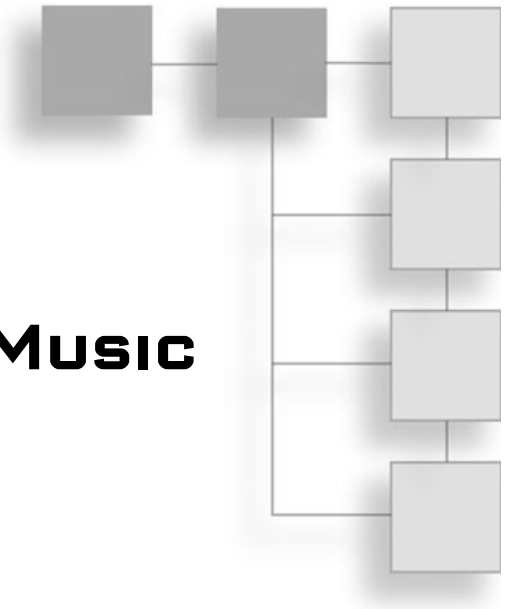
CHAPTER 9

# Sound Effects and Music



Java has a rich set of features for recording, mixing, and playing sound samples and MIDI sequences using a variety of classes that you will learn about in this chapter. You will learn about Java's rich set of sound support classes for loading and playing audio files in a variety of formats through Java's sound mixer. You will then learn about MIDI files and how to load and play them through Java's MIDI sequencer. Here is a rundown of the key topics in this chapter:

- Loading and playing digital files
- Loading and playing MIDI files
- Writing some reusable audio classes

## Playing Digital Sample Files

Java's Sound API provides a package for working with digital sample files, which has methods for loading a sample file (AIFF, AU, or WAV) and playing it through the sound mixer. The package is called javax.sound.sampled and includes numerous classes, most of which we will ignore. Some of these classes provide support for recording sound and manipulating samples, so you could write a complete sound editing program in Java that is similar to full-blown sound editing programs. One good example is Audacity–a freeware, open-source


**Figure 9.1**
Audacity is an excellent freeware sound editing program with many advanced features.

sound editor that is available for download at http://audacity.sourceforge.net (see Figure 9.1).

The Java Sound API supports the three main audio file formats used in web and desktop applications:

- AIFF
- AU
- WAV

The digital sample files can be 8-bit or 16-bit, with sample rates from 8 kHz to 48 kHz (which is CD quality). Java's Sound API includes a software sound mixer that supports up to 64 channels for both sound effects and background music for a Java applet.

**Tip**

For the latest information about the Java Sound API, point your web browser to java.sun.com/products/java-media/sound.

## Getting Started with Java Sound

The first step to writing some Java sound code is to include the javax.sound.sampled package at the top of your program:

```
import javax.sound.sampled.*;
```

If you are using JBuilder, you will see a pop-up menu to help you narrow down the class names within `javax`, which can be very educational. You'll see that when you type in `import javax.sound.`, JBuilder will show you the two classes available in `javax.sound.`, which are `sampled` and `midi`. By adding `.*` to the end of the import statement, you are telling the Java compiler to import every class within javax.sound.sampled, of which there are many.

In fact, when working with the sound system, you will need access to several classes, so it is convenient to import the associated packages at the start of a program so those classes are easier to use. For instance, without importing javax.sound.sampled, you would need to create a new sound sample variable using the full class path, such as:

```
javax.sound.sampled.AudioInputStream sample =
javax.sound.sampled.AudioSystem.getAudioInputStream(new File("woohoo.wav"));
```

Could you imagine what it would look like if you had to write all of your code like this? It would be illegible for the most part. Here is what the code looks like after you have imported javax.sound.sampled.*:

```
AudioInputStream sample = AudioSystem.getAudioInputStream(new File("woohoo.
wav"));
```

`AudioSystem` and `AudioInputStream` are classes within the javax.sound.sampled package and are used to load and play a sample in your Java applet. Later in this chapter, when I show you how to do background music, you'll get the hang of using some classes in a package called javax.sound.midi.

**Caution**

> You may run into a problem with the audio portion of your game, where your source code seems to be well written, without bugs, but you still get unusual errors. One of the most common sources of problems when working with audio data is an unsupported file format error. This type of exception is called `UnsupportedAudioFileException` and will be discussed later in this chapter.

If the program's flow runs through the `UnsupportedAudioFileException` block in your error handler, then the audio file may be encoded with an unsupported

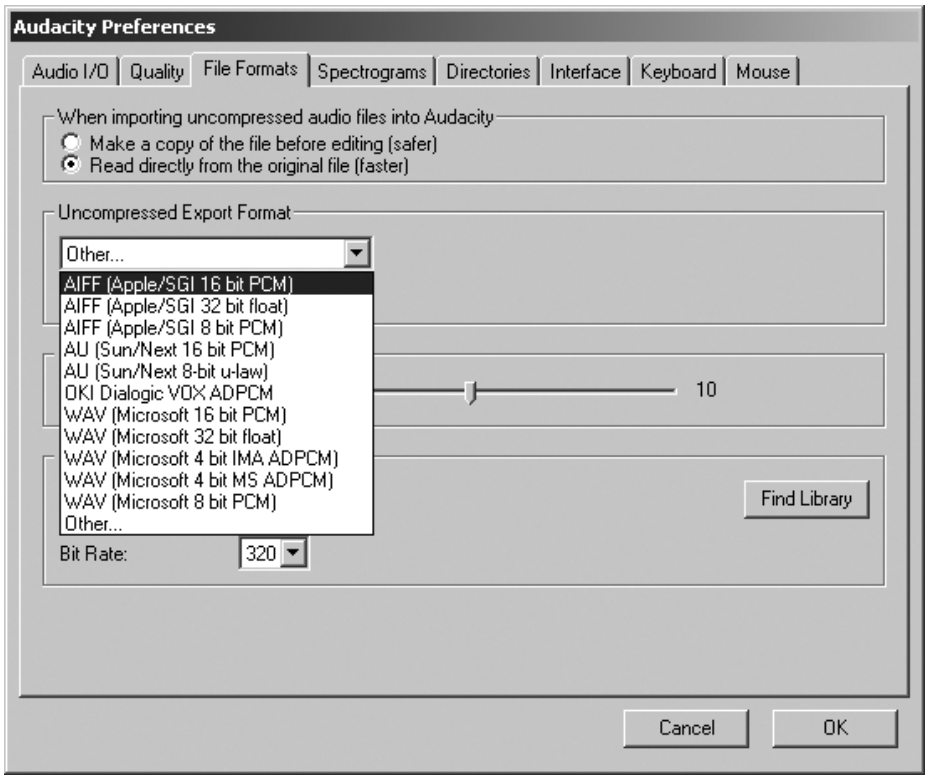



**Figure 9.2**
Changing the digital sample format settings in Audacity.

file format. The other, more obvious, problem is that the file itself might be missing.

