CHAPTER

17

MULTITHREADING AND ANIMATION

In this chapter, you will:

- Understand multithreading
- Learn about a Thread's life cycle
- Use the Thread class
- Use the sleep() method
- Set Thread priority
- Use the Runnable interface
- Create an animated figure
- Reduce flickering
- Use pre-drawn animated Image objects
- Understand garbage collection
- Put animation in a Web browser page

I thought I had learned a lot about Java," you tell Lynn Greenbrier during your six-month performance review at Event Handlers Incorporated. "I've learned to write applets and applications, extend classes, and write data files. I can even use sound and images in my applets."

"You've come very far," Lynn agrees. "I'm proud of you!"

"But there's something missing. I want to create applets with moving figures for Event Handlers. I want programs in which different activities take place on the screen at the same time."

"You almost know everything you need to accomplish your goals," Lynn explains. "You know about applets, exceptions, inheritance, and graphics. All you have to do is put it all together. Let me tell you a little about threads, and then you can get started on animation."

Previewing a Program That Displays Animation

Event Handlers Incorporated wants an animated stick figure to appear on its Web site. The figure appears to be using the Event Handlers company name as a yo-yo; that is, the words "Event Handlers" move up and down as if controlled by the moving arm of the stick figure.You will create a similar program in this chapter, but now you can use a completed version of the Chap17AnimationApplet applet that is saved in the Chapter.17 folder on your Student Disk.

To use the Chap17AnimationApplet class:

1. Go to the command prompt for the Chapter.17 folder on your Student Disk, type **`appletviewer Chap17Animation.html`**, and then press **[Enter]**. You will see a moving figure similar to the one captured in Figure 17-1.

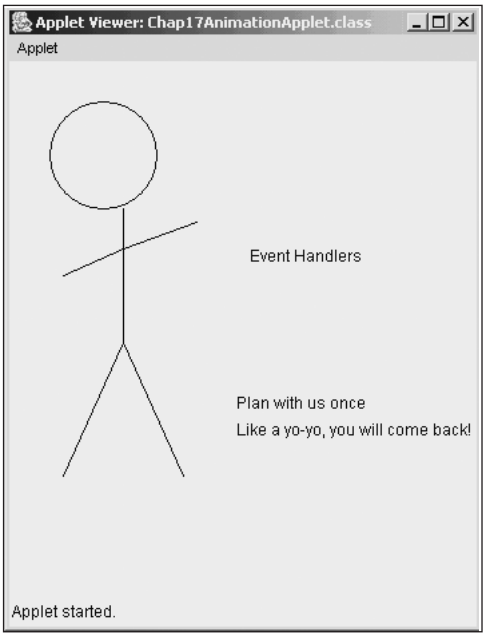

Figure 17-1 Chap17AnimationApplet applet

2. When you are finished viewing the applet, click the Applet Viewer's **Close** button.

UNDERSTANDING MULTITHREADING

A **thread** is the flow of execution of one set of program statements. When you execute a program statement by statement, from beginning to end, you are following a thread. Each of the programs you have written while working through this book has had a single thread; this means that at any one time, Java was executing only a single program statement.

Single-thread programs contain statements that execute in very rapid sequence, but only one statement executes at a time. When a computer contains a single central processing unit (CPU, or processor), it can execute only one computer instruction at a time, regardless of its processor speed. When you use a computer with multiple CPUs, the computer can execute multiple instructions simultaneously.

The Java programming language allows you to launch, or start, multiple threads, no matter which type of processor you use. Using multiple threads of execution is known as **multithreading**. As already noted, if you use a computer system that contains more than one CPU (such as a very large mainframe or supercomputer), multiple threads can execute simultaneously. Figure 17-2 illustrates how multithreading executes in a multiprocessor system.

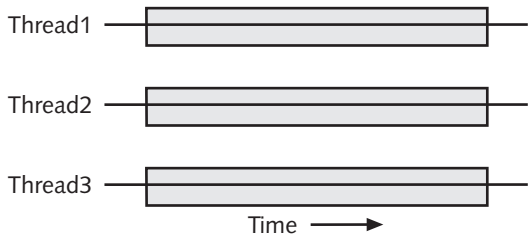

Figure 17-2 Executing multiple threads in a multiprocessor system

If you use a computer with a single processor, the multiple threads share the CPU's time, as shown in Figure 17-3. The CPU devotes a small amount of time to one task, and then devotes a small amount of time to another task. The CPU never actually performs two tasks at the same instant. Instead, it performs a piece of one task, and then a piece of another task. The CPU performs so quickly that each task seems to execute without interruption.

Perhaps you have seen an expert chess player participate in chess games with several opponents at once. The chess player makes a move on the first playing board, and then moves to the second board against a second opponent, while the first opponent analyzes his next move. The master can move to the third board, make a move, and return to the first board before the first opponent is even ready to respond. To the first opponent, it might seem as though the expert player is devoting all of her time to him. Because the expert is so fast, she can play other opponents in the first opponent's "downtime". Executing multiple threads on a single CPU works in a similar way. The CPU transfers its attention from thread to thread so quickly that the tasks don't even "miss" the CPU's attention.

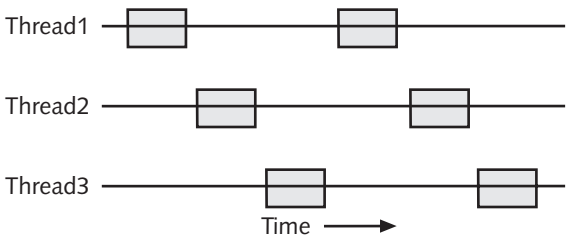

Figure 17-3 Executing multiple threads in a single-processor system

You use multithreading to improve the performance of your programs. Multithreaded programs often run faster, but what is more important is that they are more user-friendly. With a multithreaded program, your user can continue to click buttons while your program is reading a data file. With multithreading, an animated figure can display on one part of the screen while the user makes menu selections on another part of the screen. When you use the Internet, multithreading increases in importance. For example, you can begin to read a long text file or listen to an audio file while they are still downloading. Web users are likely to abandon a site if downloading a file takes too long. When you use multithreading to perform concurrent tasks, you are more likely to retain visitors to your Web site—this is particularly important if your site sells a product or service.

Programmers sometimes use the terms "thread of execution" or "execution context" to describe a thread. They also refer to a thread as a lightweight process because it is not a full-blown program. Rather, a thread must run within the context of a full, heavyweight program.

Learning About a Thread's Life Cycle

A Thread can be in one of five states during its life: new, ready, running, inactive, or finished. When you create a Thread, it is in the new state; the only method you can use with a new Thread is the method to start it. When you call the Thread's start() method, the Thread enters the ready state. A ready Thread is **runnable**, which means that it can run. However, a runnable Thread might not be in the running state because the CPU might be busy elsewhere. Just as your runnable automobile cannot pass through an intersection until the traffic officer waves you on, a runnable Thread cannot run until the CPU allocates some time to it.

If you call a Thread method that the Thread's present state does not allow, Java automatically `throws` an IllegalThreadStateException. For example, you cannot call the stop() method for a Thread that is new and has not been started.

When a Thread begins to execute, it is in the running state. A Thread runs until it becomes inactive or finishes. A Thread enters the inactive state when you call the

Thread's sleep() or suspend() method, or it might become inactive if it must wait for another Thread to finish and for the CPU to have available time. (You will see the sleep() method at work when you write the DemoHelloGoodbyeThreads program later in this chapter.) When a Thread completes the execution of its run() method, it is in the finished or dead state. A Thread can also enter the finished state before its run() method completes if you call the Thread's stop() method. You can use the isAlive() method to determine whether a Thread is currently alive, which means that it has started but has not stopped. The isAlive() method returns the Boolean value `false` if a Thread is new or finished, and `true` if it is in any other state.

Table 17-1 describes several useful methods of the Thread class.

