3 Programs

Thus far this book has not used the word “program” and, instead, has used the phrase “software product”. This chapter explores a (somewhat) formal definition of the word “program”, and considers different types of Java programs. It then borrows from the existing types of Java programs to design and build a type of Java program that is especially useful in the context of multimedia software products.

3.1 Java Programs

For the purposes of this book, a program (in an object-oriented programming language) is defined as follows:

Definition 3.1 A *program* in an object-oriented programming language is a group of cooperating classes with a well-defined *entry point* (i.e., a method that should be executed first) and, perhaps, a reentry point and/or an exit point.

Java has supported many different kinds of programs over the years. On ‘desktop’ and ‘laptop’ computers, the two most common programs are ¡em¿applications¡/em¿ and ¡em¿applets¡/em¿. On ‘small’ devices (like mobile phones), the two most common are ¡em¿MIDlets¡/em¿ (where MID is an acronym for “mobile information devices”) and ¡em¿apps¡/em¿. On ‘servers’ (especially HTTP servers) the most common are ¡em¿servlets¡/em¿. The differences between them are summarized in Table 3.1. In short, they differ in the way they are executed and in the top-level container they use.

	Environment	**Top-Level Container**	**Entry Point**
Applet	Browser	`JApplet`	`init()` then `start()`
Application	Operating System	`JFrame`	`main()`
MIDlet	Operating System	`Screen`	`startApp()`
Servlet	(HTTP) Server	N/A	`init()` then `service()`

Table 3.1 Java Programs

Applets normally run in a virtual machine inside of a WWW browser.[1] They are executed

[1] Applets can actually run inside of a variety of different desktop/laptop programs. For simplicity, this book focuses on the most common situation. When run inside of a browser, applets are generally limited/restricted in a variety of ways. For example, applets in a browser typically can’t read or write files on the client, run executables on the client, or communicate with any machine other than the originating host.

when the browser loads an HTML page that contains an `<applet>` element. When such a page is first loaded, the applet's `init()` and `start()` methods are called. An applet's top-level container is normally a `JApplet` object. Applications run in a virtual machine directly under the operating system (OS), and are normally executed by clicking on them (or using the `java` command) which causes the OS to call the `main()` method. An application's principle top-level container is normally a `JFrame` object (though it may use multiple `JFrame` and/or `JDialog` objects). MIDlets run in a virtual machine directly under the operating system, and are normally executed by clicking on them which causes a MIDlet manager to invoke the `startApp()` method. A MIDlet's top-level container is normall a `Screen` object (though it may be a `Canvas` object or, indeed, any `Displayable`). Finally, servlets run in a virtual machine inside of a server, typically a WWW (i.e., HTTP) server. They are constructed by the server and initialized with a call to `init()`. When a server receives a GET or POST request that requires a particular servlet, it calls its `service()` method. Since midlet run inside of a server, they are "headless" (i.e., they do not have a top-level container.

Servlets, while very important in some settings, are not relevant in a book that focuses on user-facing multimedia products. MIDlets, while important for many years, have been supplanted on phones by apps (which are not "pure" Java). Applets were also important for many years, but are no longer supported on any major browser (for reasons that are beyond the scope of this book). Hence, this book will only consider applications from here on. However, it will make use of what has been learned by the other kinds of programs (especially applets) to create an improved variety of application. Most importantly, it will create a application with an improved life-cycle.

When an application is started the `main()` method is executed in a non-daemon thread that this book refers to as the main thread. A single-threaded application terminates when the `System.exit()` method is called, in response to a platform-specific event such as a `SIGINT` or a `Ctrl-C`, or when the main thread 'drops out of' the `main()` method.[2] A multi-threaded application terminates when the `System.exit()` method is called, in response to a platform-specific event, or when all non-daemon threads have died.[3] Note that all GUI applications are, intrinsically, multi-threaded since they have both a main thread and an event dispatch thread.