You can check and convert audio files using the freeware Audacity program that I mentioned earlier. Just load up an audio file that you suspect is encoded in a weird format, and then save the file to a new format. Figure 9.2 shows the File Formats tab in the Audacity Preferences dialog box. Here you can change the default file format for exporting audio files from the File menu. If you choose the Other option from the drop-down list, you will be presented with *even more* audio formats, but most of them are obsolete. (For instance, you can save to Creative Labs' old VOC format, which was popular in MS-DOS games many years ago.) Some of the custom formats require an additional download of a plug-in for that particular sound format.

The key to sound programming is a class called `AudioInputStream`. This class is used to load a sound file (which can be an AIFF, AU, or WAV file) from either a local file on the current web server, where the applet is located, or from a remote URL anywhere on the Internet. An input stream is a source of data. You can create a new instance of the class like so:

```
AudioInputStream sample;
```

This statement is usually specified as a global variable within the class, defined up at the top of the class before any methods. You can define this variable as private, public, or protected. (The default, if you do not specify it, is public.) In object-oriented terms, *public* specifies that the variable is visible to other classes outside the current class, *private* means the variable is hidden to the outside world, and *protected* is similar to private, except that subclasses (through inheritance) have access to otherwise hidden variables defined as protected.

The code to load a sound from a file or URL is usually called from an applet's `init` method. The method used to load a sound is `AudioSystem.getAudioInputStream`. This method accepts a `File`, `InputStream`, or `URL` object; in addition, there are two other ways to create an audio stream (`AudioFormat` and `Encoding`), neither of which is useful for our needs.

```
sample = AudioSystem.getAudioInputStream(new File("humbug.wav"));
```

Note that the return value of this method is an `AudioInputStream`. Also, since `getAudioInputStream` does not offer an overloaded version that just accepts a `String` for a filename, you must pass a `File` object to it instead. This is easy enough using a `new File` object, passing the filename to the `File`'s constructor method. If you want to grab a file from a URL, your code might look something like this:

```
URL url = new URL("http://www.jharbour.com/test.wav");
sample = AudioSystem.getAudioInputStream(url);
```

Either way, you then have access to a sound file that will be loaded when needed. However, you can't just use an `AudioInputStream` to play a sound; as the class name implies, this is just a source of sample data without the ability to play itself. To play a sample, you use another class called `Clip` (`javax.sound.sampled.Clip`). This class is the return value of an `AudioSystem` method called `getClip`:

```
Clip clip = AudioSystem.getClip();
```

### Loading Resources

The code presented here will load a sound file correctly when your Java program is running either on your local PC or in a web browser. However, we need to use a slightly different method to load a file out of a Java archive. This subject is covered in Chapter 16, "Galactic War: Web Deployment," which covers web deployment.

But I want to prepare you for distributing your Java programs on the web *now,* so that your programs will *already* be ready for deployment. To that end, you must replace the `new File()` and `new URL()` methods to load a resource (such as an image or sound file) with the following code instead: `this.getClass().getResource()`. The `getResource()` method is found in the current class instance, `this.getClass()`. You will find it most useful if you use

`this.getClass().getResource()` anytime you need to build a URL. Here is a method I've written that accomplishes that goal:

```
private URL getURL(String filename) {
    URL url = null;
    try {
        url = this.getClass().getResource(filename);
    }
    catch (Exception e) { }
    return url;
}
```

Then, when you get to Chapter 16, the programs you've written will be ready for web deployment in a compressed Java archive (JAR)! During your explorations of the Java language while writing games and other programs, you will likely come up with many useful methods such as `getURL()`. You may want to store them in a reusable package of your own designation. The root package might be called `jharbour`, and then I would add subpackages to this, such as `jharbour.graphics`, `jharbour.util`, and so on. Since `getURL()` is the only custom reusable method repeatedly used in the book, it is more convenient to just include it in every class.

Since we don't need to pass a parameter to `getClip`, you might be wondering how this object knows what to play. There's actually one more step involved because at this point, all you have is a sound clip object with the *capability* to load and play an audio file or stream. This method actually returns a sound clip object from the default system mixer.

### *Loading the Sound Clip*

At this point, you have an `AudioInputStream` and a `Clip`, so you just need to open the audio file and play it. These steps are both performed by the `Clip` class. First, let's open the sound file:

```
clip.open(sample);
```

### *Playing the Sound Clip*

Next, there are two ways to play a clip, using the `Clip` class. You can use the `start()` method or the `loop()` method to play a sample. The `start()` method simply plays the sound clip.

```
narrator.start();
```

On the other hand, the `loop` method provides an option that lets you specify how many times the clip will repeat, either with a specific number of repeats or continuously. Here is how you might play a clip *one time* using the `loop` method:

```
explosion.loop(0);
```

Remember, the parameter specifies the number of times it will replay, as it's a given that the clip will always play at least once using the `loop` method. Here's how you can play a clip continuously:

```
thrusters.loop(Clip.LOOP_CONTINUOUSLY);
```

You might use this option if you have a music track that you would like to play repeatedly for the soundtrack of the game. Keep in mind, though, that sample files (AIFF, AU, and WAV) are quite large, so you wouldn't want the user to wait five minutes or longer (especially on dial-up) while the sound file is downloaded by your applet. This happens when you call the `open()` method, so if you try to open a huge sound file it will force the user to sit there and wait for an indeterminate length of time while the clip downloads. This is why I recommend using a MIDI sequence rather than a digital soundtrack for your game's background music.

**Tip**

MIDI is the acronym for *Musical Instrument Digital Interface*. MIDI is a synthesized music format, not a sampled format, meaning that MIDI music was not recorded using an analog-to-digital converter (which is built into your computer's soundcard). Professional musical instruments use the MIDI format to record *notes* rather than *samples*.

You may feel free to use the `Clip` class' `start()` method to play a sound clip, but I recommend using `loop(0)` instead. This type of call will give you the same result, and it will be easy to modify the method call if you ever want to repeat a sound clip once or several times. For instance, you might use this technique to save some bandwidth. Instead of downloading a two-second explosion sound effect, go for a one-half-second clip, and then repeat it four times. Always keep your mind open to different ways to accomplish a task, and look for ways to optimize your game.

**Tip**

As you will learn in Chapter 16, the Java Runtime Environment (JRE) provides an attractive applet download screen with a progress bar when you use a Java archive (JAR) to store the applet and all of its media files.

### *Stopping the Sound Clip*

Most of the time you will simply forget about a sound clip after it has started playing. After all, how often do you need to stop a sound effect from playing when there's a sound mixer taking care of all the details for you? Seldom, if ever. However, if you do need to stop a clip during playback, you can use the `stop()` method. I suspect the only time you will need this method is when you are looping a sample.

```
kaboom.stop();
```

### *Handling Errors*

One interesting aspect of the sound classes is that they *require* that errors be caught. The compiler will refuse to build a program using some of the sound classes without appropriate `try...catch` error-handling blocks. Since this is a new concept, I'll quickly explain it.