Table 17-1 Selected Thread class methods

Method	Description
start()	Starts the Thread causing the run() method to execute
stop()	Stops the Thread
suspend()	Suspends the Thread until you use the resume() method
resume()	Resumes the Thread you suspended
isAlive()	Returns `true` or `false` to indicate whether the Thread is currently running
setPriority(int)	Lets you set a priority from 1 to 10 for a Thread by passing an integer (You will learn about Thread priorities later in this chapter.)
sleep(int)	Lets you pause thread execution for a specified number of milliseconds

Using the Thread Class

Technically, every program you have created is a thread. You can also create a thread by extending the Thread class, which is defined in the java.lang package. The Thread class contains a method named run(). You override the run() method in your extended class to tell the system how to execute the Thread. For example, you can write the HelloThread class shown in Figure 17-4, which prints a "Hello" message to the screen 100 times. The HelloThread class contains a single method—the run() method—which prints a space, "Hello", and another space in a loop that executes 100 times.


```
class HelloThread extends Thread
{
        public void run()
        {
               for(int x=0; x<100; ++x)
                       System.out.print(" Hello ");
        }
}
```

Figure 17-4 The HelloThread class

When you create a class that extends Thread, you inherit the start() method. You use the start() method with an instantiated Thread object; it tells the system to start execution of the Thread. For example, you can write a program that instantiates and starts a HelloThread object, as shown in Figure 17-5.

```
class DemoHelloThread
{
        public static void main(String[] args)
        {
              HelloThread hello = new HelloThread();
              hello.start();
        }
}
```

Figure 17-5 The DemoHelloThread class

The DemoHelloThread class instantiates a HelloThread object named hello. When you use the start() method with the hello object, the run() method within the HelloThread class executes. The output appears in Figure 17-6; the "Hello" message prints 100 times.

Figure 17-6 Output of the DemoHelloThread program

You can achieve multithreading by starting more than one Thread object. For example, consider the GoodbyeThread class in Figure 17-7; it is identical to the HelloThread class except for the message it prints. When you create a demonstration class like DemoHelloGoodbyeThreads (see Figure 17-8), and instantiate one HelloThread and one GoodbyeThread object within its main() method, then the output might appear as shown in Figure 17-9.

```
class GoodbyeThread extends Thread
{
        public void run()
        {
                for(int x=0; x<100; ++x)
                System.out.print(" Goodbye ");
        }
}
```

Figure 17-7 GoodbyeThread class

```
class DemoHelloGoodbyeThreads
{
        public static void main(String[] args)
        {
                 HelloThread hello = new HelloThread();
                 GoodbyeThread goodbye = new GoodbyeThread();
                 hello.start();
                 goodbye.start();
        }
}
```

Figure 17-8 DemoHelloGoodbyeThread class

Figure 17-9 Output of the DemoHelloGoodbyeThreads program

17

When you run a program like DemoHelloGoodbyeThreads, your output might not appear exactly like Figure 17-9. In fact, if you run your program multiple times, your output might look like Figure 17-10, which shows two subsequent runs of the program. Notice that the first execution displays 16 "Hello" messages before the first "Goodbye" message appears, but the second execution displays 13 "Hello" messages prior to the first "Goodbye". Other differences also appear in the order in which the two executions are displayed. Depending on available resources, the operating system might alternate between "Hello" and "Goodbye", or execute several repetitions of the HelloThread in

a row, followed by several repetitions of the GoodbyeThread. Each Thread completes its 100 repetitions, but you have no guarantee as to the exact pattern of execution; your only guarantee is that the threads execute concurrently.

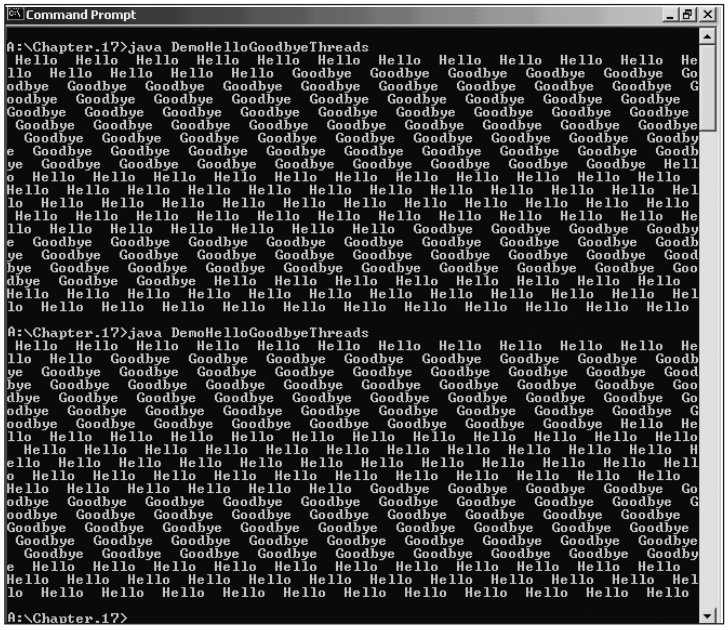

Figure 17-10 Multiple runs of the DemoHelloGoodbyeThreads program

Next you will extend the Thread class so it repeats a character on the screen. Then you will create three Thread objects to observe how they execute concurrently.

To create a class that extends Thread:

1. Open a new file in your text editor, and then type the following first few lines of the ShowThread class. This class has two data members: a character that you will display on the screen and an integer that holds the number of repetitions to be displayed.

```
class ShowThread extends Thread
{
        private char oneChar;
        private int rep = 100;
```

2. Create the following constructor for the ShowThread class. The constructor accepts a character and assigns it to the oneChar variable.

```
public ShowThread(char printChar)
{
        oneChar = printChar;
}
```

3. Enter the following run() method, which prints the character as many times as the rep value specifies:

```
public void run()
{
        for(int x = 0; x < rep; ++x)
                System.out.print(oneChar);
}
```

4. Add a closing curly brace for the class.
5. Save the file as **ShowThread.java** in the Chapter.17 folder on your Student Disk, and then compile the file using the `javac` command.

Next you will write a DemoThreads class containing a main() method that declares and uses three ShowThread objects.

To write the DemoThreads class:

1. Open a new file in your text editor, and then type the following class:

```
class DemoThreads
{
        public static void main(String[] args)
        {
                ShowThread showA = new ShowThread('A');
                ShowThread showB = new ShowThread('B');
                ShowThread showC = new ShowThread('C');
                showA.start();
                showB.start();
                showC.start();
        }
}
```

2. Save the file as **DemoThreads.java** in the Chapter.17 folder on your Student Disk, and then compile and execute the program. The output appears similarly to Figure 17-11, although the execution sequence of your ShowThread objects might vary.
3. Run the program several more times and examine the output for changes in the execution sequence.
4. Open the **ShowThread.java** file in your text editor, and then change the value in the rep variable definition so that it is `500` (for example, `private int rep = 500;`). Save the class using the same filename, and then compile and execute it.

Figure 17-11 Output of the DemoThreads program

5. Recompile the **DemoThreads.java** file, and then execute it. Observe any differences in your output.

USING THE SLEEP() METHOD

One interesting and useful member of the Thread class is the sleep() method. You use the sleep() method to pause a Thread for a specified number of milliseconds. For example, writing `sleep(500);` within a Thread class's run() method causes the execution to rest for 500 milliseconds, or half a second. You might use the sleep() method to pause a program to give a user time to respond to a question or to absorb an image before displaying subsequent images in a series.

You might also use sleep() to give lower priority threads a chance to execute. The next section explains the concept of priority.

When you use the sleep() method, you must `catch` an InterruptedException—the type of Exception thrown when a program stops running before its natural end. When you use the sleep() method, you intend for your program to be interrupted, so you usually `catch` the Exception and take no action.

To demonstrate using the sleep() method:

1. Open a new file in your text editor, and then type the following lines to begin the SleepThread class:

```
import java.awt.*;
class SleepThread extends Thread
{
```

2. Enter the following first few lines of the run() method, which uses a `for` loop to print 100 integers:

```
public void run()
{
  for(int x = 0; x < 100; ++x)
  {
          System.out.print(x + " ");
```

3. Within the `for` loop, add the following `try` block. You will test the loop counter variable, x, and when it equals 25, pause for three seconds; when it equals 75, pause for five seconds. End the `try` block, and then `catch` the InterruptedException.

```
try
{
   if(x == 25)
          sleep(3000);
   if(x == 75)
          sleep(5000);
}
catch(InterruptedException e)
{
}
```

4. Add three closing curly braces: one for the `for` loop, one for the run() method, and one for the SleepThread class.

5. Save the file as **SleepThread.java** in the Chapter.17 folder on you Student Disk, and then compile it using the `javac` command.

6. Open a new file in your text editor, and then type the following DemoSleepThread class that will create a SleepThread object and demonstrate its use:

```
class DemoSleepThread
{
    public static void main(String[] args)
    {
          SleepThread sThread = new SleepThread();
          sThread.start();
    }
}
```

7. Save the file as **DemoSleepThread.java** in the Chapter.17 folder, and then compile and execute the program. When the program runs, 25 integers display, and then you must wait during a three-second sleep. When the program resumes, 50 more integers display, and then you must wait during a five-second sleep. When the program completes, all 100 integers (0 through 99) appear on the screen.

Setting Thread Priority

Every Java Thread has a **priority**, or rank, in terms of preferential access to the operating system's resources. Threads with the same priority are called **peers**. With some operating systems (like Windows), threads are timesliced; that is, each peer receives a set amount of processor time during which it can execute. When that time is up, even if the thread has not finished its execution, the next thread that has equal priority receives a slice of time. Without timeslicing (as in the Solaris operating system), a thread runs to completion before one of its peers can execute.