The life-cycle of an applet is quite different than that of an application, and much more appropriate for multimedia software products. When an HTML page containing an `<applet>` element is loaded into a browser for the first time, the appropriate object (i.e., the descendent of the `Applet` class referred to in the `<applet>` element) is constructed and its `init()` and `start()` methods are called in a thread other than the event dispatch thread. Then, each time the user leaves the page containing the applet, the `stop()` method is called (again, not

[2]This is actually referred to as an *orderly* termination. An application can also by terminated *abruptly* by calling the `Runtime.halt()` method in the `Runtime` class.

[3]The `addShutdownHook()` method in the `RunTime` class can be used to perform specific tasks during the termination process. To do so, one creates a *shutdown hook* (a `Thread` object that has not been started) and passes it to the `Runtime` object (obtained by a call to the static method `Runtime.getRuntime()`). At the onset of the termination process, each of the shutdown hooks is started. Unfortunately, the shutdown hooks are started in no particular order. So, if you need to have the shutdown tasks performed in a particular order they should be performed by a single shutdown hook. Some people argue that shutdown tasks should be performed in the `finalize()` method of each object. It is certainly possible to do it this way since, after all of the shutdown hooks have died, the `finalize()` method will be called on all objects that have not had their `finalize()` method called previously (i.e., after they were garbage collected). However, it is very difficult to write `finalize()` methods that behave correctly.

in the event dispatch thread). Similarly, each time the user reloads the page containing the applet, the `start()` method is called. When the browser is shut down, the `destroy()` method is called (again, not in the event dispatch thread). As with GUI applications, all applets are multithreaded (since all applets have a GUI even if they don't use it in any meaningful way).

To further understand the shortcomings of the standard application lifecycle, and how it can be improved, it is helpful to consider a simple example. In particular, consider a variant of the application from Chapter 2 that displays a random message when a button is pressed. In this variant, rather than requiring input from the user, the application changes the message every second.

```
// Java libraries
import javax.swing.*;

// Multimedia libraries
import event.*;

public class BadTimedMessageSwingApplication implements MetronomeListener,
                                                        Runnable
{
  private static final String[] MESSAGES = {
     "What a great book.","Bring on the exercises.",
     "Author, author!","I hope it never ends."};

  private int       index;
  private JLabel    label;
  private Metronome metronome;

  public static void main(String[] args)
  {
    try
    {
      SwingUtilities.invokeAndWait(new BadTimedMessageSwingApplication());
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }


  public void handleTick(int millis)
  {
    index = (index + 1) % MESSAGES.length;
    label.setText(MESSAGES[index]);
  }
```

```
  public void run()
  {
    // Setup the window
    JFrame frame = new JFrame();
    frame.setSize(400,  200);

    // Setup the content pane
    JPanel contentPane = (JPanel)frame.getContentPane();
    contentPane.setLayout(null);

    // Add a component to the container
    label = new JLabel("  ", SwingConstants.CENTER);
    label.setBounds(0, 0, 400, 200);
    contentPane.add(label);

    metronome = new Metronome(1000);
    metronome.addListener(this);
    metronome.start();

    frame.setVisible(true);
  }
}
```

The shortcoming of this application is that the message continues to be changed when the `JFrame` is iconified meaning the user might miss one of the very important messages (and wasting system resources). If, on the other hand, this program were an applet, the `Metronome` could be stopped in its `stop()` method (i.e., when the user left the page containing the applet) and re-started in its `start()` method (i.e., when the user returned to the page containing the applet).

Fortunately, with a little thought, these same kinds of "hooks" can be added to an application.

3.2 An Applet-Like Application

Since the same kind of functionality will be required in every multimedia application, it makes sense to construct a class that can be specialized for specific purposes. For applets (that use the Swing windowing toolkit), this is the role played by the `JApplet` class. So, the objective here is to design and implement a `JApplet` class that plays the same role for applications.

