

Applets and Graphics

CHAPTER GOALS

To be able to write simple applets

- ▶ To display graphical shapes such as lines and ellipses
- ▶ To use colors
- ▶ To display text in multiple fonts
- ▶ To select appropriate units for drawing
- ▶ To develop test cases that validate the correctness of your programs

There are three kinds of Java programs that you will learn to write: *console applications*, *graphical applications*, and *applets*. Console applications run in a single, usually rather plain-looking terminal window (see Figure 1). Applications with a graphical user interface use one or more windows filled with *user interface components* such as buttons, text input fields, and menus (see Figure 2). *Applets* are similar to applications with a graphical

user interface, but they run *inside a web browser*.

Console programs are simpler to write than programs with a graphical user interface, and we will continue to use console programs frequently in this book to learn about fundamental concepts. Graphical user interface programs can show more interesting output, however, and are often more fun to develop. In this chapter you will learn how to write simple applets that display graphical shapes.

CHAPTER CONTENTS

- 4.1 Why Applets? 135
- 4.2 A Brief Introduction to HTML 136
- Random Fact 4.1: The Evolution of the Internet 139
- 4.3 A Simple Applet 140
- Advanced Topic 4.1: The Java Runtime Environment and Java Plug-ins 144
- 4.4 Graphical Shapes 145
- 4.5 Colors 147
- 4.6 Fonts 148
- Advanced Topic 4.2: Accurate Positioning of Text 150
- 4.7 Drawing Complex Shapes 154
- HOWTO 4.1: Drawing Graphical Shapes 157
- Random Fact 4.2: Computer Graphics 162
- 4.8 Reading Text Input 164
- Advanced Topic 4.3: Applet Parameters 166
- 4.9 Comparing Visual and Numerical Information 167
- Quality Tip 4.1: Calculate Sample Data Manually 171
- 4.10 Coordinate Transformations 171
- Advanced Topic 4.4: Let the Graphics Context Transform the Coordinates 176
- Productivity Hint 4.1: Choose Convenient Units for Drawing 178

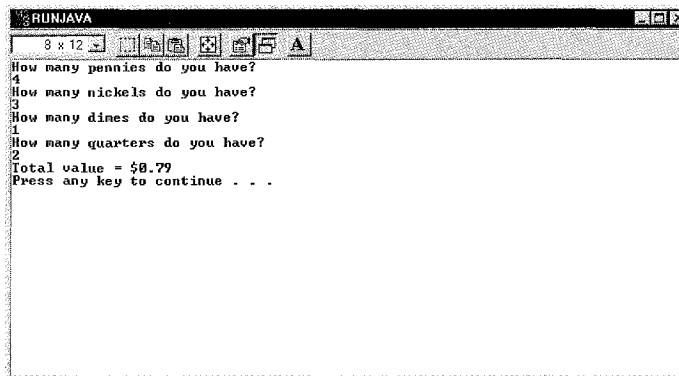


Figure 1

A Console Application

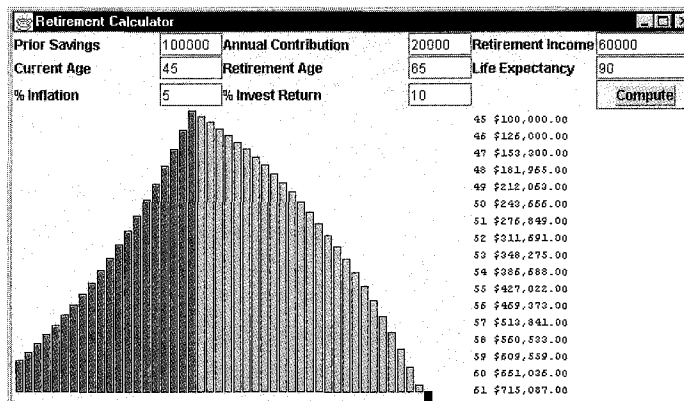


Figure 2

A Graphical Application

4.1 Why Applets?

There are three types of Java programs: *console applications, applets, and graphics applications.*

Applets are programs that run inside a web browser.

The World Wide Web makes a huge amount of information available to anyone with a web browser. When you use a web browser, you connect to a *web server*, which sends web pages and images to your browser (see Figure 3). Because the web pages and images have standard formats, your web browser can display them. To retrieve the daily news or your bank balance, you don't have to be at home, in front of your own computer. You can use a browser at school or in an airport terminal or Web café anywhere in the world. This ubiquitous access to information is one reason why the World Wide Web is so hugely popular.

Applets are programs that run inside a web browser. The code for an applet is stored on a web server and downloaded into the browser whenever you access a web page that contains the applet. That has one big advantage: You don't have to be at your own computer to run a program that is implemented as an applet. There is also an obvious disadvantage: You have to wait for the applet code to download into the browser, which can take a long time if you have a slow Internet connection. For that reason, complex applets are rare in web pages with a wide audience. Applets work very nicely, however, over a fast connection. For example, employees in a company often have a fast local area network connection. Then the company can deploy applets for schedule planning, health benefit access, product catalog lookup, and so on, and gain a big cost savings over the traditional process of developing corporate applications to be rolled out on the desktop of every user. When the program application changed, a system administrator had to make sure that every desktop was updated—a real hassle. When an applet changes, on the other hand, the code needs to be updated in one location: on the web server.

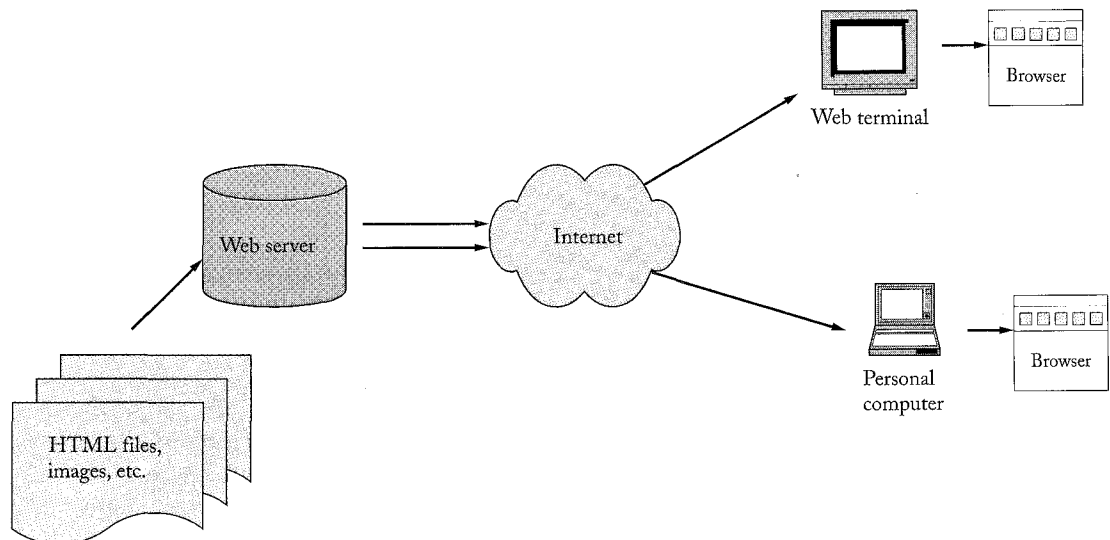


Figure 3

Web Browsers Accessing a Web Server

Web browser programs run on PCs, Macintoshes, UNIX workstations, and special web devices. For this reason, it is important that the applet code be able to execute on multiple platforms. Most traditional programs (such as word processors, computer games, and browsers) are written for a single platform, or perhaps written twice, for the two most popular platforms. Applets, on the other hand, are delivered as Java bytecode. Any computer or device that can execute Java bytecode can execute the applet.

The applet security mechanisms let you safely execute applets on your computer.

Whenever you run a program on your computer, you run a risk that the program might do some kind of damage either because it is poorly written or because it is outright malicious. A poorly written program might accidentally corrupt or erase some of your files. A virus program might do the same intentionally. In the old days, before computers were networked, you had to install every program yourself, and it was a relatively easy matter to check for viruses on the floppy disks that you used to add new programs or data to your computer. Nowadays, programs can come from anywhere—from another machine on a local area network, as email attachments, or as code that is included in a web page. Code that is part of a web page is particularly troublesome, because it starts running immediately when the browser loads the page. It would be an easy matter for a malicious person to set up a web page with enticing content, have many people visit the page, and attempt to infect each visitor with a virus. The designers of Java anticipated this problem and came up with two safeguards: Java applets can run at specified *security privileges*, and they can be *signed*. If you allow a machine language program to run, there is no control over what it can do, and it can create many kinds of problems. In contrast, the Java virtual machine can limit the actions of an applet. By default, an applet runs in a *sandbox*, where it can display information and get user input, but it can't read or touch anything else on the user's computer. You can give an applet more privileges, such as the ability to read and write local files, if you notify your browser that the applet is *trusted*. An applet can carry a *certificate* with a *signature* from an authentication firm such as VeriSign, which tells you where the applet originated. If the applet comes from a source you trust, such as a vendor who has delivered quality code to you in the past, you can tell the browser that you trust it. Signatures are not limited to applets; some browsers support certificates for machine language code.

4.2

A Brief Introduction to HTML

Applets are embedded inside web pages, so you need to know a few facts about the structure of web pages. A web page is written in a language called HTML (Hypertext Markup Language). Like Java code, HTML code is made up of text that follows certain strict rules. When a browser reads a web page, the browser *interprets* the code and *renders* the page, displaying characters, fonts, paragraphs, tables, and images.

Web pages are written in HTML, using tags to format text and include images.

HTML files are made up of text and *tags* that tell the browser how to render the text. Nowadays, there are dozens of HTML tags. Fortunately, you need only a few to get started. Most HTML tags

come in pairs consisting of an opening tag and a closing tag, and each pair applies to the text between the two tags. Here is a typical example of a tag pair:

```
Java is an <i>object-oriented</i> programming language.
```

The tag pair `<i> </i>` directs the browser to display the text inside the tags as *italics*: Java is an *object-oriented* programming language.

The closing tag is just like the opening tag, but it is prefixed by a slash (`/`). For example, bold-faced text is delimited by ` `, and a paragraph is delimited by the tag pair `<p> </p>`.

```
<p><b>Java</b> is an <i>object-oriented</i>
programming language.</p>
```

The result is the paragraph

Java is an *object-oriented* programming language.

Another common construct is a bulleted list.

Java is

- object-oriented
- safe
- platform-independent

Here is the HTML code to display it:

```
<p>Java is</p>
<ul><li>object-oriented</li>
<li>safe</li>
<li>platform-independent</li></ul>
```

Each item in the list is delimited by ` ` (for “list item”), and the whole list is surrounded by ` ` (for “unnumbered list”).

As in Java code, you can freely use white space (spaces and line breaks) in HTML code to make it easier to read. For example, you can lay out the code for a list as follows:

```
<p>Java is</p>
<ul>
  <li>object-oriented</li>
  <li>safe</li>
  <li>platform-independent</li>
</ul>
```

The browser ignores the white space.

If you omit a tag (such as a ``), most browsers will try to guess the missing tags—sometimes with differing results. It is always best to include all tags.

You can include images in your web pages with the `img` tag. In its simplest form, an image tag has the form

```

```

This code tells the browser to load and display the image that is stored in the file `hamster.jpeg`. This is a slightly different type of tag. Rather than text inside a tag pair ` `, the

`img` tag uses an *attribute* to specify a file name. Attributes have names and values. For example, the `src` attribute has the value `"hamster.jpeg"`. It is considered polite to use several additional attributes with the `img` tag, namely the *image size* and an *alternate description*:

```

```

These additional attributes help the browser lay out the page and display a temporary description while gathering the data for the image (or if the browser cannot display images, such as a voice browser for blind users). Users with slow network connections really appreciate this extra effort.

Because there is no closing `` tag, we put a slash `/` before the closing `>`. This is not a requirement of HTML, but it is a requirement of the emerging XHTML standard, the XML-based successor to HTML. See [1] for more information on XHTML.

The most important tag in web pages is the `<a>` `` tag pair, which makes the enclosed text into a *link* to another file. The links between web pages are what makes the Web into, well, a web. The browser displays a link in a special way (for example, underlined text in blue color). Here is the code for a typical link:

```
<a href="http://java.sun.com">Java</a> is an object-oriented
programming language.
```

When the viewer of the web page clicks on the word Java, the browser loads the web page located at `java.sun.com`. (The value of the `href` attribute is a *Universal Resource Locator (URL)*, which tells the browser where to go. The prefix `http:`, for *Hypertext Transfer Protocol*, tells the browser to fetch the file as a web page. Other protocols allow different actions, such as `ftp:` to download a file, `mailto:` to send email to a user, and `file:` to view a local HTML file.)