Each Thread object's priority is represented by an integer in the range of 1 to 10. If you do not assign a priority to a Thread object, it assumes a default priority of 5. You can change a Thread's priority by using the setPriority() method, or determine a Thread's priority with the getPriority() method. If you extend a Thread from an existing Thread, the child Thread assumes its parent's priority.

The Thread class contains three constants. They are MIN_PRIORITY, NORM_PRIORITY, and MAX_PRIORITY, which represent 1, 5, and 10, respectively. When you set a Thread's priority, you can use an integer, as in `myThread.setPriority(10);`, or you can use one of the three constants, as in `myThread.setPriority (Thread.MAX_PRIORITY);`. You also can use an arithmetic expression, such as `yourThread.setPriority(Thread.MAX_PRIORITY - 2);`.

If you use the priority constants with your Thread objects, then your Threads will have the appropriate relative priority even if the developers of Java decide to change the priority values in the future.

When you run a Java program, the runnable Thread with the highest priority runs first. If several Threads have the same priority, they run in rotation. Lower-priority Threads can run only when higher-priority Threads are not runnable (such as when they are finished, suspended, or asleep).

In general, when ThreadA has higher priority than ThreadB, ThreadA will be running and ThreadB will be waiting. However, sometimes Java will choose to run ThreadB to avoid starvation. **Starvation** occurs when a Thread cannot make any progress because of the priorities of other Threads. The result of starvation can be similar to creating an infinite loop—the program runs continuously without completing because one or more threads never get the opportunity to execute.

The ultimate form of starvation is called deadlock. Deadlock occurs when two Threads must wait for each other to do something before either can progress.

Next you will set the priorities of some Threads and observe the effects of those settings.

To demonstrate Thread priorities:

1. Open the **ShowThread.java** file in your text editor, and immediately save it as **ShowThread2.java**.
2. Change the class name and the constructor name to **ShowThread2**.
3. Change the value of the rep variable to `5000`. This step is necessary on most systems so that there will be enough executions of the output loop for you to observe the effect of the priority settings.
4. Save the file and compile it using the `javac` command.
5. Open the **DemoThreads.java** file in your text editor, and immediately save it as **DemoThreadsPriority.java**.
6. Change the class header to `class DemoThreadsPriority`.
7. Change the declaration of the three Threads to use the newly created ShowThread2 class. In other words, the declarations become the following:

```
ShowThread2 showA = new ShowThread2('A');
ShowThread2 showB = new ShowThread2('B');
ShowThread2 showC = new ShowThread2('C');
```

8. Position your insertion point to the right of the line with the third ShowThread2 declaration (the showC declaration), and then press **[Enter]** to start a new line.
9. Type the following lines to set the showB Thread's priority to 4 and the showC Thread's priority to 6:

```
showB.setPriority(Thread.NORM_PRIORITY-1);
showC.setPriority(Thread.NORM_PRIORITY+1);
```

10. Save the file, and then compile and run the program. The output is more than will fit on your screen, and the middle of the execution looks similar to Figure 17-12. Although the As, Bs, and Cs are somewhat intermingled, the showC Thread, with a priority of 6, finishes before showA and showB. Notice that although the highest priority Thread (showC) finishes first, and the next highest (showA) finishes second, showB is allowed some processor time before showA finishes.

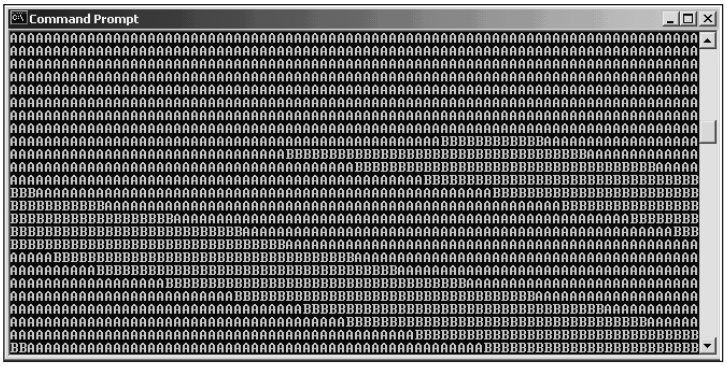

Figure 17-12 Middle part of the DemoThreadsPriority program output

USING THE RUNNABLE INTERFACE

You can create a Thread subclass by inheriting from the Thread class, but this approach won't work if you want your Thread subclass to inherit from another class as well. You have already learned that Java does not allow a class to inherit from more than one superclass. So, for example, if you want to create a JApplet that can run as a Thread, you cannot inherit from both JApplet and Thread; instead you must implement an interface. You can implement the Runnable interface as an alternative to inheriting from the Thread class.

You learned about implementing interfaces in Chapter 12.

You can write an applet that acts as a timer; the applet counts and displays seconds as they pass. You declare an integer variable—for example, `int secs = 0;`. The applet's paint() method contains a statement that adds one to the secs variable, and then displays the current count of seconds.

To let the applet run indefinitely, you can create an infinite loop that repaints the screen every 1,000 milliseconds (or every second), as in the following code:

```
while(true)  // execute forever
       repaint();
try
{
       clock.sleep(1000);
}
catch(InterruptedException e)
{
}
```

The problem with this approach is that as long as the infinite while loop is running, the CPU is occupied and cannot perform any other actions, such as carrying out the repaint() method. Instead of displaying a counter, the applet appears to freeze the screen.

The solution is to place the `while` loop in a Thread that can share time with the operating system's default Thread in which the applet runs. Next you will produce a JApplet that uses this approach as you create a JApplet that shares processor time with the clock Thread that hosts it.

To create a JApplet that implements the Runnable interface:

1. Open a new file in your text editor, and then type the following first few lines of the TimerApplet applet:

```
import javax.swing.*;
import java.awt.*;
public class TimerApplet extends JApplet
  implements Runnable
{
```

2. Define three JLabels. The first and third contain literal Strings. The second will display the changing time count.

```
private JLabel label1 = new JLabel("Time is passing.");
private JLabel label2 = new JLabel();
private JLabel label3 = new
  JLabel("Plan your event today.");
```

3. Declare a Container, a variable to hold the seconds, and a Thread named clock:

```
private Container c = getContentPane();
private int secs = 0;
private Thread clock;
```

4. The init() method establishes the layout manager, adds the three JLabels to the content pane, and declares a Thread object named clock. When the applet is initialized, the clock Thread will have value `null` because the constructor has not been called yet, so within the init() method, you instantiate the Thread. The run() method of the TimerApplet will execute when the Thread is created because the `this` in `clock = new Thread(this);` refers to "this" applet. Finally, add start() to the Thread.

```
public void init()
{
  c.setLayout(new FlowLayout());
  c.add(label1);
  c.add(label2);
  c.add(label3);
  if(clock == null)
       clock = new Thread(this);
  clock.start();
}
```

5. The applet contains the following run() method because the applet implements Runnable. Within the run() method, you place the infinite loop that calls repaint() every second and provides for the InterruptedException thrown by the sleep() method.

```
public void run()
{
  while(true)
  {
      repaint();
      try
      {
             clock.sleep(1000);
      }
      catch(InterruptedException e)
      {
      }
  }
}
```

6. Add the following paint() method that adds one to the secs variable (because paint() is called once every second) and updates the displayed count in label2:

```
public void paint(Graphics gr)
{
  ++secs;
  label2.setText("Time: " + secs);
}
```

7. Add a closing curly brace for the applet.
8. Save the applet as **TimerApplet.java** in the Chapter.17 folder on your Student Disk. Compile the applet using the **`javac`** command.
9. Open a new file in your text editor, and then type the following HTML document to host your applet:

```
<HTML>
<APPLET CODE = "TimerApplet.class"
    WIDTH = 300 HEIGHT = 50>
</APPLET>
</HTML>
```

10. Save the HTML file as **TestTimer.html** in the Chapter.17 folder, and then execute it using the **`appletviewer`** command. The output appears similar to Figure 17-13. As you watch the timer, note that the time value changes every second.
11. Click the **Close** button to close the Applet Viewer window.

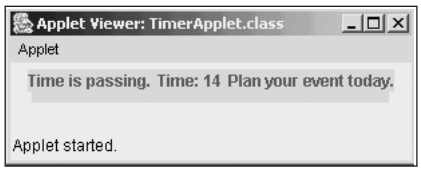

Figure 17-13 Output of the TimerApplet program

The TimerApplet that you created uses the default system Thread, and your clock object uses its own Thread. You could not have created this applet without using the power of multithreading.