Java errors are handled with a special error-handling feature called a `try...catch` block. This feature was simply borrowed from the C++ language, on which Java was based. Here is the basic syntax of a `try...catch` block:

```
try {
    //do something bad
} catch (Exception e) {
}
```

When you add error handling to your program, you are "wrapping" an error handler around your code by literally wrapping a `try...catch` block around a section of code that you need to track for errors. The Java sound classes require `try...catch` blocks with specific types of error checks. The generic `Exception` class is used to catch most errors that are not caught by a more specific type of error handler. You can have many catch blocks in your error handler, from the more specific down to the more generic in nature.

**Tip**

In some cases, a `try...catch` error handler is *required* to handle exception errors that a particular method throws (on purpose). In those cases, your program *must* implement the appropriate error handler (such as `IOException`).

Another available version of the error handler is called `try...catch...finally`. This type of error-handling block allows you to put code inside the `finally` section in order to perform any cleanup or closing of files. The code in a `finally` block will

be run *regardless* of whether an error occurred. It gets executed if there *are* errors and if there are *no* errors.

For instance, if you are loading a file, you will first check for an `IOException` before providing a generic `Exception` handler. The `AudioSystem`, `AudioInputStream`, and `Clip` classes require the following error handlers:

- `IOException`
- `LineUnavailableException`
- `UnsupportedAudioFileException`

Let me show you how to implement an error handler for the audio code you're about to write for the PlaySound program. The following code is found in the `Applet.init()` event:

```
public void init() {
    try {
        //source code lines clipped
    } catch (MalformedURLException e) {
    } catch (IOException e) {
    } catch (LineUnavailableException e) {
    } catch (UnsupportedAudioFileException e) {
    }
}
```

I'll be the first person to admit that this is some ugly code. Error handling is notoriously ugly because it adds all kinds of unpleasant-looking management methods and events around your beautifully written source code. However, error handling is necessary and prevents your program from crashing and burning. I like to think of a `try...catch` block as a rev limiter that prevents a car engine from blowing itself up when a foolish driver hits the accelerator too hard.

### *Wrapping Sound Clips*

Since error handling is a necessary evil, it supports the argument that you may want to put some oft-used code into reusable methods of your own. A couple of methods to load and play a sound file would be useful (and that error-handling code could be bottled up out of sight). It would be logical to encapsulate the `AudioInputStream` and `Clip` objects into a new class of your own design with your

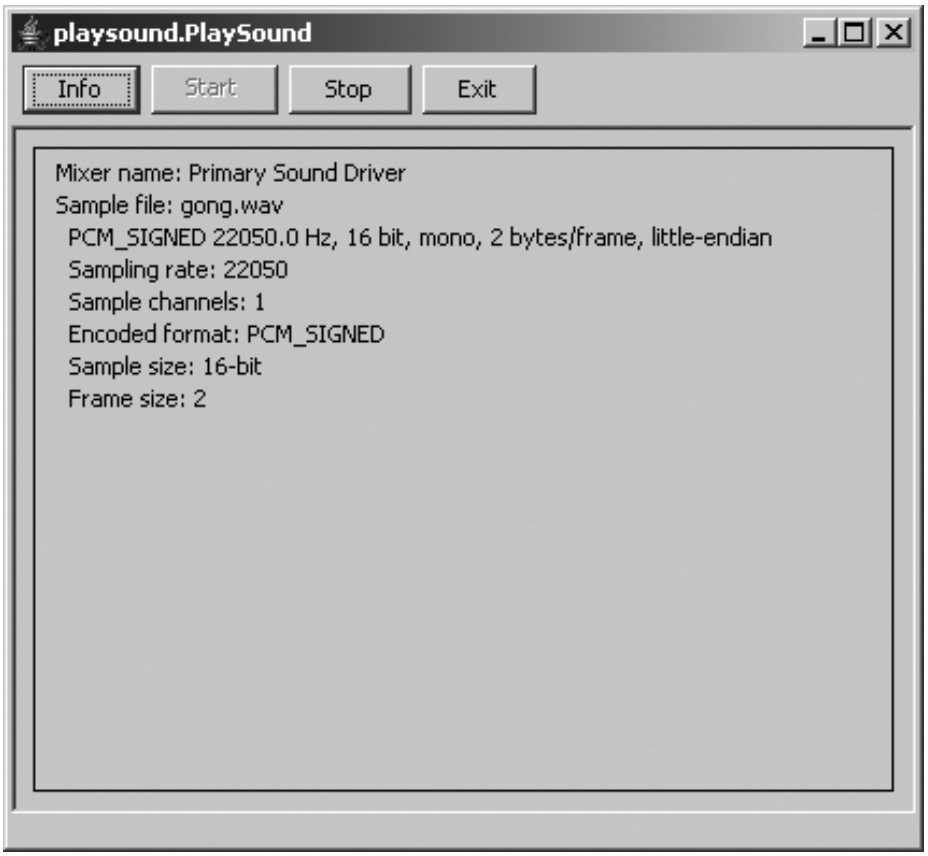



**Figure 9.3**
The PlaySound program demonstrates how to load and play a sound clip.

own methods to load and play a sound file or URL. Later in this chapter you will find source code for a class called `SoundClip` that does just that.

## Playing Sounds

The Java sound classes are not quite a "turnkey" programming solution, because you must perform several steps to load and play a sound file. I think it would be convenient to write a class that has a collection of sound clips you can load and play at any time from that single class, but I hesitate to "wrap" any Java code inside another class when it is such a heavily object-oriented language in the first place. Let's just write an example program to see how to put all this code to work. The resulting program, called PlaySound, is shown in Figure 9.3. The relevant code to this chapter is highlighted in bold.

```
import java.awt.*;
import java.applet.*;
import java.io.*;
```

```
import java.net.*;
import javax.sound.sampled.*;

public class PlaySound extends Applet {
    String filename = "gong.wav";
    AudioInputStream sample;

     private URL getURL(String filename) {
         URL url = null;
         try {
             url = this.getClass().getResource(filename);
         }
         catch (Exception e) { }
         return url;
    }
    //initialize the applet
    public void init() {
        try {
            sample = AudioSystem.getAudioInputStream(getURL(filename));
            //create a sound buffer
            Clip clip = AudioSystem.getClip();
            //load the audio file
            clip.open(sample);
            //play the sound clip
            clip.start();
    } catch (MalformedURLException e) {
        } catch (IOException e) {
        } catch (LineUnavailableException e) {
        } catch (UnsupportedAudioFileException e) {
        } catch (Exception e) { }
    }

    //the paint event handles the screen refresh
    public void paint(Graphics g) {
        int y = 1;
        g.drawString("Sample file: " + filename, 10, 15*y++);
        g.drawString("   " + sample.getFormat().toString(), 10, 15*y++);
        g.drawString("  Sampling rate: " +
            (int)sample.getFormat().getSampleRate(), 10, 15*y++);
        g.drawString("  Sample channels: " +
            sample.getFormat().getChannels() , 10, 15*y++);
        g.drawString("  Encoded format: " +
            sample.getFormat().getEncoding().toString(), 10, 15*y++);
```

```
            g.drawString("  Sample size: " +
                sample.getFormat().getSampleSizeInBits() + "-bit", 10, 15*y++);
            g.drawString("  Frame size: " +
                sample.getFormat().getFrameSize(), 10, 15*y++);
        }
    }
```