To run an applet, you need an HTML page with the applet tag.

Finally, the `applet` tag includes an applet in a web page. To display an applet, you need first to write and compile a Java file to generate the applet code—you will see how in the next section. Then you tell the browser how to find the code for the applet and how much

screen space to reserve for the applet. Here is an example:

```
<applet code="HamsterApplet.class" width="400" height="300">An
animation of Harry, the Horrible Hamster</applet>
```

The text between the `<applet>` and `</applet>` tags is only displayed in lieu of the actual applet by browsers that can't run Java applets.

You have noticed that tags are enclosed in angle brackets (less-than and greater-than signs). What if you want to show an angle bracket on a web page? HTML provides the notations `<` and `>`; produce the `<` and `>` symbols, respectively. Other codes of this kind produce symbols such as accented letters. The `&` (ampersand) symbol introduces these codes; to get that symbol itself, use `&`;

You may already have created web pages with a web editor that works like a word processor, giving you a WYSIWYG (what you see is what you get) view of your web page. But the tags are still there, and you can see them when you load the HTML file into a text editor. If you are comfortable using a WYSIWYG web editor, and if your editor

can insert applet tags, you don't need to memorize HTML tags at all. But many programmers and professional web designers prefer to work directly with the tags at least some of the time, because it gives them more control over their pages.



Random Fact

4.1

The Evolution of the Internet

Home computers and laptops are usually self-contained units with no permanent connection to other computers. Office and lab computers, however, are usually connected with each other and with larger computers: so-called *servers*. A server can store application programs and make them available on all computers on the network. Servers can also store data, such as schedules and mail messages, that everyone can retrieve. Networks that connect the computers in one building are called local area networks, or LANs.

Other networks connect computers in geographically dispersed locations. Such networks are called *wide area networks* or WANs. The most prominent wide area network is the *Internet*. At the time of this writing, the Internet is in a phase of explosive growth. Nobody knows for certain how many users have access to the Internet, but the user population is estimated in the hundreds of millions. The Internet grew out of the ARPAnet, a network of computers at universities that was funded by the Advanced Research Planning Agency of the U.S. Department of Defense. The original motivation behind the creation of the network was the desire to run programs on remote computers. Using remote execution, a researcher at one institution would be able to access an underutilized computer at a different site. It quickly became apparent, though, that remote execution was not what the network was actually used for. Instead, the “killer application” was *electronic mail*: the transfer of messages between computer users at different locations. To this day, electronic mail is one of the most compelling applications of the Internet.

Over time, more and more *information* became available on the Internet. The information was created by researchers and hobbyists and made freely available to anyone, either out of the goodness of their hearts or for self-promotion. For example, the *GNU* (GNU's Not UNIX) project is producing a set of high-quality operating system utilities and program development tools that can be used freely by anyone (<ftp://prep.ai.mit.edu/pub/gnu>), and Project Gutenberg makes available the text of important classical books, whose copyright has expired, in computer-readable form (<http://www.gutenberg.org>).

The first interfaces to retrieve this information were clumsy and hard to use. All that changed with the appearance of the *World Wide Web* (WWW). The World Wide Web brought two major advances to Internet information. The information could contain *graphics* and *fonts*—a great improvement over the older text-only format—and it became possible to embed *links* to other information pages. Using a *browser* such as Netscape or Internet Explorer, surfing the Web becomes easy and fun (Figure 4).

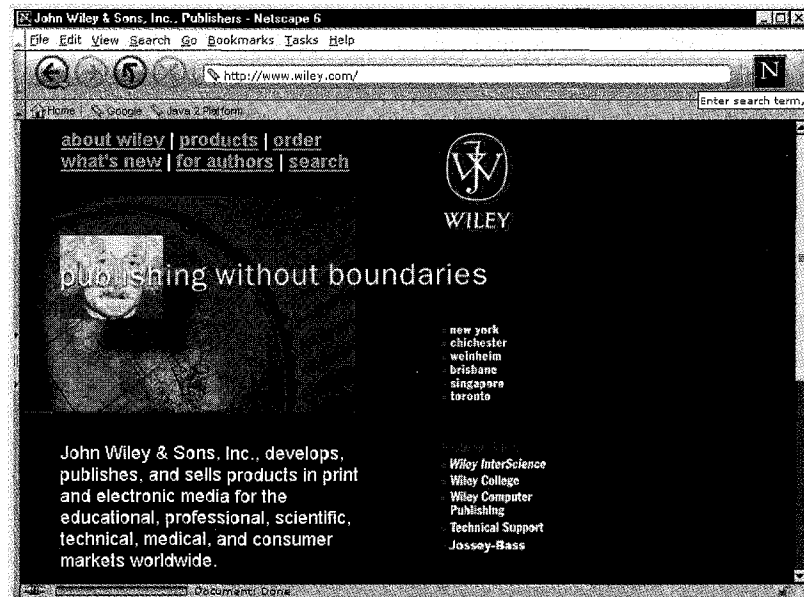


Figure 4

A Web Browser

4.3 A Simple Applet

In our first applet we will simply draw a couple of rectangles (see Figure 5). You'll soon see how to produce more interesting drawings. The purpose of this applet is to show you the basic outline of an applet that creates a drawing.

This applet will be implemented in a single class `RectangleApplet`. To run this applet, you need an HTML file with an `applet` tag. Here is the simplest possible file to display the applet:

File `RectangleApplet.html`

```
1 <applet code="RectangleApplet.class" width="300" height="300">
2 </applet>
```

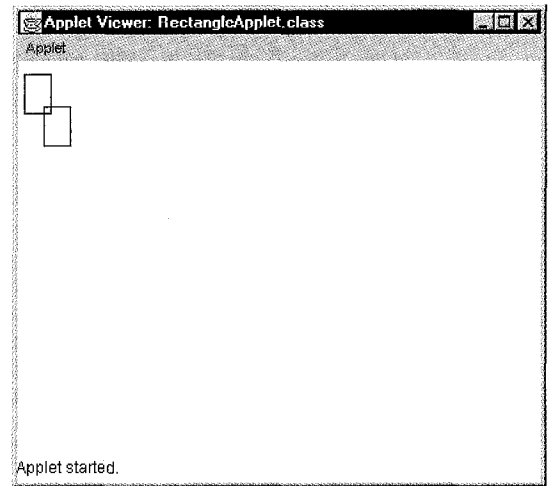
Or you can proudly explain your creation, by adding text and more HTML tags:

File `RectangleAppletExplained.html`

```
1 <p>Here is my <i>first applet</i>:</p>
2 <applet code="RectangleApplet.class" width="300" height="300">
3 </applet>
```

Figure 5

The Rectangle Applet in the Applet Viewer



An HTML file can have multiple applets. For example, you can place all your homework solutions into a single HTML file.

You can give the HTML file any name you like. It is easiest to give the HTML file the same name as the applet. But some development environments already generate an HTML file with the same name as your project to hold your project notes; then you must give the HTML file containing your applet a different name.

You view applets with the applet viewer or a Java-enabled browser.

To run the applet, you have two choices. You can use the *applet viewer*, a program that is included with the Java Software Development Kit from Sun Microsystems. You simply start the applet viewer, giving it the name of the HTML file that contains your applets:

```
appletviewer RectangleApplet.html
```

The applet viewer brings up one window for each applet in the HTML file. It ignores all other HTML tags. Figure 5 shows the applet inside the applet viewer.

You can also show the applet inside any Java 2-enabled web browser such as Netscape 6 (or later) or Opera. Figure 6 shows the applet running in a browser. As you can see, both the text and the applet are displayed.

Your applet class needs to extend the Applet class.

An applet is programmed as a class, like any other program in Java. However, the class is declared *public*, and it *extends* Applet.

That means that our rectangle applet *inherits* the behavior of the Applet class. We will discuss inheritance in Chapter 11.

```
public class RectangleApplet extends Applet
{
    public void paint(Graphics g)
    {
        . . .
    }
}
```

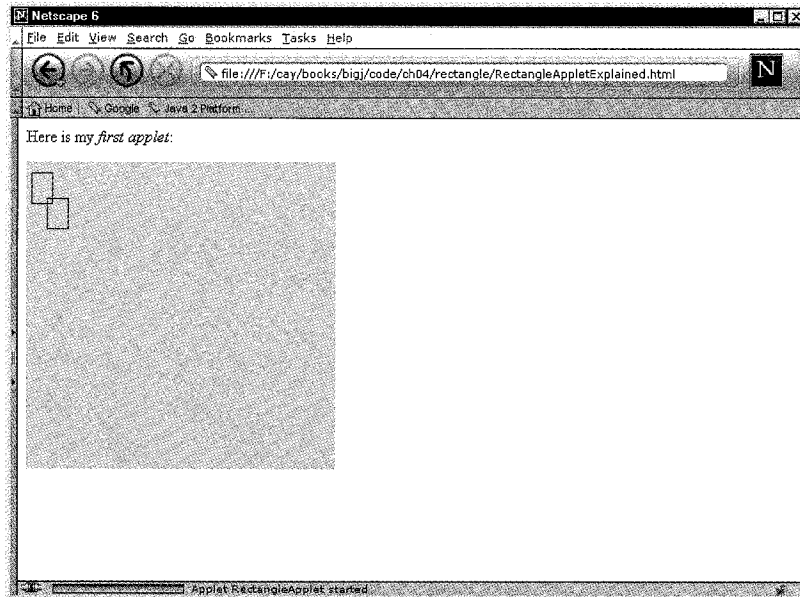


Figure 6

The Rectangle Applet in a Browser

You draw graphical shapes in an applet by placing the drawing code inside the `paint` method. The `paint` method is called whenever the applet needs to be refreshed.

The `Graphics2D` class stores the graphics state (such as the current color) and has methods to draw shapes. You need to cast the `Graphics` parameter of the `paint` method to `Graphics2D` to use these methods.

Unlike applications, applets don't have a `main` method. The web browser (or applet viewer) is responsible for starting up the Java virtual machine, for loading the applet code, and for starting the applet. This applet implements only one method: `paint`. There are other methods that you *may* implement; see Advanced Topic 4.3.

The window manager calls the `paint` method whenever the surface of the applet needs to be filled in. Of course, when the applet is shown for the first time, its contents need to be painted. If the user visits another web page and then goes back to the web page containing the applet, the surface must be painted again, and the window manager calls the `paint` method once more. Thus, you must put all drawing instructions inside the `paint` method, and you must be aware that the window manager can call the `paint` method many times.

The `paint` method receives an object of type `Graphics`. The `Graphics` object stores the *graphics state*: the current color, font, and so on, that are used for the drawing operations. For the drawing programs that we explore in this book, we always convert the `Graphics` object to an object of the `Graphics2D` class:

```
public class RectangleApplet extends Applet
{
    public void paint(Graphics g)
```

```
    {  
        // recover Graphics2D  
        Graphics2D g2 = (Graphics2D)g;  
        . . .  
    }  
}
```

To understand why, you need to know a little about the history of these classes. The `Graphics` class was included with the first version of Java. It is suitable for very basic drawings, but it does not use an object-oriented approach. After some time, programmers clamored for a more powerful graphics package, and the designers of Java created the `Graphics2D` class. They did not want to inconvenience those programmers who had produced programs that used simple graphics, so they did not change the `paint` method. Instead, they made the `Graphics2D` class extend the `Graphics` class, a process that will be discussed in Chapter 9. Whenever the window manager calls the `paint` method, it actually passes a parameter of type `Graphics2D`. Programs with simple graphics needs do not need to know about this, but if you want to use the more sophisticated 2D graphics methods, you recover the `Graphics2D` reference by using a cast as discussed in Chapter 3.

You use the `draw` method of the `Graphics2D` class to draw shapes such as rectangles, ellipses, line segments, polygons, and arcs. Here we draw a rectangle:

```
public class RectangleApplet extends Applet  
{  
    public void paint(Graphics g)  
    {  
        . . .  
        Rectangle cerealBox = new Rectangle(5, 10, 20, 30);  
        g2.draw(cerealBox);  
        . . .  
    }  
}
```

The `Graphics`, `Graphics2D`, and `Rectangle` classes are part of the `java.awt` package. As was mentioned in Chapter 1, the acronym AWT stands for Abstract Windowing Toolkit. This is the original user interface toolkit that Sun supplied for Java. It defines many classes for graphics programming, a mechanism for event handling, and a set of user interface components. In this chapter, we focus on the graphics classes of the AWT.

