

3

Introduction to Java Applets

Objectives

- To observe some of Java's exciting capabilities through the Java 2 Software Development Kit's demonstration applets.
- To differentiate between applets and applications.
- To be able to write simple Java applets.
- To be able to write simple Hypertext Markup Language (HTML) files to load an applet into the **appletviewer** or a World Wide Web browser.
- To understand the difference between variables and references.
- To execute applets in World Wide Web browsers.

*He would answer to "Hi!" or to any loud cry
Such as "Fry me!" or "Fritter my wig!"
To "What-you-may-call-um!" or "What-was-his-name!"
But especially "Thing-um-a-jig!"*

Lewis Carroll

*Painting is only a bridge linking the painter's mind with that
of the viewer.*

Eugène Delacroix

*My method is to take the utmost trouble to find the right thing
to say, and then to say it with the utmost levity.*

George Bernard Shaw

Though this be madness, yet there is method in 't.
William Shakespeare



Outline

- 3.1 Introduction
- 3.2 Sample Applets from the Java 2 Software Development Kit
 - 3.2.1 The TicTacToe Applet
 - 3.2.2 The DrawTest Applet
 - 3.2.3 The Java2D Applet
- 3.3 A Simple Java Applet: Drawing a String
 - 3.3.1 Compiling and Executing `WelcomeApplet`
- 3.4 Two More Simple Applets: Drawing Strings and Lines
- 3.5 Another Java Applet: Adding Floating-Point Numbers
- 3.6 Viewing Applets in a Web Browser
 - 3.6.1 Viewing Applets in Netscape Navigator 6
 - 3.6.2 Viewing Applets in Other Browsers Using the Java Plug-In
- 3.7 Java Applet Internet and World Wide Web Resources
- 3.8 (Optional Case Study) Thinking About Objects: Identifying the Classes in a Problem Statement

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

3.1 Introduction

In Chapter 2, we introduced Java application programming and several important aspects of Java applications. This chapter introduces another type of Java program called a Java applet. Applets are Java programs that can be embedded in *Hypertext Markup Language (HTML) documents* (i.e., Web pages). When a browser loads a Web page containing an applet, the applet downloads into the Web browser and begins execution.

The browser that executes an applet is generically known as the *applet container*. The Java 2 Software Development Kit (J2SDK) includes an applet container (called the **appletviewer**) for testing applets before you embed them in a Web page. Most Web browsers in use today do not support Java 2 directly. For this reason, we normally demonstrate our applets using the **appletviewer**. One browser that does support Java 2 is Netscape Navigator 6. To execute applets in other Web browsers such as Microsoft Internet Explorer or earlier versions of Netscape Navigator requires the *Java Plug-in*, which we discuss in Section 3.6.2 of this chapter.



Portability Tip 3.1

Most Web browsers in use today do not support applets written in Java 2. To execute applets in such browsers, you must use the Java Plug-in (see Section 3.6.2).



Testing and Debugging Tip 3.1

Test your applets in the **appletviewer** applet container before executing them in a Web browser. This enables you to see error messages that may occur. Also, once an applet is executing in a browser, it is sometimes difficult to reload the applet after making changes to the applet's class definition.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/2/01



Testing and Debugging Tip 3.2

Test your applets in every Web browser in which the applets will execute to ensure that they operate correctly in each browser.

One of our goals in this chapter is to mimic several features presented in Chapter 2. This provides positive reinforcement of previous concepts. Another goal of this chapter is to begin using the object-oriented programming terminology introduced in Section 1.15.

As in Chapter 2, there are a few cases where we do not as yet provide all the details necessary to create complex applications and applets in Java. It is important to build your knowledge of fundamental programming concepts first. In Chapter 4 and Chapter 5, we present a detailed treatment of *program development* and *program control* in Java. As we proceed through the text, we present many substantial applications and applets.

3.2 Sample Applets from the Java 2 Software Development Kit

We begin by considering several sample applets provided with the Java 2 Software Development Kit (J2SDK) version 1.3. The applets we demonstrate give you a sense of Java's capabilities. Each of the sample programs provided with the J2SDK also comes with *source code* (the **.java** files containing the Java applet programs). This source code is helpful as you enhance your Java knowledge—you can read the source code provided to learn new and exciting features of Java. Remember, all programmers initially learn new features by mimicking their use in existing programs. The J2SDK comes with many such programs and there are a tremendous number of Java resources on the Internet and World Wide Web that include Java source code.

The demonstration programs provided with the J2SDK are located in your J2SDK install directory in a subdirectory called **demo**. For the Java 2 Software Development Kit version 1.3, the default location of the **demo** directory on Windows is

```
c:\jdk1.3\demo
```

On UNIX/Linux it is the directory in which you install the J2SDK followed by **jdk1.3/demo**—for example

```
/usr/local/jdk1.3/demo
```

For other platforms, there will be a similar directory (or folder) structure. For the purpose of this chapter, we assume on Windows that the J2SDK is installed in **c:\jdk1.3** and on UNIX that the J2SDK is installed in your home directory in **~/jdk1.3**. [Note: You may need to update these locations to reflect your chosen install directory and/or disk drive, or a newer version of the J2SDK.]

If you are using a Java Development tool that does not come with the Sun Java demos, you can download the J2SDK (with the demos) from the Sun Microsystems Java Web site

```
java.sun.com/j2se/1.3/
```

3.2.1 The TicTacToe Applet

The first applet we demonstrate from the J2SDK demos is the **TicTacToe** applet, which allows you to play Tic-Tac-Toe against the computer. To execute this applet, open a com-

mand window (*MS-DOS Prompt* on Windows 95/98/ME, *Command Prompt* on Windows NT/2000 or a *command tool/shell tool* on UNIX) and change directories to the J2SDK's **demo** directory. Both Windows and UNIX use command **cd** to *change directories*. For example, the command

```
cd c:\jdk1.3\demo
```

changes to the **demo** directory on Windows and the command

```
cd ~/jdk1.3/demo
```

changes to the **demo** directory on UNIX.

The **demo** directory contains four subdirectories—**applets**, **jfc**, **jpa** and **sound** (you can see these directories by issuing in the command window the **dir** command on Windows or the **ls** command on UNIX). The **applets** directory contains many demonstration applets. The **jfc** (Java Foundation Classes) directory contains many examples of Java's newest graphics and GUI features (some of these examples are also applets). The **jpa** directory contains examples of the *Java Platform Debugging Architecture* (beyond the scope of this book). The sound directory contains examples of the *Java Sound API* (covered in Chapter 18). For the demonstrations in this section, change directories to the **applets** directory by issuing the command

```
cd applets
```

on either Windows or UNIX.

Listing the contents of the **applets** directory (with the **dir** command on Windows or the **ls** command on UNIX) indicates that there are many examples. Figure 3.1 shows the subdirectories and provides a brief description of the examples in each subdirectory.

Example	Description
Animator	Performs one of four separate animations.
ArcTest	Demonstrates drawing arcs. You can interact with the applet to change attributes of the arc that is displayed.
BarChart	Draws a simple bar chart.
Blink	Displays blinking text in different colors.
CardTest	Demonstrates several GUI components and a variety of ways in which GUI components can be arranged on the screen (the arrangement of GUI components is also known as the <i>layout</i> of the GUI components).
Clock	Draws a clock with rotating “hands,” the current date and the current time. The clock is updated once per second.
DitherTest	Demonstrates drawing with a graphics technique known as dithering that allows gradual transformation from one color to another.

Fig. 3.1 The examples from the **applets** directory (part 1 of 2).

Example	Description
DrawTest	Allows the user to drag the mouse to draw lines and points on the applet in different colors.
Fractal	Draws a fractal. Fractals typically require complex calculations to determine how they are displayed.
GraphicsTest	Draws a variety of shapes to illustrate graphics capabilities.
GraphLayout	Draws a graph consisting of many nodes (represented as rectangles) connected by lines. Drag a node to see the other nodes in the graph adjust on the screen and demonstrate complex graphical interactions.
ImageMap	Demonstrates an image with <i>hot spots</i> . Positioning the mouse pointer over certain areas of the image highlights the area and a message is displayed in the lower-left corner of the appletviewer window. Position over the mouth in the image to hear the applet say “hi.”
JumpingBox	Moves a rectangle randomly around the screen. Try to catch it by clicking it with the mouse!
MoleculeViewer	Presents a three-dimensional view of several different chemical molecules. Drag the mouse to view the molecule from different angles.
NervousText	Draws text that jumps around the screen.
SimpleGraph	Draws a complex curve.
SortDemo	Compares three sorting techniques. Sorting (described in Chapter 7) arranges information in order—like alphabetizing words. When you execute the applet, three appletviewer windows appear. Click in each one to start the sort. Notice that the sorts all operate at different speeds.
SpreadSheet	Demonstrates a simple spreadsheet of rows and columns.
SymbolTest	Draws characters from the Java character set.
TicTacToe	Allows the user to play Tic-Tac-Toe against the computer.
WireFrame	Draws a three-dimensional shape as a wire frame. Drag the mouse to view the shape from different angles.

Fig. 3.1 The examples from the **applets** directory (part 2 of 2).

Change directories to subdirectory **TicTacToe**. In that directory you will find the HTML file **example1.html** that is used to execute the applet. In the command window, type the command

```
appletviewer example1.html
```

and press the *Enter* key. This executes the **appletviewer**. The **appletviewer** loads the HTML file specified as its *command-line argument* (**example1.html**), determines from the file which applet to load (we discuss the details of HTML files in Section 3.3) and begins executing the applet. Figure 3.2 shows several screen captures of playing Tic-Tac-Toe with this applet.

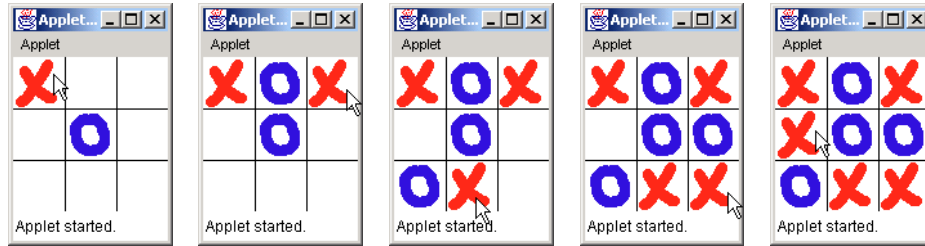


Fig. 3.2 Sample execution of the **TicTacToe** applet.



Testing and Debugging Tip 3.3

If the **appletviewer** command does not work and/or the system indicates that the **appletviewer** command cannot be found, the **PATH** environment variable may not be defined properly on your computer. Review the installation directions for the Java 2 Software Development Kit to ensure that the **PATH** environment variable is defined correctly for your system (on some computers, you may need to restart your computer after modifying the **PATH** environment variable).

You are player **X**. To interact with the applet, point the mouse at the square where you want to place an **X** and click the mouse button (normally, the left mouse button). The applet plays a sound (assuming your computer supports audio playback) and places an **X** in the square if the square is open. If the square is occupied, this is an invalid move and the applet plays a different sound indicating that you cannot make the specified move. After you make a valid move, the applet responds by making its own move (this happens quickly).

To play again, re-execute the applet by clicking the **appletviewer**'s **Applet** menu and selecting the **Reload** menu item (Fig. 3.3). To terminate the **appletviewer**, click the **appletviewer**'s **Applet** menu and select the **Quit** menu item.

Reload the applet to execute it again.

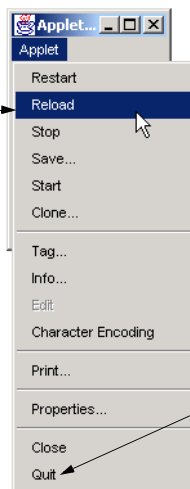


Fig. 3.3 Selecting **Reload** from the **appletviewer**'s **Applet** menu.

3.2.2 The DrawTest Applet

The next applet we demonstrate allows you to draw lines and points in different colors. To draw, you simply drag the mouse on the applet by pressing a mouse button and holding it while you drag the mouse. For this example, change directories to directory **applets**, then to subdirectory **DrawTest**. In that directory is the **example1.html** file that is used to execute the applet. In the command window, type the command

```
appletviewer example1.html
```

and press the *Enter* key. This executes the **appletviewer**. The **appletviewer** loads the HTML file specified as its command-line argument (**example1.html** again), determines from the file which applet to load and begins execution of the applet. Figure 3.4 shows a screen capture of this applet after drawing some lines and points.

The default shape to draw is a line and the default color is black, so you can draw black lines by dragging the mouse across the applet. To drag the mouse, press and hold the mouse button and move the mouse. Notice that the line follows the mouse pointer around the applet. The line is not permanent until you release the mouse button. You can then start a new line by repeating the process.

Select a color by clicking the circle inside one of the colored rectangles at the bottom of the applet. You can select from red, green, blue, pink, orange and black. The GUI components used to present these options are commonly known as *radio buttons*. If you think of a car radio, only one radio station can be selected at a time. Similarly, only one drawing color can be selected at a time.