## Playing MIDI Sequence Files

Although using MIDI is not as popular as it used to be for background soundtracks in games, you have an opportunity to save a lot of bandwidth by using MIDI files for background music in a web-based game delivered as a Java applet. On the web, bandwidth is crucial, since a game that takes too long to load may cause a potential player to abandon the game and go elsewhere. For this reason, I would like to recommend the use of MIDI for in-game music when delivering a game through the web. Java supports three types of MIDI formats:

- MIDI Type 1
- MIDI Type 2
- Rich Music Format (RMF)

### Loading a MIDI File

Loading a MIDI file in Java is just slightly more involved than loading a digital sample file because a MIDI file is played through a sequencer rather than being played directly by the audio mixer. The `Sequence` class is used to load a MIDI file:

```
Sequence song = MidiSystem.getSequence(new File("music.mid"));
```

Although this code does prepare a MIDI file to be played through the sequencer, we haven't actually created an instance of the sequencer yet, so let's do that now:

```
Sequencer seq = MidiSystem.getSequencer();
```

Note that the `Sequencer` class can be accessed through `MidiSystem` directly, but it requires less typing in of code to create a local variable to handle the setup of the MIDI sequencer. Next, let's tell the `Sequencer` class that we have a MIDI file available (via the `Sequence` class):

```
seq.setSequence(song);
```

This line of code establishes a link between the sequencer and this particular MIDI sequence file. Now all that remains to do is actually open the file and play it:

```
seq.open();
seq.start();
```

At this point, the MIDI sequence should start playing when the applet window comes up.

## Playing Music

The following program listing demonstrates how to load and play a MIDI file in a Java applet window. The PlayMusic program is shown in Figure 9.4. As you can see, there are some minor details about the MIDI file that are displayed in the applet window, which is basically just an easy way to determine that the MIDI file has been loaded correctly. The key portions of code are highlighted in bold.


**Figure 9.4**
The PlayMusic program demonstrates how to load and play a MIDI sequence.

```
import java.awt.*;
import java.applet.*;
import java.io.*;
import java.net.*;
import javax.sound.midi.*;

public class PlayMusic extends Applet {
    String filename = "titlemusic.mid";
    Sequence song;

     private URL getURL(String filename) {
         URL url = null;
         try {
             url = this.getClass().getResource(filename);
         }
         catch (Exception e) { }
         return url;
    }

    //initialize the applet
    public void init() {
        try {
            song = MidiSystem.getSequence(getURL(filename));
            Sequencer sequencer = MidiSystem.getSequencer();
            sequencer.setSequence(song);
            sequencer.open();
            sequencer.start();

        } catch (InvalidMidiDataException e) {
        } catch (MidiUnavailableException e) {
        } catch (IOException e) { }
    }

    //repaint the applet window
    public void paint(Graphics g) {
        int x=10, y = 1;
        if (song != null) {
            g.drawString("Midi File: " + filename, x, 15 * y++);
            g.drawString("Resolution: " + song.getResolution(), x, 15 * y++);
            g.drawString("Tick length: " + song.getTickLength(), x, 15 * y++);
            g.drawString("Tracks: " + song.getTracks().length, x, 15 * y++);
            g.drawString("Patches: " + song.getPatchList().length, x, 15 * y++);
        } else {
```

```
            g.drawString("Error loading sequence file " + filename, 10, 15);
        }
    }
}
```

## Reusable Classes

Now that you understand how to load and play sound clips and sequence files, let's put all of this useful (but scattered) code into two reusable classes that can be easily dropped into a project and used. Instead of dealing with all of the Java sound classes and packages, you will be able to simply create a new object from the `SoundClip` and `MidiSequence` classes, and then load up and play either a sample or sequence with a couple lines of code.

I should disclaim the usefulness of these classes for you, so you will know what to expect. Java's Sound API has a sound mixer that works very well, but we can't tap into the mixer directly using the `Clip` class that I've shown you in this chapter. The sound files that you load using the `Clip` class *do support mixing,* but a single clip will interrupt itself if played repeatedly. So, in the case of Galactic War, when your ship fires a weapon, the weapon sound is restarted every time you play the sound. However, if you have another clip for explosions (or any other sound), then it *will be mixed* with any other sound clips currently playing.

In other words, a single `Clip` object cannot mix *with itself,* only *with other sounds.* This process works quite well if you use short sound effects, but can sound odd if your sound clips are one second or more in length. (They sound fine at up to about half a second, which is typical for arcade-game sound effects.) If you want to repeatedly mix a single clip, there are two significant options (and one other unlikely option):

- Load the sound file into multiple `Clip` objects (such as an array), and then play each one in order. Whenever you need to play this specific sound, just iterate through the array and locate a clip that has finished playing, and then start playing it again.
- Load the sound file into a single `Clip` object, then copy the sample bytes into multiple `Clip` objects in an array, and then follow the general technique described in the first option for playback. This saves time from loading the clip multiple times.

- Write a threaded sound playback class that creates a new thread for every sound that is played. The thread will terminate when the sound has completed playing. This requires some pretty complex code, and there is a lot of overhead involved in creating and destroying a thread for every single sound that is played. One way to get around this overhead is to create a *thread pool* at the start of the program and then reuse threads in the pool. Again, this is some very advanced code, but it is how professional Java games handle sound playback. If you write a great Java game suitable for publishing (such as Galactic War, which you will start in the next chapter and develop throughout the book), I would recommend one of the first two options as good choices for a simple game. You don't want to deal with the overhead (or weighty coding requirements) of a threaded solution, and an array of five or so duplicates of a sound clip can be played to good effect—with mixing.

## The SoundClip Class

The `SoundClip` class encapsulates the `AudioSystem`, `AudioInputStream`, and `Clip` classes, making it much easier to load and play an audio file in your applets. On the CD-ROM there is a project called SoundClass that demonstrates this class. This class simply includes all of the code we've covered in the last few pages, combined into a single entity. Note the key portions of code that I've discussed in this section, which are highlighted in bold.