3.2.1 An Applet-Like Entry Point

Two things are required to provide an applet-like entry point. First, an `JApplication` object must have an `init()` method that is called at initialization-time. Second, this method must

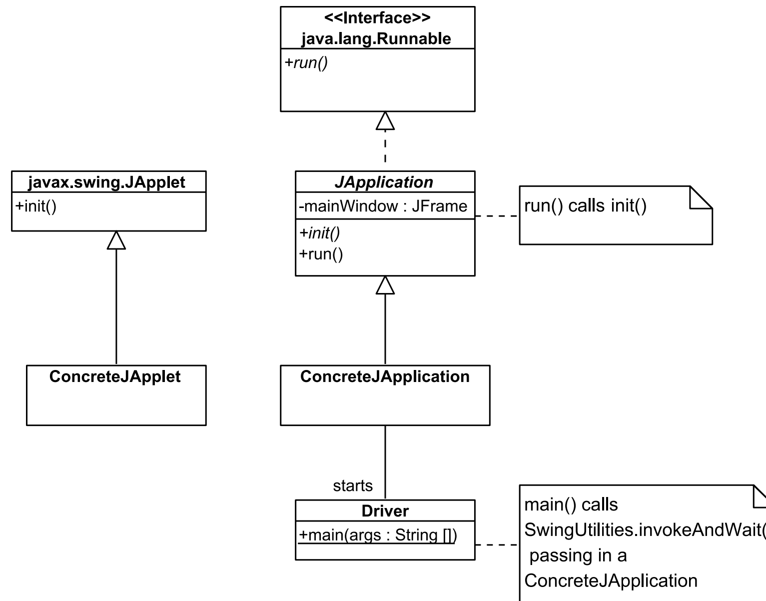


Figure 3.1 Initial Design of a `JApplication` Class

be called in the event dispatch thread (since all GUI tasks, including initialization tasks, must be performed in the event dispatch thread.

As mentioned in Chapter 2 the second specification can be satisfied using the `invokeAndWait()` method in the `SwingUtilities` class, which is passed an object that implements the `Runnable` interface. So, the `JApplication` class must implement the `Runnable` interface. The second specification can then be satisfied by having the `run()` method in the `JApplication` class call an `init()` method. This is illustrated in Figure 3.1.

As a convenience, the `JApplication` class includes a static `invokeInEventDispatchThread()` method. This method calls the `invokeAndWait()` method in the `SwingUtilities` class and, in the event of an exception, displays an appropriate error message.

```
    protected static void invokeInEventDispatchThread(Runnable runnable)
    {
      try
      {
        SwingUtilities.invokeAndWait(runnable);
      }
      catch (Exception e)
      {
        JOptionPane.showMessageDialog(null,
            "Unable to start the application.",
```

```
            "Error", JOptionPane.ERROR_MESSAGE);
      }
   }
```

The run() method in the JApplication class is then implemented as follows:

```
   public final void run()
   {
      constructMainWindow();
      init();
      mainWindow.setVisible(true);
   }
```

The constructMainWindow() method in the JApplication class constructs a JFrame, sets its properties, and sets the properties of the content pane. Note that this JFrame does not allow the user to resize it. This is both for convenience and to make it consistent with the main container in a JApplet (which cannot be resized).

```
      mainWindow = new JFrame();
      mainWindow.setTitle("Multimedia Software - jblearning.com");
      mainWindow.setResizable(false);

      contentPane = (JPanel)mainWindow.getContentPane();
      contentPane.setLayout(null);
      contentPane.setDoubleBuffered(false);
```

The init() method is abstract so that it must be implemented by all concrete specializations.

```
   public abstract void init();
```

3.2.2 An Applet-Like Lifecycle

At this point only the entry point of JApplication and JApplet objects are similar. In particular, eecall that a JApplet has its transition methods called by the browser when the page containing the JApplet is loaded/unloaded (see the discussion on page 44). Ideally, the transition methods in JApplication objects would be called at corresponding times. This can be accomplished by making JApplication a WindowListener on its main window.