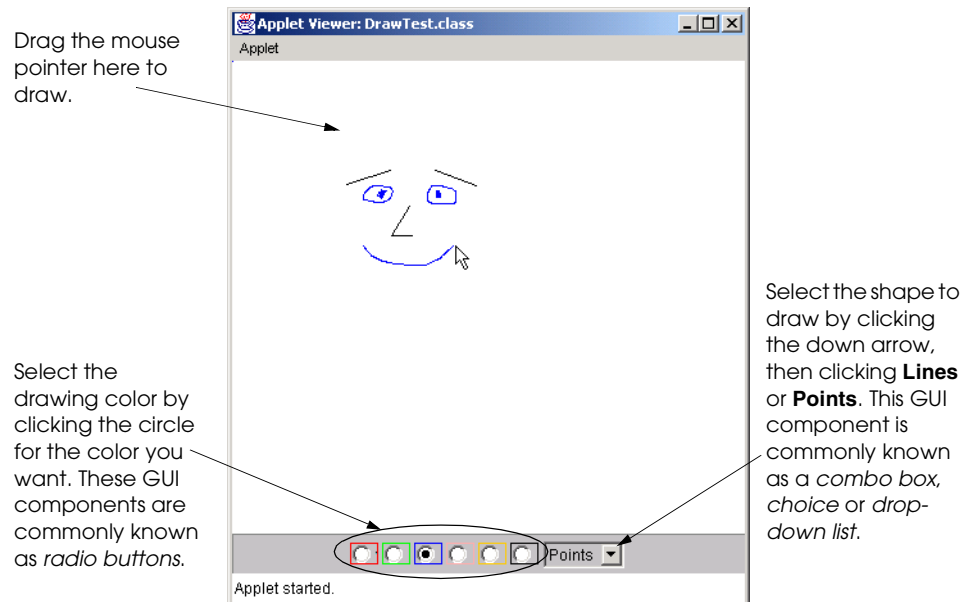


Fig. 3.4 Sample execution of applet **DrawTest**.

Try changing the shape to draw from **Lines** to **Points** by clicking the down arrow to the right of the word **Lines** at the bottom of the applet. A list drops down from the GUI component containing the two choices—**Lines** and **Points**. To select **Points**, click the word **Points** in the list. The GUI component closes the list and the current shape type is now **Points**. This GUI component is commonly known as a *choice*, *combo box* or *drop-down list*.

To start a new drawing, select **Reload** from the **appletviewer**'s **Applet** menu. To terminate the applet, select **Quit** from the **appletviewer**'s **Applet** menu.

3.2.3 The Java2D Applet

The last applet we demonstrate before defining applets of our own shows many of the complex new two-dimensional drawing capabilities built into Java 2—known as the *Java2D API*. For this example, change directories to the **jfc** directory in the J2SDK's **demo** directory, then change to the **Java2D** directory (you can move up the directory tree toward **demo** using the command "**cd ..**" in both Windows and UNIX/Linux). In that directory is an HTML file (**Java2Demo.html**) that is used to execute the applet. In the command window, type the command

```
appletviewer Java2Demo.html
```

and press the *Enter* key. This executes the **appletviewer**. The **appletviewer** loads the HTML file specified as its command-line argument (**Java2Demo.html**), determines from the file which applet to load and begins execution of the applet. This particular demo takes some time to load as it is quite large. Figure 3.5 shows a screen capture of one of this applet's many demonstrations of Java's two-dimensional graphics capabilities.

At the top of this demo you see tabs that look like file folders in a filing cabinet. This demo provides 12 different tabs with several different features on each tab. To change to a different part of the demo, simply click one of the tabs. Also, try changing the options in the upper-right corner of the applet. Some of these affect the speed with which the applet draws the graphics. For example, click the small box with a check in it (a GUI component known as a *checkbox*) to the left of the word **Anti-Aliasing** to turn off anti-aliasing (a graphics technique for producing smoother on-screen graphics in which the edges of the graphic are blurred). When this feature is turned off (i.e., its *checkbox* is unchecked), the animation speed increases for the animated shapes at the bottom of the demo shown in Fig. 3.5. This occurs because an animated shape displayed with anti-aliasing takes longer to draw than an animated shape without anti-aliasing.

3.3 A Simple Java Applet: Drawing a String

Now, let's get started with some applets of our own. Remember, we are just getting started—we have many more topics to learn before we can write applets similar to those demonstrated in Section 3.2. However, we will cover many of the same techniques in this book.

We begin by considering a simple applet that mimics the program of Fig. 2.1 by displaying the string **"Welcome to Java Programming!"**. The applet and its screen output are shown in Fig. 3.6. The HTML document to load the applet into the **appletviewer** is shown and discussed in Fig. 3.7.

Click a tab to select a two-dimensional graphics demo.

Try changing the options to see their effect on the demonstration.

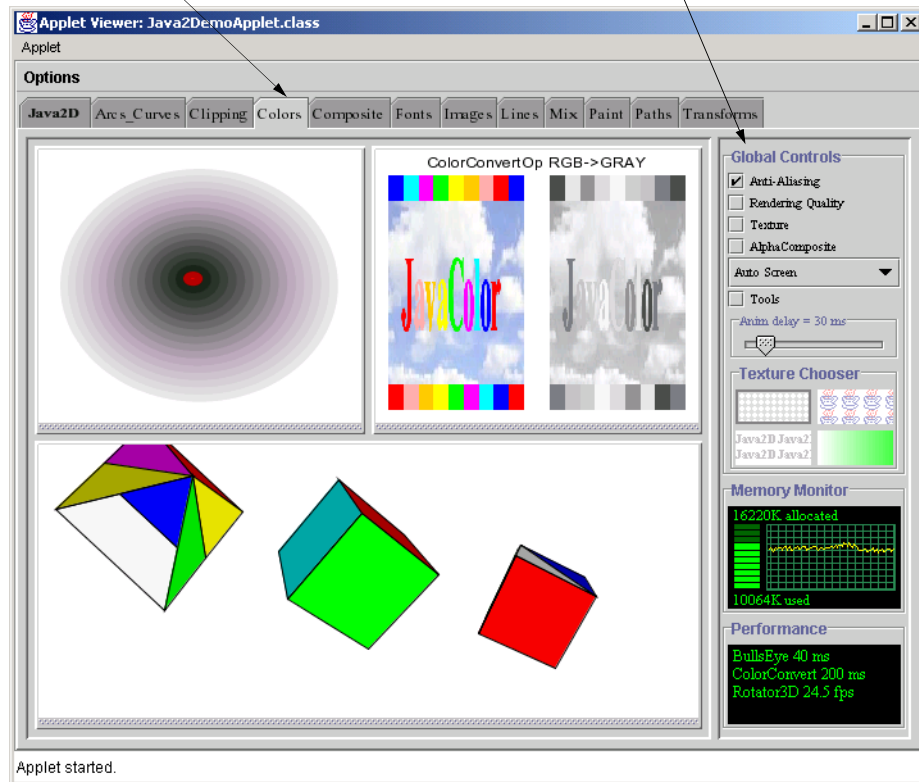


Fig. 3.5 Sample execution of applet **Java2D**.

```

1  // Fig. 3.6: WelcomeApplet.java
2  // A first applet in Java.
3
4  // Java core packages
5  import java.awt.Graphics;    // import class Graphics
6
7  // Java extension packages
8  import javax.swing.JApplet; // import class JApplet
9
10 public class WelcomeApplet extends JApplet {
11
12     // draw text on applet's background
13     public void paint( Graphics g )
14     {
15         // call inherited version of method paint
16         super.paint( g );

```

Fig. 3.6 A first applet in Java and the applet's screen output (part 1 of 2).

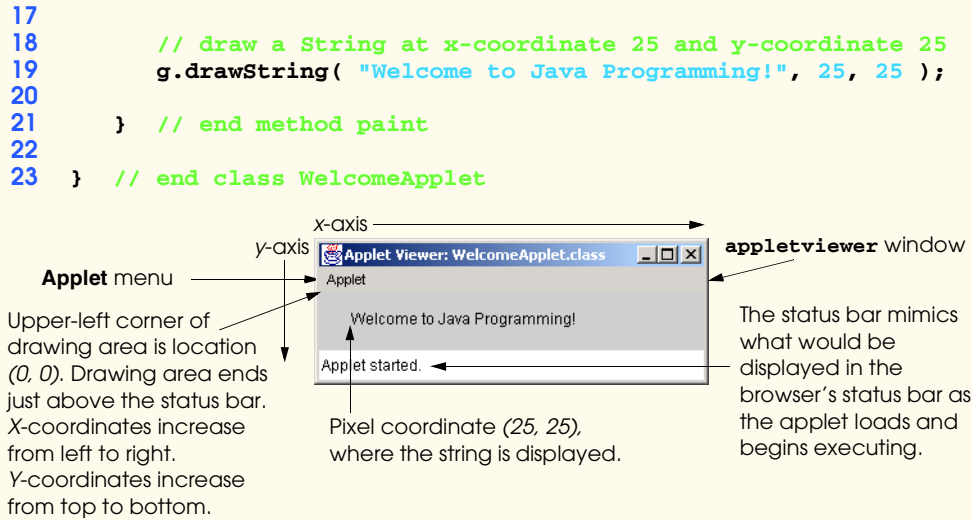


Fig. 3.6 A first applet in Java and the applet's screen output (part 2 of 2).

This program illustrates several important Java features. We consider each line of the program in detail. Line 19 does the “real work” of the program, namely drawing the string **Welcome to Java Programming!** on the screen. But let us consider each line of the program in order. Lines 1–2

```

// Fig. 3.6: WelcomeApplet.java
// A first applet in Java.

```

begin with `//`, indicating that the remainder of each line is a comment. The comment on line 1 indicates the figure number and file name for the applet source code. The comment on line 2 simply describes the purpose of the program.

As stated in Chapter 2, Java contains many predefined components called classes that are grouped into packages in the Java API. Line 5

```
import java.awt.Graphics; // import class Graphics

```

is an **import** statement that tells the compiler load class **Graphics** from package **java.awt** for use in this Java applet. Class **Graphics** enables a Java applet to draw graphics such as lines, rectangles, ovals and strings of characters. Later in the book, you will see that class **Graphics** also enables applications to draw.

Line 8

```
import javax.swing.JApplet; // import class JApplet

```

is an **import** statement that tells the compiler load class **JApplet** from package **javax.swing**. When you create an applet in Java, you normally import the **JApplet** class. [Note: There is an older class called **Applet** from package **java.applet** that is not used with Java's newest GUI components from the **javax.swing** package. In this book, we use only class **JApplet** with applets.]

As with applications, every Java applet you create contains at least one class definition. One key feature of class definitions that was not mentioned in Chapter 2 is that programmers rarely create class definitions “from scratch.” In fact, when you create a class definition, you normally use pieces of an existing class definition. Java uses *inheritance* (introduced in Section 1.15 and discussed in detail in Chapter 9, “Object-Oriented Programming”) to create new classes from existing class definitions. Line 10

```
public class WelcomeApplet extends JApplet {
```

begins a **class** definition for class **WelcomeApplet**. At the end of line 10, the left brace, **{**, begins the body of the class definition. The corresponding right brace, **}**, on line 23 ends the class definition. Keyword **class** introduces the class definition. **WelcomeApplet** is the class name. Keyword **extends** indicates that class **WelcomeApplet** inherits existing pieces from another class. The class from which **WelcomeApplet** inherits (**JApplet**) appears to the right of **extends**. In this inheritance relationship, **JApplet** is called the *superclass* or *base class* and **WelcomeApplet** is called the *subclass* or *derived class*. Using inheritance here results in a **WelcomeApplet** class definition that has the *attributes* (data) and *behaviors* (methods) of class **JApplet** as well as the new features we are adding in our **WelcomeApplet** class definition (specifically, the ability to draw **Welcome to Java Programming!** on the applet).

A key benefit of extending class **JApplet** is that someone else previously defined “what it means to be an applet.” The **appletviewer** and World Wide Web browsers that support applets expect every Java applet to have certain capabilities (attributes and behaviors). Class **JApplet** already provides all those capabilities—programmers do not need to “reinvent the wheel” and define all these capabilities on their own. In fact, applet containers expect applets to have over 200 different methods. In our programs to this point, we defined one method in each program. If we had to define over 200 methods just to display **Welcome to Java Programming!**, we would never create an applet, because it would take too long to define one! Using **extends** to inherit from class **JApplet** enables applet programmers to create new applets quickly.

The inheritance mechanism is easy to use; the programmer does not need to know every detail of class **JApplet** or any other superclass from which a new class inherits. The programmer needs to know only that class **JApplet** defines the capabilities required to create the minimum applet. However, to make the best use of any class, programmers should study all the capabilities of the superclass.



Good Programming Practice 3.1

Investigate the capabilities of any class in the Java API documentation (java.sun.com/j2se/1.3/docs/api/index.html) carefully before inheriting a subclass from it. This helps ensure that the programmer does not unintentionally “reinvent the wheel” by redefining a capability that the superclass already provides.