Here is the complete applet for displaying the rectangle shapes.

File `RectangleApplet.java`

```
1 import java.applet.Applet;  
2 import java.awt.Graphics;  
3 import java.awt.Graphics2D;  
4 import java.awt.Rectangle;  
5  
6 /**  
7     An applet that draws two rectangles.
```

```
8 */
9 public class RectangleApplet extends Applet
10 {
11     public void paint(Graphics g)
12     {
13         // recover Graphics2D
14
15         Graphics2D g2 = (Graphics2D)g;
16
17         // construct a rectangle and draw it
18
19         Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
20         g2.draw(cerealBox);
21
22         // move rectangle 15 units sideways and 25 units down
23
24         cerealBox.translate(15, 25);
25
26         // draw moved rectangle
27
28         g2.draw(cerealBox);
29     }
30 }
```



Advanced Topic

4.1

The Java Runtime Environment and Java Plug-ins

- ▼ You can run applets inside the applet viewer program, but applets are meant to be executed inside a browser. The first versions of browser programs that supported Java contained a built-in Java virtual machine to execute the downloaded Java applets. That turned out not to be a good idea. New versions of Java appeared rapidly, and the browser manufacturers were unable to keep up. Also, browser manufacturers made minor changes to the Java implementations, causing compatibility problems.
- ▼ In 1998, Sun Microsystems, the company that invented the Java language, realized that it is best to separate the browser and the virtual machine. Sun now packages the virtual machine as the “Java Runtime Environment”, which is to be installed in a standard place on each computer that supports Java. Browser manufacturers are encouraged to use the Java Runtime Environment, not their own Java implementation, to execute applets. That way, users can update their Java implementations and browsers separately. If your browser uses this approach, then it will be able to execute your Java applets without problems. As of this writing, Netscape 6 and Opera work in this way.
- ▼ However, Microsoft’s Internet Explorer browser does not currently support a modern version of the Java virtual machine. To solve this problem, Sun Microsystems provided a second tool, the “Java Plug-in”. The Java Plug-in uses the Microsoft ActiveX component

- ▼ architecture to add Java support to Internet Explorer. If you want to display your applets inside Internet Explorer, you need to download the Java Plug-in from <http://java.sun.com/products/plugin/index.html>.
- ▼ Unfortunately, the HTML required to activate the browser extension is quite a bit more arcane than the simple `applet` tag. Sun has developed a program, the “Java Plug-in HTML Converter”, that can translate HTML pages containing `applet` tags to HTML pages with the appropriate tags to launch the Java Plug-in. You can download the converter from <http://java.sun.com/products/plugin/1.3/converter.html>.

4.4 Graphical Shapes

In Section 4.3 you learned how to write an applet that draws rectangles. In this section you will learn how to draw other shapes: ellipses and lines. With these graphical elements you can draw quite a few interesting pictures.

The `Rectangle2D.Double`, `Ellipse2D.Double`, and `Line2D.Double` classes describe graphical shapes.

To draw an ellipse, you specify its *bounding box* (see Figure 7) in the same way that you would specify a rectangle, namely by the *x*- and *y*-coordinates of the top-left corner and the width and height of the box.

However, there is no simple `Ellipse` class that you can use. Instead, you must use one of the two classes `Ellipse2D.Float` and `Ellipse2D.Double`, depending on whether you want to store the ellipse coordinates as `float` or as `double` values. Since `double` values are more convenient to use than `float` values in Java, we will always use the `Ellipse2D.Double` class. Here is how you construct an ellipse:

```
Ellipse2D.Double easterEgg = new Ellipse2D.Double(5, 10, 15, 20);
```

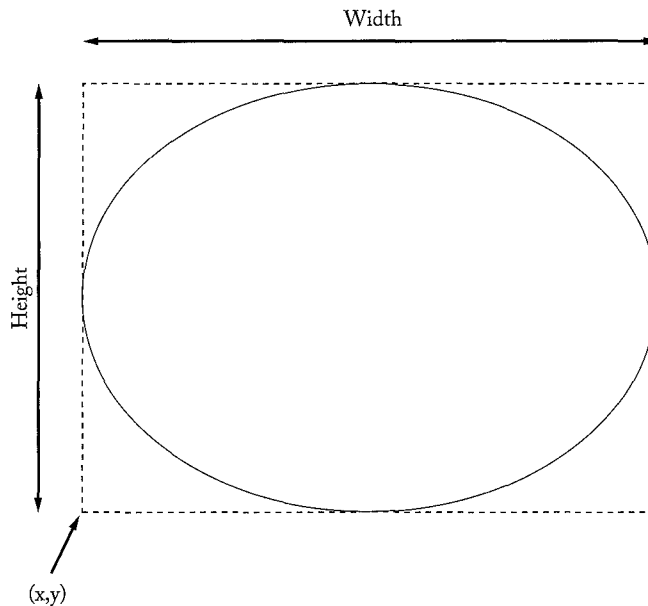
The class name `Ellipse2D.Double` looks different from the class names that you have encountered up to now. It consists of two class names `Ellipse2D` and `Double` separated by a period (`.`). This indicates that `Ellipse2D.Double` is a so-called *inner* class inside `Ellipse2D`. When constructing and using ellipses, you don't actually need to worry about the fact that `Ellipse2D.Double` is an inner class—just think of it as a class with a long name. However, in the `import` statement at the top of your program, you must be careful that you import only the *outer* class:

```
import java.awt.geom.Ellipse2D;
```

You may wonder why an ellipse would want to store its size as `double` values when the screen coordinates are measured in pixels. As you'll see later in this chapter, working with pixel coordinates can be cumbersome. It is often a good idea to switch to different units, and then it can be handy to use floating-point coordinates. (Note that the `Rectangle` class uses integer coordinates, so you will need to use a separate class called `Rectangle2D.Double` whenever you use rectangles with floating-point coordinates.)

Drawing an ellipse is easy: You use exactly the same `draw` method of the `Graphics2D` class that you used for drawing rectangles.

```
g2.draw(easterEgg);
```

**Figure 7****An Ellipse and Its Bounding Box**

To draw a circle, simply set the width and height to the same value:

```
Ellipse2D.Double circle =
    new Ellipse2D.Double(x, y, diameter, diameter);
g2.draw(circle);
```

Notice that (x, y) is the top-left corner of the bounding box, *not* the center of the circle.

To draw a line, you use an object of the `Line2D.Double` class. You construct a line by specifying its two end points. You can do this in two ways. You can simply give the x - and y -coordinates of both end points:

```
Line2D.Double segment = new Line2D.Double(x1, y1, x2, y2);
```

Or you can specify each endpoint as an object of the `Point2D.Double` class:

```
Point2D.Double from = new Point2D.Double(x1, y1);
Point2D.Double to = new Point2D.Double(x2, y2);
```

```
Line2D.Double segment = new Line2D.Double(from, to);
```

The latter is more object-oriented, and it is also often more useful, in particular if the point objects can be reused elsewhere in the same drawing.

To draw thicker lines, supply a different *stroke* object to the `Graphics2D` parameter. For example, to get lines that are 4 pixels thick, you call

```
g2.setStroke(new BasicStroke(4.0F));
```

All shapes that you draw after making that call will be drawn with thicker lines.

4.5 Colors

When you first start drawing, all shapes are drawn with a black pen. To change the color, you need to supply an object of type `Color`. Java uses the *RGB color model*. That is, you specify a color by the amounts of the *primary colors*—red, green, and blue—that make up the color. The amounts are given as `float` values, which you must identify by a suffix `F`. They vary from `0.0F` (primary color not present) to `1.0F` (maximum amount present). For example,

```
Color magenta = new Color(1.0F, 0.0F, 1.0F);
```

When you set a new color in the graphics context, it is used for subsequent drawing operations.

constructs a `Color` object with maximum red, no green, and maximum blue, yielding a bright purple color called magenta.

For your convenience, a variety of colors have been predefined in the `Color` class. Table 1 shows those predefined colors and their RGB values. For example, `Color.pink` has been predefined to be the same color as `new Color(1.0F, 0.7F, 0.7F)`.

Color	RGB Value
<code>Color.black</code>	<code>0.0F, 0.0F, 0.0F</code>
<code>Color.blue</code>	<code>0.0F, 0.0F, 1.0F</code>
<code>Color.cyan</code>	<code>0.0F, 1.0F, 1.0F</code>
<code>Color.gray</code>	<code>0.5F, 0.5F, 0.5F</code>
<code>Color.darkGray</code>	<code>0.25F, 0.25F, 0.25F</code>
<code>Color.lightGray</code>	<code>0.75F, 0.75F, 0.75F</code>
<code>Color.green</code>	<code>0.0F, 1.0F, 0.0F</code>
<code>Color.magenta</code>	<code>1.0F, 0.0F, 1.0F</code>
<code>Color.orange</code>	<code>1.0F, 0.8F, 0.0F</code>
<code>Color.pink</code>	<code>1.0F, 0.7F, 0.7F</code>
<code>Color.red</code>	<code>1.0F, 0.0F, 0.0F</code>
<code>Color.white</code>	<code>1.0F, 1.0F, 1.0F</code>
<code>Color.yellow</code>	<code>1.0F, 1.0F, 0.0F</code>

Table 1

Predefined Colors and Their RGB Values

Once you have an object of type `Color`, you can change the *current color* of the `Graphics2D` object with the `setColor` method. For example, the following code draws a rectangle in black, then switches the color to red, and draws the next rectangle in red:

```
public void paint(Graphics g)
{
    Graphics2D g2 = (Graphics2D)g;

    Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
    g2.draw(cerealBox); // draws in black

    cerealBox.translate(15, 25); // move rectangle

    g2.setColor(Color.red); // set current color to red
    g2.draw(cerealBox); // draws in red
}
```

If you want to color the inside of the shape, you use the `fill` method instead of the `draw` method. For example,

```
g2.fill(cerealBox);
```

fills the inside of the rectangle with the current color.

4.6 Fonts

The `drawString` method of the `Graphics2D` class draws a string, starting at its basepoint.

You often want to put text inside a drawing, for example to label some of the parts. You use the `drawString` method of the `Graphics2D` class to draw a string anywhere in a window. You must specify the string and the *x*- and *y*-coordinates of the *basepoint* of the first character in the string (see Figure 8). For example,

```
g2.drawString("Applet", 50, 100);
```

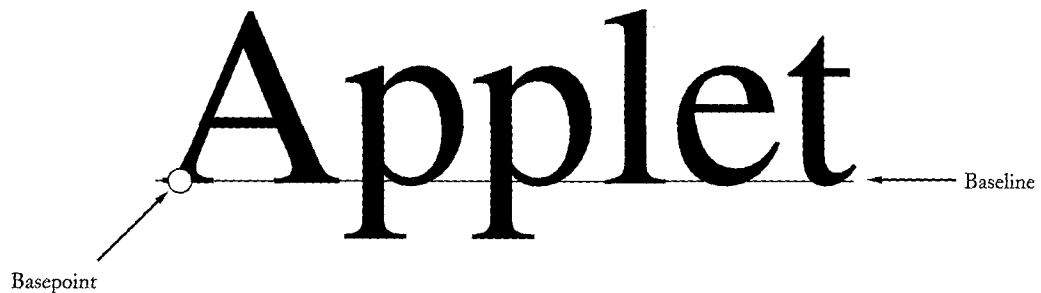


Figure 8

Basepoint and Baseline

A font is described by its face name, style, and point size.

You can select different fonts. The procedure is similar to setting the drawing color. You create a `Font` object and call the `setFont` method of the `Graphics2D` class. To construct a `Font` object, you specify

- The font face name
- The *style* (one of `Font.PLAIN`, `Font.BOLD`, `Font.ITALIC`, or `Font.BOLD + Font.ITALIC`)
- The point size

The font face name is either one of the five *logical face names* in Table 2 or a *typeface name*, the name of a typeface that is available on your computer, such as “Times Roman” or “Helvetica”. “Times Roman” and “Helvetica” are the names of popular typefaces that were designed many years ago and are in very widespread use today. These fonts differ in the shapes of their letters. The most visible difference is that the characters of the Times Roman font are composed of strokes with small cross segments at the ends, called *serifs*. The characters of the Helvetica font do not have serifs (see Figure 9). A typeface such as Helvetica is called a *sans-serif* font. It is generally believed that the serifs help make text easier to read. In

Name	Sample	Description
Serif	The quick brown fox	A serif-style font such as Times Roman
SansSerif	The quick brown fox	A sans-serif font such as Helvetica
Monospaced	The quick brown fox	A font in which all characters have the same width, such as Courier
Dialog	The quick brown fox	A screen font suitable for labels in dialogs
DialogInput	The quick brown fox	A screen font suitable for user input in text field

Table 2

Logical Font Names

Helvetica

Times Roman

Courier

Figure 9

Common Fonts

fact, the Times Roman font was designed specifically for the London *Times* newspaper, to be easy to read on newsprint paper. Most books (including this one) use a serif typeface for the body text. Sans-serif typefaces are appropriate for headlines, figure labels, and so on. Many other typefaces have been designed over the centuries, with and without serifs. For example, Garamond is another popular serif-style typeface. A third kind of typeface that you will commonly see is Courier, a font that was originally designed for typewriters.

