. If the length of the array is in fact zero, the for-statement does not execute at all and so null is returned.

8.36 Make the condition for the second if-statement in the move method the following:

```
if (itsYspot <= 1 || (itsYspot >= theBottom - UNIT && itsYmove > 0)) .
```

Or else change the second statement in the Ball constructor to the following:

```
itsYspot = theBottom - 1 - UNIT - randy.nextInt (25);
```

8.40 Change the initial value of heading to double heading = 90 * DEGREES; also, the first statement of both move and paint should be heading -= left * DEGREES.

8.43 Put the following two lines inside the definition of the steve form:

```
<INPUT  TYPE = BUTTON  NAME = "a"  VALUE = "click me!"  onClick = "doA();">
<INPUT  TYPE = BUTTON  NAME = "b"  VALUE = "click her!"  onClick = "doB();">
```

Put the following two functions in the JavaScript coding:

```
function doA()
{       window.alert (document.steve.a.value);
}
function doB()
{       window.alert (document.steve.b.value);
}
```