Classes are used as “templates” or “blueprints” to *instantiate* (or *create*) *objects* for use in a program. An object (or *instance*) resides in the computer’s memory and contains information used by the program. The term object normally implies that attributes (data) and behaviors (methods) are associated with the object. The object’s methods use the attributes to provide useful services to the *client of the object* (i.e., the code in a program that calls the methods).

When an applet container (the **appletviewer** or browser in which the applet executes) loads our **WelcomeApplet** class, the applet container creates an object (instance) of class **WelcomeApplet** that implements the applet's attributes and behaviors. [Note: The terms instance and object are often used interchangeably.] Applet containers can create only objects of classes that are **public** and extend **JApplet**. Thus, applet containers require applet class definitions to begin with the keyword **public** (line 10). Otherwise, the applet container cannot load and execute the applet. The **public** keyword and related keywords (such as **private** and **protected**) are discussed in detail in Chapter 8, "Object-Based Programming." For now, we ask you simply to start all class definitions with the **public** keyword until the discussion of **public** in Chapter 8.

When you save a **public** class in a file, the file name must be the class name followed by the **.java** file name extension. For our applet, the file name must be **WelcomeApplet.java**. Please note that the class name part of the file name must use the same spelling as the class name, including identical use of uppercase and lowercase letters. For reinforcement, we repeat two Common Programming Errors from Chapter 2.



Common Programming Error 3.1

*It is an error for a **public** class if the file name is not identical to the class name (plus the **.java** extension) in both spelling and capitalization. Therefore, it is also an error for a file to contain two or more **public** classes.*



Common Programming Error 3.2

*It is an error not to end a file name with the **.java** extension for a file containing an application's class definition. The Java compiler will not be able to compile the class definition.*



Testing and Debugging Tip 3.4

*The compiler error message "Public class ClassName must be defined in a file called ClassName.java" indicates either that the file name does not exactly match the name of the **public** class in the file (including all uppercase and lowercase letters) or that you typed the class name incorrectly when compiling the class (the name must be spelled with the proper uppercase and lowercase letters).*

Line 13

```
public void paint( Graphics g )
```

begins the definition of the applet's **paint method**—one of three methods (behaviors) that the applet container calls for an applet when the container begins executing the applet. In order, these three methods are **init** (discussed later in this chapter), **start** (discussed in Chapter 6) and **paint**. Your applet class gets a "free" version of each of these methods from class **JApplet** when you specify **extends JApplet** in the first line of your applet's class definition. If you do not define these methods in your own applet, the applet container calls the versions inherited from **JApplet**. The inherited versions of methods **init** and **start** have empty bodies (i.e., their bodies do not contain statements, so they do not perform a task) and the inherited version of method **paint** does not draw anything on the applet. [Note: There are several other methods that an applet container calls during an applet's execution. We discuss these methods in Chapter 6, "Methods."]

To enable our applet to draw, class **WelcomeApplet** *overrides* (replaces or redefines) the default version of **paint** by placing statements in the body of **paint** that draw

a message on the screen. When the applet container tells the applet to “draw itself” on the screen by calling method **paint**, our message **Welcome to Java Programming!** appears rather than a blank screen.

Lines 13–21 are the definition of **paint**. The task of method **paint** is to draw graphics (such as lines, ovals and strings of characters) on the screen. Keyword **void** indicates that this method does not return any results when it completes its task. The set of parentheses after **paint** defines the method’s *parameter list*. The parameter list is where methods receive data required to perform their tasks. Normally, this data is passed by the programmer to the method through a *method call* (also known as *invoking a method* or *sending a message*). For example, in Chapter 2 we passed data to **JOptionPane**’s **showMessageDialog** method such as the message to display or the type of dialog box. However, when writing applets, the programmer does not call method **paint** explicitly. Rather, the applet container calls **paint** to tell the applet to draw and the applet container passes to the **paint** method the information **paint** requires to perform its task, namely a **Graphics** object (called **g**). It is the applet container’s responsibility to create the **Graphics** object to which **g** refers. Method **paint** uses the **Graphics** object to draw graphics on the applet. The **public** keyword at the beginning of line 13 is required so the applet container can call your **paint** method. For now, all method definitions should begin with the **public** keyword. We introduce other alternatives in Chapter 8.

The left brace, **{**, on line 14 begins method **paint**’s body. The corresponding right brace, **}**, on line 21 ends **paint**’s body.

Line 16

```
super.paint( g );
```

calls the version of method **paint** inherited from superclass **JApplet**.¹

Line 19

```
g.drawString( "Welcome to Java Programming!", 25, 25 );
```

instructs the computer to perform an action (or task), namely to draw the characters of the string **Welcome to Java Programming!** on the applet. This statement uses method **drawString** defined by class **Graphics** (this class defines all the drawing capabilities of a Java program, including strings of characters and shapes such as rectangles, ovals and lines). The statement calls method **drawString** using the **Graphics** object **g** (in **paint**’s parameter list) followed by a *dot operator* (**.**) followed by the method name **drawString**. The method name is followed by a set of parentheses containing the argument list **drawString** needs to perform its task.

The first argument to **drawString** is the **String** to draw on the applet. The last two arguments in the list—**25** and **25**—are the *x-y coordinates* (or *position*) at which the bottom-left corner of the string should be drawn on the applet. Drawing methods from class **Graphics** require coordinates to specify where to draw on the applet (later in the text we

1. For reasons that will become clear later in the text, this statement should be the first statement in every applet’s **paint** method. Although the early examples of applets will work without this statement, omitting this statement causes subtle errors in more elaborate applets that combine drawing and GUI components. Including this statement now will get you in the habit of using it and will save time and effort as you build more substantial applets later.

demonstrate drawing in applications). The first coordinate is the *x-coordinate* (the number of pixels from the left side of the applet), and the second coordinate is the *y-coordinate* (representing the number of pixels from the top of the applet). Coordinates are measured from the upper-left corner of the applet in *pixels* (just below the **Applet** menu in the sample output window of Fig. 3.6). A pixel (“picture element”) is the unit of display for your computer’s screen. On a color screen, a pixel appears as one colored dot on the screen. Many personal computers have 800 pixels for the width of the screen and 600 pixels for the height of the screen, for a total of 800 times 600 or 480,000 displayable pixels. Many computers today have screens with higher screen resolutions, i.e., they have more pixels for the width and height of the screen. The size of an applet on the screen depends on the size and resolution of the screen. For screens with the same size, the applet will appear smaller on the screen with the higher resolution. Note that some older computers have screen resolutions lower than 800 by 600. The most common lower resolution is 640 by 480.

When line 19 executes, it draws the message **Welcome to Java Programming!** on the applet at the coordinates **25** and **25**. Note that the quotation marks enclosing the string are *not* displayed on the screen.

As an aside, why would you want free copies of methods **init**, **start** and **paint** if they do not perform a task? The predefined start-up sequence of method calls made by the **appletviewer** or browser for every applet is always **init**, **start** and **paint**—this provides an applet programmer a guarantee that these methods will be called as every applet begins execution. Every applet does not need all three of these methods. However, the **appletviewer** or browser expects each of these methods to be defined so it can provide a consistent start-up sequence for an applet. [Note: This is similar to applications always starting execution with **main**.] Inheriting the default versions of these methods guarantees the browser that it can treat each applet uniformly by calling **init**, **start** and **paint** as applet execution begins. Also, the programmer can concentrate on defining only the methods required for a particular applet.

3.3.1 Compiling and Executing WelcomeApplet

As with application classes, you must compile applet classes before they can execute. After defining class **WelcomeApplet** and saving it in **WelcomeApplet.java**, open a command window, change to the directory in which you saved the applet class definition and type the command

```
javac WelcomeApplet.java
```

to compile class **WelcomeApplet**. If there are no syntax errors, the resulting bytecodes are stored in the file **WelcomeApplet.class**.

Before you can execute the applet you must create an *HTML (Hypertext Markup Language)* file to load the applet into the applet container (**appletviewer** or a browser). Typically, an HTML file ends with the “**.html**” or “**.htm**” file name extension. Browsers display the contents of documents that contain text (also known as *text files*). To execute a Java applet, an HTML text file must indicate which applet the applet container should load and execute. Figure 3.7 shows a simple HTML file—**WelcomeApplet.html**—that loads into the applet defined in Fig. 3.6 into an applet container. [Note: For the early part of this book, we always demonstrate applets with the **appletviewer** applet container.]

```
1 <html>
2 <applet code = "WelcomeApplet.class" width = "300" height = "45">
3 </applet>
4 </html>
```

Fig. 3.7 **WelcomeApplet.html** loads class **WelcomeApplet** of Fig. 3.6 into the **appletviewer**.



Good Programming Practice 3.2

Always test a Java applet in the **appletviewer** and ensure that it is executing correctly before loading the applet into a World Wide Web browser. Browsers often save a copy of an applet in memory until the current browsing session terminates (i.e., all browser windows are closed). Thus, if you change an applet, recompile the applet, then reload the applet in the browser, you may not see the changes because the browser may still be executing the original version of the applet. Close all your browser windows to remove the old version of the applet from memory. Open a new browser window and load the applet to see your changes.



Software Engineering Observation 3.1

If your World Wide Web browser does not support Java 2, most of the applets in this book will not execute in your browser. This is because most of the applets in this book use features that are specific to Java 2 or are not provided by browsers that support Java 1.1. Section 3.6.2 discusses how to use the Java Plug-in to view applets in Web browsers that do not support Java 2.

Many HTML codes (or tags) come in pairs. For example, lines 1 and 4 of Fig. 3.7 indicate the beginning and the end, respectively, of the HTML tags in the file. All HTML tags begin with a *left angle bracket*, **<**, and end with a *right angle bracket*, **>**. Lines 2 and 3 are special HTML tags for Java applets. They tell the applet container to load a specific applet and define the size of the applet's display area (its *width* and *height* in pixels) in the **appletviewer** (or browser). Normally, the applet and its corresponding HTML file are stored in the same directory on disk. Typically, a browser loads an HTML file from a computer (other than your own) connected to the Internet. However, HTML files also can reside on your computer (as we demonstrated in Section 3.2). When an applet container encounters an HTML file that specifies an applet to execute, the applet container automatically loads the applet's **.class** file (or files) from the same directory on the computer in which the HTML file resides.

The **<applet>** tag has several *attributes*. The first attribute of the **<applet>** tag on line 2 (**code = "WelcomeApplet.class"**) indicates that the file **WelcomeApplet.class** contains the compiled applet class. Remember, when you compile your Java programs, every class is compiled into a separate file that has the same name as the class and ends with the **.class** extension. The second and third attributes of the **<applet>** tag indicate the **width** and the **height** of the applet in pixels. The upper-left corner of the applet's display area is always at *x*-coordinate 0 and *y*-coordinate 0. The width of this applet is 300 pixels and its height is 45 pixels. You may want (or need) to use larger **width** and **height** values to define a larger drawing area for your applets. The **</applet>** tag (line 3) terminates the **<applet>** tag that began on line 2. The **</html>** tag (line 4) specifies the end of the HTML tags that began on line 1 with **<html>**.



Software Engineering Observation 3.2

Generally, each applet should be less than 800 pixels wide and 600 pixels tall (most computer screens support these dimensions as the minimum width and height).



Common Programming Error 3.3

Placing additional characters such as commas (,) between the attributes in the **<applet>** tag may cause the **appletviewer** or browser to produce an error message indicating a **MissingResourceException** when loading the applet.



Common Programming Error 3.4

Forgetting the ending **</applet>** tag prevents the applet from loading into the **appletviewer** or browser properly.



Testing and Debugging Tip 3.5

If you receive a **MissingResourceException** error message when loading an applet into the **appletviewer** or a browser, check the **<applet>** tag in the HTML file carefully for syntax errors. Compare your HTML file to the file in Fig. 3.7 to confirm proper syntax.

The **appletviewer** understands only the **<applet>** and **</applet>** HTML tags, so it is sometimes referred to as the “minimal browser” (it ignores all other HTML tags). The **appletviewer** is an ideal place to test an applet and ensure that it executes properly. Once the applet’s execution is verified, you can add the applet’s HTML tags to an HTML file that will be viewed by people browsing the Internet.

To execute the **WelcomeApplet** in the **appletviewer** open a command window, change to the directory containing your applet and HTML file and type the command

```
appletviewer WelcomeApplet.html
```

Note that the **appletviewer** requires an HTML file to load an applet. This is different from the **java** interpreter for applications which requires only the class name of the application class. Also, the preceding command must be issued from the directory in which the HTML file and the applet’s **.class** file are located.



Common Programming Error 3.5

Running the **appletviewer** with a file name that does not end with **.html** or **.htm** is an error that prevents the **appletviewer** from loading your applet for execution.



Portability Tip 3.2

Test your applets in every browser used by people who view your applet. This will help ensure that people who view your applet experience the functionality you expect. [Note: A goal of the Java Plug-In (discussed later in the book) is to provide consistent applet execution across many different browsers.]