Creating an Animated Figure

Cartoonists create animated films by drawing a sequence of frames or cells. These individual drawings are shown to the audience in rapid succession to give the illusion of natural movement. You create computer animation using the same techniques. If you display computer images as fast as your CPU can process them, you might not be able to see anything. Most computer animation employs the Thread class sleep() method to pause for short periods of time between animation cells, so the human brain has time to absorb each image's content.

Artists often spend a great deal of time creating the exact images they want to use in an animation sequence. As a much simpler example, Event Handlers Incorporated wants you to create a stick figure whose arm moves up and down. The only difference between creating a stick figure and a more complex graphic image is in the amount of time and degree of artistic talent you have; the programming skills you use are the same.

You begin to create computer animation by using graph paper to sketch a figure, as shown in Figure 17-14.

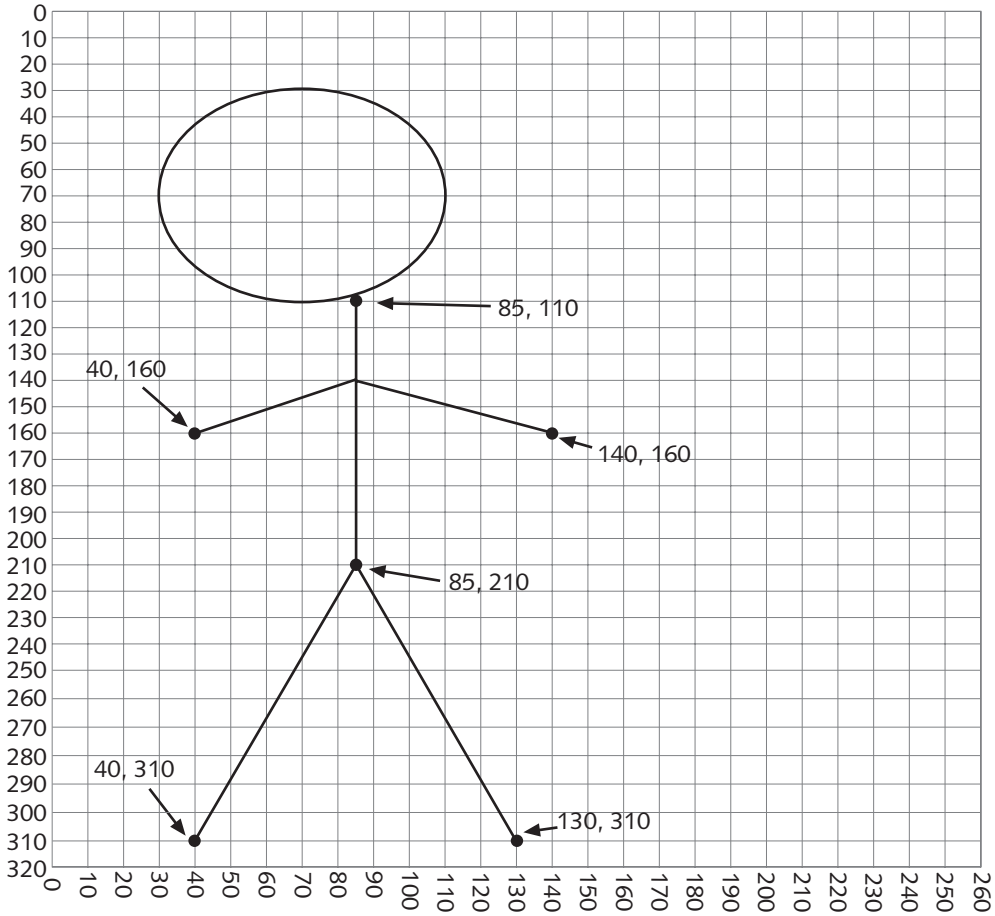

Figure 17-14 Sketch of a stick figure

To create the illusion that this figure's arm moves, you create a second sketch in which the arm is slightly raised from its position in the first sketch. In your third drawing the arm is slightly higher, and so on. Figure 17-15 shows four potential arm positions for the stick figure.

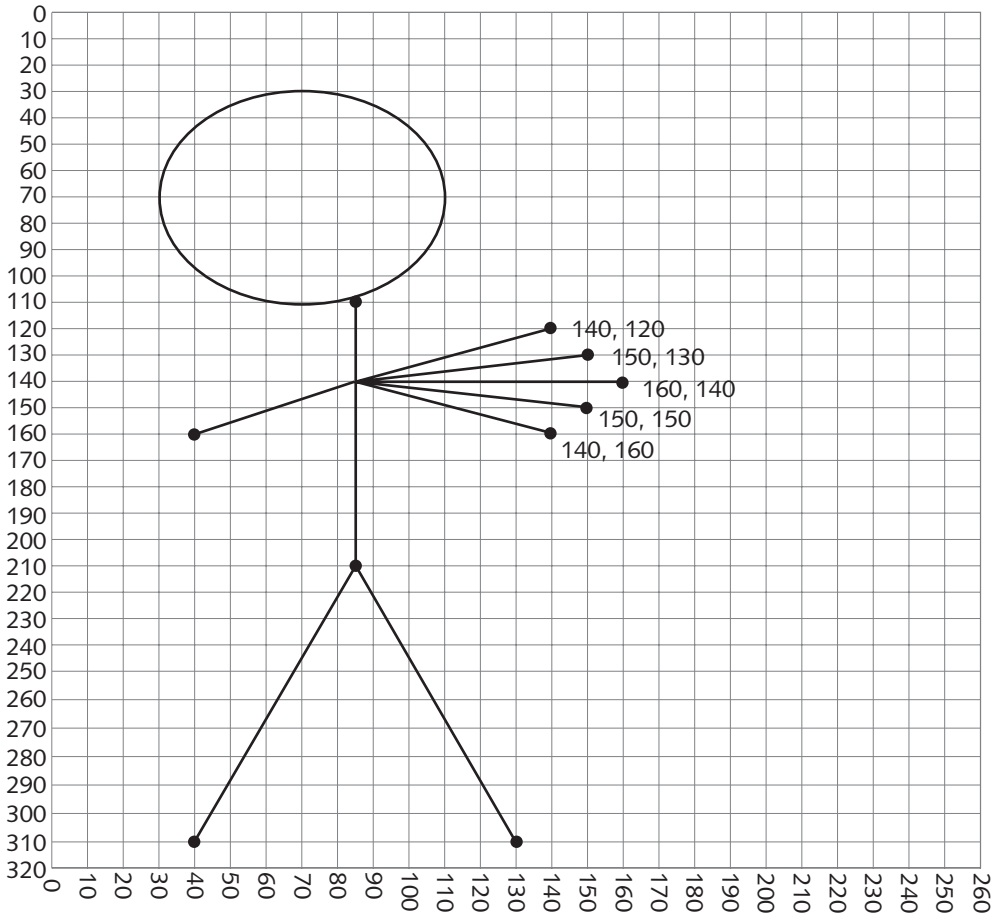

Figure 17-15 Sketch of four arm positions for the stick figure

Each time you draw the stick figure on the screen, the arm will appear slightly higher, giving the viewer the illusion of movement. The head, body, and legs of the stick figure never change. You can draw the stick figure's body using constant values for the drawing coordinates, but you need variables to store the horizontal and vertical positions of the end of the moving arm. One efficient approach is to store the x- and y-coordinates for the end location of each of the arm positions in parallel arrays, as in the following example:

```
int[] horiz = {140, 150, 160, 150, 140};
int[] vert =  {160, 150, 140, 130, 120};
```

Each time you redraw the stick figure, you can increase a subscript and use the next horizontal and vertical coordinate pair to indicate the end of the figure's arm. When you have used all five arm coordinates, you can reduce the subscript to a value of zero, and then begin again. Next you will create a stick figure and use an array of coordinates to animate its arm.