The design of a good typeface requires artistic judgment and substantial experience. However, in the United States, the shapes of letters are considered industrial design that cannot be protected by copyright. For that reason, typeface designers protect their rights by trademarking the *names* of the fonts. For example, the names “Times Roman” and “Helvetica” are trademarks of the Linotype Corporation. Although other companies can create lookalike fonts (just as certain companies create imitations of famous perfumes), they have to give them different names. That’s why you find fonts with names such as “Times New Roman” or “Arial”. That makes it a bit of a bother to select fonts by their names, especially if you don’t know what fonts are available on a particular computer. For that reason, we will specify fonts by their *logical* face names. Java recognizes five logical font names (see Table 2) that are mapped to fonts that exist on every computer system. For example, if you request the “SansSerif” font, then the Java font mapper will go and search for the best general-purpose sans-serif font available. On a computer running the Windows operating system, you will get the “Arial” font. On a Macintosh, you will get “Helvetica”.

To create a font, you must specify the *point size*: the height of the font in the typesetter’s unit called *points*. The height of a font is measured from the top of the *ascender* (the top part of letters such as *b* and *l*) to the bottom of the *descender* (the bottom part of letters such as *g* and *p*). There are 72 points per inch. For example, a 12-point font has a height of $\frac{1}{6}$ inch. Actually, the point size of a font is only an approximate measure, and you can’t necessarily be sure that two fonts with the same point size have matching character sizes. The actual sizes will in any case depend on the size and resolution of your monitor screen. Without getting into fine points of typography, it is best if you simply remember a few typical point sizes: 8 point (“small”), 12 point (“medium”), 18 point (“large”), and 36 point (“huge”).

Here is how you can write “Applet” in huge pink letters:

```
final int HUGE_SIZE = 36;
String message = "Applet";
Font hugeFont = new Font("Serif", Font.BOLD, HUGE_SIZE);
g2.setFont(hugeFont);
g2.setColor(Color.pink);
g2.drawString(message, 50, 100);
```

The *x*- and *y*-positions in the `drawString` method can be specified either as `int` or as `float`.

▼ Advanced Topic

4.2

▼ **Accurate Positioning of Text**

When drawing strings on the screen, you usually need to position them accurately. For example, if you want to draw two lines of text, one below the other, then you need to



Figure 10

Text Layout Measurements

know the distance between the two basepoints. Of course, the size of a string depends on the shapes of the letters, which in turn depends on the font face and point size. You will need to know a few typographical measurements (see Figure 10):

- The *ascent* of a font is the height of the largest letter above the baseline.
- The *descent* of a font is the depth below the baseline of the letter with the lowest descender.

These values describe the *vertical* extent of strings. The *horizontal* extent depends on the individual letters in a string. In a *monospaced* font, all letters have the same width. Monospaced fonts are still used for computer programs, but for plain text they are as outdated as the typewriter. In a *proportionally spaced font*, different letters have different widths. For example, the letter *l* is much narrower than the letter *m*.

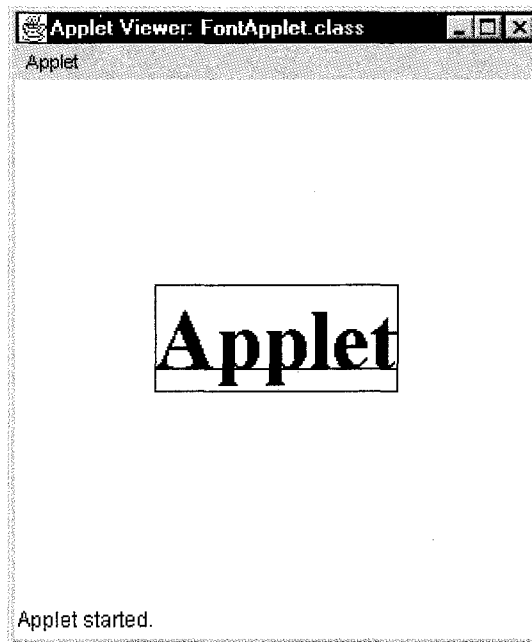
To measure the *size* of a string, you need to construct a `FontRenderContext` object, which you obtain from the `Graphics2D` object by calling `getFontRenderContext`. A font render context is an object that knows how to transform letter shapes (which are described as curves) into pixels. In general, a “context” object is usually an object that has some specialized knowledge how to carry out complex tasks. You don’t have to worry how the context object works; you just create it and pass it along as required. The `Graphics2D` object is another example of a context object—many people call it a “graphics context”.

To get the size of a string, you call the `getStringBounds` method of the `Font` class. For example,

```
String message = "Applet";
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D bounds = hugeFont.getStringBounds(message, context);
```

The returned rectangle is positioned so that the origin (0, 0) falls on the basepoint (see Figure 8). Therefore, you can get the ascent, descent, height, and width as

```
double yMessageAscent = -bounds.getY();
double yMessageDescent = bounds.getHeight() + bounds.getY();
double yMessageHeight = bounds.getHeight();
double xMessageWidth = bounds.getWidth();
```

**Figure 11**

The Font Applet Displays a Centered String

The following program uses these measurements to center a string precisely in the middle of the applet window (see Figure 11). To center the string, you need to know the size of the applet. (The user might have resized it, so you can't simply use the values of `width` and `height` in the HTML file.) The `getWidth` and `getHeight` methods return the applet size in pixels. To center the string horizontally, think of the amount of blank space that you have available. The width of the applet window is `getWidth()`. The width of the string is `xMessageWidth`. Therefore, the blank space is the difference,

```
getWidth() - xMessageWidth
```

Half of that blank space should be distributed on either side. Therefore, the string should start at

```
double xLeft = (getWidth() - xMessageWidth) / 2;
```

For the same reason, the top of the string is at

```
double yTop = (getHeight() - yMessageHeight) / 2;
```

But the `drawString` method needs the basepoint of the string. You get to the baseposition by adding the ascent:

```
double yBase = yTop + yMessageAscent;
```

Following this note is the complete program. When you run it in the applet viewer, try resizing the applet window and observe that the string always stays centered.

File FontApplet.java

```
1 import java.applet.Applet;
2 import java.awt.Font;
3 import java.awt.Graphics;
4 import java.awt.Graphics2D;
5 import java.awt.font.FontRenderContext;
6 import java.awt.geom.Line2D;
7 import java.awt.geom.Rectangle2D;
8
9 /**
10  * This applet draws a string that is centered in the
11  * applet window.
12  */
13 public class FontApplet extends Applet
14 {
15     public void paint(Graphics g)
16     {
17         Graphics2D g2 = (Graphics2D)g;
18
19         // select the font into the graphics context
20
21         final int HUGE_SIZE = 48;
22         Font hugeFont =
23             new Font("Serif", Font.BOLD, HUGE_SIZE);
24         g2.setFont(hugeFont);
25
26         String message = "Applet";
27
28         // measure the string
29
30         FontRenderContext context =
31             g2.getFontRenderContext();
32         Rectangle2D bounds =
33             hugeFont.getStringBounds(message, context);
34
35         double yMessageAscent = -bounds.getY();
36         double yMessageDescent =
37             bounds.getHeight() + bounds.getY();
38         double yMessageHeight = bounds.getHeight();
39         double xMessageWidth = bounds.getWidth();
40
41         // center the message in the window
42
43         double xLeft = (getWidth() - xMessageWidth) / 2;
44         double yTop = (getHeight() - yMessageHeight) / 2;
45         double yBase = yTop + yMessageAscent;
46
47         g2.drawString(message, (float)xLeft, (float)yBase);
48         // draw bounding rectangle
49     }
50 }
```

```
50     g2.draw(new Rectangle2D.Double(xLeft, yTop,
51         xMessageWidth, yMessageHeight));
52
53     // draw base line
54     g2.draw(new Line2D.Double(xLeft, yBase,
55         xLeft + xMessageWidth, yBase));
56 }
57 }
```

4.7 Drawing Complex Shapes

It is a good idea to make a class for each complex graphical shape.

The next program shows how you can put shapes together to draw a simple figure of a car—see Figure 12. It is a good idea to make a separate class for each complex shape that you want to draw. For example, the program at the end of this section defines a Car class.

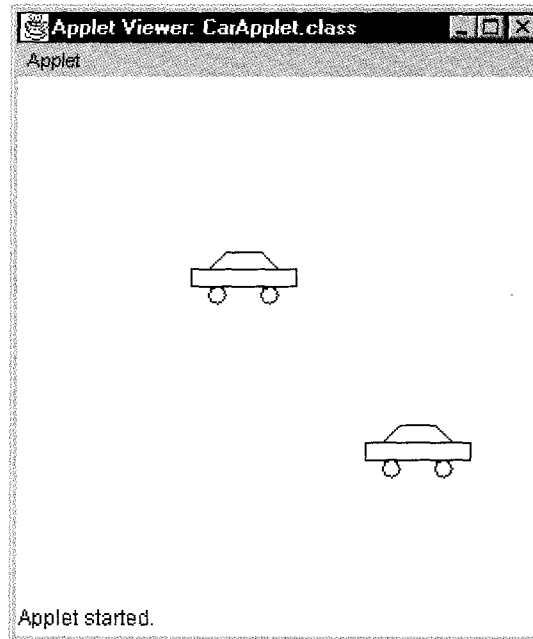
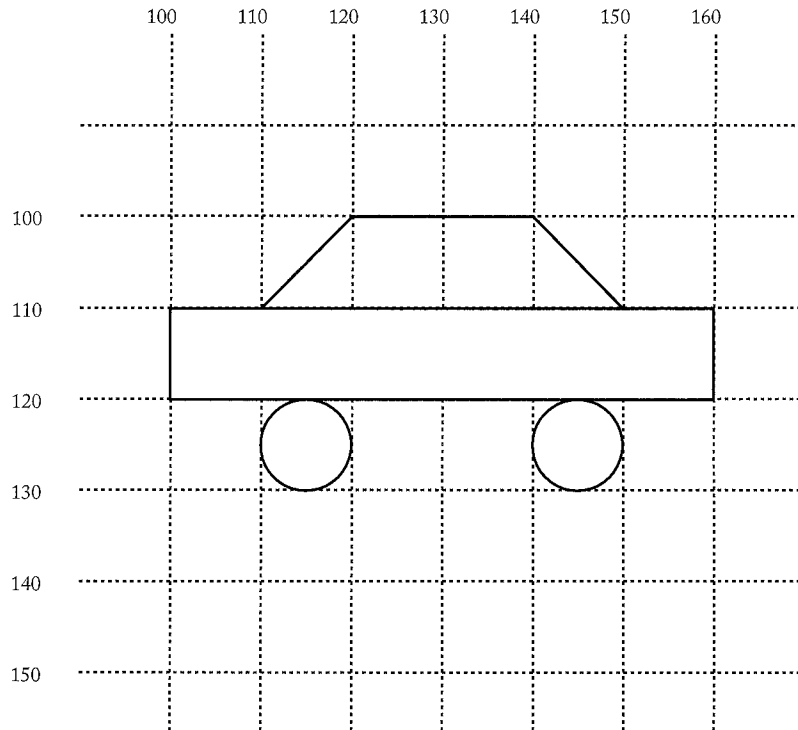


Figure 12

The Car Applet Draws Two Car Shapes

**Figure 13**

Using Graph Paper to Find Shape Coordinates

```
class Car
{
    . . .
    public void draw(Graphics2D g2)
    {
        // drawing instructions
        . . .
    }
}
```

To figure out how to draw a complex shape, make a sketch on graph paper.

The coordinates of the car parts seem a bit arbitrary. To come up with suitable values, you want to draw the image on graph paper and read off the coordinates—see Figure 13.

Here is the program. Unfortunately, in programs such as these it is difficult to avoid the “magic numbers” for the coordinates of the various shapes.