**Tip**

A complete project demonstrating this class is available on the CD-ROM in the \sources\chapter09\SoundClassfolder.

```
import javax.sound.sampled.*;
import java.io.*;
import java.net.*;

public class SoundClip {
    //the source for audio data
    private AudioInputStream sample;

    //sound clip property is read-only here
    private Clip clip;
    public Clip getClip() { return clip; }
```

```
//looping property for continuous playback
private boolean looping = false;
public void setLooping(boolean _looping) { looping = _looping; }
public boolean getLooping() { return looping; }

//repeat property used to play sound multiple times
private int repeat = 0;
public void setRepeat(int _repeat) { repeat = _repeat; }
public int getRepeat() { return repeat; }

//filename property
private String filename = "";
public void setFilename(String _filename) { filename = _filename; }
public String getFilename() { return filename; }

//property to verify when sample is ready
public boolean isLoaded() {
    return (boolean)(sample != null);
}

//constructor
public SoundClip() {
    try {
        //create a sound buffer
        clip = AudioSystem.getClip();
    } catch (LineUnavailableException e) {   }
}

//this overloaded constructor takes a sound file as a parameter
public SoundClip(String audiofile) {
    this();  //call default constructor first
    load(audiofile); //now load the audio file
}

 private URL getURL(String filename) {
     URL url = null;
     try {
         url = this.getClass().getResource(filename);
     }
     catch (Exception e) { }
     return url;
}
```

```
    //load sound file
    public boolean load(String audiofile) {
        try {

            //prepare the input stream for an audio file
            setFilename(audiofile);
            //set the audio stream source
            sample = AudioSystem.getAudioInputStream(getURL(filename));
            //load the audio file
            clip.open(sample);
            return true;

        } catch (IOException e) {
            return false;
        } catch (UnsupportedAudioFileException e) {
            return false;
        } catch (LineUnavailableException e) {
            return false;
        }
    }

    public void play() {
        //exit if the sample hasn't been loaded
        if (!isLoaded()) return;

        //reset the sound clip
        clip.setFramePosition(0);

        //play sample with optional looping
        if (looping)
            clip.loop(Clip.LOOP_CONTINUOUSLY);
        else
            clip.loop(repeat);
    }

    public void stop() {
        clip.stop();
    }
}
```

## The MidiSequence Class

The `MidiSequence` class is another custom class that makes it easier to work with the Java sound code. This class encapsulates the `MidiSystem`, `Sequencer`, and

`Sequence` classes, making it much easier to load and play a MIDI file with just two lines of code instead of many. Take note of the key portions of code, which have been highlighted in bold.

**Tip**

A complete project demonstrating this class is available on the CD-ROM in the \sources\chapter09\MidiSequence folder.

```
import java.io.*;
import java.net.*;
import javax.sound.midi.*;

public class MidiSequence {
    //primary midi sequencer object
    private Sequencer sequencer;

    //provide Sequence as a read-only property
    private Sequence song;
    public Sequence getSong() { return song; }

    //filename property is read-only
    private String filename;
    public String getFilename() { return filename; }

    //looping property for looping continuously
    private boolean looping = false;
    public boolean getLooping() { return looping; }
    public void setLooping(boolean _looping) { looping = _looping; }

    //repeat property for looping a fixed number of times
    private int repeat = 0;
    public void setRepeat(int _repeat) { repeat = _repeat; }
    public int getRepeat() { return repeat; }

    //returns whether the sequence is ready for action
    public boolean isLoaded() {
        return (boolean)(sequencer.isOpen());
    }

    //primary constructor
    public MidiSequence() {
        try {
            //fire up the sequencer
```

```
            sequencer = MidiSystem.getSequencer();
        } catch (MidiUnavailableException e) {   }
    }
    //overloaded constructor accepts midi filename
    public MidiSequence(String midifile) {
        this();  //call default constructor first
        load(midifile); //load the midi file
    }

    private URL getURL(String filename) {
         URL url = null;
         try {
              url = this.getClass().getResource(filename);
         }
         catch (Exception e) { }
         return url;
    }

    //load a midi file into a sequence
    public boolean load(String midifile) {
        try {
            //load the midi file into the sequencer
            filename = midifile;
            song = MidiSystem.getSequence(getURL(filename));
            sequencer.setSequence(song);
            sequencer.open();
            return true;
        } catch (InvalidMidiDataException e) {
            return false;
        } catch (MidiUnavailableException e) {
            return false;
        } catch (IOException e) {
            return false;
        }
    }

    //play the midi sequence
    public void play() {
        if (!sequencer.isOpen()) return;
        if (looping) {
            sequencer.setLoopCount(Sequencer.LOOP_CONTINUOUSLY);
            sequencer.start();
        } else {
```

```
            sequencer.setLoopCount(repeat);
            sequencer.start();
        }
    }

    //stop the midi sequence
    public void stop() {
        sequencer.stop();
    }

}
```

## What You Have Learned

This chapter explained how to incorporate sound clips and MIDI sequences into your Java applets. Game audio is a very important subject because a game is just no fun without sound. You learned:

- How to load and play a digital sound file
- How to load and play a MIDI sequence file
- How to encapsulate reusable code inside a class

## Review Questions

The following questions will help you to determine how well you have learned the subjects discussed in this chapter. The answers are provided in Appendix A, "Chapter Quiz Answers."

1. What is the name of Java's digital sound system class?
2. What is the name of Java's MIDI music system class?
3. Which Java class handles the loading of a sample file?
4. Which Java class handles the loading of a MIDI file?
5. What type of exception error will Java generate when it cannot load a sound file?
6. Which method of the MIDI system returns the sequencer object?
7. What is the main Java class hierarchy for the audio system class?

8. What is the main Java class hierarchy for the MIDI system class?
9. What three digital sound file formats does Java support?
10. What rare exception error will occur when no MIDI sequencer is available?

## On Your Own

Use the following exercises to test your grasp of the material covered in this chapter. Are you ready to put sound and music to the test in a real game yet? These exercises will challenge your understanding of this chapter.

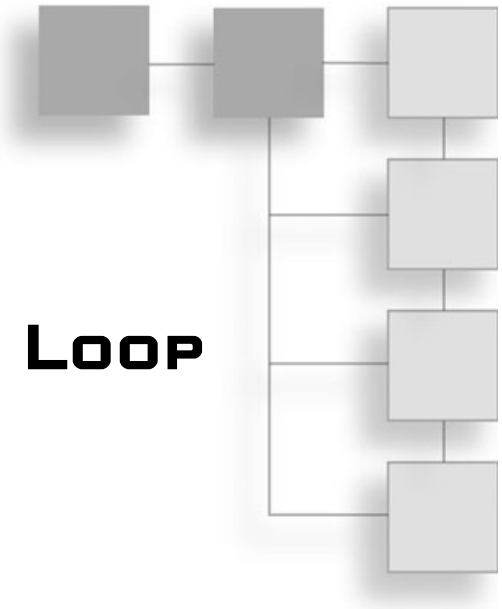
### Exercise 1

Write your own sound-effects generating program to try out a large list of sound files. You can acquire sound files of various types by searching the web. Have the program play a specific sound file by pressing keys on the keyboard.