3.4 Two More Simple Applets: Drawing Strings and Lines

Let us consider another applet. An applet can draw **Welcome to Java Programming!** several ways. For example, an applet can use two **drawString** statements in method **paint** to print multiple lines of text as in Fig. 3.8. The HTML file to load the applet into an applet container is shown in Fig. 3.9.

```

1  // Fig. 3.8: WelcomeApplet2.java
2  // Displaying multiple strings in an applet.
3
4  // Java core packages
5  import java.awt.Graphics;    // import class Graphics
6
7  // Java extension packages
8  import javax.swing.JApplet;  // import class JApplet
9
10 public class WelcomeApplet2 extends JApplet {
11
12     // draw text on applet's background
13     public void paint( Graphics g )
14     {
15         // call inherited version of method paint
16         super.paint( g );
17
18         // draw two Strings at different locations
19         g.drawString( "Welcome to", 25, 25 );
20         g.drawString( "Java Programming!", 25, 40 );
21
22     } // end method paint
23
24 } // end class WelcomeApplet2

```

Pixel coordinate (25, 25), where
Welcome to is displayed

Pixel coordinate (25, 40), where
Java Programming! is displayed

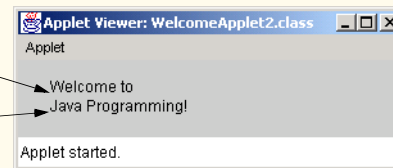


Fig. 3.8 Applet that displays multiple strings.

```

1  <html>
2  <applet code = "WelcomeApplet2.class" width = "300" height = "60">
3  </applet>
4  </html>

```

Fig. 3.9 **WelcomeApplet2.html** loads class **WelcomeApplet2** of Fig. 3.8 into the **appletviewer**.

Note that each call to method **drawString** can draw at any pixel location on the applet. The reason the two output lines appear left aligned as shown in Fig. 3.8 is that both use the same *x* coordinate (25). Also, each **drawString** method call uses different *y* coordinates (25 on line 19 and 40 on line 20), so the strings appear at different vertical locations on the applet. If we reverse lines 19 and 20 in the program, the output window will still appear as shown because the pixel coordinates specified in each **drawString** statement are independent of the coordinates specified in all other **drawString** statements (and all other drawing operations). When drawing graphics, lines of text are not separated by newline characters (as shown with methods **System.out**'s method **println** and **JOptionPane**'s method **showMessageDialog** in Chapter 2). In fact, if you try

to output a string containing a newline character (`\n`), you will simply see a small black box at that position in the string.

To make drawing more interesting, the applet of Fig. 3.10 draws two lines and a string. The HTML file to load the applet into an applet container is shown in Fig. 3.11.

```

1  // Fig. 3.10: WelcomeLines.java
2  // Displaying text and lines
3
4  // Java core packages
5  import java.awt.Graphics;    // import class Graphics
6
7  // Java extension packages
8  import javax.swing.JApplet; // import class JApplet
9
10 public class WelcomeLines extends JApplet {
11
12     // draw lines and a string on applet's background
13     public void paint( Graphics g )
14     {
15         // call inherited version of method paint
16         super.paint( g );
17
18         // draw horizontal line from (15, 10) to (210, 10)
19         g.drawLine( 15, 10, 210, 10 );
20
21         // draw horizontal line from (15, 30) to (210, 30)
22         g.drawLine( 15, 30, 210, 30 );
23
24         // draw String between lines at location (25, 25)
25         g.drawString( "Welcome to Java Programming!", 25, 25 );
26
27     } // end method paint
28
29 } // end class WelcomeLines

```

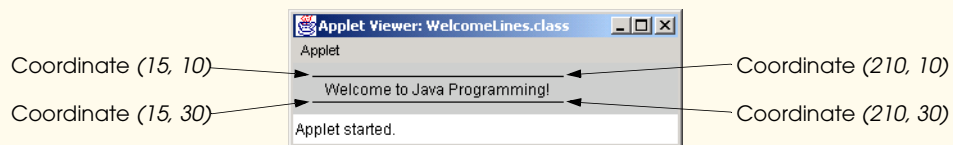


Fig. 3.10 Drawing strings and lines.

```

1  <html>
2  <applet code = "WelcomeLines.class" width = "300" height = "40">
3  </applet>
4  </html>

```

Fig. 3.11 The `WelcomeLines.html` file, which loads class `WelcomeLines` of Fig. 3.10 into the `appletviewer`.

Lines 19 and 22 of method **paint**

```
g.drawLine( 15, 10, 210, 10 );
g.drawLine( 15, 30, 210, 30 );
```

use *method* **drawLine** of class **Graphics** to indicate that the **Graphics** object that **g** refers to should draw lines. Method **drawLine** requires four arguments that represent the two end points of the line on the applet—the *x*-coordinate and *y*-coordinate of the first end point in the line and the *x*-coordinate and *y*-coordinate of the second end point in the line. All coordinate values are specified with respect to the upper-left corner (0, 0) coordinate of the applet. Method **drawLine** draws a straight line between the two end points.

3.5 Another Java Applet: Adding Floating-Point Numbers

Our next applet (Fig. 3.12) mimics the application of Fig. 2.9 for adding two integers. However, this applet requests that the user enter two *floating-point numbers* (i.e., numbers with a decimal point such as 7.33, 0.0975 and 1000.12345). To store floating-point numbers in memory we introduce primitive data type **double**, which represents *double-precision floating-point numbers*. There is also primitive data type **float** for *storing single-precision floating-point numbers*. A **double** requires more memory to store a floating-point value, but stores it with approximately twice the precision of a **float** (15 significant digits for **double** vs. seven significant digits for **float**).

Once again, we use **JOptionPane.showInputDialog** to request input from the user. The applet computes the sum of the input values and displays the result by drawing a string inside a rectangle on the applet. The HTML file to load this applet into the **applet-viewer** is shown in Fig. 3.13.

```
1 // Fig. 3.12: AdditionApplet.java
2 // Adding two floating-point numbers.
3
4 // Java core packages
5 import java.awt.Graphics; // import class Graphics
6
7 // Java extension packages
8 import javax.swing.*; // import package javax.swing
9
10 public class AdditionApplet extends JApplet {
11     double sum; // sum of values entered by user
12
13     // initialize applet by obtaining values from user
14     public void init()
15     {
16         String firstNumber; // first string entered by user
17         String secondNumber; // second string entered by user
18         double number1; // first number to add
19         double number2; // second number to add
20
21         // obtain first number from user
22         firstNumber = JOptionPane.showInputDialog(
23             "Enter first floating-point value" );
```

Fig. 3.12 An addition program “in action” (part 1 of 2).

```

24
25      // obtain second number from user
26      secondNumber = JOptionPane.showInputDialog(
27          "Enter second floating-point value" );
28
29      // convert numbers from type String to type double
30      number1 = Double.parseDouble( firstNumber );
31      number2 = Double.parseDouble( secondNumber );
32
33      // add numbers
34      sum = number1 + number2;
35  }
36
37      // draw results in a rectangle on applet's background
38      public void paint( Graphics g )
39      {
40          // call inherited version of method paint
41          super.paint( g );
42
43          // draw rectangle starting from (15, 10) that is 270
44          // pixels wide and 20 pixels tall
45          g.drawRect( 15, 10, 270, 20 );
46
47          // draw results as a String at (25, 25)
48          g.drawString( "The sum is " + sum, 25, 25 );
49
50      } // end method paint
51
52  } // end class AdditionApplet

```

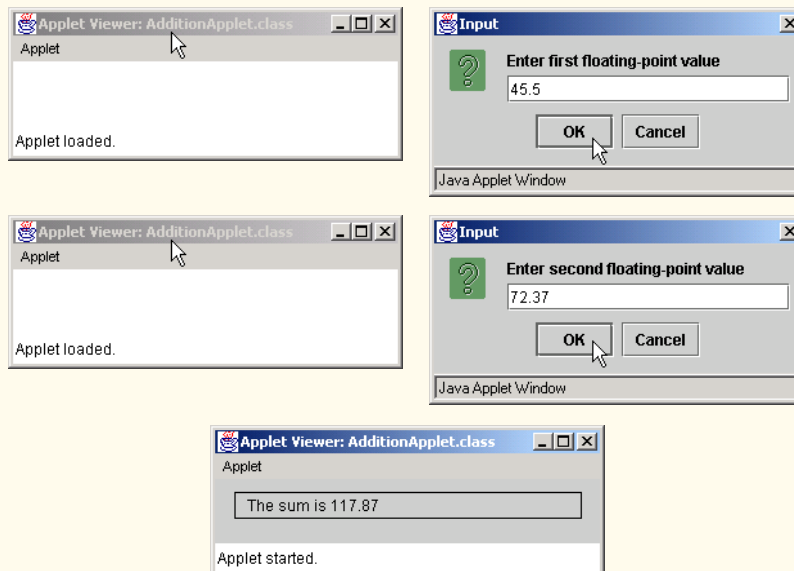


Fig. 3.12 An addition program “in action” (part 2 of 2).

```

1  <html>
2  <applet code = "AdditionApplet.class" width = "300" height = "65">
3  </applet>
4  </html>

```

Fig. 3.13 `AdditionApplet.html` loads class `AdditionApplet` of Fig. 3.12 into the `appletviewer`.

Lines 1–2

```

// Fig. 3.12: AdditionApplet.java
// Adding two floating-point numbers.

```

are single-line comments stating the figure number, file name and purpose of the program.

Line 5

```
import java.awt.Graphics;    // import class Graphics

```

imports class **Graphics** (package **java.awt**) for use in this applet. Actually, the **import** statement at line 5 is not required if we always use the complete name of class **Graphics**—**java.awt.Graphics**—which includes the full package name and class name. For example, the first line of method **paint** can be defined as

```
public void paint( java.awt.Graphics g )

```



Software Engineering Observation 3.3

The Java compiler does not require **import** statements in a Java source code file if the complete class name—the full package name and class name (e.g., **java.awt.Graphics**)—is specified every time a class name is used in the source code.

Line 8

```
import javax.swing.*;        // import package javax.swing

```

specifies to the compiler that several classes from package **javax.swing** are used in this applet. The asterisk (*) indicates that all classes in the **javax.swing** package (such as **JApplet** and **JOptionPane**) should be available to the compiler so the compiler can ensure that we use the classes correctly. This allows programmers to use the *shorthand name* (the class name by itself) of any class from the **javax.swing** package in the program. Remember that our last two programs imported only class **JApplet** from package **javax.swing**. In this program, we use classes **JApplet** and **JOptionPane** from that package. Importing an entire package into a program is also a shorthand notation so the programmer is not required to provide a separate **import** statement for every class used from that package. Remember that you can always use the complete name of every class, i.e., **javax.swing.JApplet** and **javax.swing.JOptionPane** rather than **import** statements.



Software Engineering Observation 3.4

The compiler does not load every class in a package when it encounters an **import** statement that uses the * (e.g., **javax.swing.***) notation. The compiler loads from the package only those classes the program uses.



Software Engineering Observation 3.5

Many packages have subpackages. For example, the `java.awt` package has subpackage `event` for the package `java.awt.event`. When the compiler encounters an `import` statement that uses the `*` (e.g., `java.awt.*`) notation to indicate that a program uses multiple classes from a package, the compiler does not load classes from the subpackage `event`. Thus, you cannot define an `import` of `java.*` to search for classes from all Java core packages.



Software Engineering Observation 3.6

When using `import` statements, separate `import` statements must be specified for each package used in a program.



Common Programming Error 3.6

Assuming that an `import` statement for an entire package (e.g., `java.awt.*`) also `imports` classes from subpackages in that package (e.g., `java.awt.event.*`) results in syntax errors for the classes from the subpackages. There must be separate `import` statements for every package from which classes are used.

Remember that applets inherit from the `JApplet` class, so they have all the methods that an applet container requires to execute the applet. Line 10

```
public class AdditionApplet extends JApplet {
```

begins class `AdditionApplet`'s definition and indicates that it inherits from `JApplet`.

All class definitions start with an opening left brace (end of line 10), `{`, and end with a closing right brace, `}` (line 52).



Common Programming Error 3.7

If braces do not occur in matching pairs, the compiler indicates a syntax error.

Line 11

```
double sum; // sum of values entered by user
```

is an *instance variable declaration*—every instance (object) of the class contains one copy of each instance variable. For example, if there are 10 instances of this applet executing, each instance has its own copy of `sum`. Thus, there would be 10 separate copies of `sum` (one per applet). Programmers declare instance variables in the body of the class definition, but outside the bodies of all the class's method definitions. The preceding declaration states that `sum` is a variable of primitive type `double`.

A benefit of instance variables is that all the methods of the class can use the instance variables. Until now, we declared all variables in an application's `main` method. Variables defined in the body of a method are known as *local variables* and can be used only in the body of the method in which they are defined. Another distinction between instance variables and local variables is that instance variables have default values and local variables do not. The default value of variable `sum` is 0.0, because `sum` is an instance variable.



Good Programming Practice 3.3

Explicitly initializing instance variables rather than relying on automatic initialization improves program readability.