File CarApplet.java

```
1 import java.applet.Applet;
2 import java.awt.Graphics;
3 import java.awt.Graphics2D;
```

```
4 /**
5  This applet draws two car shapes.
6  */
7  public class CarApplet extends Applet
8  {
9      public void paint(Graphics g)
10     {
11         Graphics2D g2 = (Graphics2D)g;
12
13         Car car1 = new Car(100, 100);
14         Car car2 = new Car(200, 200);
15
16         car1.draw(g2);
17         car2.draw(g2);
18     }
19 }
```

File Car.java

```
1  import java.awt.Graphics2D;
2  import java.awt.geom.Ellipse2D;
3  import java.awt.geom.Line2D;
4  import java.awt.geom.Point2D;
5  import java.awt.geom.Rectangle2D;
6
7  /**
8   A car shape that can be positioned anywhere on the screen.
9  */
10 public class Car
11 {
12     /**
13      Constructs a car with a given top left corner.
14      @param x the x-coordinate of the top left corner
15      @param y the y-coordinate of the top left corner
16     */
17     public Car(double x, double y)
18     {
19         xLeft = x;
20         yTop = y;
21     }
22
23     /**
24      Draws the car.
25      @param g2 the graphics context
26     */
27     public void draw(Graphics2D g2)
28     {
29         Rectangle2D.Double body = new
```

```
30         Rectangle2D.Double(xLeft, yTop + 10, 60, 10);
31         Ellipse2D.Double frontTire = new
32             Ellipse2D.Double(xLeft + 10, yTop + 20, 10, 10);
33         Ellipse2D.Double rearTire = new
34             Ellipse2D.Double(xLeft + 40, yTop + 20, 10, 10);
35
36         // the bottom of the windshield
37         Point2D.Double r1
38             = new Point2D.Double(xLeft + 10, yTop + 10);
39         // the front of the roof
40         Point2D.Double r2
41             = new Point2D.Double(xLeft + 20, yTop);
42         // the rear of the roof
43         Point2D.Double r3
44             = new Point2D.Double(xLeft + 40, yTop);
45         // the bottom of the rear window
46         Point2D.Double r4
47             = new Point2D.Double(xLeft + 50, yTop + 10);
48
49         Line2D.Double frontWindshield
50             = new Line2D.Double(r1, r2);
51         Line2D.Double roofTop
52             = new Line2D.Double(r2, r3);
53         Line2D.Double rearWindow
54             = new Line2D.Double(r3, r4);
55
56         g2.draw(body);
57         g2.draw(frontTire);
58         g2.draw(rearTire);
59         g2.draw(frontWindshield);
60         g2.draw(roofTop);
61         g2.draw(rearWindow);
62     }
63
64     private double xLeft;
65     private double yTop;
66 }
```

▼ ? HOWTO

4.1

Drawing Graphical Shapes

▼ You can program applets that display a wide variety of graphical shapes. These instructions give you a step-by-step procedure how to decompose a drawing into parts and implement a program that produces the drawing.

▼ **Step 1** Determine the shapes that you need for the drawing

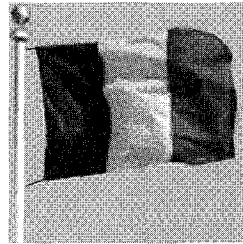
You can use the following shapes:

- ▼ ● Squares and rectangles
- ▼ ● Circles and ellipses
- ▼ ● Lines

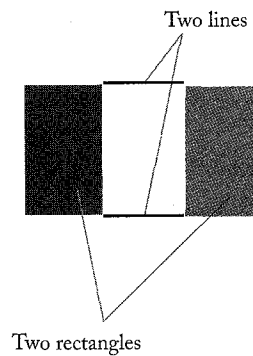
▼ You can draw the outlines of these shapes in any color, or you can fill the insides of these shapes with any color.

▼ You can also use text to label parts of your drawing.

▼ For example, many national flag designs consist of three equally wide sections of different colors, side by side:



▼ You could draw such a flag using three rectangles. But if the middle rectangle is white, as it is, for example, in the flag of Italy (green, white, red), it is easier and looks better just to draw two lines on the top and bottom of the middle portion:



▼ **Step 2** Find the coordinates for the shapes

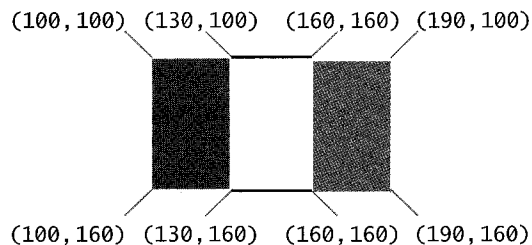
▼ You now need to find the exact positions for the geometric shapes.

- ▼ ● For rectangles, you need the x - and y -position of the top left corner, the width, and height.
- ▼ ● For ellipses, you need the top left corner, width, and height of the bounding rectangle.

- ▼ ● For lines, you need the x - and y -positions of the starting point and the end point.
- ▼ ● For text, you need the x - and y -positions of the basepoint.

A typical size for an applet is 300 by 300 pixels. You may not want the flag crammed all the way to the top, so perhaps the upper left corner of the flag should be at the point (100, 100).

- ▼ Many flags, such as the flag of Italy, have a width : height ratio of 3 : 2. (You can often find exact proportions for a particular flag by doing a bit of Internet research on one of several Flags of the World sites.) For example, if you make the flag 90 pixels wide, then it should be 60 pixels tall. (Why not make it 100 pixels wide? Then the height would be $100 \cdot 2 / 3 \approx 67$, which seems more awkward.)
- ▼ Now you can compute the coordinates of all the important points of the shape:



Step 3 Write Java statements to draw the shapes

In our example, there are two rectangles and two lines:

- ▼

```
Rectangle2D.Double leftRectangle
    = new Rectangle2D.Double(100, 100, 30, 60);
```
- ▼

```
Rectangle2D.Double rightRectangle
    = new Rectangle2D.Double(160, 100, 30, 60);
```
- ▼

```
Line2D.Double topLine
    = new Line2D.Double(130, 100, 160, 100);
```
- ▼

```
Line2D.Double bottomLine
    = new Line2D.Double(130, 160, 160, 160);
```

- ▼ If you are more ambitious, then you can express the coordinates in terms of a few variables. In the case of the flag, we have arbitrarily chosen the top left corner and the width. All other coordinates follow from those choices. If you decide to follow the ambitious approach, then the rectangles and lines are determined as follows:

- ▼

```
Rectangle2D.Double leftRectangle
    = new Rectangle2D.Double(xLeft, yTop,
        width / 3, width * 2 / 3);
```
- ▼

```
Rectangle2D.Double rightRectangle
    = new Rectangle2D.Double(xLeft + width / 3, yTop,
        width / 3, width * 2 / 3);
```
- ▼

```
Line2D.Double topLine
    = new Line2D.Double(xLeft + width / 3, yTop,
        xLeft + width * 2 / 3, yTop);
```

```

▼ Line2D.Double bottomLine
    = new Line2D.Double(
        xLeft + width / 3, yTop + width * 2 / 3,
▼        xLeft + width * 2 / 3, yTop + width * 2 / 3);

```

Now you need to fill the rectangles and draw the lines. For the flag of Italy, the left rectangle is green and the right rectangle is red. Remember to switch colors before the filling and drawing operations:

```

▼ g2.setColor(Color.green);
g2.fill(leftRectangle);
▼ g2.setColor(Color.red);
g2.fill(rightRectangle);
▼ g2.setColor(Color.black);
g2.draw(topLine);
g2.draw(bottomLine);

```

▼ **Step 4** Combine the drawing statements with the applet “plumbing”

The simplest form of the “plumbing” looks like this:

```

▼ public class MyApplet extends Applet
    {
        public void paint(Graphics g)
▼        {
            Graphics2D g2 = (Graphics2D)g;
            // your drawing code goes here
            . . .
▼        }
    }

```

▼ In our example, you can simply add all shapes and drawing instructions inside the `paint` method:

```

▼ public class ItalianFlagApplet extends Applet
    {
        public void paint(Graphics g)
▼        {
            Graphics2D g2 = (Graphics2D)g;
            Rectangle2D.Double leftRectangle
▼            = new Rectangle2D.Double(100, 100, 30, 60);
            . . .
▼            g2.setColor(Color.green);
            g2.fill(leftRectangle);
            . . .
▼        }
    }

```

▼ That approach is acceptable for simple drawings, but it is not very object-oriented. After all, a flag is an object. It is better to make a separate class for the flag. Then you can draw different flags at different positions and sizes. Specify the sizes in a constructor and supply a `draw` method:

```

▼      public class ItalianFlag
        {
          public ItalianFlag(double x, double y, double aWidth)
          {
            xLeft = x;
            yTop = y;
            width = aWidth;
          }

          public void draw(Graphics2D g2)
          {
            Rectangle2D.Double leftRectangle
              = new Rectangle2D.Double(xLeft, yTop,
                width / 3, width * 2 / 3);
            . . .
            g2.setColor(Color.green);
            g2.fill(leftRectangle);
            . . .
          }

          private double xLeft;
          private double yTop;
          private double width;
        }

```

You still need a separate class for the applet, but it is very simple:

```

▼      public class ItalianFlagApplet extends Applet
        {
          public void paint(Graphics g)
          {
            Graphics2D g2 = (Graphics2D)g;
            ItalianFlag flag = new ItalianFlag(100, 100, 90);
            flag.draw(g2);
          }
        }

```

▼ You may wish to modify this code to make the flag fit comfortably in the applet area even if the applet is not 300 by 300. Use the applet `getWidth()` and `getHeight()` methods to find the actual size of the applet window, and use these dimensions to compute the constructor parameters, similar to what was done for text in Advanced Topic 4.2.

▼ **Step 5** Write the HTML file for the applet

▼ To show an applet in the applet viewer or a Java 2-enabled browser, you need an HTML file that specifies the name of the applet class and the desired width and height of the applet. Here is a minimal HTML file for the Italian flag applet:

```

<applet code="ItalianFlagApplet.class"
width="300" height="300"></applet>

```

- ▼ If you like, you can add more HTML code around the applet tag to describe your applet. The description shows up when you view the HTML file in a browser, but the applet viewer displays only the applet.



Random Fact

4.2

Computer Graphics

- ▼ Generating and manipulating visual images are among the most exciting applications of the computer. We distinguish different kinds of graphics.
- ▼ *Diagrams, such as numeric charts or maps, are artifacts that convey information to the viewer* (see Figure 14). They do not directly depict anything that occurs in the natural world but are a tool for visualizing information.
- ▼ *Scenes are computer-generated images that attempt to depict images of the real or an imagined world* (see Figure 15). It turns out to be quite challenging to render light and shadows accurately. Special effort must be taken so that the images do not look overly

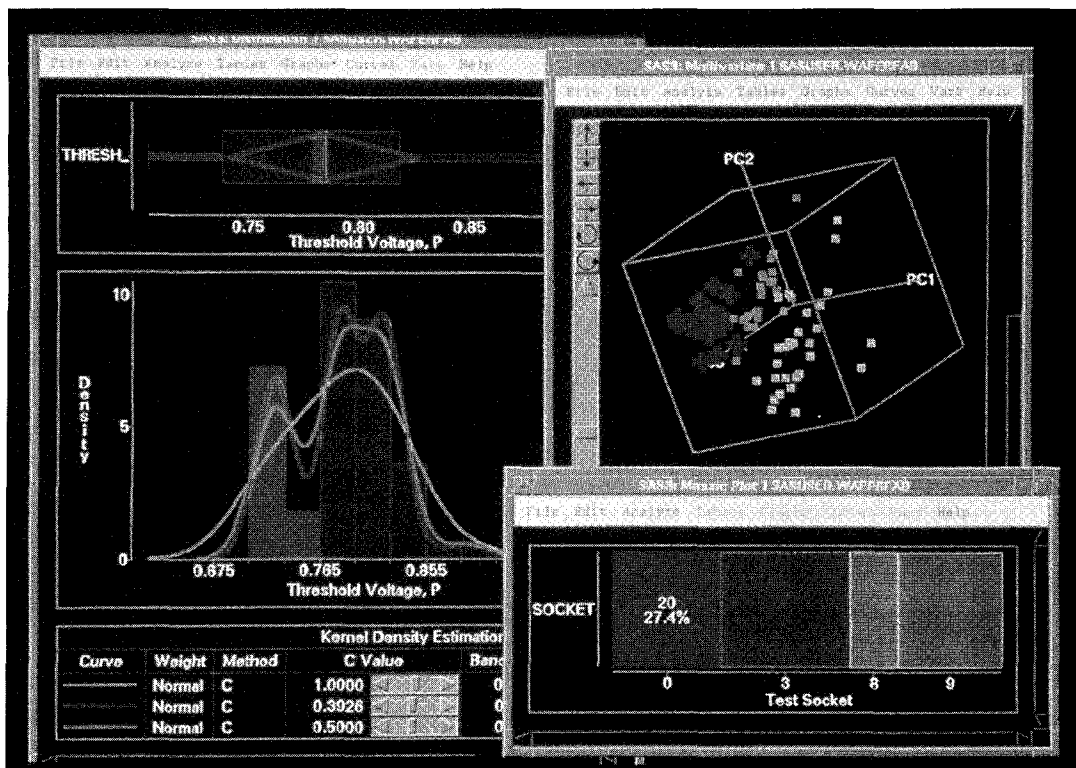


Figure 14

Diagrams



Figure 15

Scene

neat and simple; clouds, rocks, leaves, and dust in the real world have a complex and somewhat random appearance. The degree of realism in these images is constantly improving.