To do so, the following is added to the constructMainWindow() method:

```
    mainWindow.setDefaultCloseOperation(
                              JFrame.DO_NOTHING_ON_CLOSE);
    mainWindow.addWindowListener(this);
```

The first statement instructs the main window to do nothing when the "close" button is clicked. The second statement registers the `JApplication` as a `WindowListener`.

Now, it is necessary to actually implement the `WindowListener` interface. When a `windowOpened()` message is generated, the `start()` method must be called.

```
 public void windowOpened(WindowEvent event)
 {
    resize();
    start();
 }
```

The same is true when a `windowDeiconfied()` message is generated.

```
 public void windowDeiconified(WindowEvent event)
 {
    start();
 }
```

When a `windowIconified()` message is generated, the `stop()` method must be called.

```
 public void windowIconified(WindowEvent event)
 {
    stop();
 }
```

When a `windowClosing()` message is generated, the `exit()` method must be called.

```
 public void windowClosing(WindowEvent event)
 {
    exit();
 }
```

The `exit()` method asks the user to confirm and then calls the `stop()` method.

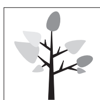
```
private void exit()
{
   int        response;

   response = JOptionPane.showConfirmDialog(mainWindow,
                                     "Exit this application?",
                                     "Exit?",
                                     JOptionPane.YES_NO_OPTION);

   if (response == JOptionPane.YES_OPTION)
   {
      mainWindow.setVisible(false);
      stop();
      mainWindow.dispose();
   }
}
```

Finally, when a `windowClosed()` message is generated (which happens after the `windowClosing()` message is generated and the `stop()` method is called), the `destroy()` method is called.

```
public void windowClosed(WindowEvent event)
{
   destroy();
   System.exit(0);
}
```

3.2.3 Providing an Applet-Like Top-Level Container

As discussed in Chapter 2, a program shouldn't use its top-level container directly. Indeed, it shouldn't even know what kind of top-level container it has. To that end, the `JApplication` class should play the role of a `RootPaneContainer` (as does the `JApplet` class). It does so by delegating to the `JFrame` attribute it constructs in its `init()` method.



The final design is summarized in Figure 3.2 on the next page. The "getters" are implemented as follows:

```
public Container getContentPane()
{
   return mainWindow.getContentPane();
```