The applet of Fig. 3.12 contains two methods—**init** (lines 14–35) and **paint** (lines 38–50). When an applet container loads an applet, the container creates an instance of the applet class and calls its **init** method. The applet container calls method **init** only once during an applet's execution. Method **init** normally *initializes* the applet's instance variables (if they need to be initialized to a value other than their default value) and performs tasks that should occur only once when the applet begins execution. As we will see in later chapters, the applet's **init** method typically creates the applet's graphical user interface.



Software Engineering Observation 3.7

The order in which methods are defined in a class definition has no effect on when those methods are called at execution time. However, following conventions for the order in which methods are defined improves program readability and maintainability.

The first line of the **init** method always appears as

```
public void init()
```

indicating that **init** is a **public** method that returns no information (**void**) when it completes and receives no arguments (empty parentheses after **init**) to perform its task.

The left brace (line 15) marks the beginning of **init**'s body, and the corresponding right brace (line 35) marks the end of **init**. Lines 16–17

```
String firstNumber; // first string entered by user
String secondNumber; // second string entered by user
```

declare local **String** variables **firstNumber** and **secondNumber** in which the program stores the **Strings** input by the user.

Lines 18–19

```
double number1; // first number to add
double number2; // second number to add
```

declare local variables **number1** and **number2** of primitive data type **double**—these variables hold floating-point values. Unlike **sum**, **number1** and **number2** are not instance variables, so they are not initialized to 0.0 (the default value of **double** instance variables).

As an important aside, there are actually two types of variables in Java—*primitive data type variables* (normally called *variables*) and *reference variables* (normally called *references*). The identifiers **firstNumber** and **secondNumber** are actually references—names that are used to *refer to objects* in the program. Such references actually contain the location of an object in the computer's memory. In our preceding applets, method **paint** actually receives a reference called **g** that refers to a **Graphics** object. Statements in method **paint** use that reference to send messages to the **Graphics** object. These messages are calls to methods (like **drawString**, **drawLine** and **drawRect**) that enable the program to draw. For example, the statement

```
g.drawString( "Welcome to Java Programming!", 25, 25 );
```

sends the **drawString** message to the **Graphics** object to which **g** refers. As part of the message, which is simply a method call, we provide the data that **drawString** re-

quires to do its task. The **Graphics** object uses this data to draw the **String** at the specified location.

The identifiers **number1**, **number2** and **sum** are the names of *variables*. A variable is similar to an object. The primary difference between a variable and an object is that an object is defined by a class definition that can contain both data (instance variables) and methods, whereas a variable is defined by a *primitive (or built-in) data type* (one of **char**, **byte**, **short**, **int**, **long**, **float**, **double** or **boolean**) that can contain only data. A variable can store exactly one value at a time, whereas one object may contain many individual pieces of data. The distinction between a variable and a reference is based on the data type of the identifier, which is stated in a declaration. If the data type is a class name, the identifier is a reference to an object and that reference can be used to send messages to (call methods on) that object. If the data type is one of the primitive data types, the identifier is a variable that can be used to store in memory or retrieve from memory a single value of the declared primitive type.



Software Engineering Observation 3.8

*A hint to help you determine if an identifier is a variable or a reference is the variable's data type. By convention all class names in Java start with a capital letter. Therefore, if the data type starts with a capital letter, normally you can assume that the identifier is a reference to an object of the declared type (e.g., **Graphics g** indicates that **g** is a reference to a **Graphics** object).*

Lines 22–23

```
// obtain first number from user
firstNumber = JOptionPane.showInputDialog(
    "Enter first floating-point value" );
```

read the first floating-point number from the user. **JOptionPane** method **showInputDialog** displays an input dialog that prompts the user to enter a value. The user types a value in the input dialog's text field, then clicks the **OK** button to return the string the user typed to the program. If you type and nothing appears in the text field, position the mouse pointer in the text field and click the mouse to make the text field active. Variable **firstNumber** is assigned the result of the call to **JOptionPane.showInputDialog** operation with an assignment statement. The statement is read as "**firstNumber** gets the value of **JOptionPane.showInputDialog** (**"Enter first floating-point value"**)."

In lines 22–23, notice the method call syntax. At this point, we have seen two different ways to call methods. This statement uses the **static** method call syntax introduced in Chapter 2. All **static** methods are called with the syntax

ClassName.methodName (arguments)

Also in this chapter, we have called methods of class **Graphics** with a similar syntax that started with a reference to a **Graphics** object. Generically, this syntax is

referenceName.methodName (arguments)

This syntax is used for most methods calls in Java. In fact, the applet container uses this syntax to call methods **init**, **start** and **paint** on your applets.

Lines 26–27

```
// obtain second number from user
secondNumber = JOptionPane.showInputDialog(
    "Enter second floating-point value" );
```

read the second floating-point value from the user by displaying an input dialog.

Lines 30–31

```
number1 = Double.parseDouble( firstNumber );
number2 = Double.parseDouble( secondNumber );
```

convert the two strings input by the user to **double** values for use in a calculation. Method **Double.parseDouble** (a **static** method of class **Double**) converts its **String** argument to a **double** floating-point value. Class **Double** is in package **java.lang**. The floating-point value returned by **parseDouble** in line 30 is assigned to variable **number1**. The floating-point value returned by **parseDouble** in line 31 is assigned to variable **number2**.



Software Engineering Observation 3.9

*Each primitive data type (such as **int** or **double**) has a corresponding class (such as **Integer** or **Double**) in package **java.lang**. These classes (commonly known as type-wrapper classes) provide methods for processing primitive data type values (such as converting a **String** to a primitive data type value or converting a primitive data type value to a **String**). Primitive data types do not have methods. Therefore, methods related to a primitive data type are located in the corresponding type-wrapper class (e.g., method **parseDouble** that converts a **String** to a **double** value is located in class **Double**). See the online API documentation for the complete details of the methods in the type-wrapper classes.*

The assignment statement at line 34

```
sum = number1 + number2;
```

calculates the sum of the values stored in variables **number1** and **number2** and assigns the result to variable **sum** using the assignment operator **=**. The statement is read as “**sum** gets the value of **number1 + number2**.” Notice that instance variable **sum** is used in method **init** even though **sum** was not defined in method **init**. We can use **sum** in **init** (and all other methods of the class), because **sum** is an instance variable.

At this point the applet’s **init** method returns and the applet container calls the applet’s **start** method. We did not define **start** in this applet, so the one inherited from class **JApplet** is called here. Normally, the **start** method is used with an advanced concept called multithreading. See Chapter 15 and Chapter 18 for typical uses of **start**.

Next, the applet container calls the applet’s **paint** method. In this example, method **paint** draws a rectangle in which the result of the addition will appear. Line 45

```
g.drawRect( 15, 10, 270, 20 );
```

sends the **drawRect** message to the **Graphics** object to which **g** refers (calls the **Graphics** object’s **drawRect** method). Method **drawRect** draws a rectangle based on its four arguments. The first two integer values represent the *upper-left x-coordinate* and *upper-left y-coordinate* where the **Graphics** object begins drawing the rectangle. The third and fourth arguments are non-negative integers that represent the *width* of the rectan-

gle in pixels and the *height* of the rectangle in pixels, respectively. This particular statement draws a rectangle starting at coordinate (15, 10) that is 270 pixels wide and 20 pixels tall.



Common Programming Error 3.8

*It is a logic error to supply a negative width or negative height as an argument to **Graphics** method **drawRect**. The rectangle will not be displayed and no error will be indicated.*



Common Programming Error 3.9

*It is a logic error to supply two points (i.e., pairs of x- and y-coordinates) as the arguments to **Graphics** method **drawRect**. The third argument must be the width in pixels and the fourth argument must be the height in pixels of the rectangle to draw.*



Common Programming Error 3.10

*It is normally a logic error to supply arguments to **Graphics** method **drawRect** that cause the rectangle to draw outside the applet's viewable area (i.e., the width and height of the applet as specified in the HTML document that references the applet). Either increase the applet's width and height in the HTML document or pass arguments to method **drawRect** that cause the rectangle to draw inside the applet's viewable area.*

Line 48

```
g.drawString( "The sum is " + sum, 25, 25 );
```

sends the **drawString** message to the **Graphics** object to which **g** refers (calls the **Graphics** object's **drawString** method). The expression

```
"The sum is " + sum
```

from the preceding statement uses the string concatenation operator **+** to concatenate the string **"The sum is "** and **sum** (converted to a string) to create the string **drawString** displays. Notice again that the preceding statement uses the instance variable **sum** even though method **paint** does not define **sum** as a local variable.

The benefit of defining **sum** as an instance variable is that we were able to assign **sum** a value in **init** and use **sum**'s value in the **paint** method later in the program. All methods of a class are capable of using the instance variables in the class definition.



Software Engineering Observation 3.10

*The only statements that should be placed in an applet's **init** method are those that are directly related to the one-time initialization of an applet's instance variables. The applet's results should be displayed from other methods of the applet class. Results that involve drawing should be displayed from the applet's **paint** method.*



Software Engineering Observation 3.11

*The only statements that should be placed in an applet's **paint** method are those that are directly related to drawing (i.e., calls to methods of class **Graphics**) and the logic of drawing. Generally, dialog boxes should not be displayed from an applet's **paint** method.*

3.6 Viewing Applets in a Web Browser

We demonstrated several applets in this chapter using the **appletviewer** applet container. As we mentioned, applets also can execute in Java-enabled Web browsers. Unfortunately,

ly, there are many different browser versions being used worldwide. Some support only Java 1.0 and many support Java 1.1. However, few support the Java 2 Platform. Also, even the browsers that support Java 1.1 do so inconsistently. In Section 3.6.1, we demonstrate an applet executing in Netscape Navigator 6, which supports Java 2. In Section 3.6.2, we demonstrate how to use the Java Plug-in to execute Java 2 applets in other Web browsers such as Microsoft Internet Explorer or earlier versions of Netscape Navigator.



Portability Tip 3.3

Not all Web browsers support Java. Those that do often support different versions and are not always consistent across all platforms.

3.6.1 Viewing Applets in Netscape Navigator 6

When you install Netscape Navigator 6, one of the browser components in the default installation is Java 2. Once installed, you can simply load an applet's HTML file into the browser to execute the applet. You can download and install Netscape 6 from

www.netscape.com

by clicking the **Download** button at the top of the Web page.

After installing the browser, open the program. On Windows, Netscape 6 typically places an icon on your desktop during the install process. In the **File** menu, click **Open File...** to select an HTML document from your local computer's hard disk. In the **Open File** dialog, navigate to the location of the HTML file of Fig. 3.11. Select the file name **WelcomeLines.html** by clicking it, then click the **Open** button to open the file in the browser. In a few moments, you should see the applet of Fig. 3.10 appear in the browser window as shown in Fig. 3.14.

3.6.2 Viewing Applets in Other Browsers Using the Java Plug-In

If you would like to use the features of the Java 2 platform in an applet and execute that applet in a browser that does not support Java 2, Sun provides the *Java Plug-in* to bypass a browser's Java support and use a complete version of the *Java 2 Runtime Environment (J2RE)* that is installed on the user's local computer. If the J2RE does not already exist on the client machine, it can be downloaded and installed dynamically.



Performance Tip 3.1

Because of the size of the Java Plug-in, it is difficult and inefficient to download the Plug-in for users with slower Internet connections. For this reason, the Plug-in is ideal for corporate intranets where users are connected to a high-speed network. Once the Plug-in is downloaded, it does not need to be downloaded again.

You must indicate in the HTML file containing an applet that the browser should use the Java Plug-in to execute the applet. To do so, requires that you convert the **<applet>** and **</applet>** tags into tags that load the Java Plug-in and execute the applet. Sun provides a conversion utility called the *Java Plug-in 1.3 HTML Converter*² that performs the conversion for you. Complete information on downloading and using the Java Plug-in and the HTML Converter are available at the Web site

java.sun.com/products/plugin/

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/2/01

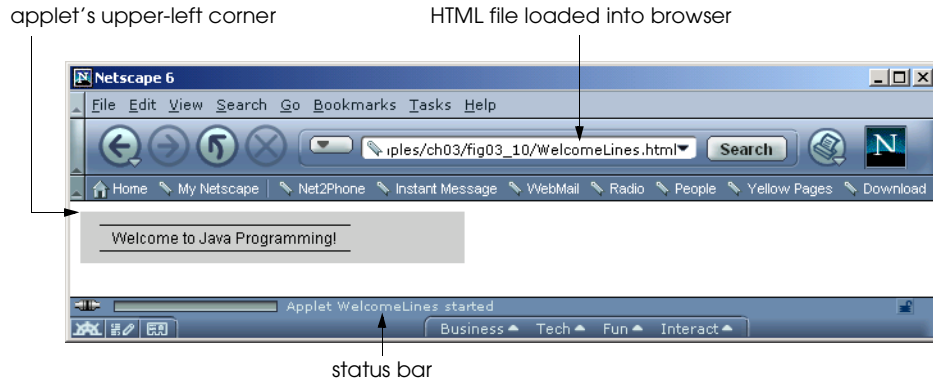


Fig. 3.14 Applet of Fig. 3.10 executing in Netscape Navigator 6.