Manipulated images are photographs or film footage of actual events that have been converted to digital form and edited by the computer (see Figure 16). For example, film



Figure 16

Manipulated Image

- ▼ sequences of the movie *Apollo 13* were produced by starting from actual images and changing the perspective, showing the launch of the rocket from a more dramatic viewpoint.
- ▼ Computer graphics is one of the most challenging fields in computer science. It requires processing of massive amounts of information at very high speed. New algorithms are constantly invented for this purpose. Displaying an overlapping set of three-dimensional objects with curved boundaries requires advanced mathematical tools. Realistic modeling of textures and biological entities requires extensive knowledge of mathematics, physics, and biology.

4.8 Reading Text Input

The applets that you have seen so far are quite nice for drawing, but they aren't interactive—you can't change the positions of the shapes that are drawn on the screen. Interactive input in a graphical program turns out to be more complex than in a console program. In a console program, the programmer dictates the control flow and forces the user to enter input in a predetermined order. A graphical program, however, generally makes available to the program user a large number of controls (buttons, input fields, scroll bars, and so on), which users can manipulate in any order they please. Therefore, the program must be prepared to process input from multiple sources in random order. You will learn how to do that in Chapter 10.

In the meantime, we will simply read input with the `JOptionPane.showInputDialog` method that you saw in Chapter 3.

You should place any calls to the `showInputDialog` method into the applet constructor, not the `paint` method. That way, the user is prompted for input once, when the applet is first displayed. (If you place the calls to `showInputDialog` inside the `paint` method, then you are prompted for new input every time the applet is repainted.) To get another chance to supply input, select "Reload" or "Refresh" from the browser or applet viewer menu.

If you put the call to `showInputDialog` inside the applet constructor, then your applet needs to remember the user input in one or more instance fields and refer to these variables in the `paint` method. The following program is an example. It prompts the user for red, green, and blue values, and then fills a rectangle with the color that the user specified. For example, if you enter 1.0, 0.7, 0.7, then the rectangle is filled with pink color.

File `ColorApplet.java`

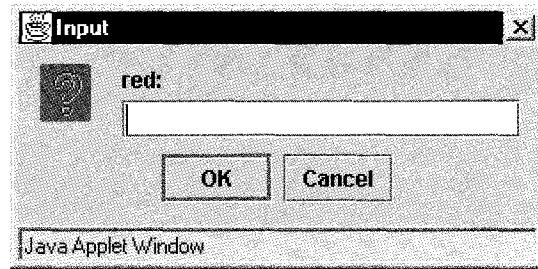
```
1 import java.applet.Applet;
2 import java.awt.Color;
3 import java.awt.Graphics;
4 import java.awt.Graphics2D;
5 import java.awt.Rectangle;
6 import javax.swing.JOptionPane;
7
8 /**
9  * An applet that lets a user choose a color by specifying
10 * the fractions of red, green, and blue.
11 */
12 public class ColorApplet extends Applet
13 {
```

```
14 public ColorApplet()
15 {
16     String input;
17     // ask the user for red, green, blue values
18
19     input = JOptionPane.showInputDialog("red:");
20     float red = Float.parseFloat(input);
21
22     input = JOptionPane.showInputDialog("green:");
23     float green = Float.parseFloat(input);
24
25     input = JOptionPane.showInputDialog("blue:");
26     float blue = Float.parseFloat(input);
27
28     fillColor = new Color(red, green, blue);
29 }
30
31 public void paint(Graphics g)
32 {
33     Graphics2D g2 = (Graphics2D)g;
34
35     // select color into graphics context
36
37     g2.setColor(fillColor);
38
39     // construct and fill a square whose center is
40     // the center of the window
41
42     Rectangle square = new Rectangle(
43         (getWidth() - SQUARE_LENGTH) / 2,
44         (getHeight() - SQUARE_LENGTH) / 2,
45         SQUARE_LENGTH,
46         SQUARE_LENGTH);
47
48     g2.fill(square);
49 }
50
51 private static final int SQUARE_LENGTH = 100;
52
53 private Color fillColor;
54 }
```

When you run this program, you will note that the input dialog has a label identifying it as an “applet window” (see Figure 17). That is an applet security feature. It would be an easy matter for a cracker to write an applet that pops up a dialog “Your password has expired. Please reenter your password.” and then sends the input back to the web server. If you should happen to visit the web page containing that applet, you might be confused and reenter the password to your computer account, giving it to the cracker. The window label tips you off that it is an applet, and not your operating system, that displays the dialog. All windows that pop up from an applet have this warning label.

Figure 17

Applet Dialog with Warning Label



While the input dialog window is on the screen, the user is not allowed to do anything in the browser except type input and click on the OK or Cancel button. This kind of dialog is called a *modal dialog*. A sequence of modal input dialogs can be frustrating to the user, so it is not really a good user interface design. However, it is easy to program, and it makes sense for you to use it until you learn how to gather input in a more professional way—the topic of Chapter 12.



Advanced Topic

4.3

Applet Parameters

You have seen how to use the `showInputDialog` method of the `JOptionPane` class to supply user input to an applet. Another way of supplying input is sometimes useful: Use the `param` tag in the HTML page that loads the applet. The `param` tag has the form

```
<param name="..." value="..." />
```

You place one or more `param` tags between the `<applet>` and `</applet>` tags, like this:

```
<applet code="ColorApplet.class" width="300" height="300">
  <param name="Red" value="1.0" />
  <param name="Green" value="0.7" />
  <param name="Blue" value="0.7" />
</applet>
```

The applet can read these values with the `getParameter` method. For example, when the `ColorApplet` is loaded with the HTML tags given above, then `getParameter("Blue")` returns the string `"0.7"`. Of course, you then need to convert the string into a number.

You cannot call the `getParameter` method in the applet constructor. When the applet is first constructed, the parameters are not yet ready. Instead, you can read them either in the `paint` method or in a special `init` method. The applet viewer or browser calls the `init` method after the constructor but before the first call to `paint`.

```
public class ColorApplet extends Applet
{
    public void init()
```

```

▼      {
        float r = Float.parseFloat(getParameter("Red"));
        float g = Float.parseFloat(getParameter("Green"));
▼      float b = Float.parseFloat(getParameter("Blue"));
        fillColor = new Color(r, g, b);
        }

▼      public void paint(Graphics g)
        {
▼      . . .
        }

        private Color fillColor;
▼    }

```

Now you can change the color values simply by editing the HTML page.

4.9 Comparing Visual and Numerical Information

The next example shows how one can look at the same problem both visually and numerically. You want to figure out the intersection between a circle and a line. The circle has radius 100 and center (100, 100). Ask the user to specify the position of a vertical line. Then draw the circle, the line, and the intersection points (see Figure 18). Label them to display the exact locations.

Exactly where do the two shapes intersect? We need a bit of mathematics. The equation of a circle with radius r and center point (a, b) is

$$(x - a)^2 + (y - b)^2 = r^2$$

If you know x , then you can solve for y :

$$(y - b)^2 = r^2 - (x - a)^2$$

or

$$y - b = \pm \sqrt{r^2 - (x - a)^2}$$

hence

$$y = b \pm \sqrt{r^2 - (x - a)^2}$$

That is easy to compute in Java:

```

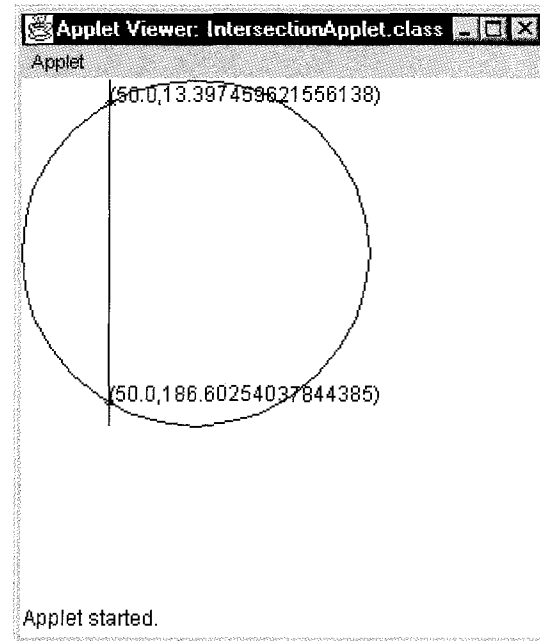
double root = Math.sqrt(r * r - (x - a) * (x - a));
double y1 = b + root;
double y2 = b - root;

```

But how do you know that you did both the math and the programming right?

Figure 18

Intersection of a Line and a Circle



If your program is correct, these two points will show up right on top of the actual intersections in the picture. If not, the two points will be at the wrong place.

If you look at Figure 18, you will see that the results match perfectly, which gives us confidence that everything is correct. See [Quality Tip 4.1](#) for more information on verifying that this program works correctly.

Here is the complete program.

File IntersectionApplet.java

```
1 import java.applet.Applet;
2 import java.awt.Graphics;
3 import java.awt.Graphics2D;
4 import java.awt.geom.Ellipse2D;
5 import java.awt.geom.Line2D;
6 import javax.swing.JOptionPane;
7
8 /**
9  * An applet that computes and draws the intersection points
10  * of a circle and a line.
11  */
12 public class IntersectionApplet extends Applet
13 {
14     public IntersectionApplet()
15     {
```


```
16     String input
17         = JOptionPane.showInputDialog("x:");
18     x = Integer.parseInt(input);
19 }
20
21 public void paint(Graphics g)
22 {
23     Graphics2D g2 = (Graphics2D)g;
24
25     double r = 100; // the radius of the circle
26
27     // draw the circle
28
29     Ellipse2D.Double circle = new
30         Ellipse2D.Double(0, 0, 2 * RADIUS, 2 * RADIUS);
31     g2.draw(circle);
32
33     // draw the vertical line
34
35     Line2D.Double line
36         = new Line2D.Double(x, 0, x, 2 * RADIUS);
37     g2.draw(line);
38
39     // compute the intersection points
40
41     double a = RADIUS;
42     double b = RADIUS;
43
44     double root =
45         Math.sqrt(RADIUS * RADIUS - (x - a) * (x - a));
46     double y1 = b + root;
47     double y2 = b - root;
48
49     // draw the intersection points
50
51     LabeledPoint p1 = new LabeledPoint(x, y1);
52     LabeledPoint p2 = new LabeledPoint(x, y2);
53
54     p1.draw(g2);
55     p2.draw(g2);
56 }
57
58 private static final double RADIUS = 100;
59 private double x;
60 }
```

File LabeledPoint.java

```
1 import java.awt.Graphics2D;
2 import java.awt.geom.Ellipse2D;
```

```
3 /**
4   A point with a label showing the point's coordinates.
5  */
6  public class LabeledPoint
7  {
8      /**
9       Construct a labeled point.
10      @param anX the x-coordinate
11      @param aY the y-coordinate
12     */
13     public LabeledPoint(double anX, double aY)
14     {
15         x = anX;
16         y = aY;
17     }
18
19     /**
20      Draws the point as a small circle with a coordinate label.
21      @param g2 the graphics context
22     */
23     public void draw(Graphics2D g2)
24     {
25         // draw a small circle centered around (x, y)
26
27         Ellipse2D.Double circle = new Ellipse2D.Double(
28             x - SMALL_CIRCLE_RADIUS,
29             y - SMALL_CIRCLE_RADIUS,
30             2 * SMALL_CIRCLE_RADIUS,
31             2 * SMALL_CIRCLE_RADIUS);
32
33         g2.draw(circle);
34
35         // draw the label
36
37         String label = "(" + x + "," + y + ")";
38
39         g2.drawString(label, (float)x, (float)y);
40     }
41
42     private static final double SMALL_CIRCLE_RADIUS = 2;
43
44     private double x;
45     private double y;
46 }
```

At this point you should be careful to specify only lines that intersect the circle. If the line doesn't meet the circle, then the program will attempt to compute a square root of a negative number, and a math error will occur. We have not yet discussed how to implement a test to protect against this situation. That will be the topic of the next chapter.