### Exercise 2

Write a similar program for playing back multiple MIDI music sequence files by pressing various keys on the keyboard. For an even greater challenge, try combining this program with the one in Exercise 1 so that you can try out playing music and sound effects at the same time!

CHAPTER 10



# Timing and the Game Loop

You have learned how to use the `Graphics2D` class to program graphics using vector shapes and bitmap images, and you have even seen a nearly complete game written from scratch. You have learned how to load and play sound files and MIDI music files, and how to program the keyboard and mouse. By all accounts, you have the tools to create many different games already. But there are some tricks of the trade—secrets of the craft—that will help you to make your games stand out in the crowd and impress. This chapter discusses the game loop and its vital importance to a smooth-running game. You will learn about threads and timing, and you will take the *Asteroids*-style game created in Chapter 5 into completely new territory, as it is modified extensively in the following pages.

Here are the specific topics you will learn about:

- Overriding default applet methods
- Using timing methods
- Starting and stopping a thread
- Using a thread for the game loop
- Building the Galactic War game

## The Potency of a Game Loop

The key to creating a game loop to facilitate the needs of a high-speed game is Java's multithreading capability. Threads are such an integral part of Java that it makes a special thread available to your program just for this purpose. This special thread is called `Runnable`, an interface class. However, it's entirely possible to write a Java game without threads by just using a simple game loop. I'll show you how to do this first, and then we'll take a look at threads as an even better form of game loop.

**Tip**

An *interface class* is an abstract class with properties and methods that are defined but not implemented. A program that uses an interface class is said to *consume* it, and must implement all of the public methods in the interface. Typical examples include `KeyListener` and `Runnable`.

## A Simple Loop

The `Runnable` interface gives your program its own awareness. I realize this concept sounds a lot like artificial intelligence, but the term *awareness* is a good description of the `Runnable` interface. Before `Runnable`, your Java programs have been somewhat naive, capable of only processing during the screen refresh. Let's take a look at an example. Figure 10.1 shows the SimpleLoop program. As you

**Figure 10.1**
The SimpleLoop program.

can see, this program doesn't do anything in real time; it waits until you press a key or click the mouse before drawing a rectangle.

```
/*************************************************************
* SimpleLoop program
 *************************************************************/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.util.*;

public class SimpleLoop extends Applet implements KeyListener, MouseListener {

    Random rand = new Random();

    public void init() {
        addKeyListener(this);
        addMouseListener(this);
    }

    public void paint(Graphics g) {
        Graphics2D g2d = (Graphics2D) g;

        //create a random rectangle
        int w = rand.nextInt(100);
        int h = rand.nextInt(100);
        int x = rand.nextInt(getSize().width - w);
        int y = rand.nextInt(getSize().height - h);
        Rectangle rect = new Rectangle(x,y,w,h);

        //generate a random color
        int red = rand.nextInt(256);
        int green = rand.nextInt(256);
        int blue = rand.nextInt(256);
        g2d.setColor(new Color(red,green,blue));

        //draw the rectangle
        g2d.fill(rect);
    }

    //handle keyboard events
    public void keyReleased(KeyEvent k) { }
    public void keyTyped(KeyEvent k) { }
```

```
    public void keyPressed(KeyEvent k) {
        repaint();
    }

    //handle mouse events
    public void mouseEntered(MouseEvent m) { }
    public void mouseExited(MouseEvent m) { }
    public void mouseReleased(MouseEvent m) { }
    public void mouseClicked(MouseEvent m) { }
    public void mousePressed(MouseEvent m) {
        repaint();
    }
}
```

**Tip**

The `Random` class is located in the `java.awt.util` class along with many other utility classes that provide commonly needed tools to your program. To use `Random` in your program, you must include this class with the following `import` statement:

```
import java.awt.util.*;
```

This program has no loop whatsoever, so it cannot process anything in real time—not space ships, asteroids, jumping Italian plumbers, yellow dot-eaters, or female spelunkers packing dual Berettas. The only thing this program can do is draw a single rectangle.

There are some ways you can make the program a little more interactive. The only problem with the mouse and keyboard listener interfaces is that you have to implement *all* of them or *none* of them. That reminds me of Yoda's famous saying, "Do, or do not. There is no 'try.'" An interface class is an all-or-nothing proposition that tends to junk up your source code, not to mention that it takes a lot of work to type in all of those interface event methods every time! But there's no real workaround for the unused event methods.

**Note**

I've been thinking about a way to use all of these interface classes (such as `Runnable` and the input listeners) by tucking them away into a class outside of the main program. This support class would provide my main program with real *events* when things happen, such as key presses, mouse movement, and other events. Perhaps this is the birth of an idea that will become some sort of game engine? Let's wait and see what Part III, "The Galactic War Project," has in store.

## Overriding Some Default Applet Behaviors

There is a serious problem with this program because it was supposed to just *add* a new rectangle every time the user presses a key or mouse button, not *redraw* the entire applet window—with a single rectangle left over. There is definitely something odd going on because this program *should have* worked as expected.

Well, it turns out that Java has been screwing with the screen without permission—or rather, by *default*. The `Applet` class, which is the basis for the SimpleLoop program (recall that it *extends* `Applet`), provides many default event methods that do certain things for you. You don't even need to implement `paint()` if you don't want to, and the `Applet` base class will provide it for your program. Granted, nothing will be drawn on the window as a result, but the compiler won't give you an error. This differs from an interface class (such as `KeyListener`) that *mandates* that you must implement all of its abstract methods. So it's pretty obvious by this difference in functionality that `Applet` is not an interface class, but a fully functioning class.

What happens, then, when you implement an `Applet` class method such as `init()` or `paint()`? These methods are essentially empty inside the `Applet` class. Oh, they exist and are not abstract, but they don't *do anything*. The `Applet` class defines these methods in such a way that you can *override* them in your applet. For instance, in the SimpleLoop program, SimpleLoop is actually the name of the class, and it inherits from `Applet`. Therefore, SimpleLoop has the opportunity to override any of the methods in `Applet` that it wants to, including `init()` and `paint()`.

However, there's another method that we haven't used yet called `update()`. Aha! No, I wasn't holding out on you—you've actually used this method before, in the game project back in Chapter 3. The `update()` method actually *does* do something as coded in the `Applet` class—it calls `repaint()` to refresh the applet window. (Light-bulb moment.)