Once you have downloaded and installed the Java Plug-in HTML converter, you can execute it via the batch file **HTMLConverter.bat** on Windows or the shell script **HTMLConverter.sh** on Linux/UNIX. These files are located in the **converter** directory's **classes** subdirectory. Figure 3.15 shows the **Java Plug in HTML Converter** window.

The **Java Plug-in HTML Converter** allows you to convert all the HTML files containing applets in one directory. Click the **Browse...** button to select the directory containing the files to convert.

Also, you can specify the directory in which the original HTML files are saved.

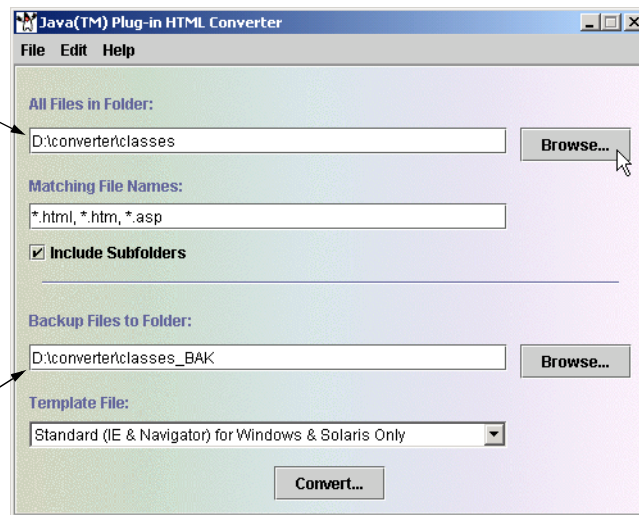


Fig. 3.15 Java Plug-in HTML Converter window.

- As of Java 2 Software Development Kit version 1.3.1, a command-line version of the Java Plug-in HTML converter is one of the tools in the J2SDK. To use the command-line version, open a command window and change directories to the location that contains the HTML file to convert. In that directory type **HTMLConverter fileName**, where *fileName* is the HTML file to convert. Visit java.sun.com/products/plugin/1.3/docs/htmlconv.html for more details on the command-line HTML converter.

To perform the conversion, you must select the directory containing the HTML files to convert. You can either type the directory name in the text field below **All Files in Folder**, or you can select the directory by clicking the **Browse...** button to the right of that text field. We clicked the **Browse...** button to display the **Open** dialog in Fig. 3.16.

After selecting the directory containing files to convert, the **Java Plug in HTML Converter** window appears as in Fig. 3.17. The converter provides several conversion templates to support different combinations of browsers. The default template supports Netscape Navigator and Microsoft Internet Explorer. Figure 3.17 shows the expanded **Template File** drop-down list containing the pre-defined conversion templates. We selected the default template that enables Microsoft Internet Explorer and Netscape Navigator to use the plug-in to execute an applet.

After selecting the appropriate template file, click the **Convert...** button at the bottom of the **Java Plug in HTML Converter** window. Figure 3.18 shows the dialog box that appears containing the status and results of the conversion. At this point the applet's HTML file can be loaded into Netscape Navigator or Microsoft Internet Explorer to execute the applet. If the Java 2 Runtime Environment does not already exist on the user's computer, the converted HTML file contains information that enables the browser to prompt users to determine if they would like to download the plug-in.

In this chapter and Chapter 2, we have introduced many important features of Java, including applications, applets, displaying data on the screen, inputting data from the keyboard, performing calculations and making decisions. In Chapter 4, we build on these techniques as we introduce *structured programming*. Here, you will become more familiar with indentation techniques. We also study how to specify and vary the order in which a program executes statements—this order is called *flow of control*.

The **Open** dialog box allows you to select the directory containing the files to convert.

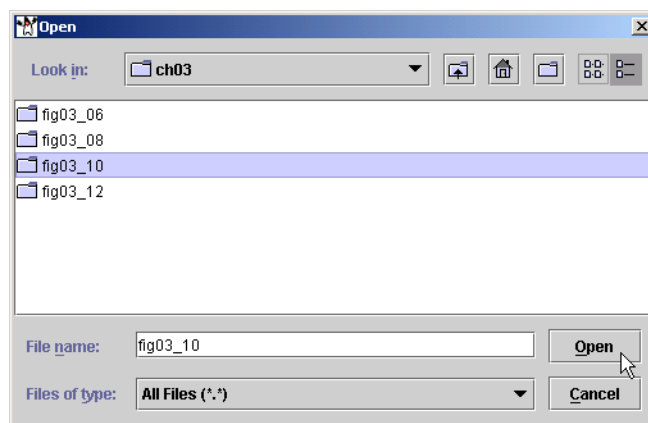


Fig. 3.16 Selecting the directory containing HTML files to convert.

The **Java Plug-in HTML Converter** window after selecting the directory containing files to convert.

The **Template File** drop-down list allows you to choose the browsers in which to use the plug-in to execute applets.

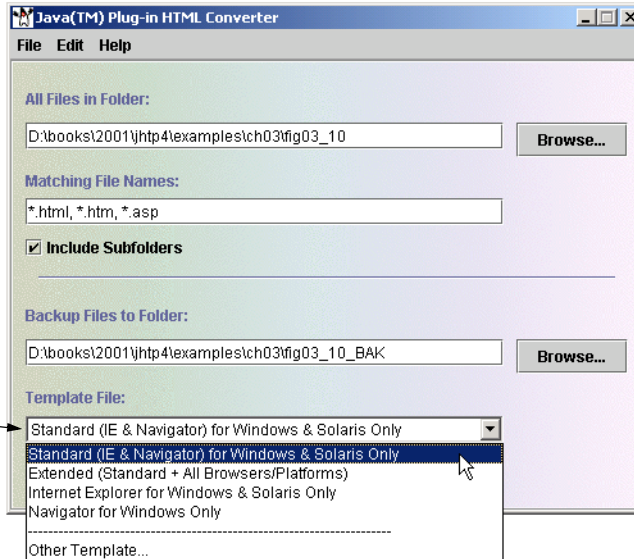


Fig. 3.17 Selecting the template used to convert the HTML files.

The confirmation dialog showing that the converter found and converted one applet in an HTML file in the specified directory.

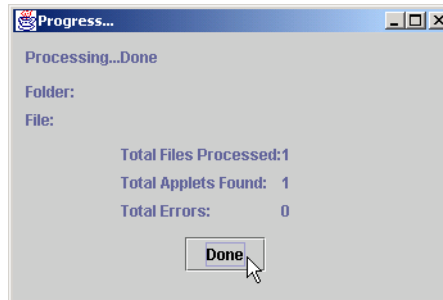


Fig. 3.18 Confirmation dialog after conversion completes.

3.7 Java Applet Internet and World Wide Web Resources

If you have access to the Internet and the World Wide Web, there are a large number of Java applet resources available to you. The best place to start is at the source—the Sun Microsystems, Inc. Java Web site **java.sun.com**. In the upper-left corner of the Web page is an **Applets** *hyperlink* that takes you to the Web page

java.sun.com/applets

This page contains a variety of Java applet resources, including free applets you can use on your own World Wide Web site, the demonstration applets from the J2SDK and a variety of other applets (many of which can be downloaded and used on your own computer).

There is also a section entitled “Applets at Work” where you can read about uses of applets in industry.

On the Sun Microsystems Java Web site, visit the *Java Developer Connection*

`developer.java.sun.com/developer`

This free site includes technical support, discussion forums, on-line training courses, technical articles, resources, announcements of new Java features, early access to new Java technologies, and links to other important Java Web sites. Even though the site is free, you must register to use it.

Another useful Web site is *JARS*—originally called the *Java Applet Rating Service*. The JARS site

`www.jars.com`

calls itself the “#1 Java Review Service.” This site originally was a large Java repository for applets. Its benefit was that it rated every applet registered at the site as top 1%, top 5% and top 25%, so you could view the best applets on the Web. Early in the development of the Java language, having your applet rated here was a great way to demonstrate your Java programming abilities. JARS is now all-around resource for Java programmers.

The resources listed in this section provide hyperlinks to many other Java-related Web sites. If you have Internet access, spend some time browsing these sites, executing applets and reading the source code for the applets when it is available. This will help you rapidly expand your Java expertise. Appendix B contains many other Web-based Java resources.

3.8 (Optional Case Study) Thinking About Objects: Identifying the Classes in a Problem Statement

Now we begin the substantial task of designing the elevator simulator model, which represents the workings of the elevator system. We will design the user interaction and display of this model in Section 12.16 and Section 22.9, respectively.

Identifying the Classes in a System

The first step of our OOD process is to identify the classes in our model. We will eventually describe these classes in a formal way and implement them in Java. First, we review the problem statement and locate all the *nouns*; it is likely that these include most of the classes (or instances of classes) necessary to implement the elevator simulation. Figure 3.19 is a list of these nouns (and noun phrases) in the order of their appearance.

Nouns (and noun phrases) in the problem statement

company	elevator system	graphical user interface (GUI)
office building	elevator shaft	elevator car
elevator	display	person
software-simulator application	model	floor (first floor; second floor)

Fig. 3.19 Nouns (and noun phrases) in problem statement (part 1 of 2).

Nouns (and noun phrases) in the problem statement

passenger	bell inside the elevator	First Floor GUI button
floor door	light on that floor	Second Floor GUI button
user of our application	energy	audio
floor button	capacity	elevator music
elevator button		

Fig. 3.19 Nouns (and noun phrases) in problem statement (part 2 of 2).

We choose only the nouns that perform important duties in our model. For this reason we omit several nouns (the next paragraph explains why each is omitted):

- company
- office building
- display
- graphical user interface (GUI)
- user of our application
- energy
- capacity
- **First Floor** and **Second Floor** GUI buttons
- audio
- elevator music

We do not need to model “company,” as a class, because the company is not part of the simulation; the company simply wants us to model the elevator. We do not model the office building, or the actual place the elevator is situated, because the building does not affect *how* our elevator simulation operates. The phrases “display,” “audio” and “elevator music” pertain to the presentation of the model, but do not pertain to the model itself. We use these phrases when we construct the presentation in Section 22.9 and Appendix I. The phrases “graphical user interface,” “user of our application” and “**First Floor** and **Second Floor** GUI buttons” pertain to how the user controls the model, but they do not represent the model. We use these phrases when we construct the user interface in Section 12.16. “Capacity” is a property of the elevator and of the floor—not a separate entity itself. Lastly, although we’ll be saving energy with the policy of not moving the elevator until requested, we do not model “energy.”

We determine the classes for our system by grouping the remaining nouns into categories. We discard “elevator system” for the time being—we focus on designing only the system’s model and disregard how this model relates to the system as a whole. (We discuss the system as a whole in Section 13.17.) Using this logic, we discard “simulation,” because the simulation *is* the system in our case study. Lastly, we combine “elevator” and “elevator car” into “elevator,” because the problem statement uses the two words interchangeably. Each remaining noun from Fig. 3.19 refers to one or more of the following categories:

- model
- elevator shaft
- elevator
- person
- floor (first floor, second floor)
- elevator door
- floor door
- elevator button
- floor button
- bell
- light

These categories are likely to be classes we will need to implement our system. Notice that we create one category for the buttons on the floors and one category for the button on the elevator. The two types of buttons perform different duties in our simulation—the buttons on the floors summon the elevator, and the button in the elevator informs the elevator to move to the other floor.

We can now model the classes in our system based on the categories we created. By convention, we capitalize class names in the design process (as we will do when we write the actual Java program that implements our design). If the name of a class contains more than one word, we run the words together and capitalize each word (e.g., **Multiple-WordName**). Using this convention, we create classes **ElevatorModel**,³ **ElevatorShaft**, **Elevator**, **Person**, **Floor**, **ElevatorDoor**, **FloorDoor**, **ElevatorButton**, **FloorButton**, **Bell** and **Light**. We construct our system using all of these classes as building blocks. Before we begin building the system, however, we must gain a better understanding of how the classes relate to one another.

Class Diagrams

The UML enables us to model, via the *class diagram*, the classes in the elevator system and their interrelationships. Class diagrams model the structure of the system by providing the classes, or “building blocks,” of the system. Figure 3.20 represents class **Elevator** using the UML. In a class diagram, each class is modeled as a rectangle. We then divide this rectangle into three parts. The top part contains the name of the class. The middle part contains the class’ *attributes*. (We discuss attributes in “Thinking About Objects” Section 4.14 and Section 5.11.) The bottom part of the rectangle contains the class’ *operations* (discussed in “Thinking About Objects,” Section 6.17).

3. When we refer to the “elevator model,” we imply all classes composing the model describing the operation of our elevator system—in other words, in our simulation, several classes comprise the model. We will see in Section 13.17 that our system requires a single class to represent the model—we create class **ElevatorModel** to act as the “representative” for the model, because, as we will see in Fig. 3.23, **ElevatorModel** is the class that aggregates all other classes comprising the model.

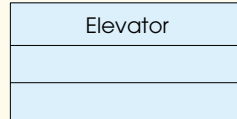


Fig. 3.20 Representing a class in the UML.