Quality Tip
4.1
Calculate Sample Data Manually

You should calculate test cases by hand to double-check that your program computes the correct answers.

It is usually difficult or impossible to prove that a given program functions correctly in all cases. For gaining confidence in the correctness of a program, or for understanding why it does not function as it should, manually calculated sample data are invaluable. If the program arrives at the same results as the manual calculation, our confidence in it is strengthened. If the manual results differ from the program results, we have a starting point for the debugging process.

Surprisingly, many programmers are reluctant to perform any manual calculations as soon as a program carries out the slightest bit of algebra. Their math phobia kicks in, and they irrationally hope that they can avoid the algebra and beat the program into submission by random tinkering, such as rearranging the + and - signs. Random tinkering is always a great time sink, but it rarely leads to useful results.

It is much smarter to look for test cases that are representative and easy to compute. In our example, let us look for three easy cases that we can compute by hand and then compare against program runs.

First, let the vertical line pass through the center of the circle. That is, x is 100. Then we expect the distance between the center and the intersection point to be the same as the radius of the circle. Now $\text{root} = \text{Math.sqrt}(100 * 100 - 0 * 0)$, which is 100. Therefore, y_1 is 0 and y_2 is 200. Those are indeed the top and bottom points on the circle. Now, that wasn't so hard.

Next, let the line touch the circle on the right. Then x is 200 and $\text{root} = \text{Math.sqrt}(100 * 100 - 100 * 100)$, which is 0. Therefore, y_1 and y_2 are both equal to 100, and indeed (200, 100) is the rightmost point of the circle. That also was pretty easy.

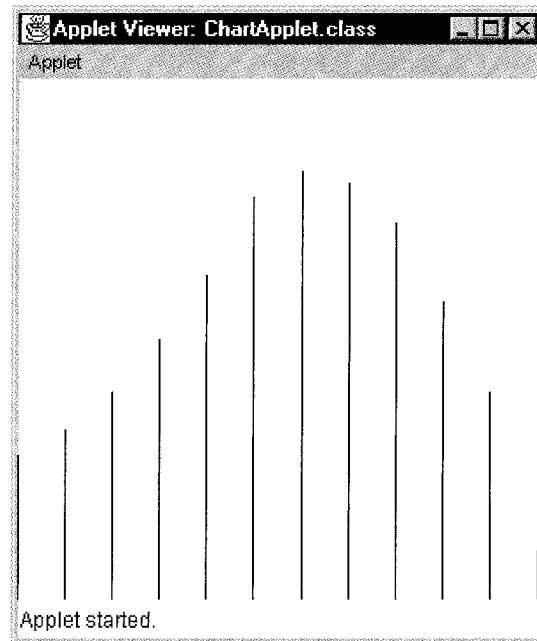
The first two cases were *boundary cases* of the problem. A program may work correctly for several special cases but still fail for more typical input values. Therefore we must come up with an intermediate test case, even if it means a bit more computation. Let us pick a simple value for x , say $x = 50$. Then $\text{root} = \text{Math.sqrt}(100 * 100 - 50 * 50) = \text{Math.sqrt}(7500)$. Using a calculator, you get approximately 86.6025. That yields $y_1 = 100 - 86.6025 = 13.3975$ and $y_2 = 100 + 86.6025 = 186.6025$. So what? Run the program and enter 50. First, you will find that the program also computes the same x - and y -values that you computed by hand. That is good—it confirms that you probably typed in the formulas correctly, and the intersection points really do fall on the right place.

4.10 Coordinate Transformations

By default, the `draw` method of the `Graphics2D` class uses *pixels* to measure screen locations. A pixel (short for “picture element”) is a dot on the screen. For example, the point (50, 100) is 50 pixels to the right and 100 pixels down from the top left corner of the panel. This default coordinate system is fine for simple test programs, but it is

Figure 19

Plotting Temperature Data



tedious when dealing with real-world data. For example, suppose you want to show a chart plotting the average temperature (degrees Celsius) in Phoenix, Arizona, for every month of the year (see Figure 19). The temperature ranges from 11 degrees Celsius in January to 33 degrees Celsius in July (see Table 3).

Pixel coordinates are not useful for real-world data sets. Pick convenient user coordinates and convert to pixels.

Here, the x -values range from 1 to 12 and the y -values range from 11 to 33. The applet pixel coordinates range from 0 to $\text{getWidth()} - 1$ and 0 to $\text{getHeight()} - 1$. If the width and height of the applet are 300 pixels each, then we can't just use pixel coordinates for the data, or the graph would only occupy a tiny area of the window.

In such a situation, you need to define *user coordinates* that makes sense for your particular application and then transform them to pixel coordinates. Suppose the application coordinates range from x_{\min} to x_{\max} and y_{\min} to y_{\max} . Then you can transform user coordinates to pixel coordinates, by using the following equations:

$$x_{\text{pixel}} = (x_{\text{user}} - x_{\min}) \cdot (\text{width} - 1) / (x_{\max} - x_{\min})$$

$$y_{\text{pixel}} = (y_{\text{user}} - y_{\min}) \cdot (\text{height} - 1) / (y_{\max} - y_{\min})$$

To see that these equations make sense, plug in $x_{\text{user}} = x_{\min}$ and $x_{\text{user}} = x_{\max}$ and check that you get $x_{\text{pixel}} = 0$ and $x_{\text{pixel}} = \text{width} - 1$. Do the same for the y values.

Note that for the y -coordinates, the roles for y_{\min} and y_{\max} are reversed. In Java, y -coordinates increase when moving down, whereas in mathematics, they increase when moving up. That is, y_{\max} corresponds to pixel 0 and y_{\min} to pixel height - 1.

Let's apply this to our temperature chart. The month range is

$$x_{\min} = 1, x_{\max} = 12$$

Table 3

Average Temperatures in Phoenix,
Arizona

Month	Temperature
January	11
February	13
March	16
April	20
May	25
June	31
July	33
August	32
September	29
October	23
November	16
December	12

We'll choose a temperature range

$$y_{\min} = 0, y_{\max} = 50$$

We draw each data point as a line from (month, 0) to (month, temperature) in user coordinates. Here is how you compute the pixel coordinates.

```
final double XMIN = 1;
final double XMAX = 12;
final double YMIN = 0;
final double YMAX = 50;

double xpixel = (month - XMIN) * (getWidth() - 1) /
    (XMAX - XMIN);
double y1pixel = getHeight() - 1; // 0 in user coordinates
double y2pixel = (temperature - YMAX)
    * (getHeight() - 1) / (YMIN - YMAX);

Line2D.Double bar =
    new Line2D.Double(xpixel, y1pixel, xpixel, y2pixel);
```

This approach works, but it is tedious and can make your code hard to read. It is better to write a couple of simple helper methods for the coordinate transformation.

```

public class Phoenix extends Applet
{
    . . .
    public double xpixel(double xuser)
    {
        return
            (xuser - XMIN) * (getWidth() - 1) / (XMAX - XMIN);
    }

    public double ypixel(double yuser)
    {
        return
            (yuser - YMAX) * (getHeight() - 1) / (YMIN - YMAX);
    }

    private static final double XMIN = 1;
    private static final double XMAX = 12;
    private static final double YMIN = 0;
    private static final double YMAX = 50;
}

```

Now you can compute pixel coordinates conveniently:

```

Line2D.Double stick = new Line2D.Double(
    xpixel(month), ypixel(0),
    xpixel(month), ypixel(temperature));

```

Advanced Topic 4.4 shows another solution that is even more elegant.

Here is the complete program for drawing the temperature bar chart. Figure 18 shows the output.

File ChartApplet.java

```

1 import java.applet.Applet;
2 import java.awt.Graphics;
3 import java.awt.Graphics2D;
4 import java.awt.geom.Line2D;
5
6 /**
7  * This applet draws a chart of the average monthly
8  * temperatures in Phoenix, AZ.
9  */
10 public class ChartApplet extends Applet
11 {
12     public void paint(Graphics g)
13     {
14         Graphics2D g2 = (Graphics2D)g;
15
16         month = 1;

```

```
17     drawBar(g2, JAN_TEMP);
18     drawBar(g2, FEB_TEMP);
19     drawBar(g2, MAR_TEMP);
20     drawBar(g2, APR_TEMP);
21     drawBar(g2, MAY_TEMP);
22     drawBar(g2, JUN_TEMP);
23     drawBar(g2, JUL_TEMP);
24     drawBar(g2, AUG_TEMP);
25     drawBar(g2, SEP_TEMP);
26     drawBar(g2, OCT_TEMP);
27     drawBar(g2, NOV_TEMP);
28     drawBar(g2, DEC_TEMP);
29 }
30
31 /**
32  * Draws a bar for the current month and increments
33  * the month.
34  * @param g2 the graphics context
35  * @param temperature the temperature for the month
36  */
37 public void drawBar(Graphics2D g2, int temperature)
38 {
39     Line2D.Double bar
40         = new Line2D.Double(xpixel(month), ypixel(0),
41                             xpixel(month), ypixel(temperature));
42
43     g2.draw(bar);
44
45     month++;
46 }
47
48 /**
49  * Converts from user coordinates to pixel coordinates.
50  * @param xuser an x value in user coordinates
51  * @return the corresponding value in pixel coordinates
52  */
53 public double xpixel(double xuser)
54 {
55     return
56         (xuser - XMIN) * (getWidth() - 1) /
57         (XMAX - XMIN);
58 }
59
60 /**
61  * Converts from user coordinates to pixel coordinates.
62  * @param yuser a y-value in user coordinates
63  * @return the corresponding value in pixel coordinates
64  */
65 public double ypixel(double yuser)
66 {
```

```

67     return
68         (yuser - YMAX) * (getHeight() - 1) /
69         (YMIN - YMAX);
70     }
71
72     private static final int JAN_TEMP = 11;
73     private static final int FEB_TEMP = 13;
74     private static final int MAR_TEMP = 16;
75     private static final int APR_TEMP = 20;
76     private static final int MAY_TEMP = 25;
77     private static final int JUN_TEMP = 31;
78     private static final int JUL_TEMP = 33;
79     private static final int AUG_TEMP = 32;
80     private static final int SEP_TEMP = 29;
81     private static final int OCT_TEMP = 23;
82     private static final int NOV_TEMP = 16;
83     private static final int DEC_TEMP = 12;
84
85     private static final double XMIN = 1;
86     private static final double XMAX = 12;
87     private static final double YMIN = 0;
88     private static final double YMAX = 40;
89
90     private int month;
91 }

```

▼ Advanced Topic

4.4

▼ Let the Graphics Context Transform the Coordinates

▼ You can change the coordinate system of the graphics context. At the beginning of the paint method, insert the following two calls:

```

▼     double xscale = (getWidth() - 1.0) / (XMAX - XMIN);
▼     double yscale = (getHeight() - 1.0) / (YMIN - YMAX);
▼     g2.scale(xscale, yscale);
▼     g2.translate(-XMIN, -YMAX);
▼     g2.setStroke(new BasicStroke(0));

```

▼ Now the graphics context translates user coordinates to pixels, leaving the programmer to focus on more important issues. You simply draw objects in user coordinates, such as

```

▼     Line2D.Double rect =
▼         new Line2D.Double(month, 0, month, temperature);

```

▼ The call to `setStroke` is necessary to set the line thickness. The default drawing operation draws lines of width 1 in user coordinates, which would result in lines that are much

- ▼ too fat. A stroke with thickness zero is always drawn one pixel wide. Font size is also specified in user coordinates. If your drawing contains fonts, you need to divide the point size by `Math.max(xscale, -yscale)`.
 - ▼ The program after this note shows how the chart applet is implemented with graphics context transformations. Note that the `xpixel` and `ypixel` methods are no longer required.
-