To create an animated stick figure for Event Handlers Incorporated:

1. Open a new file in your text editor, and then type the following first few lines of the AnimatedFigure class. (You will inherit from Applet instead of from JApplet because Applets' surfaces are automatically updated when repainted.)

```
import java.applet.*;
import java.awt.*;
public class AnimatedFigure extends Applet
{
```

2. Declare an integer index that you can use to access the horizontal and vertical position arrays. Also declare the arrays themselves, which are loaded with values taken from the preliminary sketches:

```
private int index = 0;
int[] horiz = {140,150,160,150,140};
int[] vert =  {160,150,140,130,120};
```

3. Add the following variable to store the sleep time. If you want to speed up or slow down the figure later, it will be convenient to change the value in this variable. For now, set the sleep time to 100 milliseconds as follows:

```
private int sleep = 100;
```

4. In the applet's start() method, initialize the index to 0:

```
public void start()
{
  index = 0;
}
```

5. Within the applet's paint() method, add the following code to draw the figure's head, body, and two legs. Use the sketch as a reference for the coordinates:

```
public void paint(Graphics gr)
{
  gr.drawOval(30,30,80,80);
  gr.drawLine(85,110,85,210);
  gr.drawLine(85,210,40,310);
  gr.drawLine(85,210,130,310);
```

6. You can create the figure's left arm by using the coordinates in the sketch. However, creating the right arm is more complicated. Although the right arm always starts at the same position on the body, the hand end of the arm might be in any one of four positions taken from the horiz and vert arrays. After you draw the right arm using position 0 from each array, you want to increase the index, so you will use the next pair of coordinates the next time you draw the arm. When the index eventually exceeds the highest subscript allowed, reset the index to 0 so the process of drawing each of the arm positions can start over.

```
gr.drawLine(85,140,40,160);
gr.drawLine(85,140,horiz[index],vert[index]);
++index;
if(index == horiz.length)
     index = 0;
```

7. Add the following `try` block to make the thread sleep for the designated amount of time, and then add a `catch` block to handle the InterruptedException.

```
try
{
  Thread.sleep(sleep);
}
catch(InterruptedException e)
{
}
```

8. The last step in the paint() method is to call the repaint() method. This restarts paint() to draw the figure with a new arm position, update the index, and sleep. Add two curly braces—one to close the paint() method and one to close the class.

```
    repaint();
  }
}
```

9. Save the file as **AnimatedFigure.java** in the Chapter.17 folder on your Student Disk, and then compile it.

10. Open a new file in your text editor and create the following HTML document to host the applet:

```
<HTML>
<APPLET CODE = "AnimatedFigure.class"
  WIDTH = 350 HEIGHT = 400>
</APPLET>
</HTML>
```

11. Save the HTML file as **TestAnim.html**, and then use the **appletviewer** command to run the applet. The stick figure's arm waves up and down. (You might notice some flickering on the screen; you will learn to eliminate any flickering in the next section.) One frame of the output appears in Figure 17-16.

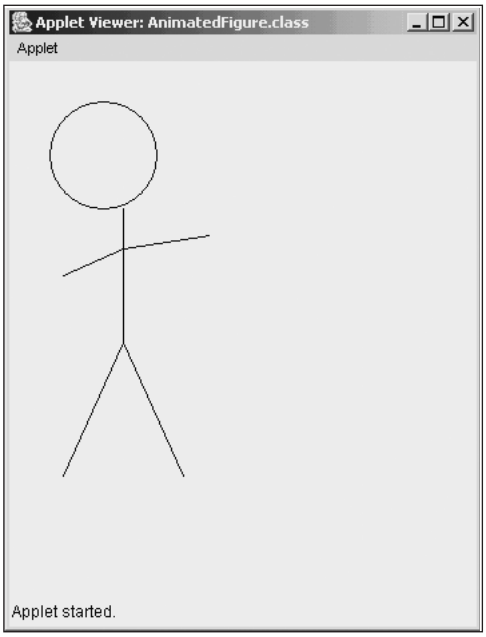

Figure 17-16 Output of the AnimatedFigure applet

12. Close the Applet Viewer by clicking its **Close** button.

Next you will add some text to the applet. Event Handlers Incorporated wants the company name to move up and down on the screen, as though the stick figure is using the name as a yo-yo. You will also add text which will appear in a fixed position in the applet.

To add moving text to the applet:

1. Open the **AnimatedFigure.java** file in your text editor if it is not still open, and then immediately save it as **AnimatedFigure2.java** in the Chapter.17 folder on your Student Disk.
2. Change the class name to `AnimatedFigure2`.
3. Within the paint() method, position your insertion point at the end of the last drawLine() method call (the statement that draws the moving arm), and then press **[Enter]** to start a new line.
4. To draw the "Event Handlers" name as though it is moving along with the stick figure's arm, indicate its vertical position as 30 pixels below the arm's current vertical position as follows:

```
gr.drawString("Event Handlers",180, vert[index] + 30);
```

5. Add the following text lines at lower positions on the next two lines:

```
gr.drawString("Plan with us once",170,260);
gr.drawString("Like a yo-yo,
  you will come back!",170,280);
```

6. Save the file, and then compile it using the **javac** command.

7. Open the **TestAnim.html** file in your text editor. Change the APPLET CODE reference to **"AnimatedFigure2.class"**, and then save the HTML document as **TestAnim2.html** in the Chapter.17 folder on your Student Disk.

8. Use the **appletviewer TestAnim2.html**command to run the applet. The stick figure appears to use the Event Handlers company name as a yo-yo. One snapshot of the output is shown in Figure 17-17.

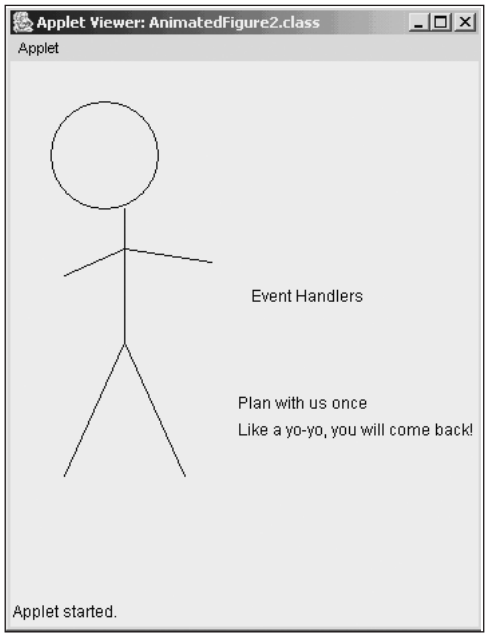

Figure 17-17 Output of the AnimatedFigure2 applet

9. Close the Applet Viewer window.

REDUCING FLICKERING

When you create an applet with animation, your screen might flicker. To understand why screens flicker, recall the applet life cycle that you learned about in Chapter 9.

The faster your processor speed is, the less flickering you will see. The stick figure you created does not require a lot of drawing, so even with a processor of moderate speed you may not experience flickering. However, if you create a detailed drawing, you might experience flickering even with a fast processor, and you will want to employ the techniques you learn in this section.

When you change a drawing in an applet, such as when you reposition the stick figure's arm and the moving text String, you call the repaint() method to repaint the applet surface. The repaint() method calls the update() method, which clears the viewing area, and then calls the paint() method, which contains the instructions for drawing the figure and String in their new positions. If the repaint() method did not call update() to clear the screen, then all previous versions of the applet would remain visible. After the first five passes through the paint() method you would see all five of the figure's arms and all five messages, as shown in Figure 17-18.

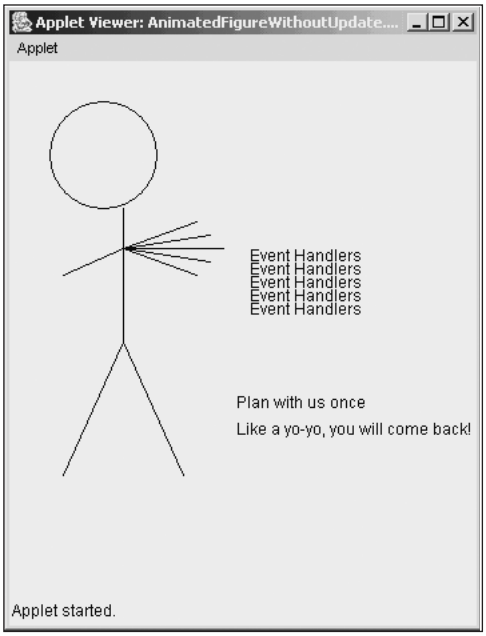

Figure 17-18 AnimatedFigure without the update() method

Your program must clear the screen so only one version of the arm and message appears at a time. However, clearing the screen takes enough time that your eye detects it; this results in the flickering you see on your screen. One way to reduce or eliminate flickering is to override the applet's update() method so that the viewing area is not cleared. After all, every time you call update() to clear the screen, the head, body, legs, and left arm of the stick figure and the two messages are immediately redrawn in the same positions from which they were just cleared. It is more efficient to draw the unchanging

parts of the screen just once when the applet starts, and then constantly redraw only those portions of the screen that change.

The trick for erasing a portion of a drawing on the screen is to use the applet's background color to redraw the portion of the screen that you want to erase. When you create graphics using the background color, the graphics seem to disappear. In other words, to appear to move the stick figure's arm, you draw over its old arm in the background color, and then draw the new arm position in the previous drawing color.

For example, if the applet background color is white and the drawing color is black, one way to draw over the old black arm line is to save the arm coordinates after you draw the arm. Then before you draw a new black arm line, draw over the old line in white, using the saved coordinates.