Since the default `update()` method has been refreshing the screen for our programs, we have had absolutely no control over this process. That explains why only one rectangle was being drawn at a time in the SimpleLoop program—`update()` was refreshing the screen on its own. In a sense, your Java program is a *slave* to the master `Applet` class until you override the functionality by rewriting its methods. It's time to break the bonds of object-oriented slavery. Let's add the `update()` method to the program:

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

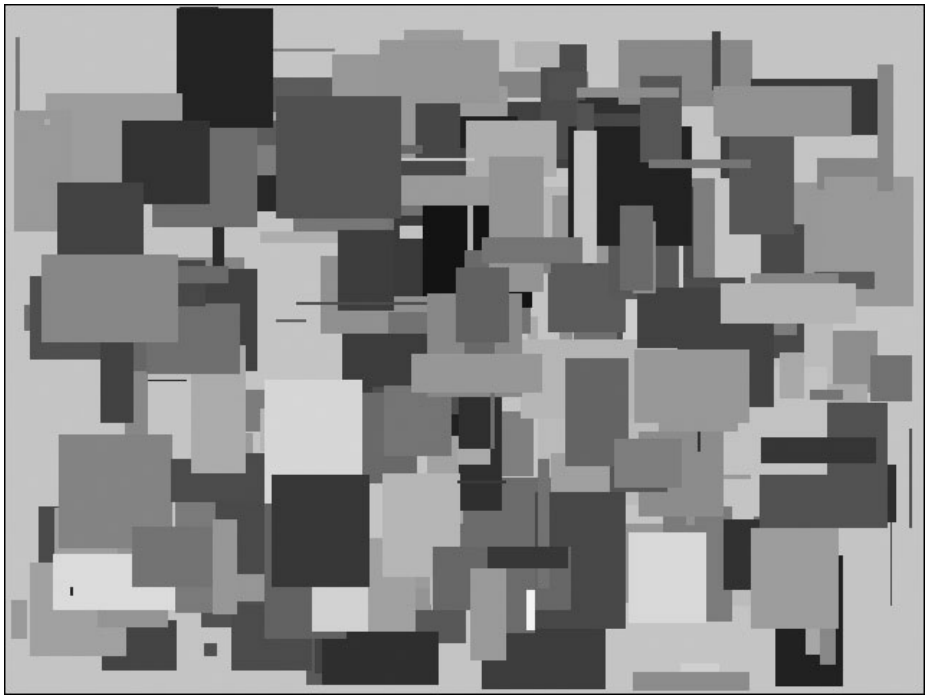


**Figure 10.2**
The SimpleLoop program now draws many rectangles.

As you can see, the `update()` method is bearing a single line of code, a call to the `paint()` method. This gives you complete control over the screen refresh because the default `update()` would not just repaint the screen, it would also *clear the screen*. Now you can run the SimpleLoop program and see a bunch of rectangles as originally expected, as shown in Figure 10.2.

I think this example makes it pretty clear that any serious game will need to override the `update()` method as well as `paint()`. But that doesn't really address the subject at hand—what about the loop? I've been calling this program SimpleLoop when no loop is even being used. Let's get to that now.

## Feeling Loopy

There are quite a few `Applet` methods available that we haven't implemented yet, in addition to the standard methods you've seen so far. I'll go over the remaining key `Applet` methods at the end of this section to make you aware of them. For now, I'd like to introduce you to the next `Applet` method: `start()`. The `start()` method is invoked in your Java applet by the web browser right after calling

`init()`. So, you can use `init()` to get the game ready to go, and then use `start()` to get things moving.

Now you finally have an opportunity to add a real loop to this program. But just for kicks, what do you think would happen if you added a `while()` loop to the `init()` method? I tried it, so you should try it too. Doing this will lock up the applet, which will not even be displayed. Why? Because `init()` is called before the applet has even brought up the web browser or applet viewer, so putting a loop here prevents the applet from drawing. However, this is not the case with the `start()` method, which is called after the applet has been initialized and is ready to go. The engine has been started and is just waiting for you to press the accelerator at this point.

You can build on this game loop, too. The call to repaint the window is only the last step in a game. First, you move your game objects around on the screen, perform collision testing, and so forth. You can perform all of these steps in the `start()` event and then call `repaint()` at the very end. Here is a suggestion. This is *not* actual code that you should add to the SimpleLoop program—it's just an example.

```
public void start() {
    //the game loop
    while (true) {
        //move the game objects
        updateObjects();

        //perform collision testing
        testCollisions();

        //redraw the window
        repaint();
    }
}
```

## Recovering Long-Lost Applet Methods

Learning about the `start()` and `update()` events was intriguing. Let's take a look at some more helpful methods in the `Applet` class that will be useful.

The `showStatus()` method is used to print some text in the bottom status bar of the browser window or applet viewer program:

```
showStatus(String msg)
```

The `isActive()` method determines whether the applet is active, which is marked just before the `start()` event method is called:

```
boolean isActive()
```

The `start()` method is called after `init()` and is often used to resume the applet after the user has browsed away from the applet page and then returned.

```
void start()
```

The `stop()` method is called by the web browser when the web page changes, signifying that the applet is about to be destroyed.

```
void stop()
```

The `destroy()` method is called by the browser when the applet is about to be destroyed (removed from memory).

```
void destroy()
```

## Stepping Up to Threads

We use the `Runnable` interface class to add threading support to a game. This interface has a single event method called `run()` that represents the thread's functionality. We can create this `run()` method in a game, and that will act like the game loop. The thread just calls `run()` once, so you must add a `while` loop to this method. The `while` loop will be running inside a thread that is separate from the main program. By implementing `Runnable`, a game becomes multithreaded. In that sense, it will support multiple processors or processor cores! For instance, if you have an Intel Core2 Duo or another multi-core processor, you should be able to see the applet running in a thread that is separate from the web browser or applet viewer program.

### Starting and Stopping the Thread

To get a thread running in an applet, you have to create an instance of the `Thread` class, and then start it running using the applet's `start()` event. First, let's create the thread object:

```
Thread thread;
```

Next, create the thread in the `start()` event method:

```
public void start() {
```

```
    thread = new Thread(this);
    thread.start();
}
```

Then you can write the code for the thread's loop in the `run()` event method (part of `Runnable`). We'll take a look at what goes inside `run()` in a moment. First, let's take a look at the `stop()` event. This method is provided by the `Applet` class and can be overridden. It has no functionality by default. This is a convenient place to kill the thread, so let's do that:

```
public void stop() {
    thread = null;
}
```