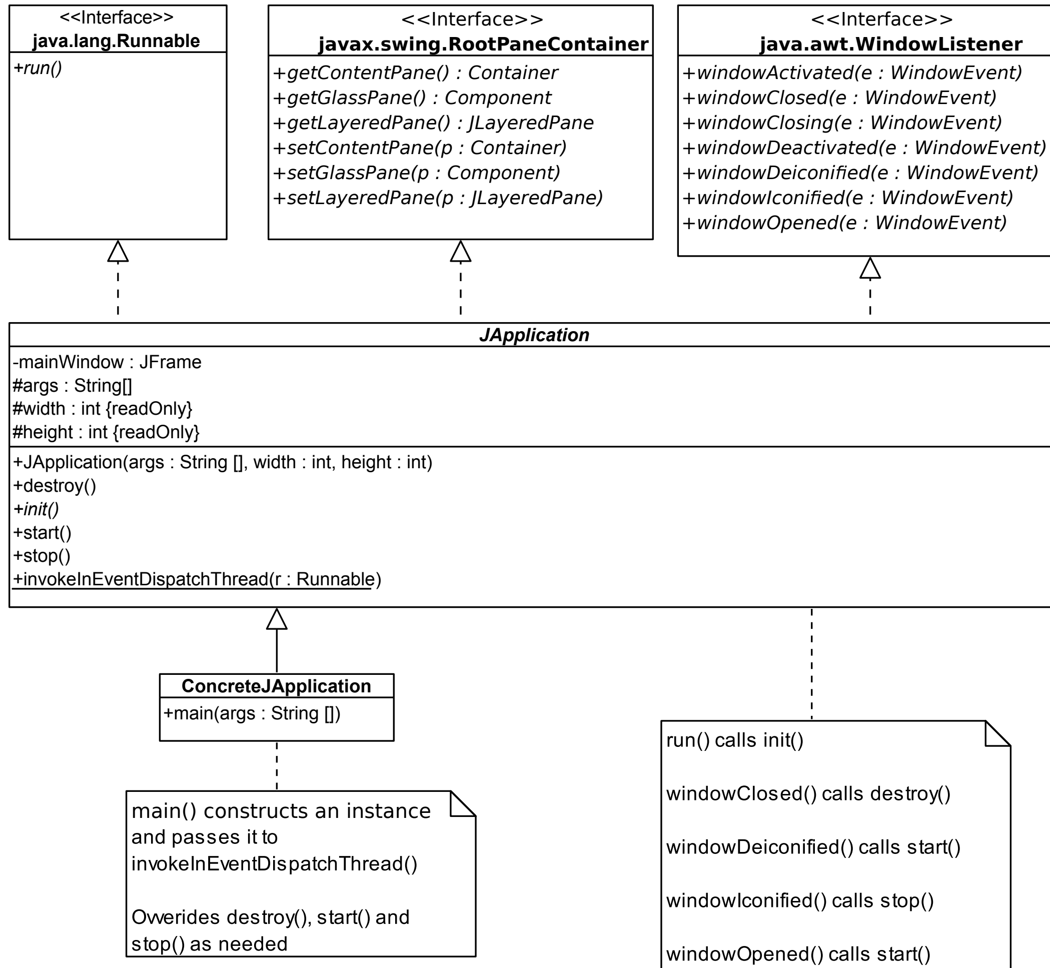


Figure 3.2 Final Design of the JApplication Class

```
   }

   public Component getGlassPane()
   {
      return mainWindow.getGlassPane();
   }

   public JLayeredPane getLayeredPane()
   {
      return mainWindow.getLayeredPane();
   }

   public JRootPane getRootPane()
```

```
    {
       return mainWindow.getRootPane();
    }
```

and the "setters" are implemented as follows:

```
    public void setContentPane(Container contentPane)
    {
       mainWindow.setContentPane(contentPane);
    }

    public void setGlassPane(Component glassPane)
    {
       mainWindow.setGlassPane(glassPane);
    }

    public void setLayeredPane(JLayeredPane layeredPane)
    {
       mainWindow.setLayeredPane(layeredPane);
    }
```

3.2.4 Specializing the `JApplication` Class

In the days when they were supported, one wrote an applet by extending the `JApplet` class. It is now possible to do the same kind of thing by extending the `JApplication` class (and writing any necessary supporting classes). Any code that needs to be executed when the window is iconified must be put in the `stop()` method (that overrides the empty implementation in the `JApplication` class. Simialrly, any code that (and writing any necessary supporting classes). Any code that needs to be executed when the window is first opened or de-iconified must be put in the `start()` method, and any code that needs to be executed when the window is closed must be put in the `destroy()` method. While it is usually easy to identify the code that must be in each of these methods, confusion can arise with respect to the `init()` method.

In particular, it may not be immediately clear what code should be in the `init()` method and what code should be in the constructor. While some "initialization" code can go in either location, it is essential to follow one rule – **all code that involves GUI components must be in the `init()` method**. This is because all instructions involving GUI components must be executed in the event dispatch thread (and the constructor will be executed in the main thread).

3.2.5 An Example

It's now easy to create a `TimedMessageJApplication` that provides the desired functionality discussed earlier. As before, it must implement the `MetronomeListener` interface but now it must also extend the `JApplication` class.

```
// Java libraries
import javax.swing.*;

// Multimedia libraries
import app.*;
import event.*;

public class TimedMessageJApplication extends JApplication
                                      implements MetronomeListener
{
  private static final String[] MESSAGES = {
     "What a great book.","Bring on the exercises.",
     "Author, author!","I hope it never ends."};

  private int       index;
  private JLabel    label;
  private Metronome metronome;
}
```