Classes relate to one another via *associations*. Figure 3.21 shows how our classes **ElevatorShaft**, **Elevator** and **FloorButton** relate to one another. Notice that the rectangles in this diagram are not subdivided into three sections. The UML allows the suppression of class attributes and operations in this manner to create more readable diagrams. Such a diagram is said to be an *elided diagram*, (i.e., some information, such as the contents for the second and bottom compartments), is not modeled. We place information in these compartments in Section 4.14 and Section 6.17, respectively.

In this class diagram, a solid line that connects classes represents an *association*. An association is a relationship between classes. The numbers near the lines express *multiplicity* values. Multiplicity values indicate how many objects of a class participate in the association. From the diagram, we see that two objects of class **FloorButton** participate in the association with one object of class **ElevatorShaft**, because the two **FloorButtons** are located on the **ElevatorShaft**. Therefore, class **FloorButton** has a *two-to-one* relationship with class **ElevatorShaft**; we can also say that class **ElevatorShaft** has a *one-to-two* relationship with class **FloorButton**. We also see that class **ElevatorShaft** has a *one-to-one* relationship with class **Elevator** and vice versa. Using the UML, we can model many types of multiplicity. Figure 3.22 shows the multiplicity types and how to represent them.

An association can be named. For example, the word **Requests** above the line connecting classes **FloorButton** and **Elevator** indicates the name of that association—the arrow shows the direction of the association. This part of the diagram reads “one object of class **FloorButton** requests one object of class **Elevator**.” Note that associations are directional with the direction indicated by the arrowhead next to the association name—so it would be improper, for example, to read the preceding association as “one object of class **Elevator** requests one object of class **FloorButton**.” In addition, the word **Resets** indicates that “one object of class **ElevatorShaft** resets two objects of class **FloorButton**.” Lastly, the phrase **Signals arrival** indicates that “one object of class **Elevator** signals the **Elevator** object’s arrival to one object of class **ElevatorShaft**.”

The diamond attached to the association lines of class **ElevatorShaft** indicates that class **ElevatorShaft** has an *aggregation* relationship with classes **FloorButton** and **Elevator**. Aggregation implies a whole/part relationship. The class that has the aggregation symbol (the hollow diamond) on its end of an association line is the whole (in this case, **ElevatorShaft**), and the class on the other end of the association line is the part (in this case, classes **FloorButton** and **Elevator**). In this example, the elevator shaft “has an” elevator and two floor buttons. The “has a/has an” relationship defines aggregation (we will see in Section 9.23 that the “is a/is an” relationship defines inheritance).

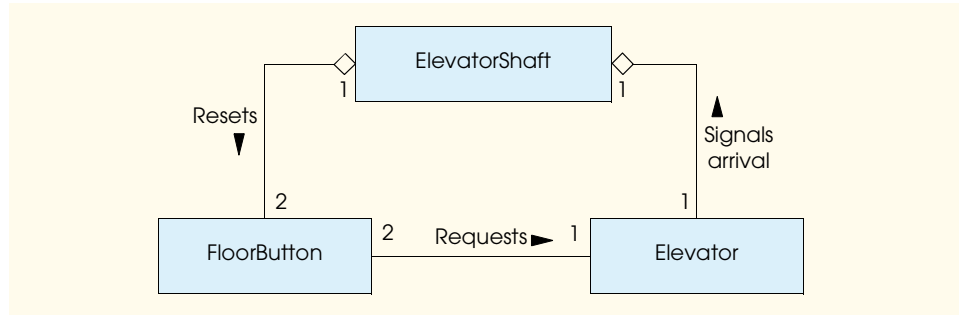


Fig. 3.21 Class diagram showing associations among classes.

Symbol	Meaning
0	None.
1	One.
<i>m</i>	An integer value.
0..1	Zero or one.
<i>m</i> , <i>n</i>	<i>m</i> or <i>n</i>
<i>m</i> .. <i>n</i>	At least <i>m</i> , but not more than <i>n</i> .
*	Zero or more.
0..*	Zero or more
1..*	One or more

Fig. 3.22 Multiplicity types.

Figure 3.23 shows the complete class diagram for the elevator model. We model all classes that we created, as well as the associations between these classes. [Note: In Chapter 9, we expand our class diagram by using the object-oriented concept of *inheritance*.]

Class **ElevatorModel** is represented near the top of the diagram and aggregates one object of class **ElevatorShaft** and two objects of class **Floor**. The **ElevatorShaft** class is an aggregation of one object of class **Elevator** and two objects each of classes **Light**, **FloorDoor** and **FloorButton**. (Notice the two-to-one relationships between each of these classes and **ElevatorShaft**.) Class **Elevator** is an aggregation of classes **ElevatorDoor**, **ElevatorButton** and **Bell**. Class **Person** has associations with both **FloorButton** and **ElevatorButton** (and other classes, as we will soon see). The association name **Presses** and the name-direction arrowheads indicate that the object of class **Person** presses these buttons. The object of class **Person** also rides the object of class **Elevator** and walks across the object of class **Floor**. The name **Requests** indicates that an object of class **FloorButton** requests the object of class **Elevator**. The name **Signals to move** indicates that the object of class **ElevatorButton** signals the object of class **Elevator** to move to the other floor. The diagram indicates many other associations, as well.

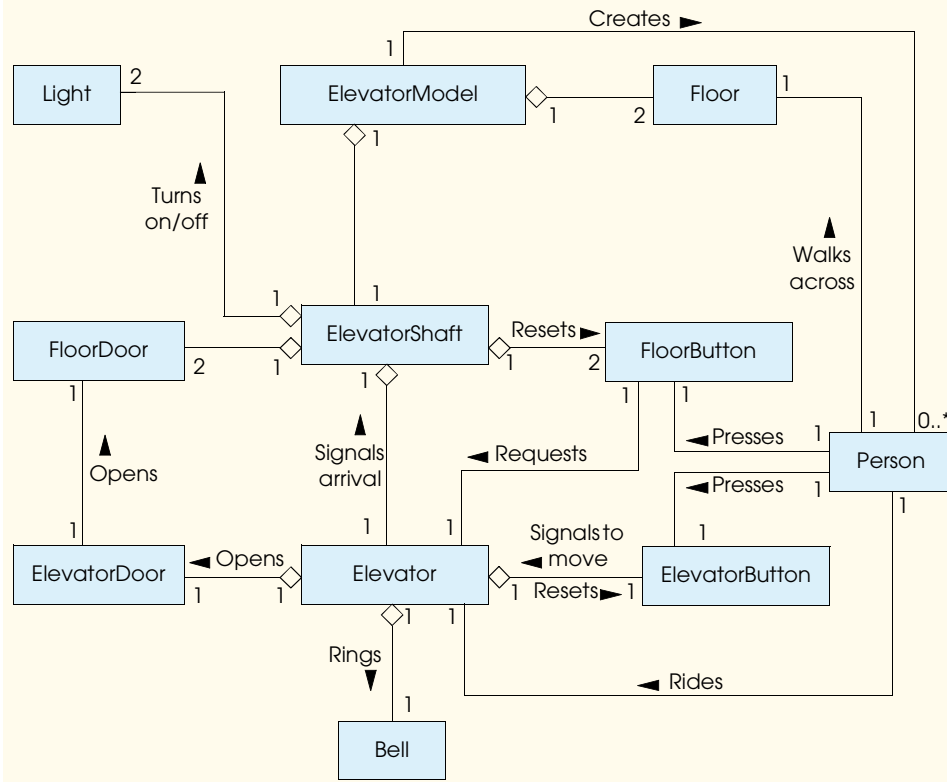


Fig. 3.23 Class diagram for the elevator model.

Object Diagrams

The UML also defines *object diagrams*, which are similar to class diagrams in that both diagrams model the structure of the system. However, object diagrams model objects (instances of classes) *at a specific time* in program execution. Object diagrams present a snapshot of the structure while the system is running, providing information about which objects are participating in the system at a definite point in time. Object diagrams represent relationships between objects as solid lines—these relationships are called *links*.

Figure 3.24 models a snapshot of the system when no people are in the building (i.e., no objects of class **Person** exist in the system at this point in time). Objects usually are written in the form **objectName** : **ClassName**—**objectName** refers to the name of the object, and **ClassName** refers to the class to which that object belongs. All names in an object diagram are underlined. The UML permits us to omit the object names for objects in the diagram where there exists only one object of that class (e.g., one object of class **Bell** at the bottom of the diagram). In large systems, object diagrams can contain many objects. This can result in cluttered, hard-to-read diagrams. If the name of a particular object is unknown, or if it is not necessary to include the name (i.e., we care only about the object type), we can disregard the object name and display only the colon and the class name.

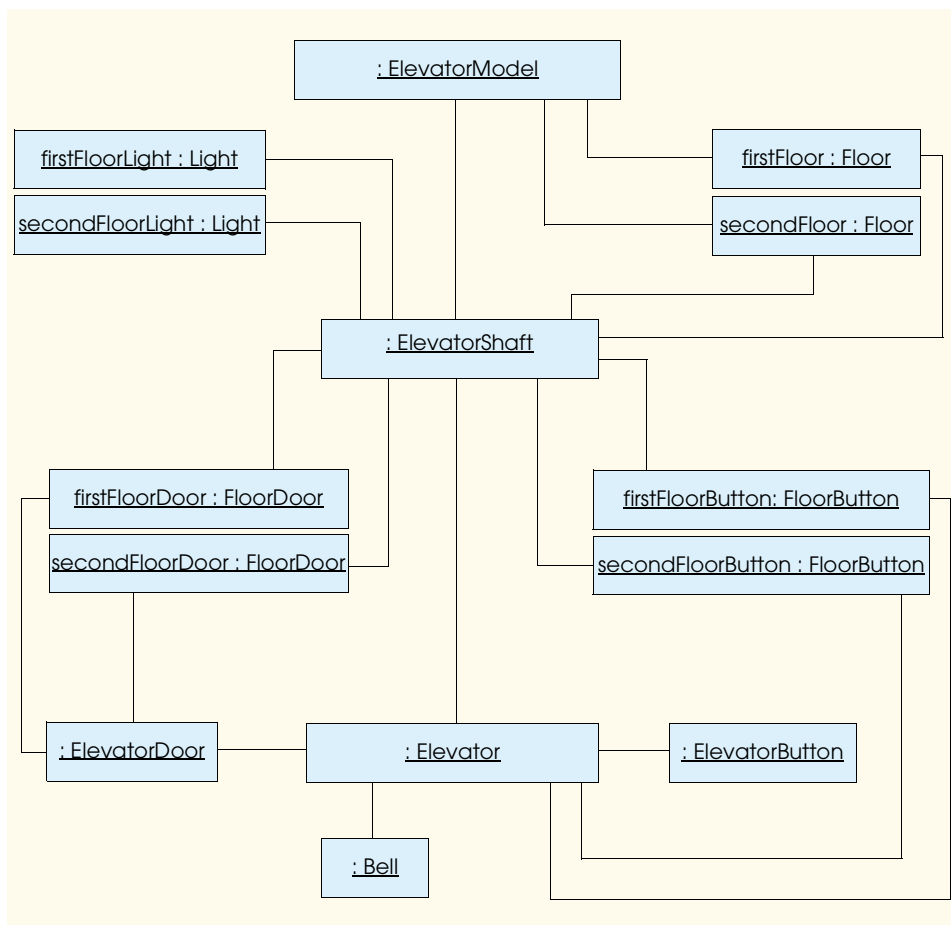


Fig. 3.24 Object diagram of an empty building in our elevator model.

Now we have identified the classes for our system (although we may discover others in later phases of the design process). In “Thinking About Objects,” Section 4.14, we determine the attributes for each of these classes, and in “Thinking About Objects,” Section 5.11, we use these attributes to examine how the system changes over time and to introduce its behavioral aspects. As we expand our knowledge, we will discover new information that will enable us to describe our classes more completely. Because the real world is inherently object oriented, it will be quite natural for you to pursue this project, even though you might have just begun your study of object orientation.

Questions

1. Why might it be more complicated to implement a three-story (or taller) building?
2. It is common for large buildings to have many elevators. We will see in Chapter 9 that once we have created one elevator object, it is easy to create as many as we like. What problems or opportunities do you foresee in having several elevators,

each of which may pick up and discharge passengers at every floor in a large building?

3. For simplicity, we have given our elevator and each floor a capacity of one passenger. What problems and opportunities do you foresee in being able to increase these capacities?