To reduce flickering in the AnimatedFigure2 applet:

1. If it is not open, open the **AnimatedFigure2.java** file in your text editor. Immediately save the file as **AnimatedFigure3.java** in the Chapter.17 folder on your Student Disk.
2. Change the class name from AnimatedFigure2 to `AnimatedFigure3`.
3. Position the insertion point at the end of the statement that declares the vert array, and then press **[Enter]** to start a new line. Create the following two new integer variables that will hold the previous arm coordinates. Initialize these variables to the first pair of coordinates. Then create a variable to indicate whether the paint() method is on its first or a subsequent execution.

```
int oldx = horiz[0], oldy = vert[0];
boolean firstTime = true;
```

4. Position your insertion point to the right of the closing curly brace for the start() method, and then press **[Enter]** to start a new line. Type the following code to override the applet's automatic update() method. The replacement update() method calls paint(), passing along the Graphics object received from the repaint() method.

```
public void update(Graphics gr)
{
  paint(gr);
}
```

5. Delete the current paint() method statements up to, but not including, the statement `++index;`. Begin the following replacement paint() method by drawing the head, body, legs, left arm, and constant text only if this is the first pass through the paint() method. Conclude the `if` block by setting the firstTime variable to `false`.

```
public void paint(Graphics gr)
{
       if(firstTime)
       {
          gr.drawOval(30,30,80,80);
          gr.drawLine(85,110,85,210);
          gr.drawLine(85,210,40,310);
          gr.drawLine(85,210,130,310);
          gr.drawLine(85,140,40,160);
          gr.drawString("Plan with us once",170,260);
          gr.drawString("Like a yo-yo, you will come
            back!",170,280);
          firstTime = false;
       }
```

6. Continue adding the following paint() method statements that execute every time the paint() method executes—that is, whether or not it is the first execution. First, set the drawing color to the background color, and redraw the arms and moving String in the background color to make them invisible.

```
gr.setColor(getBackground());
gr.drawLine(85,140,oldx, oldy);
gr.drawString("Event Handlers",180, oldy + 30);
```

7. Add the following code to change the drawing color back to the usual drawing color, and then to draw the new arm and the new moving message in their new positions:

```
gr.setColor(getForeground());
gr.drawLine(85,140,horiz[index],vert[index]);
gr.drawString("Event Handlers",180, vert[index] + 30);
```

8. Before increasing the index for the next execution of the paint() method, use the following code to save the current arm coordinates by storing them in the oldx and oldy variables:

```
oldx = horiz[index];
oldy = vert[index];
```

9. Save the file, and then compile it using the **javac** command.

10. Open the **TestAnim2.html** file in your text editor. Change the APPLET CODE reference to **"AnimatedFigure3.class"**, and then save the HTML document as **TestAnim3.html** in the Chapter.17 folder on your Student Disk.

11. Run the applet using the **appletviewer** command. When the applet runs, you will not see any flickering. (If your animation appears to run too fast or too slowly, change the value of the sleep variable.)

12. Minimize the Applet Viewer window, and then restore it. You can see only a disembodied moving arm and message. Close the Applet Viewer window.

The flickering disappears in the AnimatedFigure3 applet because the update() method no longer redraws the entire screen. However, a problem occurs when you minimize, restore, or move the applet—the figure no longer is visible and only the moving arm and message remain. When you start the applet, firstTime is set to `true`, the figure is drawn, and firstTime is set to `false`. When you minimize the applet, it is still running. When you restore the applet, the applet is redrawn, but because the applet is still running, firstTime is already set to `false`. The head and other body parts are drawn only when firstTime is true, so the applet now shows a disembodied arm. Next you will correct this problematic development.

To correct the applet problem:

1. Open the **AnimatedFigure3.java** file in your text editor, and immediately save it as **AnimatedFigure4.java**.
2. Change the class name to `AnimatedFigure4`.
3. Within the start() method, position your insertion point to the right of the statement that sets the index equal to 0, and then press **[Enter]** to start a new line.
4. Add the statement `firstTime = true;`. Now, every time the applet restarts, the applet will redraw the head and body.
5. Save the file, and then compile it using the `javac` command.
6. Open the **TestAnim3.html** file in your text editor. Change the APPLET CODE reference to `"AnimatedFigure4.class"`, and then save the HTML document as **TestAnim4.html** in the Chapter.17 folder on your Student Disk.
7. Run the applet using the `appletviewer` command. The applet runs the same as before. Minimize the Applet Viewer window and then restore it. The entire figure and complete set of messages are visible, because they are redrawn every time the start() method executes.
8. Close the Applet Viewer window.

Using Pre-Drawn Animated Image Objects

If your artistic talent is limited to drawing stick figures, you can substitute a variety of more sophisticated, pre-drawn animated images to achieve the graphic effects you want within your applets. In Chapter 10, you used the getImage() method to load a stored image into an applet. You can also use getImage() to load animated files in your programs. When you call getImage(), an image is loaded in a separate thread of execution; this allows program execution to continue while the image loads. When you used getImage() in Chapter 10, you did not create a thread—Java created one for you. Because loading images can be a time-consuming task, it is a great advantage that Java automatically creates a separate thread

for them. Recall that the getImage() method requires two arguments: a URL object and the name of an image file.

Next you will load a pre-drawn animated .gif file into an applet and execute it. The .gif file includes 16 frames that display a falling egg that Event Handlers Incorporated is using in an advertising slogan to give Event Handlers a "crack" at planning the customer's next party.

To demonstrate loading a pre-drawn animated image into an applet:

1. Open a new file in your text editor, and then enter the following first few lines of an applet that will load a pre-drawn animated image:

```
import java.applet.*;
import java.awt.*;
public class LoadImage extends Applet
{
```

2. Declare an Image object that will represent the falling egg as follows:

```
private Image egg;
```

3. The Chapter.17 folder on your Student Disk contains an animated .gif file named egg.gif. Type the following init() method, which loads the egg.gif file in the applet:

```
public void init()
{
  egg = getImage(getDocumentBase(),"egg.gif");
}
```

4. Type the following paint() method, which draws the egg Image at location 1,1 within the applet, and also draws two Strings:

```
public void paint(Graphics g)
{
  g.drawImage(egg,1,1,this);
  g.drawString("Planning your next party?",100,50);
  g.drawString
    ("Give Event Handlers a crack at it.",100,   300);
}
```

5. Type a closing curly brace for the applet.

6. Save the file as **LoadImage.java** in the Chapter.17 folder on your Student Disk, and then compile the applet using the `javac LoadImage.java` command.

7. Open a new file in your text editor, and then create the following HTML document:

```
<HTML>
<APPLET CODE = "LoadImage.class"
  WIDTH = 300 HEIGHT = 350>
</APPLET>
</HTML>
```

8. Save the HTML document as **TestImage.html** in the Chapter.17 folder of your Student Disk.
9. Execute the HTML file using the **`appletviewer`** command. The animation shows an egg that falls and cracks open at the bottom of the applet viewing area. Figure 17-19 shows the final frame of the animation.

Figure 17-19 LoadImage applet

10. Watch the egg fall and crack open repeatedly. When you are ready, close the Applet Viewer window.

Many animated images are available on the Web for you to use freely. Use your search engine to search for keywords such as gif files, jpeg files, and animation to find sources for shareware files. The egg file was found at *www.mediabuilder.com*.

17

Understanding Garbage Collection

The Java programming language supports a garbage collection feature that you seldom find in other programming languages. The **garbage collector** provides for the automatic cleanup of unnecessarily reserved memory.

With many programming languages, when you allocate memory from the memory that is available (often known as the **heap**), you must purposely deallocate it or your system's performance begins to slow as less memory is available. The Java garbage collector

automatically sweeps through memory looking for unneeded objects and destroys them. Out-of-scope objects or objects with `null` references are available for collection. Partially constructed objects that throw an Exception during construction are also available for collection.

You first learned about garbage collection in Chapter 7 when String memory addresses were introduced. Recall that when you change a String's contents, the previous contents still exist in memory. These unused characters would needlessly occupy memory if they were not collected and disposed of.

Garbage collection works by running as a very low-priority Thread. Typically, this Thread runs only if available memory is very low. That is, all other Threads must be delayed by memory limitations before the garbage collector gets the opportunity to run.

You cannot prevent the garbage collector from running, but you can request that it run by using the statement `System.gc();`. Suppose you write a program that counts seconds and uses a String named counter to hold the values "one", "two", and so on. When you assign "two" to counter, the memory that holds the character string "one" becomes garbage. Instead of allowing the useless strings to accumulate in memory, you might want to use the `System.gc();` statement. Using this statement does not force the garbage collector to run; it is only a request to the system to schedule the garbage collector. The garbage collector will run when all other Threads are delayed.