## The ThreadedLoop Program

Let's take a look at the entire ThreadedLoop project, and then I'll explain the loop for the thread inside the `run()` method. The key portions of code have been highlighted in bold.

```
/**************************************************
* ThreadedLoop program
 **************************************************/
import java.awt.*;
import java.lang.*;
import java.applet.*;
import java.util.*;

public class ThreadedLoop extends Applet implements Runnable {
    //random number generator
    Random rand = new Random();

    //the main thread
    Thread thread;

    //count the number of rectangles drawn
    long count = 0;

    //applet init event
    public void init() {
        //not needed this time
    }
```

```
//applet start event
public void start() {
    thread = new Thread(this);
    thread.start();
}

//thread run event
public void run() {
    long start = System.currentTimeMillis();

    //acquire the current thread
    Thread current = Thread.currentThread();

    //here's the new game loop
    while (current == thread) {
        try {
            Thread.sleep(0);
        }
        catch(InterruptedException e) {
            e.printStackTrace();
        }

        //draw something
        repaint();

        //figure out how fast it's running
        if (start + 1000 < System.currentTimeMillis()) {
            start = System.currentTimeMillis();
            showStatus("Rectangles per second: " + count);
            count = 0;
        }
    }
}

//applet stop event
public void stop() {
    thread = null;
}

//applet update event
public void update(Graphics g) {
    paint(g);
}
```

```
    //applet paint event
    public void paint(Graphics g) {
        Graphics2D g2d = (Graphics2D) g;

        //create a random rectangle
        int w = rand.nextInt(100);
        int h = rand.nextInt(100);
        int x = rand.nextInt(getSize().width - w);
        int y = rand.nextInt(getSize().height - h);
        Rectangle rect = new Rectangle(x,y,w,h);

        //generate a random color
        int red = rand.nextInt(256);
        int green = rand.nextInt(256);
        int blue = rand.nextInt(256);
        g2d.setColor(new Color(red,green,blue));

        //draw the rectangle
        g2d.fill(rect);

        //add another to the counter
        count++;
    }
}
```

Now let's examine this `run()` event that is called by the `Runnable` interface. There's a lot going on in this method. First, a variable called `start` gets the current time in milliseconds (`System.currentTimeMillis()`). This value is used to pause once per second to print out the total number of rectangles that have been drawn (see Figure 10.3).

Next, a local variable is set to the current thread, and then a `while` loop is created.

```
Thread current = Thread.currentThread();
```

This local thread makes sure that our loop only processes thread events intended for the game loop because you can use multiple threads in a program.

```
while (current == thread)
```

The core of the `thread` loop includes a call to `Thread.sleep(0)`, which is a placeholder for slowing the game down to a consistent frame rate. Right now it's running as fast as possible because the `sleep()` method is being passed a 0. This single method call requires an error handler because it throws an `InterruptedException` if the

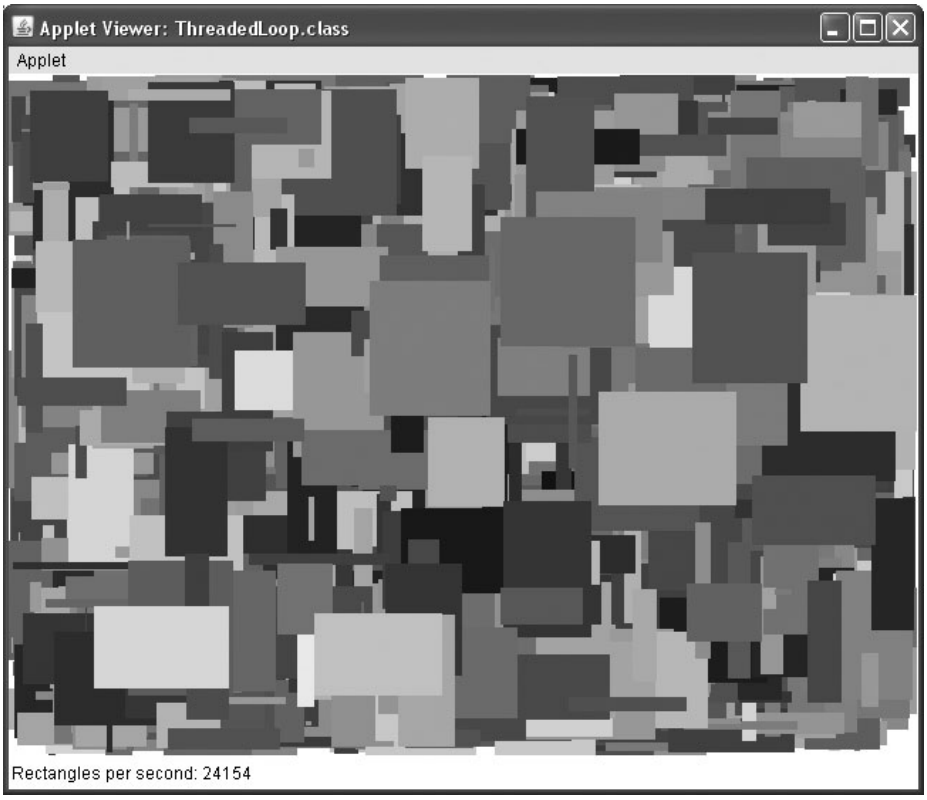



**Figure 10.3**
The ThreadedLoop program displays the number of rectangles drawn per second.

thread is ever interrupted by another thread. This is an advanced subject that doesn't concern us. If the thread is interrupted, it's not a big deal—we'll just ignore any such error that might crop up.

```
try {
    Thread.sleep(0);
}
catch(InterruptedException e) {
    e.printStackTrace();
}
```

After this call to sleep (which will slow the game to a consistent frame rate), then we can call game-related methods to update objects on the screen, perform collision testing, perform game logic to interact with enemies, and so on. In the block of code that follows, you can see some timing code inside an `if` statement. This code prints out the number of rectangles that have been drawn by the `paint()` event during the past 1,000 milliseconds (which equals 1 second).

```
//draw something
repaint();
```

```
//figure out how fast it's running
if (start + 1000 < System.currentTimeMillis()) {
    start = System.currentTimeMillis();
    showStatus("Rectangles per second: " + count);
    count = 0;
}
```

The single call to `repaint()` actually makes this program *do something*; all of the rest of the code helps this method call to do its job effectively. Here inside the `run()` event, which houses the new threaded game loop, I've used one of the new `Applet` methods. The `showStatus()` method prints out a string to the status bar at the bottom of the Web browser or applet viewer.

## Examining Multithreading

Aside from the sample game in Chapter 3, this program might have been your first exposure to multithreaded programming. Java makes the process very easy compared to other languages. I've used a thread library called pthread to add thread support in my C++ programs in Linux and Windows, and it's not very easy to use at all compared to Java's built-in support for threads. We will continue to use threads in every subsequent chapter, so you will have had a lot of exposure to this subject by the time you complete the book.

**Note**

*Game Programming All In One, Third Edition* (Thomson Course PTR, 2006) covers the pthread library and many other cross-platform game programming topics, and is based around the Allegro library and the C++ language.

# What You Have Learned

This was a heavyweight chapter that covered some very difficult subjects. But the idea is to get up the most difficult part of a hill so you can reach the peak, and then head on down the other side. That is what this chapter represents—the last few steps up to the peak. You have now learned the most difficult and challenging aspects of writing a Java game at this point, and you are ready to start heading down the hill at a more leisurely pace in the upcoming chapters. This chapter explained:

- How to create a threaded game loop
- How to override default applet methods
- How to manipulate a bitmap with transformations