SUMMARY

- Applets are Java programs that can be embedded in Hypertext Markup Language (HTML) documents (i.e., Web pages). When a browser loads a Web page containing an applet, the applet downloads into the Web browser and begins execution.
- In the **appletviewer**, you can execute an applet again by clicking the appletviewer's **Applet** menu and selecting the **Reload** option from the menu. To terminate an applet, click the **appletviewer**'s **Applet** menu and select the **Quit** option.
- Class **Graphics** is located in package **java.awt**. Import the **Graphics** class so the program can draw graphics.
- Class **JApplet** is located in package **javax.swing**. When you create an applet in Java, you must import the **JApplet** class.
- Java uses inheritance to create new classes from existing class definitions. Keyword **extends** followed by a class name indicates the class from which a new class inherits.
- In the inheritance relationship, the class following **extends** is called the superclass or base class and the new class is called the subclass or derived class. Using inheritance results in a new class definition that has the attributes (data) and behaviors (methods) of the superclass as well as the new features added in the subclass definition.
- A benefit of extending class **JApplet** is that someone else already has defined "what it means to be an applet." The **appletviewer** and World Wide Web browsers that support applets expect every Java applet to have certain capabilities (attributes and behaviors), and class **JApplet** already provides those capabilities.
- Classes are used as "templates" or "blueprints" to instantiate (or create) objects in memory for use in a program. An object (or instance) is a region in the computer's memory in which information is stored for use by the program. The term object normally implies that attributes (data) and behaviors (methods) are associated with the object and that those behaviors perform operations on the attributes of the object.
- Method **paint** is one of three methods (behaviors) that an applet container calls when any applet begins execution. These three methods are **init**, **start** and **paint**, and they are guaranteed to be called in that order.
- The parameter list is where methods receive data required to complete their tasks. Normally, this data is passed by the programmer to the method through a method call (also known as invoking a method). In the case of method **paint**, the applet container calls the method and passes the **Graphics** argument.
- Method **drawString** of class **Graphics** draws a string at the specified location on the applet. The first argument to **drawString** is the **String** to draw. The last two arguments in the list are the coordinates (or position) at which the string should be drawn. Coordinates are measured from the upper-left (0, 0) coordinate of the applet in pixels.
- You must create an HTML (Hypertext Markup Language) file to load an applet into an applet container, so the applet container can execute the applet.

- Many HTML codes (referred to as tags) come in pairs. HTML tags begin with a left angle bracket `<` and end with a right angle bracket `>`.
- Normally, the applet and its corresponding HTML file are stored in the same directory on disk.
- The first component of the `<applet>` tag indicates the file containing the compiled applet class. The second and third components of the `<applet>` tag indicate the **width** and the **height** of the applet in pixels. Generally, each applet should be less than 800 pixels wide and 600 pixels tall.
- The **appletviewer** only understands the `<applet>` and `</applet>` HTML tags, so it is sometimes referred to as the “minimal browser.” It ignores all other HTML tags.
- Method **drawLine** of class **Graphics** draws lines. The method requires four arguments representing the two end points of the line on the applet—the x-coordinate and y-coordinate of the first end point in the line and the x-coordinate and y-coordinate of the second end point in the line. All coordinate values are specified with respect to the upper-left corner (0, 0) coordinate of the applet.
- Primitive data type **double** stores double-precision floating-point numbers. Primitive data type **float** stores single-precision floating-point numbers. A **double** requires more memory to store a floating-point value, but stores it with approximately twice the precision of a **float** (15 significant digits for **double** vs. seven significant digits for **float**).
- The **import** statements are not required if you always use the complete name of a class, including the full package name and class name (e.g., **java.awt.Graphics**).
- The asterisk (*) notation after a package name in an **import** indicates that all classes in the package should be available to the compiler so the compiler can ensure that the classes are used correctly. This allows programmers to use the shorthand name (the class name by itself) of any class from the package in the program.
- Every instance (object) of a class contains one copy of each of that class’s instance variables. Instance variables are declared in the body of a class definition, but not in the body of any method of that class definition. An important benefit of instance variables is that their identifiers can be used in all methods of the class.
- Variables defined in the body of a method are known as local variables and can be used only in the body of the method in which they are defined.
- Instance variables are always assigned a default value, and local variables are not.
- Method **init** normally initializes the applet’s instance variables (if they need to be initialized to a value other than their default value) and performs any tasks that should occur only once when the applet begins execution.
- There are actually two types of variables in Java—primitive data type variables and references.
- References refer to objects in a program. References actually contain the location in the computer’s memory of an object. A reference is used to send messages to (i.e., call methods on) the object in memory. As part of the message (method call), we provide the data (arguments) that the method requires to do its task.
- A variable is similar to an object. The primary difference between a variable and an object is that an object is defined by a class definition that can contain both data (instance variables) and methods, whereas a variable is defined by a primitive (or built-in) data type (one of **char**, **byte**, **short**, **int**, **long**, **float**, **double** or **boolean**) that can contain only data.
- A variable can store exactly one value at a time, whereas one object can contain many individual data members.
- If the data type used to declare a variable is a class name, the identifier is a reference to an object and that reference can be used to send messages to (call methods on) that object. If the data type used to declare a variable is one of the primitive data types, the identifier is a variable that can be used to store in memory or retrieve from memory a single value of the declared primitive type.

- Method **Double.parseDouble** (a **static** method of class **Double**) converts its **String** argument to a **double** floating-point value. Class **Double** is part of the package **java.lang**.
- Method **drawRect** draws a rectangle based on its four arguments. The first two integer values represent the upper-left x-coordinate and upper-left y-coordinate where the **Graphics** object begins drawing the rectangle. The third and fourth arguments are non-negative integers that represent the width of the rectangle in pixels and the height of the rectangle in pixels, respectively.
- To use the features of Java 2 in an applet, Sun provides the Java Plug-in to bypass a browser's Java support and use a complete version of the Java 2 Runtime Environment (J2RE) that is installed on the user's local computer.
- To specify that an applet should use the Java Plug-in rather than the browser's Java support, use the HTML Converter to convert the applet's **<applet>** and **</applet>** tags in the HTML file to indicate that the applet container should use the Plug-in to execute the applet. Sun provides the Java Plug-in 1.3 HTML Converter to perform the conversion for you.

TERMINOLOGY

applet	int primitive type
applet container	interface
<applet> tag	invoke a method
Applet menu	JApplet class
appletviewer	java.awt package
boolean primitive type	Java Plug-in
browser	Java 2 Runtime Environment (J2RE)
built-in data type	javax.swing package
byte primitive type	local variable
char primitive type	logic error
command-line argument	long primitive type
coordinate	message
create an object	method call
derived class	Microsoft Internet Explorer
double primitive data type	Netscape Communicator
Double.parseDouble method	object
double-precision floating-point number	paint method of class JApplet
drawLine method of class Graphics	parameter list
drawRect method of class Graphics	pixel (picture element)
drawString method of class Graphics	primitive data type
extends keyword	Quit menu item
float primitive type	references
floating-point number	Reload menu item
Graphics class	short primitive type
height of an applet	single-precision floating-point number
HTML Converter	source code
HTML tag	start method of class JApplet
Hypertext Markup Language (HTML)	subclass
import statement	superclass
information hiding	text file
init method of class JApplet	width of an applet
instance variable	World Wide Web
instantiate an object	

SELF-REVIEW EXERCISES

- 3.1** Fill in the blanks in each of the following.
- Class _____ provides methods for drawing.
 - Java applets begin execution with a series of three method calls: _____, _____, and _____.
 - Methods _____ and _____ display lines and rectangles.
 - Keyword _____ indicates that a new class is a subclass of an existing class.
 - Every Java applet should extend either class _____ or class _____.
 - Java's eight primitive data types are _____, _____, _____, _____, _____, _____, _____, and _____.
- 3.2** State whether each of the following is *true* or *false*. If *false*, explain why.
- To draw a rectangle, method **drawRect** requires four arguments that specify two points on the applet.
 - Method **drawLine** requires four arguments that specify two points on the applet to draw a line.
 - Type **Double** is a primitive data type.
 - Data type **int** is used to declare a floating-point number.
 - Method **Double.parseDouble** converts a **String** to a primitive **double** value.
- 3.3** Write Java statements to accomplish each of the following:
- Display a dialog asking the user to enter a floating-point number.
 - Convert a **String** to a floating-point number and store the converted value in **double** variable **age**. Assume that the **String** is stored in **stringValue**.
 - Draw the message **"This is a Java program"** on one line on an applet (assume you are defining this statement in the applet's **paint** method) at position (10, 10).
 - Draw the message **"This is a Java program"** on two lines on an applet (assume these statements are defined in applet method **paint**) starting at position (10, 10) and where the first line ends with **Java**. Make the two lines start at the same *x* coordinate.

ANSWERS TO SELF-REVIEW EXERCISES

- 3.1** a) **Graphics**. b) **init**, **start**, **paint**. c) **drawLine**, **drawRect**. d) **extends**. e) **JApplet**, **Applet**. f) **byte**, **short**, **int**, **long**, **float**, **double**, **char** and **boolean**.
- 3.2** a) False. Method **drawRect** requires four arguments—two that specify the upper-left corner of the rectangle and two that specify the width and height of the rectangle. b) True. c) False. Type **Double** is a class in the **java.lang** package. Remember that names that start with a capital letter are normally class names. d) False. Data type **double** or data type **float** can be used to declare a floating-point number. Data type **int** is used to declare integers. e) True.
- 3.3** a)

```
stringValue = JOptionPane.showInputDialog(
    "Enter a floating-point number" );
```

b)

```
age = Double.parseDouble( stringValue );
```

c)

```
g.drawString( "This is a Java program", 10, 10 );
```

d)

```
g.drawString( "This is a Java", 10, 10 );
g.drawString( "program", 10, 25 );
```

EXERCISES

- 3.4** Fill in the blanks in each of the following:
- Data type _____ declares a single-precision floating-point variable.
- © Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/2/01

- b) If class **Double** provides method **parseDouble** to convert a **String** to a **double** and class **Integer** provides method **parseInt** to convert a **String** to an **int**, then class **Float** probably provides method _____ to convert a **String** to a **float**.
- c) Data type _____ declares a double-precision floating-point variable.
- d) The _____ or a browser can be used to execute a Java applet.
- e) To load an applet into a browser you must first define a(n) _____ file.
- f) The _____ and _____ HTML tags specify that an applet should be loaded into an applet container and executed.

3.5 State whether each of the following is *true* or *false*. If *false*, explain why.

- a) All browsers support Java 2.
- b) When using an **import** of the form **javax.swing.***, all classes in the package are imported.
- c) You do not need import statements if the full package name and class name are specified each time you refer to a class in a program.

3.6 Write an applet that asks the user to enter two floating-point numbers, obtains the two numbers from the user and draws the sum, product (multiplication), difference and quotient (division) of the two numbers. Use the techniques shown in Fig. 3.12.

3.7 Write an applet that asks the user to enter two floating-point numbers, obtains the numbers from the user and displays the larger number followed by the words “**is larger**” as a string on the applet. If the numbers are equal, print the message “**These numbers are equal.**” Use the techniques shown in Fig. 3.12.

3.8 Write an applet that inputs three floating-point numbers from the user and displays the sum, average, product, smallest and largest of these numbers as strings on the applet. Use the techniques shown in Fig. 3.12.

3.9 Write an applet that inputs from the user the radius of a circle as a floating-point number and draws the circle’s diameter, circumference and area. Use the value 3.14159 for π . Use the techniques shown in Fig. 3.12. [Note: You may also use the predefined constant **Math.PI** for the value of π . This constant is more precise than the value 3.14159. Class **Math** is defined in the **java.lang** package, so you do not need to **import** it.] Use the following formulas (r is the radius):

$$\begin{aligned} \text{diameter} &= 2r \\ \text{circumference} &= 2\pi r \\ \text{area} &= \pi r^2 \end{aligned}$$

3.10 Write an applet that draws a box, an oval, an arrow and a diamond using asterisks (*) as follows:



3.11 Write an applet that reads five integers and determines and prints the largest and smallest integers in the group. Use only the programming techniques you learned in this chapter and Chapter 2. Draw the results on the applet.

3.12 Write an applet that reads in two floating-point numbers and determines and prints if the first is a multiple of the second. (*Hint:* Use the modulus operator.) Use only the programming techniques you learned in this chapter and Chapter 2. Draw the results on the applet.

3.13 Write an applet that draws a checkerboard pattern as follows:

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

3.14 Write an applet that draws a variety of rectangles of different sizes and locations.

3.15 Write an applet that allows the user to input the four arguments required by method **drawRect**, then draws a rectangle using the four input values.

3.16 The **Graphics** class contains a **drawOval** method that takes the same four arguments as method **drawRect**. However, the arguments for method **drawOval** specify the “bounding box” for the oval. The sides of the bounding box are the boundaries of the oval. Write a Java applet that draws an oval and a rectangle with the same four arguments. You will see that the oval touches the rectangle at the center of each side.

3.17 Modify the solution to Exercise 3.16 to output a variety of ovals of different shapes and sizes.

3.18 Write an applet that allows the user to input the four arguments required by method **drawOval**, then draws an oval using the four input values.

3.19 What does the following code print?

```
g.drawString( " ", 25, 25 );
g.drawString( "****", 25, 55 );
g.drawString( "*****", 25, 85 );
g.drawString( "*****", 25, 70 );
g.drawString( "****", 25, 40 );
```

3.20 Using only programming techniques from Chapter 2 and Chapter 3, write an applet that calculates the squares and cubes of the numbers from 0 to 10 and draws the resulting values in table format as follows:

number	square	cube
0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

[*Note:* This program does not require any input from the user.]

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/2/01