Putting Animation in a Web Browser Page

If you browse the Internet, you probably have seen examples of animation on Web pages. Web page creators use animation to attract your attention and to make their pages more appealing. Next you will create an applet that displays a moving word. As a stand-alone applet it is interesting, but using the applet becomes much more interesting when you run several versions of it in a Web browser at the same time.

Each version of the applet displays the single String "Party" at x- and y-coordinates. After each display, you use the Thread class sleep() method to pause the animation. You then erase the String by drawing it in the background color; you then redraw it in a new position. The new position is three pixels to the right and down from the previous String, so the illusion is that the String is moving down and to the right. When the String reaches the right edge of the viewing area, you begin to subtract three pixels from the horizontal and vertical coordinates, so the String appears to reverse course and move up and to the left.

To create the first of three applets containing a moving word that you will run in the browser:

1. Open a new file in your text editor, and then type the following opening lines for the BouncingParty1 applet:

```
import java.applet.*;
import java.awt.*;
public class BouncingParty1 extends Applet
{
```

2. Establish variables for the step value increase for each drawString() and for the x- and y-coordinates as follows:

```
private int step = 3;
private int x = 10, y = 10;
```

3. Also establish variables to hold the maximum screen positions, the previous screen positions, and the sleep interval as follows:

```
private int maxX = 100, maxY = 100;
int oldx, oldy;
private int sleep = 50;
```

4. Type the following update() method that calls the paint() method (so Java doesn't call its own update() method, which would clear the viewing area and increase flickering):

```
public void update(Graphics gr)
{
  paint(gr);
}
```

5. Begin the following paint() method, which draws "Party" using the background color at the previous x- and y-coordinates, and then draws the String using the foreground color at the new coordinates:

```
public void paint(Graphics gr)
{
  gr.setColor(getBackground());
  gr.drawString("Party", oldx, oldy);
  gr.setColor(getForeground());
  gr.drawString("Party", x, y);
```

6. Save the x and y values in the oldx and oldy variables. Then increase x and y by the step value as follows:

```
oldx = x;
oldy = y;
x += step;
y += step;
```

7. When the String approaches the edge of the applet viewing area, you want to reverse the direction of movement. You can accomplish this by changing the step value to its negative equivalent as follows:

```
if(x < 10 || x > 90)
   step = -step;
try
{
   Thread.sleep(sleep);
}
catch(InterruptedException e)
{
}
repaint();
```

8. Add the closing curly brace for the paint() method and the closing curly brace for the class.
9. Save the file as **BouncingParty1.java** in the Chapter.17 folder on your Student Disk, and then compile the class using the `javac` command.
10. In a new file in your text editor, create the following HTML document to test the class:

```
<HTML>
<APPLET CODE = "BouncingParty1.class"
  WIDTH = 120 HEIGHT = 120>
</APPLET>
</HTML>
```

11. Save this file as **TestParty.html** in the Chapter.17 folder, and then use the `appletviewer` command to test the class. The word "Party" moves up and down the screen. Close the Applet Viewer window.

The simple BouncingParty1 applet is interesting, but it won't hold your attention for very long. If you create several similar applets and run them at the same time, the output will be more noteworthy. Next you will create two more applets that display a bouncing "Party" message. You will alter each applet so that the displayed String starts in a slightly different position within each applet.

To create applets in which the moving String starts in a different position:

1. Open the **BouncingParty1.java** applet in your text editor, and then immediately save it as **BouncingParty2.java**.
2. Change the class name to `BouncingParty2`.
3. Change the beginning x and y variable values from 10 to **40** so the statement becomes `private int x = 40, y = 40;`.
4. Save the file, and then compile it using the `javac` command.
5. To create the third applet, save the BouncingParty2.java file as **BouncingParty3.java**. Change the class name to `BouncingParty3`. Change the values for both the x and y variables to **70**. Save the file and compile it.

You now have three similar applets that each display a bouncing "Party" String; the only difference is that each initially places the String in a different position. In the next set of steps you will incorporate these three applets into an HTML document and view it in your Web browser. To make the HTML document more interesting, you will add two headings to the document.

HTML provides six levels of headings named `H1` through `H6`. `H1` headings are the largest, and `H6` are the smallest. Just as with the `<APPLET>` and `</APPLET>` tags you have used in your HTML documents, the heading tags come in pairs. You place text you want to display between a pair, such as `<H1>` and `</H1>`.

HTML document authors seldom use H6 headings; they are quite small and difficult to read. You first learned about HTML tags in Chapter 9.

To create an HTML document to use with your browser to view the three applets:

1. Open a new file in your text editor, and then enter the following HTML document, which consists of a large heading, three applets, and another heading:

```
<HTML>
<H1>We keep your party moving</H1>
<APPLET CODE = "BouncingParty1.class"
    WIDTH = 120 HEIGHT = 120>
</APPLET>
<APPLET CODE = "BouncingParty2.class"
  WIDTH = 120 HEIGHT = 120>
</APPLET>
<APPLET CODE = "BouncingParty3.class"
  WIDTH = 120 HEIGHT = 120>
</APPLET>
<H1>at Event Handlers Incorporated</H1>
</HTML>
```

2. Save the file as **TestParties.html** in the Chapter.17 folder on your Student Disk.
3. Open a Web browser such as Netscape Navigator or Microsoft Internet Explorer. (You do not need to connect to the Internet to complete this step.) From your browser's main menu, select **File**, click **Open**, and either choose the Browse button to locate your HTML document, or type its complete path; for example, **A:\Chapter.17\TestParties.html**, and then press **[Enter]**. Alternately, you can locate the TestParties.html file using Explorer or My Computer, and then double-click the file icon to open it in your default browser. The three applets run within the HTML document, as shown in Figure 17-20. Depending on your browser, the background colors for the three applets might differ.

Instead of opening the HTML document using the File menu, you can type the path to your HTML document in your browser's Location field (where you type URLs to visit Web sites), and then press **[Enter]**.

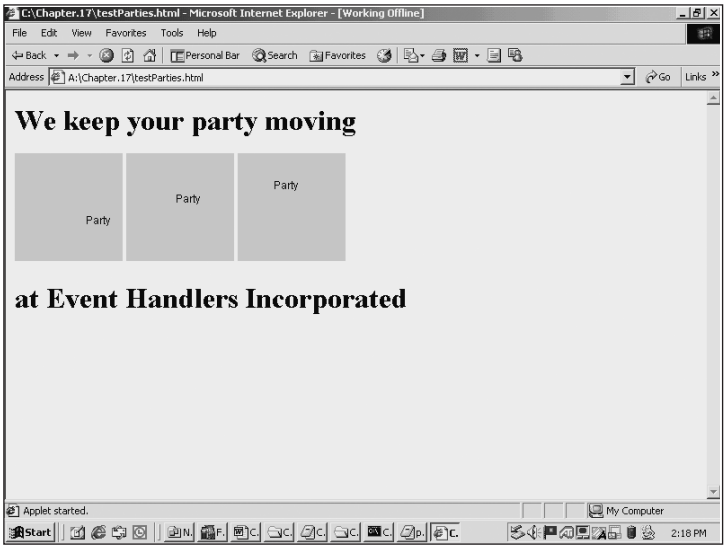

Figure 17-20 Three applets running in the browser

4. Close your Web browser.
5. Experiment with modifying the step variable value and the sleep time in the BouncingParty applets and observe the results in your browser.
6. Try to modify one of the applets so that the "Party" message moves from upper right to lower left by altering the x- and y-coordinates so that as x gets bigger, y gets smaller.
7. Experiment with modifying the HTML document to hold additional applets.

Chapter Summary

- A thread is the flow of execution of one set of program statements. The Java programming language allows you to launch, or start, multiple threads. You use multithreading to make your programs perform better.
- You can create threads by extending the Thread class, which is defined in the java.lang package. The Thread class contains a method named run() that you override to tell the system how to execute the Thread.

- A Thread can exist in one of five states during its life: new, ready, running, inactive, or finished.
- You use the sleep() method to pause a Thread for a specified number of milliseconds. When you use the sleep() method, you must `catch` an InterruptedException.
- Every Thread in Java has a priority, or rank, in terms of preferential access to the operating system's resources. The Thread class contains three constants: MIN_PRIORITY, NORM_PRIORITY, and MAX_PRIORITY.
- You can implement the Runnable interface as an alternative to inheriting from the Thread class.
- You create computer animation by displaying Images in rapid sequence. Most computer animation employs the Thread class sleep() method to pause for short periods of time between animation cells; then the viewer has time to absorb each image's content.
- Clearing the screen before new images are drawn or added takes enough time that your eye detects it. As a result, the screen appears to flicker. One way to reduce or eliminate flickering is to override the applet's update() method so that the viewing area is not cleared. To "erase" a portion of a drawing on the screen, you use the applet's background color to redraw the portion of the screen that you want to erase.
- The applet method getImage() loads a stored image into an applet. The getImage() method requires two arguments: a URL object and the name of an image file.
- Garbage collection runs as a low-priority Thread and provides for the automatic cleanup of unnecessarily reserved memory.
- Web page creators use animation to attract your attention and to make their pages interesting. You can use HTML heading tags to display text in Web documents.