The constructor performs all of the initialization tasks that are not related to the GUI.

```
  public TimedMessageJApplication(String[] args, int width, int height)
  {
    super(args, width, height);
    index = -1;
  }
```

The `init()` method performs all of the initialization tasks that are related to the GUI.

```
  public void init()
  {
    // Setup the content pane
    JPanel contentPane = (JPanel)getContentPane();
    contentPane.setLayout(null);

    // Add a component to the container
```

```
    label = new JLabel("  ", SwingConstants.CENTER);
    label.setBounds(0, 0, 400, 200);
    contentPane.add(label);

    metronome = new Metronome(1000);
    metronome.addListener(this);
  }
```

The `start())` and `stop()` method start and stop the `Metronome` object respectively.

```
  public void start()
  {
    metronome.start();
  }

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

As before the `handleTick())` method changes the message.

```
  public void handleTick(int millis)
  {
    index = (index + 1) % MESSAGES.length;
    label.setText(MESSAGES[index]);
  }
```

Finally, the `main()` method constructs an instance and calls the (inherited) `invokeInEventDispatchThread()` method.

```
  public static void main(String[] args)
  {
    JApplication demo = new TimedMessageJApplication(args, 400, 200);
    invokeInEventDispatchThread(demo);
  }
```

3.3 Program Resources

Most multimedia programs, be they applications or applets, need to 'load' resources of various kinds (e.g., artwork, preferences) at run time. While this is not a problem conceptually, it can be somewhat problematic in practice because of the different ways in which applets and applications can be 'organized' (e.g., in a `.jar` file, in a packaged set of classes, in an unpackaged set of classes) and 'delivered/installed' (e.g., by an HTTP server, by an installer, as files on a CD/DVD). Hence, in practice, it can be very difficult for a program to know where resources are. Fortunately, with a little bit of effort, one can find resources in the same way that the Java interpreter does.

3.3.1 Finding Resources

The Java interpreter obtains the byte codes that constitute a class using a `class loader`.[4] Obviously, the class loader must be able to find the byte codes regardless of how the applet/application is 'organized' and 'delivered/installed'. Fortunately, this same logic can be used to load resources. To do so, one must first understand a little about reflection.

Every interface, class, and object in Java has an associated `Class` object that can be used to obtain information about that interface's/class's/object's attributes, methods, etc. This information is encapsulated as `Constructor`, `Field`, `Method` , and `Type` objects. These objects can be used for a variety of purposes. The `ResourceFinder` class that follows uses the `getResource()` and `getResourceAsStream()` methods in `Class` objects.

Since, in the future, it might be desirable to create pools of `ResourceFinder` objects, this class uses the factory method pattern as follows:

```
package io;

import java.io.*;
import java.net.*;
import java.util.*;


public class ResourceFinder
{
    private Class                c;

    private ResourceFinder()
    {
       c = this.getClass();
    }

```

[4] In fact, the Java interpreter uses three different kinds of class loaders. The bootstrap class loader loads system classes (e.g., from `rt.jar`), the extension class loader loads standard extensions, and the system/application class loader loads application classes (e.g., from the `classpath`).

```
    private ResourceFinder(Object o)
    {
       // Get the Class for the Object that wants the resource
       c = o.getClass();
    }

    public static ResourceFinder createInstance()
    {
       return new ResourceFinder();
    }

    public static ResourceFinder createInstance(Object o)
    {
       return new ResourceFinder(o);
    }
}
```

When constructed, a `ResourceFinder` object can be 'told' to use either its class loader (by calling the default factory method) or another object's class loader (by calling the explicit-value factory method).

The `findInputStream()` method in the `ResourceFinder` class uses the appropriate `Class` object (with the help of the class loader) to get a resource as an `InputStream`.