File ChartApplet.java

```
1 import java.applet.Applet;
2 import java.awt.BasicStroke;
3 import java.awt.Graphics;
4 import java.awt.Graphics2D;
5 import java.awt.geom.Line2D;
6
7 /**
8  * This applet draws a chart of the average monthly
9  * temperatures in Phoenix, AZ.
10 */
11 public class ChartApplet extends Applet
12 {
13     public void paint(Graphics g)
14     {
15         Graphics2D g2 = (Graphics2D)g;
16
17         double xscale =
18             (getWidth() - 1.0) / (XMAX - XMIN);
19         double yscale =
20             (getHeight() - 1.0) / (YMIN - YMAX);
21         g2.scale(xscale, yscale);
22         g2.translate(-XMIN, -YMAX);
23         g2.setStroke(new BasicStroke(0));
24
25         month = 1;
26
27         drawBar(g2, JAN_TEMP);
28         drawBar(g2, FEB_TEMP);
29         drawBar(g2, MAR_TEMP);
30         drawBar(g2, APR_TEMP);
31         drawBar(g2, MAY_TEMP);
32         drawBar(g2, JUN_TEMP);
33         drawBar(g2, JUL_TEMP);
34         drawBar(g2, AUG_TEMP);
35         drawBar(g2, SEP_TEMP);
36         drawBar(g2, OCT_TEMP);
37         drawBar(g2, NOV_TEMP);
38         drawBar(g2, DEC_TEMP);
39     }
```

```
40    /**
41     * Draws a bar for the current month and increments
42     * the month.
43     * @param g2 the graphics context
44     * @param temperature the temperature for the month
45     */
46    public void drawBar(Graphics2D g2, int temperature)
47    {
48        Line2D.Double bar = new
49            Line2D.Double(month, 0, month, temperature);
50
51        g2.draw(bar);
52
53        month++;
54    }
55
56    private static final int JAN_TEMP = 11;
57    private static final int FEB_TEMP = 13;
58    private static final int MAR_TEMP = 16;
59    private static final int APR_TEMP = 20;
60    private static final int MAY_TEMP = 25;
61    private static final int JUN_TEMP = 31;
62    private static final int JUL_TEMP = 33;
63    private static final int AUG_TEMP = 32;
64    private static final int SEP_TEMP = 29;
65    private static final int OCT_TEMP = 23;
66    private static final int NOV_TEMP = 16;
67    private static final int DEC_TEMP = 12;
68
69    private static final double XMIN = 1;
70    private static final double XMAX = 12;
71    private static final double YMIN = 0;
72    private static final double YMAX = 40;
73
74    private int month;
75 }
```



Productivity Hint

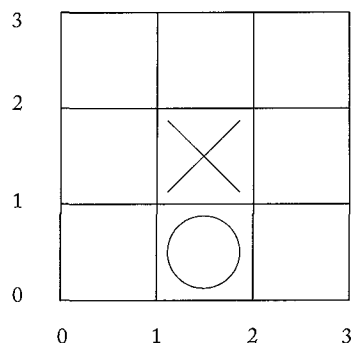
4.1

Choose Convenient Units for Drawing

- ▼ Whenever you deal with real-world data, you should use units that are matched to the data. Figure out which range of x - and y -coordinates is most appropriate for the program data. For example, suppose you want to display a tic-tac-toe board (see Figure 20) that is supposed to fill the entire applet.
- ▼

▼ **Figure 20**

▼ A Tic-Tac-Toe Board



▼ Of course, you could labor mightily and figure out where the lines are in relation to the default pixel coordinate system. Or you can simply set your own units with both x and y going from 0 to 3.

```

▼ g2.draw(new Line2D.Double(
    xpixel(0), ypixel(1), xpixel(3), ypixel(1)));
▼ g2.draw(new Line2D.Double(
    xpixel(0), ypixel(2), xpixel(3), ypixel(2)));
▼ g2.draw(new Line2D.Double(
    xpixel(2), ypixel(0), xpixel(1), ypixel(3)));
▼ g2.draw(new Line2D.Double(
    xpixel(2), ypixel(0), xpixel(2), ypixel(3)));

```

▼ Some people have horrible memories about coordinate transformations from their high-school geometry class and have taken a vow never to think about coordinates again for the remainder of their lives. If you are among them, you should reconsider. If pixel coordinates are a poor match for your drawing, you may end up spending a lot of time fussing with coordinates to get the drawing right. Pick the right coordinate system and use the standard conversion functions (or, even better, let the graphics context do them, as described in Advanced Topic 4.4). Then all the horrible algebra is done automatically for you, so you don't have to program it by hand.

CHAPTER SUMMARY

1. There are three types of Java programs: *console applications*, *applets*, and *graphics applications*.
2. Applets are programs that run inside a web browser.
3. The applet security mechanisms let you execute applets safely on your computer.
4. Web pages are written in HTML, using tags to format text and include images.
5. To run an applet, you need an HTML page with the `applet` tag.

6. You view applets with the applet viewer or a Java-enabled browser.
7. Your applet class needs to extend the `Applet` class.
8. You draw graphical shapes in an applet by placing the drawing code inside the `paint` method. The `paint` method is called whenever the applet needs to be refreshed.
9. The `Graphics2D` class stores the graphics state (such as the current color) and has methods to draw shapes. You need to cast the `Graphics` parameter of the `paint` method to `Graphics2D` to use these methods.
10. The `Rectangle2D.Double`, `Ellipse2D.Double`, and `Line2D.Double` classes describe graphical shapes.
11. When you set a new color in the graphics context, it is used for subsequent drawing operations.
12. The `drawString` method of the `Graphics2D` class draws a string, starting at its basepoint.
13. A font is described by its face name, style, and point size.
14. It is a good idea to make a class for each complex graphical shape.
15. To figure out how to draw a complex shape, make a sketch on graph paper.
16. An applet can obtain input by displaying a `JOptionPane` in its constructor.
17. You should calculate test cases by hand to double-check that your application computes the correct answer.
18. Pixel coordinates are not useful for real-world data sets. Pick convenient user coordinates and convert to pixels.

References

[1] For details about the XHTML standard, see <http://www.w3.org/MarkUp/>.

CLASSES, OBJECTS, AND METHODS INTRODUCED IN THIS CHAPTER

```
java.applet.Applet
    getHeight
    getWidth
    init
    paint
java.awt.BasicStroke
java.awt.Color
java.awt.Font
```

```
    getStringBounds
java.awt.Graphics
java.awt.Graphics2D
    draw
    drawString
    fill
    getFontRenderContext
    getParameter
    scale
    setColor
    setFont
    setStroke
    translate
java.awt.font.FontRenderContext
java.awt.geom.Ellipse2D.Double
java.awt.geom.Line2D.Double
    getX1
    getX2
    getY1
    getY2
    setLine
java.awt.geom.Point2D.Double
    getX
    getY
    setLocation
java.awt.geom.Rectangle2D.Double
java.awt.geom.RectangularShape
    getCenterX
    getCenterY
    getMaxX
    getMaxY
    getMinX
    getMinY
    getWidth
    getHeight
    setFrameFromDiagonal
java.lang.Float
    parseFloat
javax.swing.JOptionPane
    showInputDialog
```

REVIEW EXERCISES

Exercise R4.1. What is the difference between an applet and an application?

Exercise R4.2. What is the difference between a browser and the applet viewer?

Exercise R4.3. Why do you need an HTML page to run an applet?

Exercise R4.4. Who calls the paint method of an applet? When does the call to the paint method occur?

Exercise R4.5. Why does the parameter of the `paint` method have type `Graphics` and not `Graphics2D`?

Exercise R4.6. What is the purpose of a graphics context?

Exercise R4.7. How do you specify a text color?

Exercise R4.8. What is the difference between a font and a font face?

Exercise R4.9. What is the difference between a monospaced font and a proportionally spaced font?

Exercise R4.10. What are serifs?

Exercise R4.11. What is a logical font?

Exercise R4.12. How do you determine the pixel dimensions of a string in a particular font?

Exercise R4.13. Which classes are used in this chapter for drawing graphical shapes?

Exercise R4.14. What are the three different classes for specifying rectangles in the Java library?

Exercise R4.15. You want to plot a bar chart showing the grade distribution of all students in your class (where A = 4.0, F = 0). What coordinate system would you choose to make the plotting as simple as possible?

Exercise R4.16. Let e be any ellipse. Write Java code to plot the ellipse e and another ellipse of the same size that touches e . *Hint:* You need to look up the accessors that tell you the dimensions of an ellipse.

Exercise R4.17. Write Java instructions to display the letters X and T in a graphics window, by plotting line segments.

Exercise R4.18. Introduce an error in the program `Intersect.java`, by computing `double root = Math.sqrt(r * r + (x - a) * (x - a));`. Run the program. What happens to the intersection points?

Exercise R4.19. Suppose you run the `Intersect` program and give a value of 30 for the x -position of the vertical line. Without actually running the program, determine what values you will obtain for the intersection points.

PROGRAMMING EXERCISES

Exercise P4.1. Write a graphics program that draws your name in red, centered inside a blue rectangle.

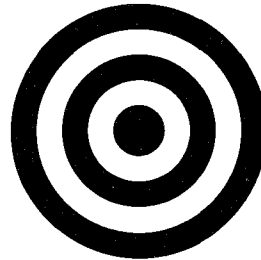
Exercise P4.2. Write a graphics program that draws your name four times, in a large serif font, in plain, bold, italic, and bold italic. The names should be stacked on top of each other, with equal distance between them. Each of them should be centered horizontally, and the entire stack should be centered vertically.

Exercise P4.3. Write a graphics program that draws twelve strings, one each for the 12 standard colors besides `Color.white`, each in its own color.

Exercise P4.4. Write a graphics program that prompts the user to enter a radius. Draw a circle with that radius.

Exercise P4.5. Write a program that draws two solid circles: one in pink and one in purple. Use a standard color for one of them and a custom color for the other.

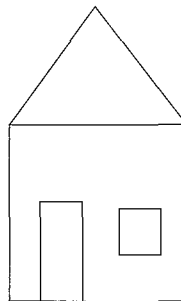
Exercise P4.6. Draw a “bull’s eye”—a set of concentric rings in alternating black and white colors. *Hint:* Fill a black circle, then fill a smaller white circle on top, and so on.



Exercise P4.7. Write a program that fills the applet window with a large ellipse, filled with your favorite color, that touches the window boundaries. The ellipse should resize itself when you resize the window.

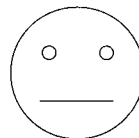
Exercise P4.8. Write a program that draws the picture of a house. It could be as simple as the accompanying figure, or if you like, make it more elaborate (3-D, skyscraper, marble columns in the entryway, whatever).

Implement a class `House` and supply a method `draw(Graphics2D g2)` that draws the house.



Exercise P4.9. Extend Exercise 4.8 by allowing the user to specify houses of different sizes in the `House` constructor. Then populate your screen with a few houses of different sizes.

Exercise P4.10. Write a program to plot the following face.



Exercise P4.11. Write a program to plot the string “HELLO”, using just lines and circles. Do not call `drawString`, and do not use `System.out`. Make classes `LetterH`, `LetterE`, `LetterL` and `LetterO`.

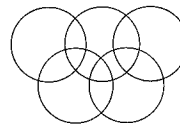
Exercise P4.12. *Plotting a data set.* Make a bar chart to plot the following data set:

Bridge Name	Longest Span (ft)
Golden Gate	4,200
Brooklyn	1,595
Delaware Memorial	2,150
Mackinac	3,800

Make the bars horizontal for easier labeling. *Hint:* Set the window coordinates to 5,000 in the x -direction and 4 in the y -direction.

Exercise P4.13. Write a graphics program that displays the values of Exercise 4.11 as a *pie chart*. Just draw a circle and the edges of the pie slices. You don't have to color the slices. (If you want to color them, look at the online API documentation for the `Arc2D` class.)

Exercise P4.14. Write a program that displays the Olympic rings.



Color the rings in the Olympic colors.

Exercise P4.15. Write a graphics program that draws a clock face with a time that the user enters in a text field. (The user must enter the time in the format `hh:mm`, for example `09:45`.)

Hint: You need to find out the angles of the hour hand and the minute hand. The angle of the minute hand is easy: The minute hand travels 360 degrees in 60 minutes. The angle of the hour hand is harder; it travels 360 degrees in 12×60 minutes.

Design a class `Clock` and supply a method `draw(Graphics2D g2)` that draws the clock.

Exercise P4.16. Change the `CarApplet` program to make the cars appear twice the size of the original example.

Exercise P4.17. Change the `CarApplet` program to make the cars appear in different colors. Each `Car` object should store its own color.

Exercise P4.18. Design a class `Truck` whose constructor takes the top left corner point of the truck. Supply a method `draw(Graphics2D g2)` that draws the truck. Then populate your screen with a few cars and trucks.