Review Questions

1. A thread is the _____ one set of program statements.
   a. amount of memory occupied by
   b. flow of execution of
   c. machine language code for
   d. area of memory occupied by
2. A modern computer with a single CPU can execute _____ statement(s) at a time.
   a. one
   b. two
   c. at least several dozen
   d. at least several thousand

3. If you use a computer with a single processor, you can execute _____ concurrently.
   a. only one thread
   b. several threads
   c. any number of threads
   d. You cannot execute multiple threads on a single-processor computer.
4. You can create threads by _____ the Thread class.
   a. making a copy of
   b. instantiating
   c. extending
   d. overriding
5. You override the _____ method to tell the system how to execute a Thread.
   a. thread()
   b. execute()
   c. run()
   d. start()
6. You achieve _____ by starting more than one Thread object.
   a. polythreading
   b. bithreading
   c. multithreading
   d. buffered threading
7. Which of the following states is not possible for a Thread?
   a. ready
   b. finished
   c. altered
   d. new
8. A Thread's rank in terms of preferential access to the operating system's resources is its _____.
   a. priority
   b. prerogative
   c. supremacy
   d. license

9. If you do not assign a priority to a Thread object, it assumes a priority of _____ by default.
   a. 0 (zero)
   b. 1
   c. 5
   d. 10
10. Which of the following Threads would most likely run first?
   a. a finished Thread with priority 10
   b. a suspended Thread with priority 5
   c. a runnable Thread with priority NORM_PRIORITY
   d. a runnable Thread with priority 2
11. Java sometimes chooses to run a low-priority Thread to avoid _____.
   a. construction
   b. death
   c. starvation
   d. sedation
12. You can use the Runnable _____ to inherit Thread methods.
   a. interface
   b. method
   c. mode
   d. formula
13. When you define a Thread as `private Thread someThread;`, the value of someThread is _____.
   a. zero
   b. `null`
   c. `false`
   d. unknown
14. You would see all versions of an animated drawing at the same time if you did not call an applet's _____ method.
   a. clear()
   b. paint()
   c. draw()
   d. update()

15. A technique to reduce screen flickering is to _____.
    a. redraw the entire image with each call to the paint() method
    b. redraw only those portions of the screen that actually change
    c. call the paint() method directly instead of repaint()
    d. buy a more expensive monitor
16. To create the illusion that an object moves up and to the left, you would _____.
    a. increase its x-coordinate and decrease its y-coordinate
    b. decrease its x-coordinate and increase its y-coordinate
    c. increase both its x- and y-coordinates
    d. decrease both its x- and y-coordinates
17. The applet method getImage() _____.
    a. allows you to draw an image on the screen
    b. loads a stored image into an applet
    c. produces a copy of a displayed image
    d. returns the name of an image when you point to it with your mouse
18. When you use getImage(), _____.
    a. you must create a thread in which it can run
    b. you must not create a thread in the same file
    c. Java automatically creates a thread for you
    d. as many threads are launched as there are frames within the image
19. HTML provides _____ levels of headings.
    a. two
    b. six
    c. 12
    d. an unlimited number of
20. Image files usually use _____ as filename extensions.
    a. .jpeg or .gif
    b. .exe or .doc
    c. .www or .url
    d. .com or .net

EXERCISES

1. a. Create a Thread class named Friend whose constructor accepts a friend's name. The Friend class run() method displays a single space 499 times before displaying the friend's name once. Write a program that instantiates three Threads to wihich you pass your first name and the first names of two friends. Start the Threads and observe which Thread wins the race. (When you run the program several times, a different Friend will win by different margins represented by the spaces between the names.) Save the Friend class as **Friend.java** and the program as **NameRace.java** in the Chapter.17 folder on your Student Disk.

   b. Set different priorities for the three Thread objects you created in Exercise 1a, and then run the program again. Save the new program as **NameRaceWithPriorities.java** in the Chapter.17 folder of your Student Disk.

2. Create two classes that extend Thread—LovesMeThread and LovesMeNotThread. Each Thread displays the phrase its name implies 1000 times. Write a program to start the two Threads; the final message is the answer to your question! Save the classes as **LovesMeThread.java** and **LovesMeNotThread.java** and the program as **LoveQuestion.java** in the Chapter.17 folder on your Student Disk.

3. Create a class named RaceHorse that extends Thread. Each RaceHorse has a name and a run() method that displays the name 5000 times. Write a program that instantiates two RaceHorse objects. The last RaceHorse to finish is the loser. Save the class as **RaceHorse.java** and the program that creates the RaceHorse objects as **Race.java** in the Chapter.17 folder on your Student Disk.

4. Create a CharacterThread class that displays a single character 500 times. Write a program that creates five CharacterThreads, each of which displays a different character. Give two Threads the minimum priority, two Threads the maximum priority, and one Thread the default priority. Run the program and observe the results. Save the class as **CharacterThread.java** and the program as **FiveThreads.java** in the Chapter.17 folder on your Student Disk.

5. a. Write an applet that displays a stick figure doing jumping jacks. Save the program as **Exercise.java** in the Chapter.17 folder on your Student Disk.

   b. Add text to the Exercise applet so it serves as an advertisement for a health club. Save the new program as **ExerciseAd.java** in the Chapter.17 folder on your Student Disk.

6. Write an applet that shows a bouncing ball by drawing a circle in a foreground color, redrawing it in the background color, and then drawing a new foreground-colored ball in a new position. The ball reverses direction when it nears the edge of the viewing area. Use the Thread.sleep() method so there is enough time for the user to absorb the changes on the screen before it is redrawn. Save the program as **Pong.java** in the Chapter.17 folder on your Student Disk.

7. a. Write an applet that shows a yo-yo moving up and down on its string. Each time you redraw the yo-yo, let it sleep 100 milliseconds. Create an HTML file named **TestYoYo.html** that you can run in your browser to test the class. Save the program as **YoYo.java** in the Chapter.17 folder and save both the program and the HTML file on your Student Disk.

   b. Save the YoYo.java program as **YoYo1.java**. Then create a new **YoYo2.java** file in which the yo-yo sleeps for 150 milliseconds—slightly longer than YoYo1. Create a file named **TestYoYos.html** that you can run in your browser to watch both yo-yos at once. Save all the files in the Chapter.17 folder on your Student Disk.

8. Locate a shareware animated GIF file on the Web, and then include it in an applet. Save the program as **MyMovie.java** in the Chapter.17 folder on your Student Disk.

9. Write an applet that creates a straight line that constantly rotates in a circle. Save the program as **Baton.java** in the Chapter.17 folder on your Student Disk.

10. Write an applet that simulates a marquee by displaying a String of characters one-at-a-time from right-to-left across the screen. Use the Thread.sleep() method to pause momentarily before each new character displays. When the String message is fully displayed, start the message again. Save the program as **Marquee.java** in the Chapter.17 folder on your Student Disk.

11. Each of the following files in the Chapter.17 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugSeventeen1.java will become FixDebugSeventeen1.java. You can use the files testDebug3.html and testDebug4.html in the Chapter.17 folder of your Student Disk to test the applets in DebugSeventeen3.java and DebugSeventeen4.java, respectively. There are five additional files accompanying the Debug files. DebugSeventeen1.java uses DebugThreadA.java and DebugThreadB.java. DebugSeventeen2.java uses DebugSeventeen2Thread.java. You can use the html files TestDebug3.html and TestDebug4.html to run DebugSeventeen3.java and DebugSeventeen4.java, respectively.

    a. DebugSeventeen1.java

    b. DebugSeventeen2.java

    c. DebugSeventeen3.java

    d. DebugSeventeen4.java

CASE PROJECT

The Healthy Heart Association is sponsoring a walk-a-thon to raise money and wants an animated figure to display on its Web site. Write an applet that displays a figure that walks from left-to-right across the screen. When the figure nears the right side of the screen, start the figure at the left again. Save the program as **Walking.java**. Write an HTML document named **TestWalkers.html** that displays three walking figures and includes text describing a fund-raising walk-a-thon. Save both files in the Chapter.17 folder on your Student Disk.