```
    public InputStream findInputStream(String name)
    {
       InputStream    is;

       is    = c.getResourceAsStream(name);

       return is;
    }
```

The resource can then be read from this `InputStream`.

In some situations, it may be more useful to have a 'pointer' to the resource, rather than an `InputStream`. In such situations, on can use a *uniform resource locator* (URL). URLs are encapsulated by the `URL` class and can be obtained (among other ways) using the `getResource()` method in the `Class` class.

```
    public URL findURL(String name)
    {
       URL            url;

       url   = c.getResource(name);
```

```
        return url;
    }
```

3.3.2 Marking the Location of Resources

The ResourceFinder makes it possible to load resources from a variety of different sources, however it does not solve every resource-related problem. In particular, by itself, it does not solve some of the resource-related problems introduced by modern integrated development environments (IDEs).

For organizational reasons, many IDEs keep the source code (i.e., the .java files) and the byte code (i.e., the .class files) in different directories/folders. For example, several keep the source code in a subdirectory (under the project directory) named src and the byte code in a subdirectory named bin. They then either copy resources into the bin directory at runtime, change the classpath at runtime, or both. This can mean that the structure of the file system can be different from the structure of the .jar file when the product is deployed.

Fortunately, a simple "trick" can be used to avoid any complications that this causes.[5] In particular, one can put the resources in a package (e.g., named resources) that includes a class (e.g., named Marker) that can be used to find them.

3.3.3 A Complete Example

The following TimedMessageDemo is almost identical to the TimedMessageJApplication except that, rather than hard-coding the messages in a String[], it reads them into an ArrayList<String> from a file named. messages.txt. It has the following structure:

```
public class TimedMessageDemo extends    JApplication
                              implements MetronomeListener
{
  private ArrayList<String> messages = new ArrayList<String>();

  private int               index;
  private JLabel            label;
  private Metronome         metronome;
}
```

and reads the messages (in the constructor) as follows:

[5] If one is intimately familiar with the way the IDE operates, this "trick" isn't necessary and should probably be avoided. However, it provides a fallback when all else fails, or when multiple different IDEs are being used.

```
    BufferedReader in     = new BufferedReader(new InputStreamReader(is));

    String line;
    try
    {
      while ((line = in.readLine()) != null)
      {
        messages.add(line);
      }
    }
    catch (IOException ioe)
    {
      messages.add("Best book ever!");
    }
```

So that it can find the messages regardless of how the application is deployed, the file named `messages.txt` is in a package named `resources`, that also contains the following `Marker` class.

```
package resources;

public class Marker
{
}
```

A `Marker` object can then be used with a `ResourceFinder` object to create the `InputStream` named `is` (that is used above to create the `BufferedReader`).

```
    ResourceFinder rf = ResourceFinder.createInstance(new resources.Marker());
    InputStream    is = rf.findInputStream("messages.txt");
```

EXERCISES

1. Using your answers to Exercises 6 on page 38, 7 on page 39, and 8 on page 39, create a class named `OnOffJApplication` providing the same functionality that extends `JApplication`. (Note: Remember that it must implement the `ActionListener` interface.)
2. Using your answers to Exercises 9 on page 40, 10 on page 40, and 11 on page 41, create a class named `TextBounceJApplication` that provides the same functionality and extends

`JApplication`. The text must stop bouncing when the window is iconified and must start bouncing again when it is deiconified.

3. Modify your answer to Exercise 2 so that `message` is initialized to the command-line argument 0.

4. Build and execute a version of the `TimedMessageDemo` using your preferred development environment.

5. Create an executable `.jar` file that contains the `TimedMessageDemo` and the file named `messages.txt`. Make sure you can execute the application **directly from the operating system**.

REFERENCES AND FURTHER READING

Boese, E.S. (2010) *An Introduction to Programming with Java Applets* Jones and Bartlett Publishers, Sudbury, MA.

Horstmann, C.S. and G. Cornell (2002) *Core Java: Volume I - Fundamentals* Sun Microsystems Press, Palo Alto, CA

Horstmann, C.S. and G. Cornell (2002) *Core Java: Volume II - Advanced Features* Sun Microsystems Press, Palo Alto, CA